Growth markets in computing have historically followed the introduction of "missing link" products that produced quantum leaps in application programming productivity

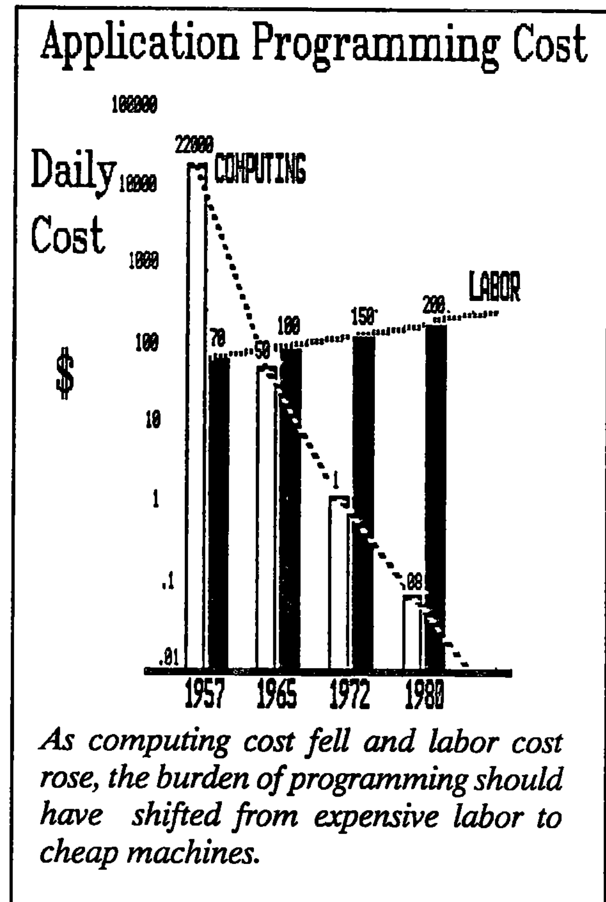
Calculus Machine Architecture the missing link to scientific productivity

Synthetic Calculus ... A Hierarchic Mathematics Plus Writable Instruction-Set Computers (WISC)

It used to be that there were two major cost factors that determined whether it was practical to use a computer to solve a particular problem in science and engineering, programming labor and computer time. In 1965 a FORTRAN programmer costing $100/day to a project could hardly spend $50/day in computer time while developing an application, because most of his time was spent finding bugs. In 1957 when FORTRAN was introduced, this same amount of computing would have cost about $22,000, so it didn't seem so bad that the burden of application programming was on the man rather than the machine. But by 1980, this same amount of computing cost about 8 cents and now in 1988 it is no longer significant at all.

Today, after 30 years, FORTRAN-level languages are still the predominant means of scientific programming. This suggests that there is a fundamental deficiency or "missing link" that is holding back progress in scientific computing. In the past such missing links severely restricted demand for computers until they became available, then they sparked expansive market growth as they opened up computer usage to new users and new applications for which there was great need.

FORTRAN itself was the first of these missing links. When FORTRAN was introduced in 1957 it quickly expanded the population of programmers. In those days, FORTRAN users were the amatuers of the computing world. They primarily used computers to solve problems that they themselves originated. In the middle sixties a second missing link, time-shared remote computing, spurred new growth with more convenience to the same class of users, i.e., problem-originators who did their own programming primarily in BASIC. More recently, a third growth market of the same kind occurred with the advent of personal computers, introducing a new kind of "do-it-yourself" programming, in spreadsheets.



As computing cost fell and labor cost rose, the burden of programming should have shifted from expensive labor to cheap machines.

What is the missing link today? Today's computerized applications are limited to highly recurrent engineering that can justify high development cost and long delays. The labor vs. computer cost trends suggest that something fundamental must change if the cost of applying computers to *new* problems is to be overcome. Certainly, force-multiplying leverage is needed to tackle the scope of new applications epitomized by SDI requirements. But SDI is merely the most ambitious of the projects of the 1980's that are being attempted with 1960's programming technology.

Hierarchic Mathematics

Today the missing link is nothing less than a new kind of mathematics, and a new kind of computer architecture designed to support it. One of the major innovations of computing that is now well established is the art of software engineering. This art is in essence not new, but is rather the enforcement of hierarchic techniques in the development of software; hierarchies having been well established in hardware engineering and in nature as the primary means of dealing with complexity. The problem in scientific computing, however, is that it is predominantly mathematical, and mathematics has not been generally thought to be hierarchical.

In the 1970's, however, a new mathematical form appeared that was definitely hierarchical, and was extremely powerful in solving very sophisticated mathematical problems. Moreover it matched the mathematical skills of engineers and physicists who were the problem originators. This new mathematics appeared in the form of a new type of programming language. The language was called PROSE (for PROblem Solution Engineering), but it was really a *software virtual machine* implemented as a *calculus architecture* using large mainframes as if they were microengines. Although this language is no-longer available due to the lack of supporting hardware architecture, the mathematics that it introduced is the basis for a new kind of computers. This mathematics is now called *synthetic calculus* -- "synthetic" because it is an opposite form of the more familiar "analytic" calculus. It is primarily numerical rather than symbolic, yet it is founded on exact formulas. These formulas are used to perform *differential arithmetic* which is the foundation of synthetic calculus.

Calculus Architecture

The architecture of synthetic calculus is composed of three kinds of numerical solution processes which serve as the building blocks of program hierarchies. Each solution process is a problem statement posed as a bilevel hierarchy, connecting two procedures or contexts (See Figure 1). In the *outer context*, the problem is invoked by an operation statement that defines the variables to be solved for and the variables that are used as criteria for solution. This statement identifies which procedure is called to invoke the *inner context* and it calls the solution method (JUPITER) that controls the solution process.

