LA-UR-

98-1440

Approved for public release;
distribution is unlimited.

Title:

Delta-f and Hydrodynamic Methods for
Semiconductor Transport

Author(s):

Lester E. Thode, X-DO
Robert S. Hotchkiss, X-CM
Mark G. Gray, X-TM
Charles M. Snell, X-TM
Daniel C. Barnes, X-PA

RECEIVED

SEP 22 1998

OSTI

Submitted to:

DOE Office of Scientific and Technical
Information (OSTI)

JUT

MASTER

DISTRIBUTION OF THIS DOCUMENT IS UNLIMITED

Los Alamos

NATIONAL LABORATORY

Los Alamos National Laboratory, an affirmative action/equal opportunity employer, is operated by the University of California for the U.S. Department of Energy under contract W-7405-ENG-36. By acceptance of this article, the publisher recognizes that the U.S. Government retains a nonexclusive, royalty-free license to publish or reproduce the published form of this contribution, or to allow others to do so, for U.S. Government purposes. Los Alamos National Laboratory requests that the publisher identify this article as work performed under the auspices of the U.S. Department of Energy. The Los Alamos National Laboratory strongly supports academic freedom and a researcher's right to publish; as an institution, however, the Laboratory does not endorse the viewpoint of a publication or guarantee its technical correctness.

DISCLAIMER

This report was prepared as an account of work sponsored by an agency of the United States Government. Neither the United States Government nor any agency thereof, nor any of their employees, makes any warranty, express or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government or any agency thereof. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States Government or any agency thereof.

DISCLAIMER

Portions of this document may be illegible in electronic image products. Images are produced from the best available original document.

Delta-f and Hydrodynamic Methods for Semiconductor Transport

L. E. Thode*, R. Hotchkiss, M. Gray, C. Snell, and D. Barnes

Abstract

This is the final report of a three-year, Laboratory Directed Research and Development (LDRD) project at Los Alamos National Laboratory. We have developed a prototype plug-and-play (PCUBED) environment based upon a C++ class called a fragment. A fragment is a universal object that can represent any data type. Fragments provide an excellent intuitive approach to the development of an efficient architecture, as well as providing a common data implementation within and between codes. As a result, the PCUBED environment allows for the generation of many different codes within a common framework. At this time, there are seven major codes implemented within the PCUBED environment. Input, output, restart, setup, and graphics are programmed using a high-level approach to insure human efficiency. In contrast, computationally intensive algorithms are programmed using a low-level approach to insure computational efficiency. Fragments provide a straightforward approach to switch between high-level and low-level programming. PCUBED has been tested on a Macintosh PowerPC; on IBM, SUN, HP, and SGI workstations; and on the CRAY YMP and Cray T3D. Using this environment, we have incorporated a drift diffusion, energy balance, hydrodynamic, and Monte Carlo model for metal-oxide semiconductor field-effect transistors (MOSFETs) into a single architecture. With all the models in a common framework, we have investigated the noise characteristics of hybrid and delta-f models. Although hybrid and delta-f models appear viable in one dimension, the noise level of higher order transport coefficients in two and three dimensions makes the utility of such combined methods questionable.

Background and Research Objectives

One of the major problems with the development of new algorithms is the inability to rapidly test and evaluate against existing algorithms in an unbiased fashion. Primarily, this problem arises because existing algorithms are buried within large codes that are not amenable to controlled testing because materials specification, setup, and execution of a specific algorithm is not independent of the rest of the code. Moreover, it is often very difficult to incorporate new algorithms into existing codes because the existing data written in different languages, such as C, C++, f90, or f77. As a result, it is difficult to determine if differences are due to the algorithm, the implementation, or the language.

Principal Investigator, email: les@lanl.gov

Many of the algorithm prototype problems could be overcome with the development of a plug-and-play environment. Such an environment could provide a common method for input, output, restart, and graphics. However, such an environment typically requires a significant rewrite of existing algorithms because of a mismatch of data structures. What is needed is a dynamic method for modifying very complex data structures.

The object-oriented C++ fragment class [1-2] has recently been developed and allows one to integrate new and existing modules in a straightforward manner. In this context, a module can be a multigrid solver; a mesh generator or a visualization tool; or an entire FORTRAN program. The fragment class provides the "glue" to bind the modules together. In this context, "glue" refers to 1) dynamic assemblage of data structures, 2) memory management, and 3) standard ASCII/binary input and output. As a result, we proposed to use the fragment class to develop an environment in which C, C++, f90, and f77 modules could be bundled into a common structure without a significant change in the existing modules. Because of its large number of different models and codes, as well as its importance to civilian industry, we proposed the area of microelectronics modeling to apply this new approach.

Truly predictive device design capability is fundamental to the future success and diversity of the United States microelectronics industry. A recurring theme in our discussions with a large number of semiconductor companies is the desire for more reliance on simulation to reduce the cost and time required to bring a new product to market. The driving forces behind the increased emphasis on simulation are decreased device size, increased device complexity, and increased fabrication cost.

Traditional semiconductor component design utilizes the drift-diffusion equations coupled with Poisson's equation. These equations are based on the fundamental assumption that the electron or hole energy density remains less than the thermal energy density.

