Physics of Computational Abstraction

Douglas J. Matzke

Texas Instruments Incorporated
PO. Box 655474, MS 446
Dallas, Texas 75265
matzke@hc.ti.com

Abstract

This paper will discuss the idea that efficient computation depends on local space and time costs as seen by the services provided to the current layer of abstraction. The approach taken will be to replace the classical notions of space and time with the unified notion of SpaceTime from modern physics. The rest of the paper will be organized as follows. The history of key topics and their perspective of space and time will be discussed. Then I will justify the need for a unified notion of SpaceTime in order to show computation and information are not the same. Finally the conclusions and other issues will be discussed.

1 Introduction

It is evident from studying physics and computation, that seamless layering is required to build complex systems. Each layer has it's own set of primitives and rules which combine to produce totally new behaviors, while ignoring the details of the layers below. This paper will discuss how layers of abstraction in computational systems requires a computation theory that is quite different than information theory applied to storage or communications systems. Efficient computation requires something other than just reducing the number of bits. Efficient computation must minimize how those bits are arranged in both space and time.

2 History

Many historical references of information and computation deal with space and time aspects separately. Information theory when applied to communications systems, deals with information movement across space. Likewise, information theory when applied to memory systems, deals with information movement across time. Conventional hardware gates contain both space and time costs to compute an answer. Since the Von Neumann architecture, computer engineers have separated the memory (or spatial) and processor (or temporal) sides of computation. Even software programmers are taught that program = data structure + algorithm (or computation = space + time) [1]. This pervasive thinking is never questioned and shows up in database's which separate queries from data records, as well as architectures with separate program caches and data caches. Carver Mead's [2] approach also reinforces this notion of separation of space and time with his two costs to computation. They are 1) the cost of information being in the wrong place which he calls spatial entropy, and 2) the cost of information being in the wrong form which he calls logical entropy. Real computation is impossible without both space (memory/communication) and time (processor or change) resources, but segregating them seems to violate what physics has learned about space and time being inseparable.

Computer software must not be ignored in thinking about physics of computation, because today's software is tomorrow's architectures and hardware. Since all modern computers are Turing equivalent, software (and the underlying compilers and operating systems) is the domain of modern computation. At the 1981 conference, Dr Hillis [3] stated that computer science is no good because it is missing many of the things that make the laws of physics so powerful -- locality, symmetry and invariance of scale. An important addition to the list is that computer science has an outdated notion of space and time (ie a classical view) and needs to adopt the modern unified spacetime of physics. This added notion coexists very nicely with the three points of Hillis, as will be discussed in more detail later. Hillis touches on the kernel of this idea in his paper when he says the memory locations are just wires turned sideways in time. Fredkin and Toffoli [4] also unified communication and memory with the unit wire primitive by stating, from a relativistic viewpoint there is no distinction between storage and transmission of signals. Fredkin's [5] [6] [7] Digital Mechanics work also takes a different view of space and time by saying they are discrete. Just as special relativity changed how physicists think about spacetime, computer scientists also need an updated view.

Computer science research on active data types is taking a more unified view of SpaceTime. This approach is different from the classical view so it persists mainly at universities and has not been adopted by industry. The two primitives from this work, functions and continuations, are both active data types. Type is an orthogonal property of a data structure other than its value. These primitives are particularly interesting for the following reasons 1) they are the building blocks of abstract layering in programs, 2) they are the basis of most modern compiler work, 3) hardware support for them is showing up in many architectures, and 4) they unify the notion of data and program.

Much of the research on functions and continuations is being done in languages that are dialects of Lisp (CLOS, Scheme) and other functional programming languages (ML). Functional programming is a style of programming that limits side effects and always expects a value to be returned (in Lisp everything is a function). Procedural languages always side effect something globally to return a result. The work on functions and continuations is mathematically rigorous (formalized syntax and semantics from lambda calculus) and has resulted in compilers producing more reliable yet faster code. Schools like MIT [8] and CMU [9] teach their introductory programming classes in Scheme or Lisp in order to teach students how to program using high level abstractions, and then this thinking can be applied to other languages. Functions represent the primitive for building layers of abstraction in software, and is a fundamental aspect of teaching computer science. A continuation is the equivalent to the process primitive for Scheme.