The three kinds of solution processes are called *simulation*, *correlation*, and *optimization*, each having a specific form of operation statement (Figure 2), relating to the kind of equations in the inner context and the way they are solved.

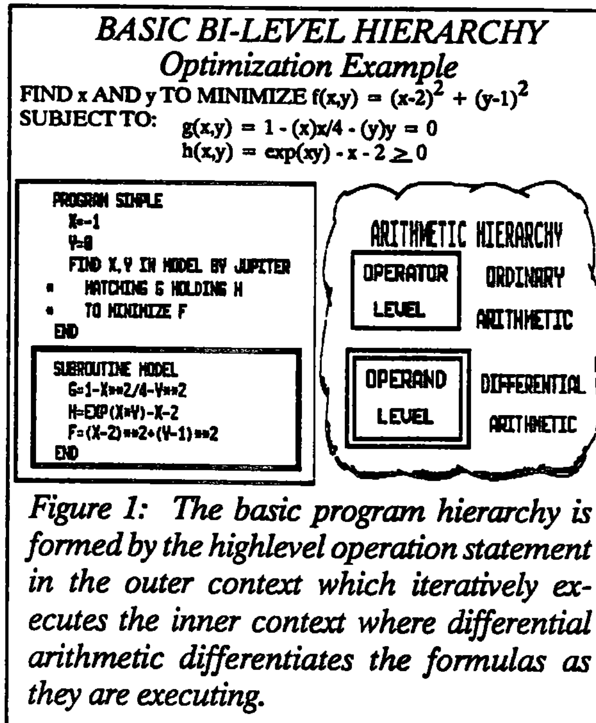


Figure 1: The basic program hierarchy is formed by the highlevel operation statement in the outer context which iteratively executes the inner context where differential arithmetic differentiates the formulas as they are executing.

Simulation usually involves the solution of differential equations, but the solution of explicit algebraic equations via a procedure call is also regarded as a degenerate form of simulation. The essential characteristic is that the solution variables are *dependent variables* of the equations.

Correlation involves the solution of implicit equations, usually simultaneous ones, by matching equality constraints. It is a search process that solves

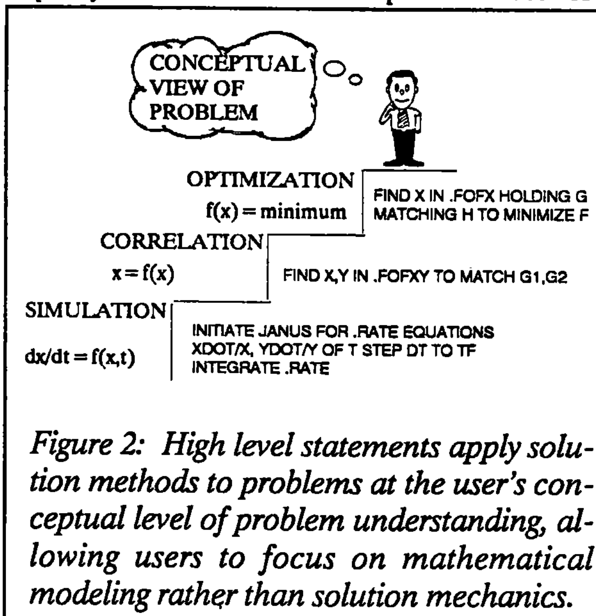


Figure 2: High level statements apply solution methods to problems at the user's conceptual level of problem understanding, allowing users to focus on mathematical modeling rather than solution mechanics.

for the *independent variables* of the equations. The typical search method applied is a Newton method which varies according to the number of equations and the number of unknowns. Correlation is hierarchic with regard to simulation, because it always involves at least the degenerate form of simulation.

Optimization is the process commonly referred to as "mathematical programming". It is a search process that solves for the independent variables of the equations by maximizing or minimizing some function, usually subject to equality and/or inequality constraints. It is also hierarchic with regard to simulation, and correlation is its degenerate form (when there are no degrees of freedom).

Differential Arithmetic

Referring again to Figure 1, it shows a simple problem posed as a bilevel optimization hierarchy. The language is FORTRAN CALCULUS, a calculus-based extension of FORTRAÑ 77, being developed by IMS. Also illustrated is the arithmetic hierarchy imposed automatically by this operation. The outer context is ordinary arithmetic and the inner context is differential arithmetic that produces exact values (to computer precision) of partial derivatives of every dependent variable of the inner context with respect to the independent variables specified in the FIND statement.

This differentiation process, further illustrated in Figure 3, is referred to in the mathematical literature as automatic derivative evalutation (ADE). It is the reason for a new kind of computer architecture. No formula manipulation is present. The formulas for differential computation, including the chain-rule, are extensions to the arithmetic of the machine. Note, also that this is inherently vector arithmetic that can automatically exploit special array processing hardware, because *partial* derivatives are being evaluated. Thus it may be viewed as a vectorized generalization of basic arithmetic that computes function shape information (e.g. the gradient) at the point where the function is evaluated. A further generalization of machine arithmetic is interval arithmetic and interval differential arithmetic. In these cases the value of each variable is the two points that bound an interval. In the figure only first-order differentiation is illustrated, but using recursive methods, any order of differentiation is feasible.