As "shrink technology" advances, the characteristic size of the device is reduced while the applied voltage remains on the order of the bandgap. Hence, large electric fields develop in critical regions of an advanced device, where charge carriers can gain energy far in excess of the thermal energy. This regime is called the "hot carrier" regime. In these regions the fundamental assumption of the drift-diffusion equations is no longer valid. For this regime, fundamental understanding of advanced device performance requires a return to a more first-principles transport approach.

The most sophisticated simulation technique for carrier transport in semiconductors is the Monte Carlo method. This method simulates the motion of carriers in k-space using random numbers and is equivalent to solving the Boltzmann transport equation. Although

the method is used for some advanced device design studies, it is computationally costly, which is a major practical limitation to the method. Nevertheless, the Monte Carlo method is used extensively for steady-state calculations of transport coefficients required for reduced models that attempt to address the hot carrier transport problem via electric field dependent transport coefficients.

From an industrial standpoint, the hydrodynamic approximation is the most sophisticated model considered for most applications. This method solves the first three moments of the Boltzmann transport equation. With respect to carrier transport, a major difficulty in the hydrodynamic approximation is the closure assumption.

The hydrodynamic equations can be simplified by assuming that the current density varies slowly compared to the momentum relaxation time. This energy-balance model is an extension over the drift-diffusion model because the mobility and diffusion coefficients can be linked to the local electric field. Although this model provides an approximation to high electric field transport, it does not adequately address the hot carrier problem.

Given the drift diffusion, energy balance, hydrodynamic, and Monte Carlo models in a single environment, it was proposed to determine if the so-called delta-f method can be applied to semiconductor transport. This approach was developed within the magnetic fusion community. It is the numerical analog of the Chapman-Enskog expansion of statistical mechanics, in which the unknown distribution, f , described by a Boltzmann equation is divided into a local equilibrium part, f_B , and a deviation, δf . Here, f_B is an absolute equilibrium solution of the Boltzmann equation neglecting spatial gradients and is described by a few parameters related to the lower moments of the distribution. The δf is calculated using a particle transport method. Closure moment information is obtained from δf to complete the scheme.

This approach toward semiconductor device design offers the potential for a method that treats the nonequilibrium character of the distribution function more accurately than the energy-balance or hydrodynamic approximations but with less computation and reduced noise than in a full-scale Monte Carlo method. This more accurate treatment of transport coupled with the reduced computational requirement could be critical to the design of near-term, three-dimensional devices.

Importance to LANL's Science and Technology Base and National R&D Needs

This project mainly applies to the Los Alamos' theory, modeling, and high-performance computing competency. A secondary Los Alamos competency is nuclear science, plasmas, and beams.

The importance of the development of new algorithms and evaluation of such algorithms against existing algorithms spans all of computational physics. With a plug-and-play environment, the effort associated with algorithm development is greatly streamlined.

To our knowledge, application of the delta-f method to bridge the gap between the carrier temperature or hydrodynamic equations and the full-scale Monte Carlo method has not been suggested for semiconductor device design. From a scientific point-of-view, this project is an example of the cross-fertilization of scientific disciplines leading to a much broader application of a new method. The method is well-matched to the future industrial approach toward object-oriented parallel distributed computation. If the method can be successfully applied to device design, the technical impact is a large reduction in the cost and time required to bring a new product to market.

The microelectronics industry relies primarily on workstations to perform routine design calculations. There is no evidence that the industry will move toward large mainframe computation. Rather, the trend is toward object-oriented distributed parallel computation using a network of high-performance workstations. Given these computational requirements, a hybrid fluid-particle approach, such as the delta-f method, can be utilized in a practical manner for device sensitivity studies. Moreover, the method can be extended to three dimensions, whereas full-scale Monte Carlo in three dimensions is not feasible.

Scientific Approach and Accomplishments

The project was split into three major objectives 1) development of a plug-and-play environment using C++ objects called fragments, 2) incorporation of drift diffusion, energy balance, hydrodynamic, and Monte Carlo microelectronics models into the plug-and-play environment, and 3) evaluation and development of hybrid and delta-f methods for MOSFETs.

The first two objectives were completed successfully during the first and second years of the project. Aspects of these efforts are discussed in the following subsections.

During the last year of the project, we developed numerical diagnostics necessary to evaluate the high-order transport coefficients needed to evaluate hybrid and delta-f

methods. In one dimension, a sufficient number of electrons could be utilized to obtain a reasonable approximation for the high-order transport coefficients. However, in two and three dimensions the noise level remained large, meaning that the fluid equations would be driven by noisy source. Estimates indicated that without smoothing, which would lead to a smearing of the depletion region, the noise would propagate into the fluid solution. From these evaluations, the proposed delta-f approach for microelectronics appeared questionable in two and three dimensions. Given these negative evaluations, the third objective was not pursued and half of the final year's funding was returned to the LDRD program.

