Handling Code Displacements in the Debugging of Optimised Programs

William S. Shu

Department of Mathematics and Computer Science,
University of Buea, P. O. Box 63, Buea, CAMEROON.

November 1995

KEY WORDS: Interactive Source-level Debugging, Debugging of Optimised Programs, Path Metric Space, Distance Function, Optimisation.

ABSTRACT: Instruction code reordering is a common consequence of many optimisation techniques and must be masked out in the source-level debugging of optimised programs. To capture the notion of "distances along [specified] paths", a function Δ is defined and used. Δ mimics the usual mathematical distance function though it is applied in a "path metric space": it gives the distance between two points in a program but also the path along which it is measured. Δ is used to correct optimisation effects of code movements at debug-time.

RESUME: Changer l'ordre d'exécution des instructions est une conséquence fréquente de plusieurs techniques d'optimisation et ceci doit être rectifié lors de la mise au point, au niveau source, des programmes optimisés. Pour capter l'idée de "distances sur une trajectoire [spécifiée]", une fonction Δ est définie et utilisée. Δ ressemble à la fonction habituelle des distances métriques en mathématique mais elle est appliquée dans un "espace métrique des trajectoires": elle donne la distance entre deux positions dans un programme mais aussi le chemin sur lequel cette dernière est effectuée. Δ est utilisée pour corriger, lors du de la mise au point, les effets de déplacement des instructions par l'optimiseur.

1. INTRODUCTION

In the source-level debugging of unoptimised programs, there is a simple match between source and target codes. Thus, setting breakpoints or accessing variables at debug time is straightforward. The source-code is matched to its target code where the breakpoint is set. Similarly, symbol table information permits one to identify the location of a variable, which could then be assigned to or read. Thus, a variable would be in scope if it was defined in the block or procedure in which the program was stopped (*i.e.* at the current breakpoint).

In the presence of optimisation, this simple match between source and target code is lost. Instruction codes are moved around or replaced. New ones are inserted, and others deleted. Conventional debugging can take place only if one monitors and corrects for the effects of optimisation. One effect of optimisation is to move code, relative to each other, and so alter their expected execution sequence. I adapt a function termed a "path metric" to measure this displacement. A *path metric* is similar to a conventional mathematical metric but the [execution] path on which the distance function is measured must be identified as well.

The metric is then used to adapt a conventional debugger to one for optimised programs. Many applications — especially real-time and fault-tolerant ones — require debugging on the final, optimised version of a program. Besides being cheaper to adapt an existing debugger than to conceive new ones, using the metric provides a simple but sound basis on which code displacement issues are systematically handled or interpreted. Furthermore, the formal nature of this metric approach eases correctness proofs on such debuggers.

1.1. Other works

A number of works have dealt with issues arising from the source-level debugging of optimised programs. For instance, Hennessy [4] considered variable access. Zellweger [8] studied control-flow issues and how to mask out adverse optimisation effects. The DOC system of Coutant *et al.* [3] defined the ranges of optimisation effects based on address ranges of instructions. These ranges identified memory locations for values of variables to be displayed. Others, such as Brooks *et al.* [2] monitored the effects of optimisation for a user who would then decide on how best to debug his/her program. Meanwhile, Holzle *et al.* [5] preferred to "deoptimise" relevant code segments by incrementally [re-]compiling an unoptimised version at debug-time.

Shu [6,7] formally characterised the effects of optimisation on debugging. He held that formal reasoning on the issues involved would lead to greater understanding and better debugger design for a wide class of programs. Thus, the effects of optimisation on debugging were examined within an algebraic framework. This paper studies a formal characterisation based on metric distances which focuses on measuring and masking code movements.

2. USE OF PATH METRIC CHARACTERIZATION IN DEBUGGING

2.1. Mapping Program Instructions to Nodes

A given program may be seen as a graph whose nodes are its instructions, and whose arcs determine control flow. Instructions added, removed or replaced by optimisation are also seen as nodes. However, the actual instructions, which could even be program units, do not matter; one is interested only in the optimising transformations applied to them and how to undo their effects.