A function represents a piece of active data. Functions exist in all computer languages, but they are usually not explicitly manipulated by user programs. In traditional languages, only special programs such as compilers, linkers, loaders and debuggers manipulate functions. In lisp dialects, functions are like any other native data type in the language (can be passed around and modified) plus it returns values when executed. In lisp languages, every function can return a value or even another function. Thus you can easily write programs that build and execute other programs. This is very useful if you are building simulators (we built a compiled code discrete time simulator using an array of functions where the designer could move forward or backward through time [10]) or other specialized execution environments that run at native machine speeds. Most computer architectures have some hardware to support calling and returning values from functions. Many RISC machines were designed with specific extra hardware to optimize function calling overhead (ie overlapping register windows). A first class function represents a primitive computational building block that has both space and time potential associated with it, and seems more in tune with the physics notion of a unified spacetime.

Each modular function can ignore the details of the functional layers below and treat those lower services (using the appropriate interfaces) as primitive and atomic. This thinking is the basis of many software reliability methodologies. In order to view the lower services as atomic, each functional layer can be thought of having it's own local operations and its own local time scale. J.T. Frazier [11], who founded the International Institute for the Study of Time, has written many books on the subject of time layers. This is important because culturally we have very little temporal intuition compared to our spatial experience. Locality, symmetry, and invariance of scale are properties that should exist for both space and time.

This notion of local time layering can be best described via an example of nested transactions. Many commercial data processing systems are implemented using transactions. These primitive atomic requests for information or action can be viewed as a quantum of data processing because each transaction either succeeds or fails atomically. Atomic action means that from the perspective of the requester, this action is indivisible (in time). This idea of local time based on abstract layering and perspective, becomes even more interesting when nested transactions are considered. One transaction can spawn a tree of subtransactions, each having the same indivisible atomic behavior in time, therefore collectively representing a hierarchy of local times.

Local perspective and local time passage seem to be a fundamental aspect of building layers of abstractions and has striking parallels to physics. Relativity has shown that both space and time are interlocked and are dependent on the motion of the reference frame. As a reference frame approaches the speed of light, yard sticks get longer and clocks slow down in contrast to other reference frames.

This warping of spacetime continues until the speed of light is reached, (only achieved by massless photons). From the perspective of a photon, time stands still, that is, the emission and absorption as seen by the photon are the exact same instant. This indivisible notion of local time passage is strikingly similar to the passage of time for transactions. This notion of relative time and relative space will be addressed more later in a discussion about incremental computation in graphs.

A function is a powerful abstract building block for implementing complex programs but some specialized programs such as debuggers and operating systems need something even more powerful. A continuation is a native data type in Scheme and is also executable like a function. It is an active data structure that preserves the state of the rest of the computation. Using functions and continuations, you are able to implement an operating system (including interrupt handling and process switching) in a high level language. You can also implement native debuggers and exception handlers. There was a research program at MIT that designed a computer called the L-machine that was a scalable, fine grained parallel architecture that used continuations as a primitive data element in the architecture. For many years, the formal definitions of continuations has been used in compiler research (continuation-passing style).

Functional programming, layers of abstraction, and space/time tradeoffs all come together with a software technique called memoization [12] (yes the spelling is correct). Memoization is a technique of caching computed functional values inorder to speed up an algorithm, usually without having to rewrite the algorithm. Memoization is brushed aside by many as just a cute optimization, but this paper will show it to be more fundamental than that. Memoization is an approach that lets us transition from thinking about computation (from a classical view) and jump to a metacomputation view, where relative space/time costs represent a quadratic cost function (energy like) for dealing with computation efficiency. The next section will show how the use of memoization, in functionally organized programs, requires an active data view of computation that has symmetry, locality, and invariance of scale.