In summary, the fundamental research result was the demonstration of a plug-and-play environment in which a large number of new or existing FORTRAN, C, and C++ programs could be integrated, leading to common methods for input, output, restart, graphics, and algorithm development. Such an environment is invaluable for testbed codes and provides an efficient approach for the development and testing for all classes of new and existing algorithms. Finally, the prototype plug-and-play environment was sufficiently powerful to be applied as a spin-off to a brain tomography problem [4] and gas-core nuclear rocket stability problem [3].

A. Introduction

A C++ data-structure class has been used to develop a prototype plug-and-play (PCUBED) environment. This fragment class allows PCUBED to integrate new and existing modules in a straightforward manner. In this context, a module can be a multigrid solver; a mesh generator or a visualization tool; or an entire FORTRAN program. Within PCUBED the fragment class provides the "glue" to bind the modules together. In this context, "glue" refers to 1) dynamic assemblage of data structures, 2) memory management, and 3) standard ASCII/binary input and output.

The fragment class was intended to 1) provide an ability to develop platform independent code, 2) provide faster code development using rapid prototyping and efficient data structures, 3) reduce code maintenance through the reduction in code size, and 4) generate code with high human and computational efficiency.

A fragment is a simple structure that can describe any type of information. For example, a fragment might represent a string, a number, a vector, a set, a function, or any combination of these data types. Independent of what data a fragment represents, all fragments have the same structure. As a result, small fragments can be combined to form larger fragments, then these larger fragments can be combined into still larger fragments. Where this integration process stops is determined by the developer or user.

An advantage associated with using a fragment-based data structure is reduction in code size. Code compression is achieved through the use of toolboxes that contain common functions and operations utilized in many application codes. An application code uses the fragment toolbox and one or more user toolboxes. The fragment toolbox contains input and output operations, communication functions, intrinsic fragment functions, and dynamic memory management operations. The user toolboxes are a collection of general functions that are important to a specific user but may not be general enough to be included in the fragment toolbox.

To insure human efficiency, input, output, restart, setup, and graphics are typically programmed using a high-level approach. In contrast, computationally intensive algorithms are programmed using a low-level approach. Fragments provide a straightforward approach to switch between high-level and low-level programming.

Fragments allow implementation of n-dimensional code in a compact form. In this context, n-dimensional code means that the same lines of code apply to a one-, two-, or three-dimensional problem. In fact, we routinely run one-, two-, and three-dimensional problems for a number of applications.

Within PCUBED there are a number of elementary linear and nonlinear n-dimensional solvers. These solvers can be used directly or in combination with other solvers, including existing FORTRAN solvers. Because we can directly test one algorithm against another, as well as evaluate a specific algorithm written in different programming languages, we can provide a valid test of computational efficiency. In particular, a well-written FORTRAN code is no more efficient than a well-written C or C++ code. Unfortunately, it is straightforward to achieve incomprehensible and inefficient code with any programming language.

PCUBED allows for the generation of many codes within a common framework, including different codes for the same application. Fragments provide an excellent intuitive approach to the development of an efficient architecture, as well as providing a common data implementation within and between codes. PCUBED has been tested on a Macintosh PowerPC; on IBM, SUN, HP, and SGI workstations; and on the CRAY YMP and Cray T3D.

At present, seven major codes can be specified within the PCUBED environment: 1) LEARN, 2) ELLIPTIC, 3) MINIMOS, 4) MONTECARLO, 5) HYBRID, 6) GASCORE, and 7) VNAP. All of these modules and codes are combinations of C++, C, and FORTRAN modules.

- LEARN is a full plug-and-play testing and learning environment. It allows for learning new constructs in C, C++, and FORTRAN. In addition, the LEARN environment allows testing and evaluation of the new constructs including graphic support.
- ELLIPTIC is a full plug-and-play elliptic partial differential equation (PDE) code with selectable modules for normalization, material assignment, initial conditions, boundary conditions, coefficient generation, and solvers. Both linear and nonlinear elliptic problems can be solved, in one, two, and three dimensions. In addition to microelectronics, the ELLIPIC environment has been used to contribute to electron-beam radiography, magnetic coil design, and brain tomography.
- MINIMOS [5] is a 100,000 line FORTRAN module that solves a number of different models for Metal-Oxide Semiconductor Field-Effect Transistors (MOSFETs) [6] in two and three dimensions. In this case, the entire code was incorporated into the PCUBED environment. The only change was to replace the graphics with the interactive graphics. The FORTRAN input, output, and restart structure were left unchanged.
- MONTECARLO is a Monte Carlo code that solves for electron transport in Silicon. This code is constructed from an object-oriented C++ module for the particle pusher and a FORTRAN Multigrid module for the electrostatic potential.
- HYBRID is a hybrid code intended to investigate hybrid and delta-f methods. It is a hybrid fluid-particle model for a MOSFET. The HYBRID environment combines MINIMOS and MONTECARLO.
- GASCORE [7] is a gas-core nuclear rocket FORTRAN module (Poston and Kammash, 1996) combined with C++ input, output, restart, and graphics.
- VNAP [8] is a two-dimensional, time-dependent, compressible, turbulent-flow FORTRAN module. The entire code was integrated into the PCUBED environment. To date, the FORTRAN input, output, and restart structure has been left unchanged. The original graphics were replaced with interactive graphics and post-processor file dumps. This hydrodynamic module was combined with C++ input, output, and restart to provide understanding for a cold-flow gas-core nuclear rocket experiment.

