New User Interface for Tiny and Other Extensions

User Guide

Wayne Kelly
Dept. of Computer Science
Univ. of Maryland, College Park, MD 20742
wak@cs.umd.edu, (301)-405-2726

Vadim Maslov
Dept. of Computer Science
Univ. of Maryland, College Park, MD 20742
vadik@cs.umd.edu, (301)-405-2726

January 29, 2013

Contents

1 Introduction

This document is intended for use by users of Michael Wolfe's Tiny Tool as extended by the University of Maryland. It is assumed that users are already familiar with the functionality of the original Tiny Tool. The extensions described below were implemented to allow Tiny to be used on non trivial sized programs. It is now possible to examine and transform programs that are longer that one screen in length by being able to change the portion of the program displayed on the screen. The new user interface also includes supplemental ways to view dependences. All existing capabilities of Tiny have been maintained (to the best of our knowledge).

November 1992 release of Tiny by University of Maryland at College Park (UMCP) research group has many new features. Here's a short list of important ones:

- A Unified system for performing transformations
- Induction variable recognition (IVR) using Static Single Assignment (SSA) graph
- Forward substitution of scalars
- Elimination of dead assignments to scalars
- Fortran-77 to Tiny converter named *f2t*
- Built-in functions
- Storage classes, Entry and Exit nodes for dependences involving global variables
- Update (assignment) statements `+=`, `*=`, `max=`, `min=`
- Reduction dependences
- DOANY loop
- Scalar and array expansion to kill anti and output dependences
- Array privatization
- Strip-mine loop transformation

2 Command line options

Tiny is invoked with any number of command line options, and, (optionally,) a initial file to read in. Options may precede file name or follow it. The command line options are:

`-a`*d* Use data dependence analysis algorithm number *d*. By default Omega test is used. Currently the following algorithms may be used (identified by algorithm number):

1 DDalg_banerjee: Banerjee test.

2 DDalg_simultaneous: ?

3 DDalg_sim_exp: ?

4 DDalg_sim_constrain: Power test.

5 DDalg_no_banerjee: ?

6 DDalg_lambda: Lambda test

7 DDalg_omega: Omega test

`-c` Ignore Tiny program comments. That is, don't display them and don't keep them in Tiny Abstract Syntax Tree (AST). By default comments are displayed.

`-d` Make arrays have AUTO storage class by default, that is, if no storage class is explicitly mentioned for the array. Normally arrays are of INOUT class (equivalent to COMMON) by default.

`-e` Don't perform elimination of dead scalar assignment statements. Performed by default.

`-g` Don't compute dependences coming from Entry node to all variables defined on entry to the Tiny procedure and from all variables used on exit from the procedure to Exit node. Done by default.

`-i` Don't perform induction variable recognition. IVR is performed by default. SSA graph is always built (even if IVR is not performed).

`-l` Replace affine expressions with their canonical form. Not done by default.

`-p` Perform privatization of arrays for the whole program. Not done by default.

`-r` Try to convert assignment statements to update statements. That is, perform transformations like `s = a+s+b --> s += a+b`. This conversion is not done by default.

`-q` Make Tiny indicate what phase of analysis it is running (parsing, IVR stuff, dependence analysis). Off by default.

`-s` Skip data dependence analysis for scalar variables.

`-t` Skip data dependence analysis for top-level dependences (dependences between references that share no common loops).

`-x` Perform array expansion for the whole Tiny program. Not done by default.

`-y` If array expansion is performed, do it only once. By default array expansion is repeated until no further changes can be made.

`-z` Don't substitute expressions containing scalar variables with unknown values. That is, substitution will be done only for expressions containing constants and loop variables. By default *integer* scalar variables which have unknown values can form a part of expression to be substituted.

`-2` Skip Omega-2 analysis, that is, dependence killing, termination, covering, and refinement.

Not skipped by default.

- `-A` Run somewhat expensive assertions in the code for static single assignment and induction variable recognition (useful for discovering bugs). Not done by default.
- `-I` Write a lot of induction variable recognition debugging information to `trace.out` file.
- `-D` Don't perform dependence analysis at all. May be useful if you want to see effects of IVR and don't need dependence information.
- `-O`*level* Set Omega test debug level. May be `-O1`, `-O2`, or `-O3`. The more the debugging level, the more debugging information will appear in `trace.out` file.
- `-P` Turn on printing of results of Omega test.
- `-Z` Print dependence zap gists.

3 New user interface

3.1 Scrolling

In the original Tiny, it was intended that the entire program be displayed on the screen. The new implementation treats the screen as a window through which the program can be viewed. At any time only a portion of the entire program can be seen through the window. The portion of the program that is currently visible through the window can be changed by the user both directly and indirectly.