3 Computational Costs

This section will a take few small programming examples use them to illustrate both the classical way of thinking about space and time, and the unified approach. These examples were chosen (Fibonacci and Towers of Hanoi) because in their simplest form they are exponential algorithms. These examples will illustrate both programming style and memoization.

The algorithm to compute the Fibonacci series can be described elegantly as a recursive function in Scheme as follows [13]. In the following form, lambda denotes a function with an argument called NUMBER that is stored in the symbol called FIB. Each call to FIB creates two lower calls to the function FIB, producing an exponentially growing number of calls [$O(2^n)$].

```
(define fib
        (lambda (number)
            (if (< number 2)
                (max 0 number)
                (+ (fib (- number 1))
                   (fib (- number 2)))) ))
```

```
(Fib 15)  ->   610  takes <1 second
(Fib 20)  ->  6765  takes >7 seconds
(Fib 25)  -> 75025  takes >70 seconds
(Fib 100) -> does not finish
```

This example seems trivial and the implementation naive, but this problem is equivalent to the Tower of Hanoi example and many other problems in industry (ie simulation [14]). The Tower of Hanoi consists of three poles with a set of graduated disks stacked on one pole. The problem requires to move the stack of disks to another pole by only moving one disk at a time, and never placing a bigger disk on top of a smaller one. People who play with this problem soon see a strategy for solving it that involves recursively moving all the disks but the last one to another pole, moving the last disk to the third pole, and then moving all the other disks back on top of this disk. Even though this problem is naturally recursive and could also be implemented using a functional programming style (with no side effects), most people implement it procedurally by globally side effecting the position of the disks on the poles. Thus, programmers tend to bind a particular space and time scenario in their programs, and most of the time this is done in an ad hoc fashion.

Compilers currently due nothing to optimize this straight forward Fibonacci algorithm into a fixed space and time volume. Notice the time is exponential, but the space is linear due to the stack depth. Formally knowing the space/time costs for an abstract algorithm would be very useful for compilers. This kind of analysis is an unsolved problem in general, but is the focus of much attention as designers use simulation languages to build silicon (ie VHDL and C). Companies like Synopsis routinely manipulate the space/time costs of circuits, but since this analysis is an exponential problem, the size of the circuit is limited to around 5000 gates at one time. Thus architects and programmers alike must rigidly define and hand tune their algorithm to get the performance they need.

In many application specific areas (like DSPs or Fuzzy Logic) the same restricted input language can map to both software and hardware implementations. Regular language compilers must also know more about architecture performance considerations. Compilers for pipelined RISC machines can effectively due useful work the instruction after a jump because the pipeline is never broken. Vectorizing compilers for vector array processors also optimize iteration variables to keep from thrashing memory systems and keep the pipe lines full. Most of these compiler optimization techniques are based on only the analysis of the code, but do no resource sharing analysis and thus can not automatically handle even the simple case of Fibonacci above. Hand tuning of the implementation costs for hardware implementations is possible due to good tools that measure/optimize the size and timing of circuits. Designers must also specify the number of wires, thus limiting the size of problem that can be solved. Good tools are practically nonexistent for general programming languages but some are available for some vector processors.

Commercial compilers that automatically optimize or control space/time tradeoffs are far off. This is important because automatic optimizing of code to give different space/time characteristics is critical for the future of our industry, especially with the general purpose multiprocessors hitting the market or the advent of different architectures (and thus compilers) for controlling nanoelectronic implementations. Some concepts and techniques that can head us in the right direction will be described next.