B. Fragment Overview

A fragment is a generalized data type that can represent a string, a number, a vector, a set, a function, or any combination of these data types. A fragment retains the same structure independent of what a given fragment represents. In PCUBED, a fragment has the following structure: variableF: | label | type | length | variable data/address |. In this structure variableF is the variable name. To work with the variableF fragment, one must access the four fragment fields: label, type, length, and data/address. This access operation is done by using so-called member functions.

Fragments allow one to construct very complex data structures including sets. In this case, sets refer to a collection of different data types. A set might be a collection of text and data to perform a specific function. For example, in PCUBED all information about boundary conditions for a given PDE forms a set. In this fashion, a set acts somewhat like a common block in FORTRAN. With fragments, on the other hand, such sets are dynamic in shape, size, and content.

C. Plug-and-Play Implementation

At the highest level of complexity, the fragment combines a number of different functions (C and C++ procedures and/or FORTRAN subroutines) into a single set. This single functional fragment represents a specific code. At the same time, one can generate a second functional fragment with a different mix of procedures and/or subroutines. This second functional fragment represents a second code. At one end of the spectrum, the difference between these two codes might just be a change in the solution method for a specific application. At the other end of the spectrum, the first code might be a microelectronics code whereas the second code might be a gas-core nuclear rocket code. Specific code generation is controlled by the developer or user.

For a given application, we might need functions to perform normalization, material specification, boundary conditions, initial conditions, coefficient generation, and algorithm solution. At the highest level, we define a fragment that will represent this functional space. This fragment will be of the type VFrag, a vector of fragments. For illustration, we consider only an application that requires the solution to a single PDE. Then, at the next level down, each fragment will be of type VFUN, a vector of function pointers, each of length one.

A functional space for a single elliptic PDE is shown in Exhibit 1. The length of the functional fragment is seven, indicating that there are seven elements that define this code: code, normalization, material specification, boundary conditions, initial conditions, coefficient generation, and solution. In general, there is no restriction on the number of

elements. The data field associated with this functional space fragment contains the address to an array of fragments. The data field of each element of that array, each element being a fragment, contains the address to the specified function.

functional_spaceF:	functional space	VFRAG	6	0x...
code_functionF:	elliptic	VFUN	1	0x...
normalization_functionF:	normalization	VFUN	1	0x...
material_functionF:	material	VFUN	1	0x...
boundary_condition_functionF:	boundary_condition	VFUN	1	0x...
initialization_functionF:	initialization	VFUN	1	0x...
coefficient_functionF:	coefficient	VFUN	1	0x...
algorithm_functionF:	algorithm	VFUN	1	0x...

Exhibit 1. Representation of a functional space plug-and-play fragment.

An actual plug-and-play functional fragment, corresponding to the representative input fragment shown in Exhibit 1, is shown in Exhibit 2. The number of function members in the actual functional fragment is ten rather than seven because the material fragment represents three materials whereas the algorithm fragment represents both the norm and solver. Thus, the actual functional fragment contains *code*, *normalization*, *diffusion*, *cross_section*, *source*, *boundary*, *initialization*, *coefficient*, *norm*, and *solver*. Note the first function fragment in the functional set specifies the code to be constructed or generated, which could be *learn()*, *elliptic()*, *minimos()*, *monte_carlo()*, *hybrid()*, *gascore()*, or *vnap()*. The bracket at the end of each name indicates that the name is associated with a function.

This functional input corresponds to a linear diffusion equation for neutron transport, an elliptic PDE. Thus, we specify the *elliptic* code. For each material specification there are actually two functions. In this example, the indexes of the material regions are first calculated, and then the materials are averaged to the correct positions on the grid. In this fashion, one can make a straightforward change from cell-center to cell-vertex material specification. An L_1 norm is specified along with a multigrid solver. By simply changing the name of a function, a new code configuration is generated. As a simple example, suppose we want to change the norm used by the solver to the L_2 norm. We simply replace *L1NormC()* with *L2NormC()* in the functional input and reexecute the functional fragment.

D. Integration of FORTRAN Modules

From a pragmatic point of view, the best module is one that works. Unfortunately, one individual's working module can often become another individual's nightmare. Moreover, when one attempts to integrate a number of different modules together, the use of existing

```

FunctionalSpace
{
  code          [ elliptic () ]
normalization   [ DSA_normalization() ]
diffusion       [ region_to_vertex_index(), diffusion_block_ce() ]
cross_section   [ region_to_vertex_index(), cross_section_block_cv() ]
source          [ region_to_vertex_index(), source_block_cv() ]
boundary        [ boundary_set_index() ]
initialization  [ flux_zero() ]
coefficient     [ coefficient_elliptic_cell() ]
norm            [ L1NormC() ]
solver          [ flux_2D_MG() ]
}

```

Exhibit 2. Actual structure of a functional space plug-and-play fragment.

software can become a significant problem. Typically, there is little or no uniformity in data structure; in the human interface; or in input, output, restart, and graphics. PCUBED directly attacks this set of problems by providing 1) a powerful technique for bundling various data structures into a new shape and 2) a standard technique for input, output, restart, and graphics. PCUBED uses the fragment toolbox to achieve these functions.