3.1.1 Direct Control

Whenever the program is displayed on the screen the user is free change which portion of the program is visible through the window by scrolling the window up or down using the keys described below. These scrolling options do not appear explicitly in any menu, but they are implicitly available in all menus where the program is displayed.

The keys used for scrolling are based on those used by the "vi" editor available under Unix. The keys used are: 'ctrl j' and 'ctrl k' cause the window to scroll down one line and up one line respectively; 'ctrl d' (d for down) and 'ctrl u' (u for up) cause the window to scroll down half a page and up half a page respectively; finally 'ctrl f' (f for forward) and 'ctrl b' (b for backward) cause the window to jump one page forward or one page backward respectively.

3.1.2 Indirect Control

The user controlled scrolling in the previous section does not affect the node and/or dependence if any, that is currently being browsed. However by modifying the node and/or dependence that is currently being browsed, the user may cause the current node and/or dependence to no longer be in the portion of the program visible through the window. When this occurs the Tiny system automatically scrolls the window up or down appropriately, so that the the current node and/or dependence is visible through the window. It is sometimes not possible to display both nodes involved in a dependence on the screen at the same time because they are more than one screen apart. When this occurs only one of one two nodes is marked on the screen. However the user can still scroll manually to the part of the program where the other node involved in the dependence is located and it will be shown marked as normal.

3.2 The Mouse

The mouse can now be used to select menu options by clicking on the appropriate word in the menu line. This allows Tiny to be used almost entirely without the use of the keyboard. The mouse can also be used to change the current node and/or dependence being browsed.

Mouse works only in `xterm` terminal emulator running under X Windows System in UNIX operating system.

Clicking the left button on a node causes it to become the current node being browsed. If the user is not already in the browse menu then they are automatically placed in the browse menu, and when they exit this menu they are returned to the menu they were working in when they clicked the button. If the user is already in the browse dependence menu when they click the left button and the node they click on is involved in a dependence then the first dependence that node is involved in becomes the current dependence being browsed.

The middle button is used change the dependence currently being browsed. If the user is not already in the browse dependence menu then they are automatically placed in the browse dependence menu, and when they exit this menu, they are returned to the menu they were working in when they clicked the button. The middle button can only be used to select nodes that could possibly be involved in dependences. The first dependence involving the selected node, if any, becomes the current dependence being browsed. The middle button also has the affect of pulling up the pop up dependence window described below.

The right button is used to toggle whether this dependence window is displayed on the screen.

3.3 Dependence status

In any listing of a dependence, the status of a dependence may be indicated by a set of letters in square brackets at the right end of the line describing the dependence. These status flags arise from two types of extended analysis. First, we attempt to determine which dependences are based on values and which are based solely on sharing memory locations. Dependences based solely on shared memory locations are termed *dead.* We also check to see if there exist interesting assertions that can be added to the program so as to eliminate the dependence. The meanings of the letters are as follows:

`R` - the direction/distance associated with this dependence has been updated (*refined*) so as to reflect dependences based on references values, not shared memory references

`C` - This is a *covering* dependence; every location accessed by the sink of the dependence is first overwritten by the source of the dependence

`T` - This is a *terminating* dependence; every location accessed by the source of the dependence is later overwritten by the sink of the dependence

`k` - This dependence was *killed* by an explicit test and therefore dead.

`r` - this dependence is the original (pre-*refined*) version of a dependence that has been refined (as is therefor as dead).

`c` - This dependence was quick killed by a *covering* dependence.

`t` - This dependence was quick killed by a *terminating* dependence.

`Z` - This dependence can be *zapped* (i.e., an interesting assertion can be used to delete this dependence).

For a description of these terms, see the SIGPLAN PLDI'92 paper on the Omega test.

3.4 The Dependence Window

The dependence window is a pop up window that appears on top of the main window. It contains a list of the dependences involving the node currently being browsed. At any time this dependence window will only show some of these of these dependences, however this window can also be scrolled to see the other dependences. This scrolling is controlled by clicking on the "Up" and "Down" options in the dependence window. By default, dependences involving the current browse node as either the source or destination node are shown in the window. The dependences involving the current browse node as the destination node are separated from the dependences involving the current browse node as the source node are separated by a line of $\sim$'s. Initially the dependences involving the current browse node as the source node are displayed. The dependences involving the current browse node as the destination node can be viewed by scrolling the dependence window up.