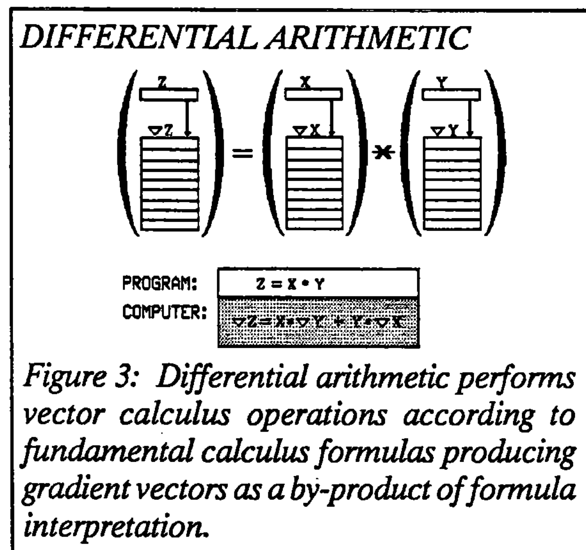


Figure 3: Differential arithmetic performs vector calculus operations according to fundamental calculus formulas producing gradient vectors as a by-product of formula interpretation.

Notice also that the differentiation process and the derivative values themselves are hidden from the programmer. Programming has become *calculus based*, but has not materially changed. The major purpose of the generalized arithmetic is to *internally* support Newton-based or Interval-Newton-based equation solving and nonlinear programming algorithms, and Jacobian-based or Interval-Jacobian-based numerical integration algorithms. The programmer merely applies these generalized methods to mathematical models using the problem-solving statements of synthetic calculus.

Structured Calculus

The differentiation machinery of synthetic calculus is invoked by the first phrase of the FIND statement:

FIND *parameter vector* IN *model subroutine*.

It is a form of ADE called *synthetic differentiation*. The primary difference of synthetic differentiation from other ADE methods is that it is a fully interpretive arithmetic, in which the *parameter vector*, serving as the contextual basis for differentiation, can be changed dynamically during execution. This contrasts with most of the other methods of ADE, in which differentiation contexts are compiler-bound (the parameter vector and differentiation linkages are specified at compile time rather than at run time).

Implicit Problem Nesting

The importance of dynamic binding of differentiation contexts is that it enables the nesting of implicit problem hierarchies. Synthetic calculus architecture contains machinery that enables higher mathematical computer programs to be constructed as nested structures of bilevel problems as illustrated in Figure 4. For example, the *model subroutine* of the previous FIND operation could contain another FIND statement referencing another parameter vector and another (nested) model subroutine. Since each parameter vector represents a coordinate system, a differential coordinate transformation is performed

following the execution of the inner FIND statement to evaluate differentials of the outer FIND corresponding to implicitly computed variables of the inner FIND. This transformation mechanism makes synthetic calculus hierarchic.

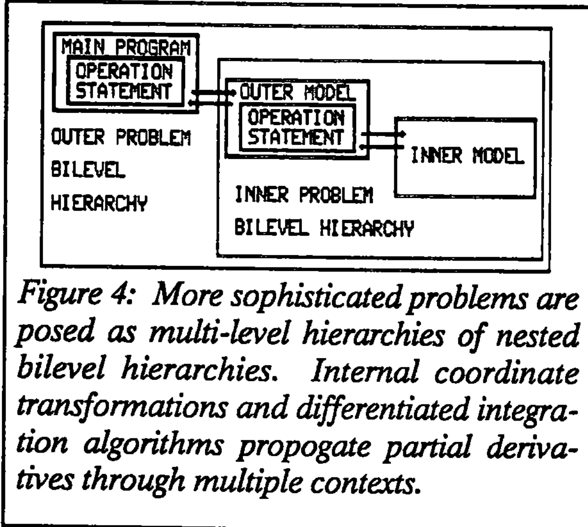


Figure 4: More sophisticated problems are posed as multi-level hierarchies of nested bilevel hierarchies. Internal coordinate transformations and differentiated integration algorithms propogate partial derivatives through multiple contexts.

Integration within Differentiation

In cases where a problem contains differential equations, two additional operation statements are used for *simulation*:

(1)INITIATE *method subroutine*
FOR *model subroutine*
EQUATIONS *derivative vector/state vector*
OF *independent variable*
STEP *size variable*
TO *integration limit*

(2) INTEGRATE model subroutine

which specify and solve an initial value problem.

In structured problems where the inner problem is an initial value problem, synthetic differentiation is used to differentiate the integration process as it is executing. This process will differentiate functions that have no analytic form and therefore could not be differentiated by symbol manipulation methods (e.g. derivatives of boundary conditions with respect to initial conditions, Figure 5).

Figure 6 is a FORTRAN CALCULUS example of a four level hierarchy involving implicit problem nesting and integration within differentiation. It is an optimal design and control problem from the textbook *Quasilinearization and Nonlinear Boundary Value Problems* by R.E. Bellman and R.E. Kalaba, (Elsevier 1965). Figure 7 is a comparison of this same program to the one in FORTRAN from the textbook.

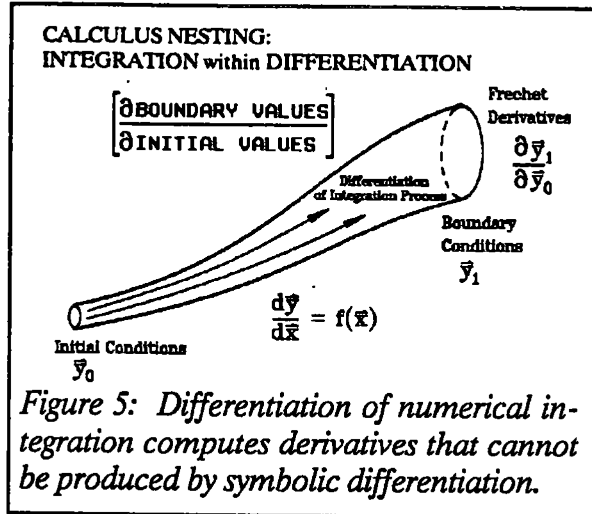


Figure 5: Differentiation of numerical integration computes derivatives that cannot be produced by symbolic differentiation.