For expository clarity, I assume a model of compilation where naive code is generated and optimisation is then applied separately, though the principles discussed apply to other compilation strategies as well. Thus, in [Figure 1] program (a) is compiled to **b.1** and then successively optimised to (**b.2**) and (**b.3**). Optimisation transformations are assumed correct.

The model of the source-level debugger is a conventional one for unoptimised programs: allowing for minor adjustments, a table associates address [ranges] of target codes to successive source codes constructs. For instance, the current execution point of a program in the source code is that source code identified with the target code address from the table. In practice, source code addresses may be given in terms of line numbers, blocks and so on.

Control is exchanged between a running program and a debugger at breakpoints. Other debugger features, such as single-stepping, are viewed as special versions of breakpointing. For interactive source-level debugging, the breakpoints are in the executed code, but have to be expressed as positions in the source-code via the address table.

2.2. Measuring Displacements

Let M be the set of all nodes that could be used in a program and $G(\mathrm{M})$ is a graph describing the program obtained from nodes in M. Each node denotes a single instruction code as discussed above. Each instruction in the program is initially mapped to a unique node in M: multiple instances of the same instruction correspond to distinct nodes. Thus, $G(\mathrm{M})$ is initially a directed graph of the unoptimised program. A path on the graph $\langle P\rangle$ is *specified* in terms of nodes found on it. For instance, $\langle AB\rangle$ (or $\langle A, B\rangle$) describes a set of directed paths from A to B, possibly through intervening nodes. I term such a description a *path specification*. A path may be expressed in terms of alternative nodes, and a set of paths may be described by the same path specification.

A path is measured using a special distance function in a *path metric space*. See [Appendix A] for a full discussion on this. $(\mathrm{M},\Delta,\delta)$ defines a path metric space. For a path specification $\langle P\rangle$, Δ is a function that returns the pair (r, l). r is the "length" of $\langle P\rangle$ as measured by δ, a conventional distance function. l is the actual path used for the measurement of r, since $\langle P\rangle$ may

Figure 1 – Source, Naive and Optimised codes

```
   ...
S0:<start node>

S2: a = 5;
S3: if (c != x)
S4:     b = c+d;
S5: else
S6:     a = 5;
S7:     x = 23;
S8: e = a+b+c;
    ...
```

source code (a)

```
      ...
T00:<start node>

      ; *** S2 ***
T02:  mov   a,5

      ; *** S3 ***
T03:  mov   ax,c
T04:  cmp   ax,x
T05:  je    @L1:

      ; *** S4 ***
T06:  mov   ax,c
T07:  add   ax,d
T08:  mov   b,ax

      ; *** S5 ***
T09:  jmp   @L2

      ; *** S6 ***
      @L1:
T10:  mov   a,5

      ; *** S7 ***
T11:  mov   a,23
      @L2:

      ; *** S8 ***
T12:  mov   ax,a
T13:  add   ax,b
T14:  add   ax,c
T15:  mov   e,ax
      ...
```

naive target code (b.1)

```
      ...
T00:<start node>

      ; *** S2 ***
T10:  mov   a,5
T02:  mov   a,5

      ; *** S3 ***
T03:  mov   ax,c
T04:  cmp   ax,x
T05:  je    @L1:

      ; *** S4 ***
T06:  mov   ax,c
T07:  add   ax,d
T08:  mov   b,ax

      ; *** S5 ***
T09:  jmp   @L2

      ; *** S6 ***
      @L1:

      ; *** S7 ***
T11:  mov   a,23
      @L2:

      ; *** S8 ***
T12:  mov   ax,a
T13:  add   ax,b
T14:  add   ax,c
T15:  mov   e,ax
      ...
```

first optimisation (b.2)

```
      ...
T00:<start node>

      ; *** S2 ***
T10:
T02:  mov a,5

      ; *** S3 ***
T03:  mov ax,c
T04:  cmp ax,x
T05:  je @L1:

      ; *** S4 ***
T06:  mov ax,c
T07:  add ax,d
T08:  mov b,ax

      ; *** S5 ***
T09:  jmp @L2

      ; *** S6 ***
      @L1:

      ; *** S7 ***
T11:  mov a,23
      @L2:

      ; *** S8 ***
T12:  mov ax,a
T14:  add ax,c
T13:  add ax,b
T15:  mov e,ax
      ...
```