The first dependence in the dependence window is always the current dependence being browsed, therefore scrolling the dependence window changes the current dependence being browsed. Correspondingly changing the current dependence being browsed using existing Tiny features, will change the dependences shown in the dependence window (e.g. "cycle" and "next").

The size and position of the dependence window can be altered by clicking on the "Resize" and "Move" options in the dependence window. A move is achieved by pressing a mouse button down on the "Move" option, and then dragging the mouse to the line representing the required top of the dependence window. A resize is achieved similarly by pressing a mouse button down on the "Resize" option and then dragging the mouse to the line representing either the required top or bottom of the dependence window. The dependence window also contains a set of toggled filter options which modify the dependences which are shown to the user as described in the next section.

3.5 Dependence Filters

The dependence window contains a set of filter options. Clicking on these options toggles whether the corresponding filter should be applied to the dependences shown to the user. If a filter is "on" then that option is highlighted in the dependence window and dependences of the type corresponding to this filter are shown to the user. Initially all filters are set to "on". The filter settings remain in affect until they are changed, even after the dependence window is closed. The filters affect all dependences that are shown to the user, not just dependences shown in the dependence window, i.e. they affect the existing "cycle" and "next" options for example.

There are four filters "flow", "anti", "output" and "reduce" (the last one for reduction dependences) which determine which types of dependences are shown to the user. The four filters "dead", "$\sim$dead", "$\sim$refine" and "$\sim$cover" determine whether dependences that have are dead (killed or covered), not dead, not refined and are not covering respectively should be shown to the user. The "∼carry" filter determines whether loop independent dependences are shown to the user. Finally the scalar filter determines whether dependences between scalars are shown to the user.

3.6 Zapping Dependences

In order to zap dependences in Tiny, select the zap option which is under the DD option, which in turn is under the browse option in the main menu. When this is done, Tiny displays the conditions that must hold in order for a dependence to exist. The user can then indicate that one of these conditions is known to be false, and Tiny will add an assertion negating that condition, and remove the dependence.

Tiny can not handle those cases in which the conjunction of the conditions is known to be false, but no individual condition is false. Thus if the conditions displayed are

1. 3 <= n
2. n <= 10

and we know that n < 3 holds then we just choose 1. But Tiny can not handle the case in which we know that either n < 3 or n > 10 is true, but neither condition by itself is always true.

4 Changes to Tiny language

Except for changes listed below some "syntactic sugar" changes were made. These changes should be checked at Tiny grammar file `src/tinyy.y`.

4.1 Assert Statements

Assert statements can now be added to a Tiny program. These can be placed anywhere in the program text and must be true for the entire program. The syntax for these statements is

`assert`(<arithmetic expression> <comparison operator> <arithmetic expression>)

where a <comparison operator> can be any one of >, <, >= and <=, and the arithmetic expressions must be comprised of numeric and symbolic constants and arithmetic operators.

4.2 Change of Operator precedences

The exponentiation operator (∗∗) now has the highest precedence and the <, <=, >, >= and <> have been assigned the lowest precedence. Thus the expression

$$a ** 2 + b * c < 10$$

is now equivalent to

$$((a ** 2) + (b * c)) < 10.$$

4.3 Built-in functions

Built-in functions can now be used in Tiny program. First, built-in function should be declared like

```
builtin real abs(), sin(), myfun()
```

This declaration informs Tiny that functions `abs`, `sin`, and `myfun` have no side effect and can be treated as conventional arithmetic operations.

F2t converter automatically declares all standard Fortran-77 functions as `builtin`.

4.4 Storage classes

Now each Tiny variable has a storage class. It's needed to tell whether variable is defined on entry to the procedure and whether variable value is used on exit from the procedure. This information is used by the IVR transformations and by the dependence analyzer.

There are following storage classes defined:

`auto` Automatic variable. No value is assigned to variable on entry to procedure, and value of variable on exit from the procedure is not used.

`common`

`static`

`formal` Common, static or formal (dummy parameter) variable. Variable has a value on entry to procedure and its value on exit from procedure is used. The class `common` corresponds to Fortran COMMON. The class `static` corresponds to Fortran SAVEd. The class `formal` corresponds to Fortran subroutine dummy parameter.

`in` Variable has a value on entry to the procedure, but its value on exit from the procedure is not used.

`out` Variable has no value on entry, but its exit value is used.

`inout` Equivalent to COMMON — both entry and exit values are present.

`private` Private variable. There exists separate instance of the private variable for each point of iteration space created by loops surrounding the declaration of this variable. Therefore no dependence can be carried by surrounding loops for private variable.

`builtin` Built-in function — see above.

`const` Named constant.