Differentiation within Integration

Figure 8 illustrates two kinds of integration processes in which synthetic differentiation performs subprocesses for integration stepping. In the solution of implicit differential equations, Newton methods are used to solve for the implicit derivative variables (rates) of integration by finding the zeros of derivative constraints, and then these rates are integrated by ordinary numerical integration methods. Figure 9 is a hierarchic PROSE example of this kind solving a pair of implicit differential equations.

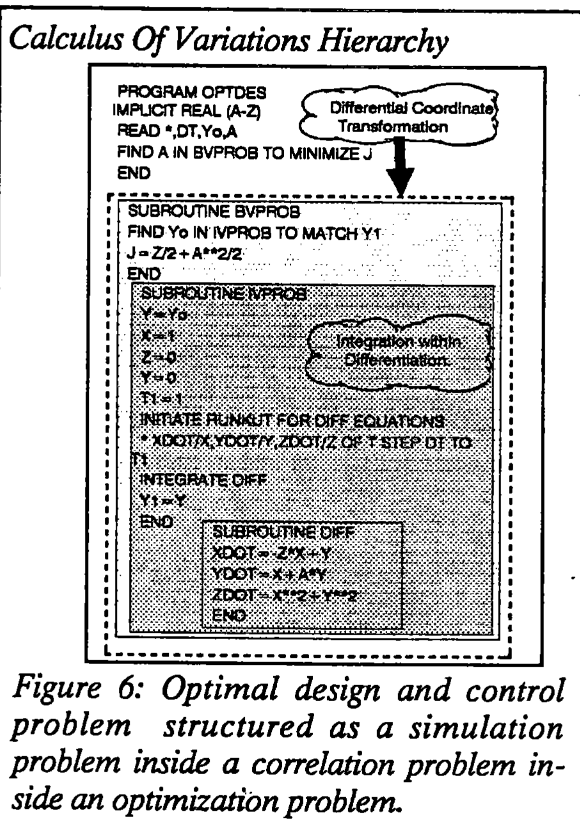


Figure 6: Optimal design and control problem structured as a simulation problem inside a correlation problem inside an optimization problem.

HIERARCHIC *VS.* *NON-HIERARCHIC*

```
      PROGRAM OPTDES
      IMPLICIT REAL (A-Z)
      READ *,DT,Y0,A
      FIND A IN BVPROB TO MINIMIZE J
      END

      SUBROUTINE BVPROB
      FIND Y0 IN IVPROB TO MATCH Y1
      J=Z/2+A**2/2
      END

      SUBROUTINE IVPROB
      Y=Y0
      X=1
      Z=0
      Y=0
      T1=1
      INITIATE RUNKUT FOR DIFF EQUATIONS
*     XDOT/X,YDOT/Y,ZDOT/Z OF T STEP DT TO T1
      INTEGRATE DIFF
      Y1=Y
      END

      SUBROUTINE DIFF
      XDOT=-Z*X+Y
      YDOT=X+A*Y
      ZDOT=X**2+Y**2
      END
```