A working FORTRAN program can be integrated into the plug-and-play environment. Once integrated, the program can be used directly with no change. This process, however, is not very useful, except to test if the module will compile on different platforms. A more useful exercise is to strip the input, output, restart, and graphics from the program. These operations are then replaced by fragment bridge functions. In this fashion, input, output, restart, and graphics are standardized and are straightforward to modify. Most importantly, dynamic graphical debugging can then be inserted at any point within the FORTRAN program.

As an example, consider the small FORTRAN code shown in Exhibit 3. This code is a good one, since it will compile and execute on a number of platforms. However, the code has limited utility because there is no input, output, or graphics; it simply calculates a set of numbers which will describe a circle.

To integrate this FORTRAN program into PCUBED, we must first convert the program into a subroutine. Then, as a demonstration for input, we let the cell size delta, the distance between x or y points, become an input variable. Finally, as a demonstration of interactive graphics, we add a plot of the variable $z(x,y)$. The converted program is shown in Exhibit 4, with changes indicated by bold type. The cell size delta is now a subroutine argument. The variables *name* and *pp* are the name of the plot and the name of the post processor. The subroutine plot is a fragment bridge function which can print a fragment to the screen, dump a fragment for interactive debugging, and/or dump a fragment for post processing. Note that it is not important to understand the details, rather it is

important to notice there are very few changes to achieve the goal of adding input, output, and graphics.

Having modified the FORTRAN program, an interlanguage naming convention as well as the FORTRAN prototype must be declared in a header file. In addition, a plug-and-play function prototype must be specified and registered in a header file. Finally, the plug-and-play function must be written into a program file. The basic forms for the interlanguage linking convention, FORTRAN prototype, plug-and-play function prototype and registration, and the plug-and-play function are shown in Exhibit 5. Again, it is not important to understand the details, rather it is important to understand that a major effort is not required to add modules to the environment. The function *get_check_ref* in the plug-and-play function *learn_to_plot* checks for consistency in the input fragment. The function searches the input fragment for the specified fragment label. If the input variable is found, the variable type is checked. If there is an inconsistency, PCUBED stops and issues an error message.

```

      program testplot
      parameter ( nx = 101 , ny =101)
      parameter ( nz = nx*ny )
      real dx, dy
      real delta
      integer i,j
      dimension x(nx),y(ny),z(nz)
      delta = 0.2
      dx = delta
      dy = delta
      xmin = -50.0*delta
      ymin = -50.0*delta
      do 10 i = 1, nx
10      x(i) = xmin + (i-1)*dx
      do 20 j = 1, ny
20      y(j) = ymin + (j-1)*dy
      do 30 i = 1, nx
      do 30 j = 1, ny
      ijk = i +(j-1)*nx
30      z(ijk) = x(i)**2 + y(j)**2
      stop
      end

```

Exhibit 3. A FORTRAN code with no input, output, or graphics.

```

subroutine testplot ( delta )
parameter ( nx = 101 , ny =101)
parameter ( nz = nx*ny )
real dx, dy
integer i,j
character name*10, pp*10
dimension x(nx),y(ny),z(nz)
dx = delta
dy = delta
xmin = -50.0*delta
ymin = -50.0*delta
do 10 i = 1, nx
  x(i) = xmin + (i-1)*dx
do 20 j = 1, ny
  y(j) = ymin + (j-1)*dy
do 30 i = 1, nx
do 30 j = 1, ny
  ijk = i +(j-1)*nx
30  z(ijk) = x(i)**2 + y(j)**2
  name = 'circle'
  pp = 'peg'
  call plot(name,x,nx,y,ny,z,nz,pp)
  return
end

```

Exhibit 4. A modified FORTRAN code with input, output, and graphics.

```

// INTERLANGUAGE LINKING CONVENTION
#ifdef CRAY
#define testplot TESTPLOT
#endif

// FORTRAN PROTOTYPE
extern "C" void testplot ( REAL & ) ;

// PLUG-AND-PLAY FUNCTION PROTOTYPE
extern FRAG & learn_to_plot ( const FRAG& , FRAG& ) ;

// REGISTRATION AND PLUG-AND-PLAY FUNCTION
register_function ( "learn_to_plot" , learn_to_plot ) ;

// PLUG-AND-PLAYFUNCTION
FRAG & learn_to_plot (const FRAG &inputF , FRAG &outF ) {
  FRAG *inputFPtr = (FRAG*)(&inputF) ;
  REAL delta =
    get_check_ref (*inputFPtr,"cell_size",_REAL).R() ;    testplot (
delta ) ;
  return ( outF ) ;
}

```

Exhibit 5. Interlanguage linking convention, FORTRAN prototype, plug-and-play prototype, plug-and-play function registration, and plug-and-play function are required to complete the integration.