When variable is declared its class should precede variable type. If class is omitted then variable is considered to be of the class `auto` for scalars and of class `inout` for arrays. Also if there is a use of scalar variable before its definition in Tiny program and no storage class is explicitly assigned to variable, the variable gets `in` class.

F2t converter generates relevant storage classes for Fortran variables.

4.5 Update statements `+=`, `*=`, `max=`, `min=`

Now it's possible to write a la C

```
  s += a(i,j) + b(k)
```

instead of

```
  s = a(i,j) + b(k) + s
```

We introduced update statements only for both commutative and associative operations (`+, *, min, max`). It was done so because we introduced reduction dependences which may exist only between two updates (see below).

There's no need to rewrite existing Tiny programs to get advantage of reduction dependences which are less restrictive than other types of dependences. Tiny now can automatically convert assignment statements to semantically equivalent update statements if it is started with `-r` command line option.

For example, assignment `s = s + a(i,j) + 2` or even `s = a(i,j) + s + 2` is converted by Tiny to `s += a(i,j) + 2`.

4.6 DOANY loop

This type of loop was suggested by Michael Wolfe in the paper "Doany: Not Just Another Parallel Loop" published at Conference record of 5th Workshop on Languages and Compilers for Parallel Computers, Yale University, August 1992. It instructs computer to execute iterations of the loop in any sequential order but not simultaneously.

When performing parallelization of a loop (either by running menu command Browse.Restr.Par or while doing parallelization of all loops by running System.Auto.AutoParallel) Tiny makes loop to be DOANY if all dependences carried by it are reduction dependences.

DOANY loop is intimately related to *reduction* dependences (see below).

4.7 RETURN statement

RETURN statement is now allowed to appear at the end of the Tiny program. It may contain expression which is ignored.

`f2t` assigns storage class OUT to Fortran function return variable, so assignments to it won't be eliminated by IVR dead statement eliminator.

4.8 Run-time dimensions and undeclared variables

Array dimensions can be run-time.

Undeclared variables are allowed to appear in the bounds of array being declared. They will be automatically declared with type `integer`. Warning (not error) message is issued on double definition of variable.

5 New analyses and transformations

5.1 Induction Variable Recognition (IVR)

A new IVR recognition technique proposed by Michael Wolfe at paper "Beyond Induction Variables" published at PLDI '92 conference proceedings is implemented. This technique is based on Static Single Assignment (SSA) Graph. Right now we recognize only basic induction variables, that is, variables which are affine functions of loop parameters.

5.2 Forward substitution of scalars

This technique is based on the SSA graph too. We substitute variable by the expression assigned to it if this expression is considered to have *known* value. The following things are considered to have known value:

- Constants,
- Loop parameters,
- Result of operation which arguments are known values,
- Value of variable assigned with known value.

It should be noted that array reference, even if all of its indexes are known values, is not a known value.

5.3 Elimination of dead scalar assignments

Assignment to scalar is considered to be dead if value assigned by it is not used later in the program.

Decision to eliminate assignment is taken using the SSA graph of program. If there are no edges going from strongly connected component (SCC) in reverse of SSA graph (in reverse graph edge goes from definition of value to its use) then this SCC can be eliminated.

By eliminating not just single statement but whole SCC we are able to eliminate mutually dependent but otherwise useless assignments.

5.4 Example of SSA graph–related techniques application

```
real a(100,100,100),b(100),c(200)
integer n, s, t, u
real c1
s = 100
t = 1
for k = 3 to 100 by 2 do
  c1 = c(k)
  a(k,s+10,t) = a(k,s,t)+c1
  b(s+10) = b(s)+c1
  s = s - 10
  t = t + 10
  c1 = c(k)
  u = t
endfor
for k = 0 to 100 by 1 do
  b(t) = b(s)+1
  s = s - 10
  t = t + 10
  c1 = b(t)
  u = s
endfor
```

is converted to the following using IVR, forward substitution and dead statement elimination:

```
auto real a(1:100,1:100,1:100), b(1:100), c(1:200)
auto integer n, s, t, u
auto real c1
for k = 3,100,2 do
  c1 = c(k)
  a(k,125-k*5,-14+k*5) = a(k,115-k*5,-14+k*5)+c1
  b(125-k*5) = b(115-k*5)+c1
endfor
for k = 0,100,1 do
  b(k*10+491) = b(-390-k*10)+1
endfor
```

5.5 Reduction dependences

This is 4th type of dependence in addition to three known types — flow, anti, and output dependences.

There is a *reduction* dependence between two instances of statements iff these two instances may be executed in any order but not simultaneously or in such a fashion that the semantics of reduction is preserved.