Figure 7: The leverage of hierarchic math vs non-hierarchic math is illustrated by the two programs, one in FORTRAN CALCULUS (above) and the other in ordinary FORTRAN (right). One is simple, non-algorithmic and therefore comprehensible. The other is complex, algorithmic and incomprehensible. One requires little mathematical expertise, even though it's subject is mathematically sophisticated. The other requires considerable mathematical expertise.

```
      PROGRAM OPTDES(INPUT,OUTPUT)
C     DESIGN PROBLEM
      COMMON N1,KMAX,NP,NMAX,HGRID,CX,T,X,
     1 Y,U,V,H,PREV,H,P,A,B,C,NEQ
      DIMENSION T(250),W(4,500),H(4,4,500),
     1 P(4,500),A(2,2),B(2),C(4),PREV(4)
1     CALL INPUT
2     CALL START
3     DO 19,K=1,KMAX
      DO 5 I=1,250
5     T(I)=0.
      T(2)=0.
      T(3)=HGRID
      T(4)=1.
      T(9)=1.
      T(14)=1.
      N=0
      PRINT 111, (T(L),L=4,431)
      NEC=20
      CALL INT(T,NEQ,N1,0.,0.,0.,0.,0.,0.)
      N=0
8     DO 11 M=1,NP
      N=N+1
      X=W(1,N)
      Y=W(2,N)
      U=W(3,N)
      V=W(4,N)
      CALL INTM
      L=3
      DO 9 I=1,4
      DO 9 J=1,4
      L=L+1
9     H(I,J,N)=T(L)
      DO 10 J=1,4
      L=L+1
10    P(J,N)=T(L)
11    CONTINUE
6     PRINT 111,N,(T,L),L=4,43)
      PRINT 111,N,((H(I,J,N),J=1,4),I=1,4),
     1 (P(J,N),J=1,4)
111   FORMAT(1H019XI10,4E20.6/(30X4E20.6))
      IF(N-NMAX)8,7,7
7     PRINT 40,K
40    FORMAT(1H0/65X9HITERATION,I3)
      N=NMAX
      A(1,1)=H(2,2,N)
      A(2,1)=H(2,3,N)
      A(1,2)=H(3,2,N) + H(4,2,N)
      A(2,2)=H(3,3,N) + H(4,3,N)
      B(1)=-P(2,N) - CX*H(1,2,N)
      B(2)=-P(3,N) - CX*H(1,3,N)
      DET=A(1,1)*A(2,2) - A(2,1)*A(1,2)
      G=ABSF(DET)
      IF(G-.000001)200,200,12
200   PRINT 201,DET,((A(I,J),J=1,2),B(I),I=1,2)
201   FORMAT(1H020X112HDETERMINANT=
     1 ,E16.6/(40X3E20.6))
      CALL EXIT
12    C(1)=CX
      C(2)=(B(1)*A(2,2) - B(2)*A(1,2))/DET
      C(3)=(B(2)*A(1,1) - B(1)*A(2,1))/DET
      C(4)=C(3)
      GRID=0.
      PRINT 45,GRID,(C(I),I=1,4)
45    FORMAT(1H026X4HGRID,14X1HX,19X1HY,
     1 18X2HMU,19X1HA/20XF10.4,4E20.6)
      N=0
13    DO 14 M=1,NP
      N=N+1
      DO 14 I=1,4
      W(I,N)=P(I,N)
      DO 14 J=1,4
14    W(I,N)=W(I,N)+C(J)*H(J,I,N)
      FN=N
      GRID=FN*HGRID
      PRINT 50, GRID,(W(I,N),I=1,4)
      IF(N-NMAX) 13,15,15
15    DO 16 I=1,4
      G=ABSF(C(I)-PREV(I))
      IF(G-.000001)16,16,17
16    CONTINUE
      GO TO 1
17    DO 18 I=1,4
18    PREV(I)=C(I)
19    CONTINUE
      GO TO 1
50    FORMAT(20F10.4,4E20.6)
      END

      SUBROUTINE INPUT
      COMMON N1,KMAX,NP,NMAX,HGRID,
     1 CX,T,X,Y,U,V,H,PREV,H,P,A,B,C
      DIMENSION T(250),W(4,500),H(4,4,500),P(4,500),
     1 A(2,2),B(2),C(2),PREV(4)
      READ 110,N1,KMAX,NP,NMAX,HGRID,CX
      PRINT 10,N1,KMAX,NP,NMAX,HGRID,CX
110   FORMAT(4I4,2E8.2)
10    FORMAT(1H120X14HDESIGN PROBLEM//
     1 38X2HN1,6X4HKMAX,8X2HNP,6X4HNMAX,
     2 13X5HHGRID,16X4HX(10)/30X4I10,2E20.6)
      RETURN
      END

      SUBROUTINE START
      COMMON N1,KMAX,NP,NMAX,HGRID,
     1 CX,T,X,Y,U,V,H,PREV,H,P,A,B,C
      DIMENSION T(250),W(4,500),H(4,4,500),
     1 P(4,500),A(2,2),B(2),C(2),PREV(4)
      DO 1 N=1,NMAX
      W(1,N)=CX
      W(2,N)=0
      W(3,N)=0
1     W(4,N)=0
      DO 2 I=1,4
2     PREV(I)=W(I,1)
      K=0
      PRINT 40,K
      GRID=0.
      PRINT 45,GRID,(W(I,1),I=1,4)
40    FORMAT(1H0/65X9HITERATION,13)
45    FORMAT(1H026X4HGRID,14X1HX,19X1HY,
     1 18H2HMU,19X1HA/20XF10.4,4E20.6
     2 /1H019X22HFOR ALL VALUES OF TIME)
```

Mathematical Software Engineering

The nesting of problems in synthetic calculus provides a simple hierarchic means of solving very complex mathematical problems. Normally, problems in these classes would not be tackled without considerable knowledge of higher mathematics, and without considerable skill with numerical methods. In short this has always been the exclusive domain of experts. Synthetic calculus changes all that. It permits the *problem originator* to pose mathematical problems in a simplified manner with little knowledge of numerical methods. There are important reasons for the simplicity:

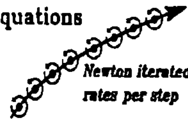
Mathematical Self Organization

Synthetic calculus hierarchies are mathematically self organized because the bilevel structures are complete problem units satisfying mathematical computability requirements. The hierarchies further divide naturally into alternating levels of prediction and control because of what might be called "paradox avoidance". In describing a predictive model, deterministic formulas are used whose output is prescribed once the input is given, and this input is held constant during the computation pass of the predictive formulas. But in order to control the prediction, the input of the predictive formulas must

CALCULUS NESTING: DIFFERENTIATION within INTEGRATION

- Implicit Differential Equations

$\left[\frac{\partial \text{ZEROS}}{\partial \text{RATES}}\right]$



- Integration Step Optimization

Stiff Equations

$\left[\frac{\partial \text{RATES}}{\partial \text{STATES}}\right]$

Spectrally Ill Conditioned Equations

Figure 8: Differentiation within numerical integration is used to solve for implicit rates to be used during integration and to apply Jacobian-based methods to the optimization of integration step size.

Implicit Differential Equations

$$g\dot{x} = \dot{x} + (1 + e^{-t/20} \sin \pi t) 3.2 \sqrt{1 - (\dot{x}+\dot{y})[1.15 + 57.5/(20000+x+y)]} = 0$$