second optimisation (b.3)

describe more than one path. Let ⟨*P*⟩ be the path ⟨*AB*⟩ where *A* and *B* are positions of a node before and after optimisation. Then *r* defines the relative displacement of the node [on path *l*] due to optimisation. For the discussion following, *r* is the number of arcs (and hence instructions) linking two nodes, but *r* assumes a special value τ if there is no path between them. (*r*, *l*) is sometimes written as (Δ_r, Δ_l) to stress that they correspond to mathematical projections of Δ.

2.3. Handling Code reordering

Consider [Figure 1] where the C program fragment in (a) is compiled to an 8086 assembly code in (**b.1**). For expository reasons, instructions have been labelled and the names of source-level variables have been used directly in the assembly code. Also, `***  Sj  ***` shows where target code for source statement `Sj` begins.

Suppose a user wants to stop just before statement `S6` at debug-time. For the unoptimised code (**b.1**) this is at `T10`, or strictly speaking, just before `T10`. Expressing this in terms of the path metric, $\Delta_r(T00,T10) = 5$ when measured on the path ⟨*T00,T05,T11*⟩. After optimisation in (**b.2**), `T10` is just before `T02` and so $\Delta_r(T00,T10) = 1$ on the same path. Unfortunately, a conventional debugger would use the map in (**b.1**) — for which Δ_r is *5* — and hence stop at `T05`. Such a debugger is adapted for optimised programs by mapping its pre-optimisation values for Δ to matching post-optimisation ones. [Figure 2] illustrates this map. It also gives the special value of τ to `T10` when it is deleted. See also [section 2.5].

2.4. Choice and iterative constructs

Measuring code movements across choice constructs, such as if-statements, is straight-forward. However, one may also have to identify the particular choice point (construct) when masking the optimisation effect. For example, a breakpoint at `T10`, just before `T05`, in [Figure 1(b.2)] must not be honoured if the comparison in `S3` evaluates to true: one needs to know the choice point provoking this decision.

Code movements across loop boundaries pose similar problems as for choice constructs. Generally, code movements involving loops have the additional problem of identifying the particular loop iteration for which the code should execute. See also [Appendix A.4].

2.5. Mapping target and source codes

Figure 2 – Optimisation and changing path metric distances

DISTANCE		PRE-Optim.	POST-Optimisation	
Source	**Target**	**(b.1)**	**(b.2)**	**(b.3)**
S0 → S8	T00 → T12	6	6	5
S0 → S6	T00 → T10	5	1	τ
S0 → S3	T00 → T03	2	3	2
	T00 → T05	4	5	4
S2 → S3	T02 → T03	1	2	1

When code is moved around, source statements no longer have contiguous ranges of target code. Identifying where a source statement is actually executed, or where it begins, in the target code becomes problematic. As has been pointed out (Zellweger [8], Shu [7]), it is desirable to select the target code range that executes the *key statement effect* of a statement. The key statement effect is core semantic effect of a statement; for an assignment statement it is the assignment operation, and for a GOTO statement it is the transfer of control.

Also, one has to keep track of the textual and semantic positions of code. For example, the textual position of `T10` in [Figure 1(b.2)] is just before `T11` but its semantic one is just before `T02`. In (b.3) `T10` is deleted. `T02` implements its semantics and hence determines where. The issues raised by source-target maps are quite complicated. It suffices to say that the path metric could be used in a measure of the distribution of target code "fragments" for a given source code. Hence, one may estimate where a key statement effect is located, perhaps by observing how the fragments are clustered, or assess how syntactic and semantic positions of statements may "migrate".