Reduction dependences may exist only between variable updates of the same type (that is, there is a reduction dependence between `s+=expr` and `s+=expr` but not between `s+=expr` and `s*=expr`). Variable update is done by an update statement (see above).

Another distinctive feature of reduction dependences is that they are not forward in time. It is so because reduction dependences don't impose any ordering on sequence of commutative and associative updates of the same memory cell. They rather indicate that instances of statements can not be executed in parallel but it's OK to execute them sequentially in any order (or to use special reduction operations hardware support present in many parallel/vector computers).

```
real a(100,100)
real s

S4:  s = 2
     for i=1 to 100 by 2 do
     for j=3 to 100 by 3 do
```

```
S7:    s += a(i,j) + 2
S8:    s *= a(i,j)
S9:    s += a(j,i) - 1
     endfor
     endfor
S12: a(50,50) = s
```

In this example we have the following dependences coming from statement S7 (with Omega-2 turned off). As you may note, reduction dependence between S7 and S9 (or between S7 and S7) has `(*,*)` direction vector which is not valid for time-directed dependences.

```
flow     7: s               -->  12: s
reduce   7: s               -->   7: s              (*,*)
output   7: s               -->   8: s              (0,0+)
output   7: s               -->   8: s              (+,*)
flow     7: s               -->   8: s              (+,*)
flow     7: s               -->   8: s              (0,0+)
anti     7: s               -->   8: s              (0,0+)
anti     7: s               -->   8: s              (+,*)
reduce   7: s               -->   9: s              (*,*)
```

Even though we say that reduction dependence is *between* statements S7 and S9, it is shown in Tiny as going both from S7 to S9 and from S9 to S7. Reduction dependence direction vector is the same for both directions.

5.6 Strip-mine loop transformation

This is done by running menu entry `Browse.Restr.Mine`. Since it is not reordering transformation, user is not asked any questions except for he/she is requested to type in strip-mine factor which should be positive integer constant.

Say, we have a loop

```
DO i = a, b, c
  ...
DOEND
```

After strip-mining it by a factor `f` we have the following two loops:

```
DO i# = a, b, c*f
  DO i = i#, MIN(i# + c*(f-1)), c
  ...
  DOEND
DOEND
```

5.7 Array and scalar renaming and expansion

This transformation allows us to kill anti and output (generic name — storage) dependences by renaming and expanding involved variables.

To kill storage dependence by expansion select menu entry `Browse.DD.Expnd` when dependence browser is on the dependence you want to kill. Please note that kill is not guaranteed because we use simplified algorithm for the expansion.

To run expansion for the whole program run menu entry `System.Auto.ExpanArr` or start Tiny with command line flag `-x`.

We use the following renaming and expansion algorithm: For each variable

1 Find connected components of dependence graph based on flow and reduction dependences only. For each connected component:

2 To kill a storage edge between two different components, rename one of them.

3 To kill a storage edge within a component:
Find a depth *d* such that all flow and reduction edges within the component has distance 0 at level *d* and the edge being killed has a non-zero distance.
Note: all references must share the same outer *d* loops.
Rename the component, and add an additional subscript to all references, containing the loop variable for level *d*.

Example. File `test/tiny/local/givens.t`:

```
real a(1:100,1:100)
integer n
real c,s,d,a1,a2
for i = 1 to n do
  for j = i+1 to n do
    a1 = a(i,i)
    a2 = a(j,i)
    d = sqrt(a1*a1+a2*a2)
    c = a1/d
    s = a2/d
    for k = i to n do
      a1 = a(i,k)
      a2 = a(j,k)
      a(i,k) = c * a1 + s*a2
      a(j,k) = -s * a1 + c*a2
    endfor
  endfor
endfor
```

After scalar and array renaming and expansion:

```
inout real a(1:100,1:100)
auto integer n
auto real c_e(1:n,i+1:n), s_e(1:n,i+1:n), d_e(1:n,i+1:n)
auto real a1_e(1:n,i+1:n), a1_r_e(i:n,1:n,i+1:n)
auto real a2_e(1:n,i+1:n), a2_r_e(i:n,1:n,i+1:n)
for i = 1,n do
  for j = i+1,n do
    a1_e(i,j) = a(i,i)
    a2_e(i,j) = a(j,i)
    d_e(i,j) = sqrt(a1_e(i,j)*a1_e(i,j)+a2_e(i,j)*a2_e(i,j))
    c_e(i,j) = a1_e(i,j)/d_e(i,j)
    s_e(i,j) = a2_e(i,j)/d_e(i,j)
    for k = i,n do
      a1_r_e(k,i,j) = a(i,k)
      a2_r_e(k,i,j) = a(j,k)
      a(i,k) = c_e(i,j)*a1_r_e(k,i,j)+s_e(i,j)*a2_r_e(k,i,j)
      a(j,k) = -s_e(i,j)*a1_r_e(k,i,j)+c_e(i,j)*a2_r_e(k,i,j)
    endfor
  endfor
endfor
```

5.8 Array privatization

The basic idea of this transformation is to make each processor of MIMD system own separate instance of a given variable thus eliminating communication and subscripts computation. Privatized variables have storage class PRIVATE.