$$g\dot{y} = \dot{y} + (1 + e^{-t/20} \sin t/2) 1.59 \sqrt{1 - (\dot{x}+\dot{y})[1.15 + 36.2/(20000+x+y)]} = 0$$

```
PROBLEM .IDE [IMPLICIT DIFFERENTIAL EQUATIONS]
  X = 1400 Y = 7000 [INITIAL CONDITIONS]
  XDOT = -50 YDOT = -25 [GUESSES FOR IMPLICIT DERIVATIVES]
  T = 0 DT = .5
  INITIATE JANUS FOR .DEQ
    EQUATIONS XDOT/X, YDOT/Y OF T STEP DT
  IMPOSE .SET [SUPPRESS SUMMARY AFTER INITIAL POINT]
  HEADING = ' TIME      XDOT      X      YDOT      Y'
  DUMP GLOBALS, EJECT PAGE, TEXT PRINT HEADING
  TPRINT = 0
  UNTIL T GE 50 DO
    INTEGRATE .DEQ
    WHILE T GE TPRINT DO
      DISPLAY T,XDOT,X,YDOT,Y IN
      '**,***E*** **,***E*** **,***E*** **,***E** ***,***E***
      TPRINT = TPRINT + 1
    REPEAT
  REPEAT
  CONTROLLER .SET FOR AJAX
    SUMMARY = 0
  END
END [.IDE]

    MODEL .DEQ
      FIND XDOT,YDOT IN .IEQ TO MATCH GX,GY
    END [.DEQ]

      MODEL .IEQ [IMPLICIT EQUATIONS]
        GX = XDOT + 3.2*.SQRT(1-(XDOT + YDOT)*(1.15 + 57.5/
          (2000 + X + Y)))*(1 + .1*.EXP(-T/20)*.SIN(1.5708*T))
        GY = YDOT + 1.59*.SQRT(1-(XDOT + YDOR)*(1.15 + 36.2/
          (2000 + X + Y)))*(1 + .1*.EXP(-T/20)*.SIN(1.508*T))
      END [.IEQ]
```

Figure 9: Hierarchic program to solve a pair of implicit differential equations, structured as a correlation problem inside a simulation problem.

be varied as parameters in multiple passes, each one being used as a sample by the control method in seeking a stationary point (root or extremum). This dual requirement of holding the input constant for prediction and varying it for control would cause a paradox in a single level of description, leaving no alternative to the use of two hierarchic levels to describe the prediction/control process.

Non-Algorithmic Modeling

The models used to describe problem-unit functions are expressed as "open loop" predictive procedures usually containing only explicit non-iterative formulas. Novice programmers usually have no difficulty learning programming when formulations are this simple. Difficulty of understanding and obscurity of syntax develops when some sort of solution process is blended with the formulation (unnecessary in synthetic calculus modeling). Then the program loses its identity as a problem statement and starts becoming an algorithm, usually having some sort of strategy that must be studied to be understood. It is at this point that programs lose correspondence with the problems that they describe.

Transparency of Mathematical Machinery

As transparent semantic processes of a programming language, synthetic differentiation, the hierarchic coordinate transformations, and the "built-in" solution methods constitute presupplied apparatus that the user does not have to build or even understand. Taken together these mechanisms uncouple mathematical modeling from numerical mathematics. Models and methods are transparent to each other, and methods are interchangable without reprogramming. As a result, methods can be learned by experimental usage as plug-in tools.

Mathematical Exactness

The differential arithmetic in its non-interval form produces partial derivatives that exact are to the precision of the computer. In its interval form, derivatives can made exact to any desired precision. Because of this exactness, unlike finite differencing, the accuracy of approximations is independent of the nonlinearity of the model. Consequently, the user does not have to be sensitive to the customary difficulties of numerical approximation.

Intrinsic Mathematical Intelligence

Perhaps the most important contrast between synthetic calculus and conventional methods of scientific computing is that it utilizes the intrinsic intelligence available in mathematical statements to a much greater degree. In pure software implementations of synthetic calculus virtual machines, the

penalty is much slower function evaluation, because each arithmetic operation at the statement level expands into a massive vector-arithmetic operation at the hardware level. Moreover, because of the incompatibility of conventional machine architecture, these vector operations are very inefficient. Nevertheless, the use of greater mathematical intelligence can drastically reduce the number of function evaluations that are necessary to solve a problem. An important example of this was demonstated in the late seventies when a synthetic calculus language PROSE was pitted against a simulation language CSSL III on the same computer (Control Data 6600). The problem was a system of about seventy differential equations. Since CSSL III generated a FORTRAN model, it was able to execute functions faster than the interpretive language PROSE by about a factor of 100. Nevertheless PROSE solved the complete problem quicker by factor of 80. The reason was that PROSE permitted the use of an exact derivative based step-optimizing integration technique whereas CSSL III used a variable-step predictor-corrector method because of the limitations of FORTRAN. The derivative-based method used much more accurate function intelligence than the predictor corrector, and this made a surprising difference.

WISC Architecture

The early synthetic calculus virtual machines (PROSE and its predecessor SLANG) were sufficient to demonstrate the mathematical power of synthetic calculus, but were limited by the slow speed of software interpretation using conventional machine architecture. This handicap has now been eliminated by the development of writable instruction set computer ("WISC") architecture. This architecture is designed to implement virtual machines to support the interpretive execution of very-high-order languages at speeds roughly equivalent to compiled languages (e.g. FORTRAN) on conventional architectures.

Migration of Software into the Machine

In as much as possible, the WISC provides the advanced system architect a "clean slate" with which to synthesize a computational architecture to optimize the performance of any language, independent of the level of its operations. In fact the higher the order of its general operations, the greater its performance can be, because these operations can be migrated into microcode or into special hardware modules. Note that this is quite the opposite of compiled languages. In general they become less efficient, the higher the order of their operations. This is one of the reasons that very-high-order languages have been so rare in the last two decades.

In the case of synthetic calculus, the WISC provides an avenue of performance that enables synthetic calculus languages to transcend the performance of conventional languages. As stated in the previous section, there have been examples in which the mathematical intelligence of synthetic calculus has been able to overcome the virtual-machine execution handicap on conventional machines, resulting in a particular benchmark of a speed gain by a factor of 80. But since the handicap itself is about a factor of 100, the removal of the handicap would have resulted in a speed gain of about 8000, without resort to highly parallel computation!