As a final step, an input file must be constructed to execute the plug-and-play fragment function *learn_to_plot* defined above, which calls the FORTRAN *testplot* subroutine. An

input file contains a single fragment that is read and parsed by the fragment toolbox. The input file for the example code is shown in Exhibit 6.

The input fragment is *learnToPlot*{}, where {} means a set of fragments. For PCUBED the complete input fragment always contains a set of five input-space fragments: { *functionalSpace*, *controlSpace*, *virtualSpace*, *physicalSpace*, and *computationalSpace* }. The fragment *functionalSpace* contains the names of the functions and files necessary to define a code or application. Next, the fragment *controlSpace* contains the information needed for numerical algorithm control. The fragment *virtualSpace* contains information needed for domain decomposition of physical space. Next, the fragment *physicalSpace* contains information needed for the physical description of the problem. Finally, the fragment *computationalSpace* contains information needed to restart the problem. In this example, we only need *functionalSpace* and *controlSpace*. In *functionalSpace*, we want to execute the *learn* code. Within the *learn* code, we want to execute the example *learn_to_plot*. In *controlSpace*, we specify the *cell_size*.

```
LearnToPlot
{
  functionalSpace
  {
    code [ learn() ]
    example [ learn_to_plot() ]
  }
  controlSpace { cell_size 0.20}
  virtualSpace { processor { } cache { } }
  physicalSpace { }
  computationalSpace { }
}
```

Exhibit 6. A PCUBED input file to execute the *learn_to_plot* example in the *learn* code for the specified *cell_size*.

A small number of fragment bridge functions can provide basic graphic information in one, two, and three dimensions. As an example, the fragment bridge function *plot* is shown in Exhibit 7. In the first block of the function, *f* is the fragment label and *p* is the post-processor's name. Both names enter as pointers to FORTRAN character variables. These names are converted to C character strings. The second block of the function makes a basis-vector fragment. The basis-vector information enters as pointers to *x* and *y*. These pointers are stored using the EXTERNAL fragment option: the data is just rebundled but memory is not allocated. The strings *basis_vector*, *x1* and *x2* are labels for the data. They are not variable names. In this context, the fragment label provides for self-documentation of variables. The third block of the function makes the output fragment *oF*. All the information is bundled into this single fragment. The higher level functions *getdims* and *mesh_coords* provide stride and coordinate information using the basis-vector fragment.

In some cases, the stride and coordinate information are known in the FORTRAN code. In that case, the stride and coordinate pointers can be stored directly using the EXTERNAL option. Finally, the pointer to the field information, z, is stored, again using the EXTERNAL fragment option. In the last block of the function, the output fragment is printed, written to a dump file, or exported to interactive graphics, depending upon compile line options.

```
void plot ( char *f , REAL *x , int &nx , REAL *y , int &ny , REAL *z ,
           int &nz , char *p )
{
// CONVERT FROM FORTRAN NAMES TO C NAMES
  int n ;
  char fname[32] ;
  for ( n = 0 ; f[n] != ' ' && f[n] != '\0' ;      n++ )
  {
    fname[n] = f[n] ;
    fname[n] = '\0' ;
  }
  char pname[32] ;
  for ( n = 0 ; p[n] != ' ' && p[n] != '\0' ;      n++ )
  {
    pname[n] = p[n] ;
    pname[n] = '\0' ;
  }

// CONSTRUCT BASIS_VECTOR FRAGMENT
  FRAG basisF ("basis_vector",_VFRAG,2) ;
  basisF.VFRAG(0).set("x1",nx,x,EXTERNAL) ;
  basisF.VFRAG(1).set("x2",ny,y,EXTERNAL) ;

// CONSTRUCT OUTPUT FRAGMENT
  FRAG oF ("contour_fragment",_VFRAG,3) ;
  oF.VFRAG(0).set("meshStride") = getdims( basisF ) ;
  oF.VFRAG(1).set("meshCoordinate") = mesh_coords( basisF ) ;
  oF.VFRAG(2).set(fname,nz,z,EXTERNAL) ;

// OUTPUT FRAGMENT AS PRINT, FILE, OR GRAPHIC }
  PR( oF.p() ; )
  DP( dump( oF, pname ) ; )
  PL( export( oF, pname ) ; )
}
```

Exhibit 7. This fragment bridge function provides data for slice and contour plots for a two-dimensional code. The choice of the postprocessor specified by pname is determined by the user.

E. Comparison - Language Performance

With the plug-and-play environment, simple or very complicated algorithms can be compared against one another. One can compare different algorithms written in the same programming language. On the other hand, one can compare the same algorithm written in different programming languages. As a simple example, we consider a SAXPY loop

(Single precision A times X Plus Y). The SAXPY loop is used by a leapfrog particle pusher or the inner loop of a matrix multiply, see Exhibit 8.

```
do 10 i = 1, n
10  y(i) = a*x(i) + y(i)
```

Exhibit 8. A SAXPY loop.