4 Computational Abstraction and Spacetime

Memoization can be applied to both examples from the last section to give linear time behavior. Given the maximum size of the problem you want to solve, can place an upper limit on the space requirement. Memoize can be written in scheme as follows.

```
(define memoize
    (lambda (function)
        (let ((table (maketable)))
            (lambda (arg) ;function is returned
                (or (gettable table arg)
                    (puttable table arg
                              (function arg)))) )))
 (define fib
     (memoize ;encapsulates Fibonacci
         (lambda (number)
             (if (< number 2) (max 0 number)
                (+ (fib (- number 1))
                   (fib (- number 2)))) ))

(Fib 15)  ->    610  takes  <1 second
(Fib 20)  ->  6765  takes  <1 second
(Fib 25)  -> 75025  takes  <1 second
(Fib 100) ->354224848179261915075 < 3 secs
```

A generalized memoize function can be written to handle any number of calling function arguments and any kind of caching strategy. It can also be given controls to turn itself off completely or control the size or sparseness of the cache [ie (if (mod arg 5) <cache> <no cache>)]. This ability to give different space/time controls without rewriting or recompiling the algorithm could allow adaptive cache controllers (ie even ones with timeouts) and allow compilers using static analysis to automatically insert cache controls or generate hardware descriptions. Parallel implementations of algorithms could also be scheduled better.

This same memoization technique can be applied to Towers of Hanoi except that when the matching operation is based on the value of the disk positions on the poles, a large space storage would be required to store all the answers. An alternative would be to use a pattern matching function that was based on the number of disks to move, and the position of the disks on the poles (upon entering and exiting). This function would create a cache that would contain knowledge about how to move N disks. This knowledge about how to implement an N disk move is based on experience (from pattern matching on poles) and if used to speed up the computation really represents a violation of the game's rules of only moving one disk at a time. Is this cheating or chunking? Psychologist Robert Ornstein [15] showed that people's perception of time passage could be changed by recoding their memory of an experience. Experts also seem to use specialized knowledge to solve problems faster.

This result is very interesting since people instinctively tend to solve problems this way. A memoized Towers of Hanoi (and Fibonacci) becomes linear in both time and space because abstract knowledge is collected that allows new primitive higher level operations to emerge. This should be the basis for any generalized adaptive caching system. No global information is required since this knowledge is collected locally inside a function based on the time of services for that level. Due to caching cost overhead, the optimum caching sparseness may be a subset of all possible values. Interpreting, labeling and formalizing this kind of knowledge is what human experts do all the time to build higher levels of abstraction and represents a transition to metacomputation.
When values are placed into these caches then time is being compressed into space, and when values are removed from the cache then space is being reconverted into time savings. Thus, memoization caches represent capacitors for space/time conversion which may have an impact in the overall spacetime volume of the computation. The many operating points for running the cache can be explored topologically (hopefully by using automatic training) to find the local minimum based on the space and time constraints for a computation.

These examples seem simple, but they are representative of the kind of experience we gained in building a large Computer Aided Design (CAD) system called Droid [16]. Droid was implemented in Lisp, and thus we were able to explore and build many high level abstractions not easily built in more traditional programming languages. We used functional based memoization to build a high speed simulator and object oriented database. The CAD tools limited the use of side effects to the caching of computed functional values (enforced due to consistency requirements described below). We built a caching control primitive called KEEP [17] (based on EGV - Extended Generalized Variables) that decouples the computing of the value from the side effect and consistency controls. A generalized variable or abstract location can be defined as an EGV, thus formalizing the binding, unbinding, getting the value, or asking if bound. Lazily computed EGVs unify spacetime by turning ordinary variables into active locations. The next section describes how we used functional programming and memoization to achieve efficient programming paradigms with good locality.

5 Efficient Computation Styles

Functional programming with memoization naturally leads to demand driven programs or lazy computation. This demand driven programming style was used in our object oriented database to produce multiple cached projections of primitive user data (ie hierarchical vs flat, vectorized vs bit view, filtered vs raw, etc). In general, each tool wants to see it's own custom perspective of the user data. We realized that by generating and caching multiple representations, a consistency problem could occur if any of the original data changed. This kind of change can occur from user inputs or from optimizer tool programs. We built incremental consistency management techniques that hook directly into KEEP, because any time a value is cached it must also be programmed for consistency. It is essential to formalize consistency inorder to be efficient and still correct. We found out that most change management could not be lazy, but must actively propagate like ripples produced by a stone thrown into a pond. These change ripples could best be described as alternate modes of execution across the database.