To apply privatization to the whole program select menu entry `System.Auto.Privatize` or start Tiny with command line flag `-p`.

We use the following privatization algorithm: For each variable

1 Find connected components, just as in expansion. For each connected component:

2 For each component, let d be the level such that all flow and reduction dependences have distance 0 at this level.

3 Privatize the component at the level d. Put variable declaration inside the appropriate loop.

4 Remove any subscripts that are all identical (e.g., the variable is privatized in the `i` loop, and the second subscript is `i` in all references). Do not refer to unprivatized loop variables or any non-loop variables.

5 Repeat steps 2, 3, 4 until no new d can be found.

Example: after the privatization of previously expanded program:

```
inout real a(1:100,1:100)
auto integer n
for i = 1,n do
  for j = i+1,n do
    private real d_e_p, s_e_p, c_e_p, a2_e_p, a1_e_p
    a1_e_p = a(i,i)
    a2_e_p = a(j,i)
    d_e_p = sqrt(a1_e_p*a1_e_p+a2_e_p*a2_e_p)
    c_e_p = a1_e_p/d_e_p
    s_e_p = a2_e_p/d_e_p
    for k = i,n do
      private real a1_r_e_p, a2_r_e_p
      a1_r_e_p = a(i,k)
      a2_r_e_p = a(j,k)
      a(i,k) = c_e_p*a1_r_e_p+s_e_p*a2_r_e_p
      a(j,k) = -s_e_p*a1_r_e_p+c_e_p*a2_r_e_p
    endfor
  endfor
endfor
```

5.9 Data dependences from Entry and to Exit nodes

Entry node symbolizes the outer context of Tiny procedure which *defines* variables of COMMON, STATIC, FORMAL, IN, INOUT storage classes.

Exit node symbolizes the outer context which *uses* variables of COMMON, STATIC, FORMAL, OUT, INOUT classes.

Dependences involving one of these nodes have empty direction and distance vectors because these nodes are not surrounded by any loops.

Flag `-g` makes Tiny ignore dependences involving Entry and Exit nodes.

6 Unified system for performing transformations

This new transformation system is a major addition to the Tiny system. The system enables the user to optimize their program using a single universal transformation operation, rather than composing a number of different transformations. In this respect our system is very similar to the unimodular transformation framework, however our system is more general. The basic idea is to associate an affine schedule with each statement, that specifies the "time" at which each of their iterations should be executed. So a transformation is represented by a set of schedules (one for each statement). For more details please refer to:

> A Framework for Unifying reordering Transformations, Wayne Kelly and Bill Pugh, Technical Report CS-TR-2995.1, Department of Computer Science, University of Maryland

Our system generates a list of schedule sets (corresponding to transformations). This list only contains schedules corresponding to transformations which are "profitable" according to our simple performance estimators. The user is able to control the size of the search space that is considered in generating these schedules. This allows the user to balance the time spent finding schedules verses the likelyhood that the most profitable transformation will be found.

6.1 Using the system

The Unified transformation system is reached by choosing the "Unif" option in the Restructuring menu, which is reached by choosing the "Restr" option in the Browse menu. The user must select the outer level loop which they wish to transform before entering the Unified transformation system.

6.2 The Unified transformation control panel

The top half of the screen contains two windows for displaying sections of code. Initially the left window contains the section of code which is about to be transformed. The schedules corresponding to the sections of code are displayed immediately above the sections of code.

The middle section of the screen contains a window for viewing the list of schedules generated by the system. The schedules are listed one per row in a tabular fashion. The first column contains reference numbers which uniquely identify the schedules. The remaining columns contain the various performance estimates (simplicity, parallelism, locality and code length). Each of the performance estimate columns has a heading identifying it. Beneath each of these column headings there are displayed two cutoff values corresponding to their respective columns. These cutoff values can be modified by the user and control how many schedules will be generated. If column A has an absolute cutoff value of $a\%$, then only schedules which are within $a\%$ of the best schedule w.r.t column A will be displayed. An absolute cutoff value of "No Limit" is equivalent to a value of infinity. If a schedule is worse than some other schedule by more than the desirable cutoff value $d_i\%$ w.r.t. to all columns i that are "Applicable", then that schedule will not be displayed.