WISC Hardware for Personal Computer Hosts

The IMS MAX-2 WISC shown in Figure 10, is an add-in computer board for PC-AT class machines, using a 20-40 MIPS 32-bit pipelined RISC microprocessor as its CPU. It contains three separate memories each connected to the CPU by an independent 32-bit bus. In the current MAX-2 implementation, the user memory, is sized from 4 to 64 megabytes depending on the use of 256K, 1 megabit, or 4 megabit chips. The smaller instruction and context memories run at twice processor clock speed and

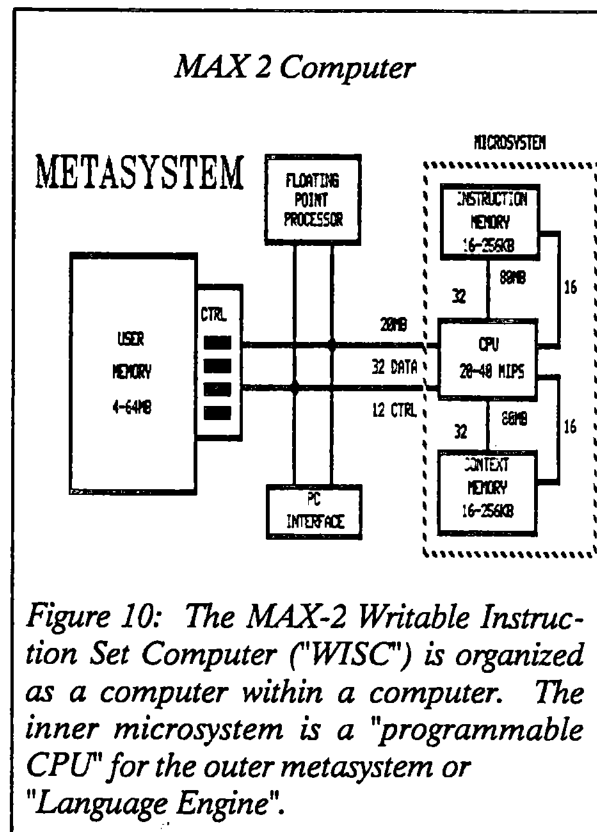


Figure 10: The MAX-2 Writable Instruction Set Computer ("WISC") is organized as a computer within a computer. The inner microsystem is a "programmable CPU" for the outer metasystem or "Language Engine".

are sized at 64K bytes, although addressing is provided for 256K bytes.

From the three memories at the current 20 MHz clock rate, information can be supplied at an aggregate rate of 180 megabytes/sec, 80 megabytes/sec for each of the static RAMS and 20 megabytes/sec for the dynamic RAM, well matched to the capabilities of a 20-40 MIPS pipelined CPU. This native performance enables high-speed interpretation of high-level languages. For example in Smalltalk, using the context memory for execution stack and a method cache, a performance level of 3-4 Dorados or 8-10x an 8 MHz PC/AT 286 is obtained.

Language Engine

The MAX-2 is organized to act as a computer within a computer. The inner computer (microsystem) consists of the microprocessor CPU and the static RAMs. The microsystem acts as the "language engine". The virtual machine microprogram is stored in the instruction memory, and the context memory is used as an execution stack/heap, as appropriate to the language being interpreted. The executable instructions (an arbitrary instruction format) are stored in the user memory. Since these instructions are a specialized encoding of the user langauge, they are more abstract and semantically intensive than assembly level languages, resulting in much smaller programs and permitting higher semantic information traffic into the CPU ("von Neumann bottleneck").

The floating point coprocessor (FPC) is a daughter board for the MAX-2 that provides single and double precision IEEE format floating point arithmetic, conversions and the transcendental functions (e.g., SQRT, SIN, COS). It may be microprogrammed to add other specialized functions. The board includes ALU and MPY units, and 4096 words of 48-bit memory for microcoding. The CPU communicates with the FPC over a 32-bit bus using BUS EMIT and BUS RECEIVE microinstructions, or may designate exchange between the FPC and the user memory using BUS FROM-TO microinstructions.

User Memory System

A four-channel memory controller chip provides key features for the execution of vector operations in high-level languages. There are four channels to memory, each with independent address registers and modes controlling access to memory. A typical use is for one channel to be the high-level instruction stream, two channels dedicated to source vector operands and one channel dedicated to destination vector operands. These channels continue to supply information according to the active modes until they are reset with new mode commands. Addressing modes are provided for bytes, halfwords, or words. Addresses are incremented or decremented following each fetch according to a control mode. There are two parameters of address incrementing: "stride" value and "tiptoe" value. Stride value is the major increment between element fetches. Values of stride may be set to provide for row element access to a column-stored array. Tiptoe counts sub-elements, for example the two words for a double precision real or the four elements of a double complex element.

High Productivity Science & Engineering

One of the side effects of stagnation of the programming art has been the growth in the army of professional programmers relative to the growth of *problem-original* scientists and engineers. Moreover, since technical people like to work on what they understand best, this growth has prolonged the stagnation, because programmers develop languages for programmers -- not for *problem originators*. This is why C is the predominant language of new software development today, it is the nearest thing we have to a portable machine language.

Needed: New Job Goals in Programming

If we are to break out of this productivity bind, we must heed the growth lessons of the past. As stated in the introduction, the growth periods of computing, sparked by the missing links (FORTRAN, timesharing, and personal computing) were all *problem-original revolutions*. Unlike the programmer, whose *job* is programming, the job of the problem originator is something else, not programming *per se*. *His* motivation is to get the programming done so he can get on with what the programming is for.

High Productivity Programming

The track record of PROSE (Figure 11) illustrates how much can be gained in productivity by reducing programming time and by increasing the sophistication of new problems being solved. The comparisons noted in this figure are based upon actual user estimates of the time and/or cost required to program the problem in FORTRAN rather than PROSE.