The program for propagation of changes can be automated [18], but a general solution seems to ignore abstraction boundaries and thus is too fine grained and time expensive. Therefore, inorder to achieve efficiency, the change propagation code was written manually at several layers of abstraction or several grain sizes. The efficient manually written change code deals with abstract change types (add, insert, delete, remove, reorder, etc) and differential values (primitive change quantum) at abstraction boundaries. Tiny changes may cause an avalanche effect invalidating everything in a large database, and resembles what is predicted by chaos theory. In our attempts to formalize automatic change management techniques, we realized that any piece of data could have an arbitrarily large number of dependencies. This can be easily seen by the observation that any piece of data could be included in an arbitrary number of sets. The priority of each kind of dependency link is also different and thus effects how the change can be propagated.

Demand driven code is efficient because it computes the minimum amount to produce the answer. Incremental computation should ideally have costs proportional to the amount of change and not the size of the database. Both of these styles have many layers of abstractions and atomic primitives (similar to nested transactions). These styles mimic the locality, symmetry, invariance of scale, and consistency demonstrated by physics. These styles go hand in hand when large demand driven, multiple representation, data graphs are built. For the kinds of problems we worked on, these graphs were more than 10 dimensional, where each tool lazily generates and caches it's own view of the primitive data, thus increasing the dimensionality of the graph on demand. Coding theory, Neural Networks, and graph theory [19; 20] also depend on the mathematics of high dimensional spaces for useful computation mechanisms.

Tools using the database required mechanisms for controlling relative motion over these high dimensional graphs. We supported general iteration mechanisms for this unified database (JOURNEYS) that allowed movement around this high dimensional graph using iterative, relative directions (up, down, forward, backward, closer, farther, east, west, north, south, etc) that could be filtered (context sensitive PERSPECTIVES) based on where they started and what they were looking for. These relative directions were overloaded functions, called methods, that returned a graph object that represents the next logical step in that abstract direction. Thus, directional functions define the topology of the graph, and not the data structures themselves, since many of the abstract slot accessor functions can lazily add to the graph. Each abstract location can be implemented as an active data element. The graph grows on demand and can add new datatypes which can enable new methods (or build new methods on fly), which can cause more graph growth, etc, again reinforcing the notion that computation can be thought of as active data. Computation organized by defining, building, and moving over high dimensional topologies has strong parallels to physics.

Much of physics is based on geometrical topology considerations and computation should also be thought of as a real topology for the following reasons. Pointers in virtual memory allow programs to build high dimensional graphs mapped onto a one dimensional space (virtual memory is a large one dimensional space). This is particularly important when one realizes that incremental modes in a high dimensional graph, operate with different clusters of data than those originally built on the same physical page. This non-optimal page clustering occurs anytime a high dimensional graph is mapped onto hardware with less than that many dimensions. This non-symmetrical paging behavior due to modes of accessing high dimensional graphs can be easily shown by initializing a 2 dimensional array (1000 x 1000) and iterating over the **wrong** index first.

Relative spatial semantics is becoming more important because of issues in paging systems, garbage collectors, and safe programming techniques. Paging optimizers for object oriented data bases and garbage collector technology [21] both require a completely local view of data objects inorder to allow reclustering. Several popular programming languages allow you to directly specify the absolute locations for data structures to be next to each other in memory, and then the programmer is allowed to cheat by building his own pointers that addresses across these data structure boundaries. This unsafe programming technique assumes global address, rather than local addresses from the beginning of the structure. This practice can produce pointers with invalid addresses, but can also prohibit the use of paging optimizers and garbage collectors that reorganize the absolute location of blocks of data in memory. These techniques that depend on locality are migrating into next generation object oriented operating systems.