The absolute cutoff values can be toggled between "No Limit" and some limit by clicking the middle button of the mouse over the corresponding values. The absolute cutoff values can be incremented and decremented respectively by clicking the left and right buttons of the mouse over the corresponding values. The desirable cutoff values can be toggled between "Not Applicable" and applicable by clicking the middle button of the mouse over the corresponding values. The desirable cutoff values can be incremented and decremented respectively by clicking the left and right buttons of the mouse over the corresponding values.

The bottom section of the screen displays other user toggled options which affect the search space which is considered when generating schedules. The options correspond to various traditional transformations (loop reversal, loop fusion, loop skewing, statement duplication, loop alignment and access normalization). If an option is selected by the user (indicated by reverse video) then schedules corresponding to that traditional type of transformation will be considered. The user can toggle each of the options by clicking over the corresponding positions.

Selecting the "Edit" option will place the cursor in the section of the screen where the schedule of the left code is displayed. The user can then modify this schedule. All valid characters typed by the user are inserted into the schedule. The delete key, cursor keys and return key can also be used to edit the schedule. When the user hits the return key in the schedule of the last statement, the modified schedule is checked for legality. If it is legal, the corresponding code is produced and the normal mode of operation of the uniform menu is restored. Modulo and integer division can be included in schedules using the syntax "MOD(expr, int)" and "DIV(expr, int)".

Selecting the "Trans" option will cause the system to start generating schedules and displaying them in the middle section of the screen. When it has finished the message " Scheduling complete" will appear on the message line. The user is then able to browse the schedules that are displayed. The middle section of the screen can be scrolled vertically using the cursor keys if there are more schedules than can be displayed in the middle section of the screen. The code corresponding to a schedule can be viewed by clicking the mouse anywhere over the line displaying that schedule. This new code is displayed in the right hand window at the top of the screen. The code displayed in the top section of the screen can be scrolled vertically using the 'j' and 'k' keys and horizontally using the 'h' and 'l' keys. The schedules displayed on the screen can be sorted based on any of the performance metrics by clicking the mouse over the heading of the corresponding column.

The "Copy" option copies the code in the right hand window to the left hand window. The "Original" option copies the original code into the left hand window. The "Write" option write the code in the left hand window to a file specified by the user. The "Delete" option removes the schedule currently being viewed from the list of schedules. The “Select” option replaces the original section of code with the code in the left hand window and returns the user to the Restructure menu. At any time the user can return to the Restructuring menu using the normal “Xcape” feature and no transformation will have been applied.

6.3 Performance estimates

Our performance metrics are forced to be fairly simple because they are based on schedules rather than actual code.

- The simplicity metric is an estimate of how conplex the transformed code will be.
- Our estimate for parallelism is based on how many outer level loops can be run in parallel.
- The locality estimate is determined by considering only self reuse, i.e. we do not consider reuse between different references even if they are in the same statement. The estimates are based on whether the accesses are stride 0, stride 1 or other. We assume that there are enough iterations of the inner loop to flush the cache lines, so there is no temporal reuse across outer loops for stride 1 accesses. Our estimates are based on an arbitary cache line size of 8 “values” and we assume all loops have 40 iterations.
- The length metric is the size of the transformed code in characters. This metric is not displayed until the corresponding code is displayed.

7 Miscellaneous

- Menu entry `Browse.Optns` is added. When activated it displays how command line options are set.
- User profile file capability is added. That is, command line options are first read from file `.tinyrc` residing in user’s home directory. After that command line options are parsed.
- Error messages are clarified and improved. If Tiny is forced to exit/abort it says whether user or Tiny is to blame.
- Declarations which were in one statement in the source program, are displayed on one line (not one name per line as it was before).
- Number of subscripts in array reference is checked against declaration.
- Indentation is added to files saved by menu command `Write`.
- `Quit` menu command is made to work in every menu.
- Original Tiny was using its own memory allocation routine which tried to overcome deficiencies of `malloc/free` functions. Since now optimized versions of `malloc/free` are available, we use “pure” `malloc/free` for memory management.

8 *f2t* — Fortran-77 to Tiny converter

It is a hack of `f2c` (Fortran-77 to C converter) which, in turn, is a hack of ancient `f77` UNIX compiler. `f2t` is given name of Fortran program to convert:

```
f2t foo.f
```

If conversion is successful file `foo.t` will appear in your working directory. If you already had `foo.t` in your working directory it will be overwritten. Your favorite `*.t` files can be protected from overwriting by making them read-only.