Figure 12 makes the productivity loss of FORTRAN more graphic. In particular this loss includes the latent productivity so critical to obtaining new business in a competitive market -- the *short-term productive capacity* needed to develop better proposals, better designs, and better evaluations in science and engineering. PROSE and its predecessor SLANG were

SAMPLE OF TRACK RECORD IN INDUSTRY

Rapid Solution to Problems

- Antenna Design Beam Synthesis (Hughes) 3 weeks vs 6 months
- Laser System Design Balancing (Ford Aeroneutronics) 3 days
- Steam Power Electric Network (Bechtel Corp) 1/80 Cost of Benchmark
- Radial Tire Design (Goodyear) 1 day vs 6 months
- Space Telescope Mirror (Rockwell) 2 days - proposal support
- Electric Motor Design Optimization (Sierracin/Magnedyne) 1 day
- Optical System Optimization (TRW) (3wks & $2400 VS 6mos & $50000)
- Airport Noize Optimization (Wyle Labs) 1 day

Design Software Tools Development

- Electron Imaging Optimization (Tektronix)
- Traveling Wave Tube Amplifier Design (Watkins-Johnson, Varian)
- Oil Tanker Unloading Optimization (National Steel & Shipbuilding)
- Forest Management Optimal Control (Western Timber Assoc.)
- Telephone System Provisional Planning (GTE)

Figure 11: Examples of sophisticated applications solved in very short times with the PROSE language. Some comparisons with times required for use of FORTRAN are based upon user estimates.

Where Programming Productivity Really Counts

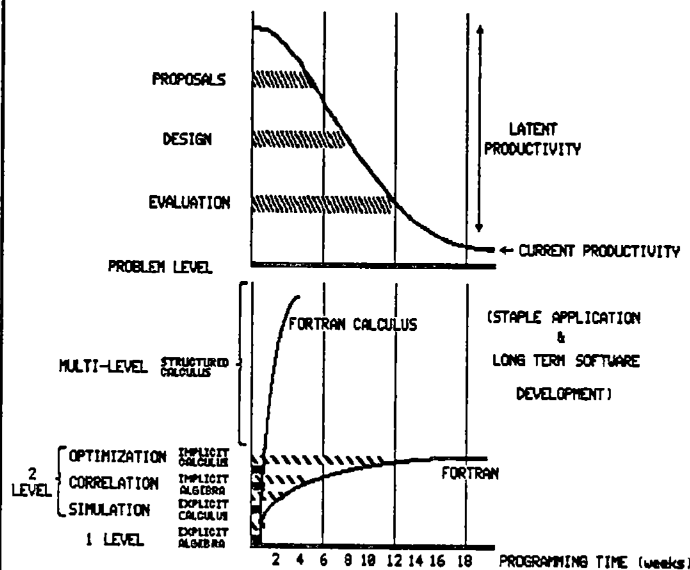


Figure 12: Rapid development of new programs provides short-term productivity so vital for new business acquisition. Current productivity levels do not meet these needs.

used to win competitive proposals by solving examples of RFP problems during the 30-45 day proposal periods and submitting the quantitative results in the proposals.

The examples of Figure 11 include several design advances. In cases like the Antenna Design Beam Synthesis problem, only 3 weeks were required to compute the 20 design parameters necessary to optimally shape the INTELSAT antenna beam to fit the footprint of the 50 states of the U.S.

The importance of synthetic calculus to test evaluation is exemplified by the fact that it was created for the evaluation of flight test data from ballistic missile and spacecraft flights. It was used to mathematically reconstruct the behavior of propulsion systems using state estimation methods. These methods were so accurate in computing payload capability that they were instrumental in decisions to carry greater payloads on manned missions, including the first EVA experiment on Gemini, and the decision to develop the Lunar Rover vehicle for Apollo.

The importance of short-term productivity in the modern environment of highly specialized developments is that multiple design approaches can be developed and quantitatively tested in the time previously employed in a single development. Second order productivity benefits include the ability to solve problems that were previously infeasible; better use of engineering resources; more adaptive flexibility in planning; and better cost control.

The Technology is Ready

The availability of the MAX-2 attached computer with FORTRAN CALCULUS enables FORTRAN-fluent problem originators to spark a synthetic calculus revolution that will break the current programming stalemate. Professional programmers will inevitably articulate this new technology once established, and a new productivity surge will unfold, bringing new calculus-based hardware and even higher-level programming media.

New RISC Chip to Emulate 486 and 68040

International Meta Systems (Torrance, CA) claims it has a new RISC microprocessor that can emulate an Intel 486 or Motorola 68040 at their full native speeds and at a fraction of their cost. IMS is pitching the CPU for pen computers that need high performance for tasks such as handwriting recognition. It also says the chip could be used in a "chameleon computer" that runs PC and Mac software.

The IMS 3250, slated for mid-1993 production, is a two-chip set with a RISC CPU and an I/O controller. IMS says the 3250 will use 0.7- or 0.8-micron CMOS technology with the equivalent of 400,000 transistors. Clocked at 100 MHz, the CPU reportedly runs at 90 MIPS in native RISC mode.

What sets the 3250 apart from other RISC chips is its programmable microcode. Although many CISC processors implement their instruction sets in microcode, most RISC chips do not. Systems designers can reprogram the 3250's microcode using assembler-like tools. IMS says it has written modules that emulate a 486 at 25 MHz and a 68040 at 30 MHz, including FPU support.

To build a computer that runs both PC and Mac applications, a designer would still have to add the appropriate system software. PC clones are easy to make, but a Mac clone would require either licensed Mac ROM chips or their legal equivalent. One possibility is a Mac "compatibility engine" such as the toolbox emulator from Quorum. IMS says the 3250 will cost just $50 to $60 in production quantities.

—Tom R. Halfhill