Huffman encoding [22] (and other kinds of statistical based information like entropy) requires global knowledge of a finite set of data. This is useful, but at odds with locality arguments in physics, and also efficiency of incremental modes. This concerns me because of the impact we have seen in the behavior of our incremental programs and in the complexity of programming various modes of change propagation. We have seen that some global answers can be easily recomputed incrementally (average of a set of numbers) but other problems (traveling salesman) have no incremental modes possible. Does physics say anything about global optimizations or incremental modes?

From our work, I believe that a formalized theory of computation should include the follow software concepts which have parallels in physics. 1) Formalization of abstract locations, relative directions, high dimensional topologies, 2) Formalized notion of hierarchical time, 3) Unification of spacetime using active data concept as the primitive building block, 4) Automatic consistency management at appropriate grain size, 5) Context fields and context sensitive actions, and 6) Theory for automatic adaption of computational efficiency which includes space/time optimization and parallelism. Others areas are also important, but these are the ones that can be supported by our work. This will become more important during the next 5-10 years when general parallel hardware is designed and built to support large object oriented database applications. Many of these enable higher productivity and reliability by automating routine software tasks.

5 Conclusions

These ideas were developed as a result of looking towards physics as a model for how to organize large software applications. Physics has added more useful insight into the software design process than studying any programmers guide, Turing machine, or book on information theory. Locality, symmetry, and invariance of scale need to be followed in our programming practices as well as our architectures. Functional programming techniques using Lisp, allowed us to step above the problem without being limited by the language or people on the program.

Computation is really the unification of SpaceTime and information. **Space and time** seem to be the ultimate computational resources. These resources should be labeled **direction** and **change** inorder to encourage a more relativistic or local view of computation. The Von Neumann bottleneck is really a Newtonian bottleneck, caused by the segregation of space and time in computer architectures and programs. Space and time must be unified at all levels of computer science and computer engineering. A practical computation theory must include the thinking about abstraction, information, space, and time such that it can be useful in practice. The cultural, conceptual, and language barriers between physics, hardware, software, and mathematics must be broken down if we are to really develop a useful theory of computation. This can only happen when abstract concepts can be shown to have physical meaning and consequences in our computers. Hopefully this will be useful for explaining psychology and exploring how humans think, thus leading to a theory of real Intelligence. The rest of this section is more speculative in order to encourage further discussion.

6 Other Issues

Logical and spatial entropy from Carver Mead can be reinterpreted within this high dimensional topology framework. The cost of information being in the wrong place can in principle always be made arbitrarily small by adding a direct pointer between two abstract locations, which represents adding another dimension to a high dimensional graph. Thus by adding the pointer, any two locations can thought of as being moved closer to each other in hamming distance, which is equivalent to warping the space between them. Knowing how to efficiently map this high dimensional graph to a one dimensional virtual memory or a 3 dimensional space of our physical world seems to be the missing technology.

Information being in the wrong form does not represent a rotation of the data, but rather a change in the position of the observer in hyperspace. In physics, all perspectives are computed in parallel, and only the observer position must change. If this were not true, then some perspectives would be biased or have special status, which we know is prohibited by special relativity. This geometric topology interpretation is interesting because both spatial and logical entropy can be unified to represent a warping or motion in hyperspace. This interpretation could suggest a computational mechanics that resembles the geodesics for photon mechanics in physics. Since the CPU represent a stationary observer perspective, all data must be rotated with respect the CPU. This is analogous to touring Europe by sitting on a New York dock and watching all of Europe's sights parade by on ships in the harbor. Standardizing active data services to support mobile computational perspectives (like friendly viruses) seems like a better model to support an efficient distributed/parallel observer.