Seven forms of the algorithm were written in FORTRAN, C, and C++ and tested on the CRAY-YMP:

- A Standard FORTRAN 77 Implementation (f77).
A procedural implementation. This implementation vectorizes on the YMP without any compiler directives. Recall that CRAY FORTRAN assumes x and y aren't aliased, which permits automatic vectorization. If the non-aliased assumption isn't true, due to parameter aliasing or equivalents, the loop will give the incorrect results.
- A Standard C Implementation (C)
A procedural implementation. This implementation will vectorize on the YMP if x and y are declared "restrict." Recall that CRAY C assumes x and y are aliased unless otherwise told. The "restrict" keyword assures the compiler that the arrays aren't aliased, so it can produce vectorized code. The "restrict" keyword will be part of the ANSI C release.
- A C Implementation Compiled with C++ (CC)
A procedural implementation. This is the same implementation as the C code, except the code is compiled with the C++ compiler rather than the C compiler.
- A "Pure" C++ Implementation (CC0)
A pure object-oriented implementation. The compiler cannot chain operator* and operator+=; thus, at best, two vector loops are needed. Further, the compiler makes no assumptions about the aliasing of the temporary and the variable x , so that loop isn't vectorized.
- A "Mixed" C++ Implementation (CC1)
A mixed procedural and object-oriented implementation. An inline operator[] is defined for vector types. The "ivdep" directive is used in an attempt to get the compiler to vectorize the loop. Unfortunately, "ivdep" specifies that the arrays are unaliased to a

recurrence threshold of 64. The possibility of a recurrence requires the compiler to generate code for a "complete memory reference," which adds an additional 30% overhead to loop execution.

- A "Local" C++ Implementation (CC2)
A mixed procedural and object-oriented implementation. Temporary restricted pointers are aliased to the vector data and used in the loop. This code could be hidden in a SAXPY member function of the vector class.
- A "Restrict" C++ Implementation (CC3)
A mixed procedural and object-oriented implementation. The vector class contains "restrict" pointers and an inline operator[ ]. With the "-h vector3" compile option (enabling the recognition of restricted pointers as member data) this produces code that looks and performs just like the FORTRAN 77 intrinsic arrays but that has all of the advantages of a user defined type.

The performance data as a function of implementation is shown in Fig. 1. The three procedural implementations (f77, C, and CC) are essentially the same. The "pure" object-oriented implementation (CC0) has a performance of less than one-half of the procedural implementations. With care, the object-oriented implementation will perform at the same level as the procedural implementations (CC2 and CC3). It is clear that performance is in the details of the implementation and not in the programming language.

Figure 1. Comparison of performance of a SAXPY loop on the Cray YMP, for Fortran 77, C, and a number of C++ implementation.

F. Comparison - Algorithm Performance

As stated previously, a MOSFET is a semiconductor device. A sketch of a MOSFET is given in Fig. 2. As part of the development of a hybrid particle-fluid code to address short-channel MOSFETs, one needs to solve for the electrostatic potential. A MOSFET has a number of material regions, including air, metal contacts, insulators, and doped semiconductors. The electrostatic potential is obtained from a nonlinear Poisson equation, with voltage, charge neutral, and Neumann boundary conditions.

With the plug-and-play environment, we investigated a number of different methods to solve for the electrostatic potential. Three classes of methods were considered: 1) elementary direct nonlinear solvers, 2) linear solvers, and 3) combinations of nonlinear and linear solvers. The elementary nonlinear solvers are Point Jacobi, Red-Black Gauss Seidel, Red-Black SOR, and Point Jacobi coupled with SOR. The second approach linearizes the nonlinear equation about the last solution, solves the linear equation, and then repeats the process until convergence is achieved. A linear multigrid routine is used for this approach. Finally, the third approach adds a nonlinear Point Jacobi solution before each linear multigrid step, then repeats the nonlinear-linear step until convergence is achieved.

Figure 2. Conceptual sketch of a MOSFET.

As an example of a performance study, in Fig. 3 the relative performance between the three approaches is shown for an HP 735 workstation. If we are in an inner loop, where the electrostatic potential needs to be recalculated at each iteration, then we have a good initial guess and want a quick reduction of the residual by about four orders-of-magnitude. In that case, the best method is the nonlinear Point Jacobi solver coupled with the nonlinear Red-Black SOR. On the other hand, if the residual must be reduced to machine accuracy,

the combined method of nonlinear Point Jacobi and iterative linear multigrid is a better choice.

Figure 3. Relative performance of the three approaches, for a residual reduction of four-orders-of magnitude for the solution of a nonlinear poisson equation. Here, PJ is direct nonlinear Point Jacobi solver, RBSOR is direct nonlinear Red-Black SOR solver, PJRBSOR is a coupled direct nonlinear method using both Red-Black Point Jacobi and SOR solvers, MG is an iterative linear multigrid solver, and PJMG is a coupled method using direct nonlinear Point Jacobi and iterative linear multigrid solvers.

G. Interactive Graphics

Suppose an existing Fortran program has been integrated into PCUBED. At this point, it can be used directly with no change. This code is not very useful, however, except to test if the module will compile on different platforms. Of course, getting a code to compile on different platforms can be a nontrivial task, but PCUBED is not necessary to complete such an task. On the other hand, PCUBED can make such an exercise a little more straightforward, because PCUBED is intended to be platform independent.