3. SOME IMPLEMENTATION ISSUES

In many uses of the path metric Δ, any path is all right if it uniquely determines the relative execution order of the desired nodes. Furthermore, measuring absolute distances and their attendant computational costs may be avoided if one is only interested in their relative execution order. Thus, a boolean-based relational algebra may be developed over Δ and used more conveniently in debugger implementations. For instance, one has comparisons, based on Δ, such as:

BEF$(A, B) \equiv A < B$. *i.e.* $A \leq B$ and $A \neq B$.
AFT$(A, B) \equiv B < A$.
NEU$(A, B) \equiv A = B$.

Also, code movements not perceived at source-code level may be ignored. For instance swapping `T13` and `T14` produces code movements totally within source statement `S7` which may thus be ignored.

Existing algorithms could be modified and used. Likewise, implementation structures naturally available in some optimisation techniques may be adapted. For instance, the notion of basic blocks (Aho *et al.* [1]), as opposed to individual target instructions, is used to a first approximation of the path metric in relational comparisons.

4. CONCLUSION

Many optimisation techniques reorder instruction codes and so a debugger for unoptimised programs operates on false execution sequences. Consequently, I developed a graph-based program model on which code displacements are measured in a path metric space [details in Appendix A]. The measurements are then used to monitor and mask out the effects of reordered codes. I then pointed out how the identification of textual and semantic positions of codes during source-target maps could be located using the metric.

An advantage of the path metric approach is that results can be obtained after approximate calculations, adding flexibility to interpretations of code displacements in it. Various interpretations of debug issues, besides source-target maps, could be built over the path metric or the associated relational algebra. Thus, though the paper focused on reordered execution sequences of instructions, the metric may also be used in optimisation-induced data/variable access (*i.e.* assignments and use) problems, including relocation of variables in memory.

5. REFERENCES

[1]. Aho, A. V., Sethi, R., and Ullman, J. D., Compilers: Principles, Techniques, and Tools, Addison-Wesley, Reading, Massachusetts, (1986).

[2]. Brooks, G., Hansen, G. J., and Simons, S., A new approach to debugging optimised code, ACM SIGPLAN'92 Conference on Programming Language Design and Implementation, San Francisco, California (June 1992), pp.1-11.

[3]. Coutant, D. S., Meloy, S., and Ruscetta, M., DOC: A practical approach to source-level debugging of globally optimised code, Proc. of the SIGPLAN'88 Conference on Programming Language Design and Implementation, Atlanta, Georgia, (Jun 1988), pp.125-134.

[4]. Hennessy, J., Symbolic debugging of optimised code, ACM Transactions on Programming Languages and Systems 4(3), (1982) pp.323-344.

[5]. Holzle, U., Chambers, C., and Ungar, D., Debugging optimised code with dynamic deoptimization, ACM SIGPLAN'92 Conference on Programming Language Design and Implementation, San Francisco, California, (June 1992), pp.32-43.

[6]. Shu, W. S., A Unified Approach to the Debugging of Optimised Programs, PhD Dissertation, Department of Computer Science, University of Nottingham, England, UK, (1989).

[7]. Shu, W. S., A new basis for debugging ... from optimised programs, in: Tchuente M. (ed.), "Proceedings of the 1st International Conference on Research in Computer Science, Yaoundé, Cameroon", INRIA, France, Oct 1992).

[8]. Zellweger, P. T., Interactions between high-level debugging and optimised code, PhD Dissertation, Computer Science Division - EECS, University of California, Berkeley, (1984).

APPENDIX A

A DEFINING THE PATH METRIC

Here I show how the path metric was developed, explaining the reasoning behind it. One wants to measure distances between nodes on a directed path and handle the special case where they have no common path — that is, when they are on a *null path*. Unfortunately, a conventional mathematical metric does not include information on the "path (or direction!) of measurement". As such, I adapt the conventional metric space to a *path metric space* so as to state my case formally; it identifies the paths along which conventional metric measurements are made. As one moves from one path to another, relational comparisons based on the metric measurements should be preserved or ensured. I examine acyclic paths then allow for cyclic graphs.