Many other areas need to be explored that suggest parallels between physics and computational abstraction, for example. Does active data suggest some primitive spacetime "compuquantum"? Does a continuation represent a point on a light cone or maybe a formalized observer? Can an observer be distributed as would be needed for parallel computation models? Does the re-execution of a continuation multiple times add support to the many worlds model of quantum physics? Does a continuation/observer have an effective mass as it moves through the computation state space? Do memoization results suggest that space and time are interchangeable like matter and energy? Is spacetime conserved in efficiency tradeoffs? If complexity increases by growing higher dimensional graphs, is this reversible? What does incremental computation mean in physics? Does entropy say anything useful about incremental computation of a graph or about inconsistency or other noise/errors in large active data graphs? Do Eigen values suggest a model for reliability of graphs that have only a few valid configurations? Does the topologies formed by the efficiency tradeoffs, suggest that relative spacetime is a field like gravity that can be distorted due to some information influence (ie memoization pattern matcher)? Could the computational efficiency of spatial locality in high dimensional graphs be mapped to the high order dimensions predicted by GUT or EPR? Is computation time either relative or hierarchical? These and many other questions will need to be addressed inorder to understand the physics of computational abstraction.

Acknowledgements

I am grateful to Rob Farrow, Derek Smith, and Riley Jackson who helped maintain the inspiration, develop the ideas, and review the text of this paper.

References

[1] Niklaus Wirth, 1976: Algorithms + Data Structures = Programs, Prentice-Hall, Englewood Cliffs, NJ.

[2] Carver Mead and Lynn Conway, 1980: Introduction to VLSI Systems, Addison-Wesley Publishing Company, Menlo Park, CA. pages 333-371.

[3] W. Daniel Hillis, 1982: "New Computer Architectures and Their Relationship to Physics or Why Computer Science Is No Good.", International Journal of Theoretical Physics, Vol. 21, Nos 3/4, Pages 255-262.

[4] Edward Fredkin and Tom Toffoli, 1982: "Conservative Logic.", International Journal of Theoretical Physics, Vol. 21, Nos 3/4, Pages 223-226.

[5] Edward Fredkin, 1990: "Digital Mechanics", Physica D, pages 254-270.

[6] Robert Wright, 1988: Three Scientists and Their Gods, Times Books, New York.

[7] Rolf Landauer, May 1991: "Information is Physical", Physics Today, page 29.

[8] Harold Abelson and Gerald Sussman, 1985: Structure and Interpretation of Computer Programs, MIT Press, Cambridge Massachusetts, Pages 1-70.

[9] Dan Friedman and Mattias Felleisen, 1986: The Little LISPer, Science Research Associates, Chicago.

[10] T. Yoshino, D. Smith, D. Matzke, 1987: A Register Transfer Language Based on Functional Programming", Texas Instruments Technical Journal, Vol 4, No. 3, pages 139-145.

[11] J. T. Frazier, 1987: Time, the Familiar Stranger, Microsoft Press, Redmond, Washington.

[12] Harold Abelson and Gerald Sussman, 1985: Structure and Interpretation of Computer Programs, MIT Press, Cambridge Massachusetts, Page 39.

[13] Harold Abelson and Gerald Sussman, 1985: Structure and Interpretation of Computer Programs, Page 218.

[14] Steven D. Johnson, 1984 : "Synthesis of Digital Designs from Recursive Equations", The ACM Distinguished Dissertation Series, MIT Press.

[15] Robert Ornstein, 1969: On the Experience of Time, Pelican Books.

[16] Paul Kollaritsch, et al. June 1989: "A Unified Design Representation Can Work." Proceedings of Design Automation Conference.

[17] Rob Farrow and Doug Matzke, 1991: "KEEP: Managing Functional Results" internal report for DROID project, Texas Instruments.

[18] Doug Matzke, 1989: Personal communication about DREAM system at ICAD.

[19] Robert W. Lucky 1989: Silicon Dreams: Information, Man, and Machine, St. Martin's Press, New York, Pages 66-89.

[20] Pentti Kanerva 1988: Sparse Distributed Memory, MIT Press, Cambridge MA.

[21] Douglas Johnson April 1991: "The Case for a Read Barrier", Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS IV), pages 96-107.

[22] Harold Abelson and Gerald Sussman, 1985: Structure and Interpretation of Computer Programs, Pg 120-121.