So why bother to integrate existing Fortran codes into PCUBED? Basically, it is the ability to add interactive graphics in a simple fashion to provide new diagnostic capability to a code. Generally speaking, adding a new diagnostic can be a complicated procedure, because changes often need to be added in many different places within the code. With PCUBED, adding a new diagnostic is a local process, typically adding only a few lines in the Fortran code. Furthermore, it is possible for the user to specify the visualization tool to be used to view the new diagnostic.

Summary

The project was split into three major objectives 1) development of a plug-and-play environment using C++ objects called fragments, 2) incorporation of drift diffusion, energy balance, hydrodynamic, and Monte Carlo microelectronics models into the plug-and-play environment, and 3) evaluation and development of hybrid and delta-f methods for MOSFETs.

The first two objectives were completed successfully during the first and second years of the project. Fragments provide an excellent intuitive approach to the development of an efficient architecture, as well as providing a common data implementation within and between codes. As a result, the PCUBED environment allows for the generation of many different codes within a common framework. PCUBED has been tested on a Macintosh PowerPC; on IBM, SUN, HP, and SGI workstations; and on the CRAY YMP and Cray T3D. Using this environment, we have incorporated a drift diffusion, energy balance, hydrodynamic, and Monte Carlo model for metal-oxide semiconductor field-effect transistors (MOSFETs) into a single architecture.

During the last year of the project, we developed numerical diagnostics necessary to evaluate the high-order transport coefficients needed to evaluate hybrid and delta-f method. In one dimension, a sufficient number of electrons could be utilized to obtain a reasonable approximation for the high-order transport coefficients. However, in two and three dimensions the noise level remained large, meaning that the fluid equations would be driven by noisy source. Estimates indicated that without smoothing, which would lead to a smearing of the depletion region, the noise would propagate into the fluid solution. From these evaluations, the proposed delta-f approach for microelectronics appeared questionable in two and three dimensions. Given these negative evaluations, the third objective was not pursued and half of the final year's funding was returned to the LDRD program.

The fundamental research result was the demonstration of a plug-and-play environment in which a large number of new or existing FORTRAN, C, and C++ programs could be integrated, leading to common methods for input, output, restart, graphics, and algorithm development. Such an environment is invaluable for testbed codes and provides an efficient approach for the development and testing for all classes of new and existing algorithms. Finally, the prototype plug-and-play environment was sufficiently powerful to be applied as a spin-off to a brain tomography problem and gas-core nuclear rocket stability problem.

Publications

1. Thode, L. E., Hotchkiss, R., Snell, M., Gray, M, Turner, S., and Plumer, E., "Plug-and-Play Environment for Algorithm Development (U)," Nuclear Weapons Code Developer's Conference (San Diego, CA, 1996), Los Alamos National Laboratory report LA-CP-96-224 (1996).
2. Hotchkiss, R., Thode, L. E., and Plumer, E., "Code Development with Fragments - The Universal Object," Nuclear Weapons Code Developer's Conference (San Diego, CA, 1996), Los Alamos National Laboratory report LA-CP-96-232 (1996).
3. Plumer, E., Thode, L. E., Hotchkiss, R., "Unified Data Interchange Using Fragments," Nuclear Weapons Code Developer's Conference (San Diego, CA, 1996), Los Alamos National Laboratory report LA-CP-96-234 (1996).
4. Thode, L. E., Cline M. C., and Howe, S. D., "Vortex Formation and Stability in a Scaled Gas-Core Nuclear Rocket Configuration," AIAA conference paper 97-2955 (Seattle, WA, 1997). Submitted for publication to Journal of Propulsion and Power.

References

- [1] Hotchkiss, R., "The Fragment—The Universal Object," Los Alamos National Laboratory Memorandum XCM:96-10 (U) (1996).
- [2] Plumer, E. and Hotchkiss, R., "Fragment Definition Language (FDL)," Los Alamos National Laboratory Memorandum XCM:96-86 (U) (1996).
- [3] Thode, L. E., Cline M. C., and Howe, S. D., "Vortex Formation and Stability in a Scaled Gas-Core Nuclear Rocket Configuration," AIAA paper 97-2955 (Seattle, WA, 1997). Submitted for publication to Journal of Propulsion and Power.
- [4] L. E. Thode, "Diffusion Approximation Invalid for Brain Tomography," unpublished.
- [5] Hansch, W. H. and Selberherr, S., "MINIMOS 3: A MOSFET Simulator that Includes Energy Balance," *IEEE ED-34*, pp. 1074-1078, (1987).
- [6] Shur, M., Physics of Semiconductor Devices (Prentice Hall, New Jersey, 1990), p.353.
- [7] Poston, D. I. and Kammash, T., "A Computational Model for an Open-Cycle Gas Core Nuclear Rocket," *Nuclear Science and Engineering* **122**, pp. 32-54, (1996).
- [8] Cline, M. C., "VNAP2: A Computer Program of Two-Dimensional, Time-Dependent, Compressible, Turbulent Flow," Los Alamos National Laboratory Report LA-8872 (1981).