If Tiny is given a Fortran source, it automatically translates it to the Tiny language by calling `f2t`. Since Tiny doesn't know path to `f2t`, it should be on your PATH.

It's not recommended to use `f2t` on files containing more that 1 Fortran program unit (procedure or function). First use `fsplit` to make each file contain exactly one program unit.

If you try to convert something which can not be expressed in Tiny, C code is likely to appear and to be rejected by Tiny. `f2t` is not designed to convert full Fortran-77 to Tiny. Moreover, it can not do it because of Tiny restrictiveness.

Here's a short list of things that are converted but are not likely to be accepted by Tiny:

- GOTO statements and any unstructured control statements. It may be recommended to use first some restructuring tool (like VAST, KAP, or even `struct`) to get rid of unstructured control.
- CHARACTER type and any manipulations with string values.
- RETURN statement in the middle of a program. In fact, RETURN statement is just special case of GOTO statement. So, if you have RETURN at the end of your SUBROUTINE, Tiny will accept it. But if you have it in the middle of a program, use third party restructurer first to make your program structured.
- COMPLEX things may cause a problem.

9 Syntax of Tiny

This is taken from file `src/tinyy.y` which is YACC parser for Tiny language.

Terminals are capitalized or are represented by input sequences in double quotes, non-terminals are lower case names.

```
Explanations for some of terminals:
TID             name
TINT            integer constant
TFLOAT          real constant
TEOS            ";" or "\n"  -- end of statement
TCOMMENT        comment starting with "!"


pgm:    stlist

stlist: stmt
|       stlist TEOS stmt

stmt:
|       stassert
|       stcomment
|       stasgn
|       stasgnoper
|       stconst
|       stfor
|       stforall
|       stif
|       stparfor
|       stdoany
|       stdecl
|       streturn

streturn:   "return"
|           "return" expr

stassert:   "assert" "(" expr ")"

stcomment:  TCOMMENT

stasgn:     lhs "=" expr

stasgnoper: lhs asgnoper expr

asgnoper:  "+="
|          "*="
|          "max="
|          "min="

lhs:       TID
|          TID "(" list ")"

stconst:   "const" constlist

constlist: constdecl
|          constlist "," constdecl

constdecl: newid "=" cexpr

stfor:     "do"  doid "=" expr forto expr optforby TEOS stmt
|          "do"  doid "=" expr forto expr optforby "{" stlist "}"
|          "for" doid "=" expr forto expr optforby "do" stlist "endfor"

optforby:
|          forby expr

stforall:  "forall" doid "=" expr forto expr optforby "do" stlist "endfor"
```

```
stparfor:  "doall"  doid "=" expr forto expr optforby "do" stlist "endfor"

stdoany:   "doany"  doid "=" expr forto expr optforby "do" stlist "endfor"

forto:  "to"
|       ","
|       ":"

forby:  "by"
|       ","
|       ":"

stif:   "if" expr "then" stlist elsepart "endif"

elsepart:
|       "else" stlist

stdecl: class type decllist

class:
|       "auto"
|       "common"
|       "static"
|       "formal"
|       "in"
|       "out"
|       "inout"
|       "private"
|       "builtin"

type:   "integer"
|       "real"

decllist: decl
|         decllist "," decl

decl:   newid init
|       newid "(" boundlist ")" init

init:
|       "=" expr

boundlist:
|       bound
|       boundlist "," bound
```

```
bound:  expr
|       expr ":" expr

newid:  TID

doid:   TID

cexpr:  TID
|       TINT
|       cexpr "+" cexpr
|       cexpr "-" cexpr
|       "-" cexpr
|       "+" cexpr
|       cexpr "*" cexpr
|       "(" cexpr ")"

oldid:  TID

expr:   oldid
|       oldid "(" list ")"
|       TINT
|       TFLOAT
|       "(" expr ")"
|       expr "+" expr
|       expr "-" expr
|       "+" expr
|       "-" expr
|       expr "*" expr
|       expr "/" expr
|       expr "**" expr
|       expr "<" expr
|       expr "<=" expr
|       expr "==" expr
|       expr "!=" expr
|       expr ">=" expr
|       expr ">"   expr
|       expr "mod" expr
|       expr "max" expr
|       expr "min" expr
|       TSQRT "(" expr ")"
|       "floor" "(" expr "/" expr ")"
|       "ceiling" "(" expr "/" expr ")"
|       assocop  "(" list ")"

assocop: "max"
```

```
|        "min"

list:   expr
|       list "," expr
```