A.1. Distances *along* acyclic paths

A conventional mathematical metric is given in [Definition 1] below. To facilitate the discussion, the term path is understood to denote a single path or a path specification [Section 2.2]: the former is a special case of the latter. Also, the following notation is adopted:

For any $A, B, C, Y, Z \in$ M,

- $\langle P\rangle$ denotes a path, P.
- $\langle ABC\rangle$ or $\langle A, B, C\rangle$ denotes a path from A to C via B, in order.
- $\langle A{+}B{+}C\rangle$ denotes a path through the nodes A, B and C but in any order.
- $\delta(\langle ABC\rangle)$ stands for $\delta(A, B) + \delta(B, C)$.
- (Δ_r, Δ_l) correspond to the pair (r, l); they are, component-wise, projections of Δ.

Definition 1: A metric space is a pair (**M**, δ) where δ is a "distance function" defined on the set **M**, and satisfies the following for all $X, Y, Z \in$ **M**:

$\delta(X,Y) \in \{ r \in \mathbb{R} \mid r \geq 0\}$; $\mathbb{R}$ is the set of real numbers. (1.1)

$\delta(X,Y) = 0 \;\Leftrightarrow\; X = Y$ (1.2)

$\delta(X,Y) = \delta(Y,X)$ [symmetry] (1.3)

$\delta(X,Y) + \delta(Y,Z) \geq \delta(X,Z)$ [triangle inequality] (1.4)

I define a metric along acyclic paths on a directed graph in terms of the number of arcs linking the two nodes. This number depends on the actual path used. This leads to [Definition 2] below. Now, of the alternatives given by a path specification [*c.f.* Section 2.2] only one — which usually denotes the desired execution path — is selected. Any relevant assertions are made on nodes from the path, though the path specification must be precise enough. In [Definition 3], I select the shortest non-null path, if there is one, that contains the nodes specified. A shortest path is used because it gives a conceptually simple and consistent way (I think!) of implementing/visualising the metric. Besides, efficient algorithms exist for shortest path problems. However, as explained in [Section 3] assertions on paths are essentially relational and so the shortest path constraint may be relaxed.

Definition 2: Each arc in an execution path contributes a *link* — the unit of measure — to the distance between two nodes. If the distance, n, is obtained from links on a path, $\langle P\rangle$, we say n is measured on $\langle P\rangle$. Two nodes are *comparable* if there exists a path linking them *i.e.* they lie on a path. Otherwise, they are *incomparable* (or *independent*).

Let n be the number of links in a path, and τ be larger than any n. Define

$$\delta(A, B) = \begin{cases} n & \text{if the path } \langle AB\rangle \text{ is non-null.} \\ \tau & \text{otherwise} \end{cases}$$

Take τ to be $\upsilon{+}1$ where υ is the maximum number of nodes in the graph. Clearly, δ is a distance function: it is applied on straight line graphs.

Definition 3: Let $\langle P\rangle$ be a path specification. Then $\alpha(\langle P\rangle)$ is a function that returns a single path defined by, or containing $\langle P\rangle$. For our purposes, let $\alpha(\langle P\rangle)$ be a shortest path linking the points explicitly stated in $\langle P\rangle$.

A.2. Distances *over* acyclic paths

Suppose that for optimisation purposes I need a path specification containing `T00` and `T15` [Figure 1(b.1)]. Let the path $\langle T00,T06,T15\rangle$ be given and suppose it establishes the relative positions of `T00` and `T15`. At some later point in time I want to optimise `T10` as in [Figure 1(b.2)]. The path $\langle T00,T11,T15\rangle$ thus becomes the better choice. One should be able to switch over to the latter path without invalidating earlier decisions on `T00` and `T15`.

In effect, assertions over one acyclic path should carry over to other paths, if their premises are still valid. To this end, I define a metric Δ that takes into account the path along which it is measured. I call it a *path metric* — and from it define a path metric space — since it mimics a conventional mathematical metric [space]. Δ is given in [Definition 4] and it identifies valid distances (Eqns 2.1 – 2.2) as well as how they could be added or compared (Eqns 2.3, 2.4).

Definition 3: Let δ_e be δ measured on an acyclic path $\langle e\rangle$. Let $\varnothing_P$ be the null path and $\langle P\rangle = \alpha(\langle AB+CD\rangle)$. Define Δ to have the following properties and operations:

$$\Delta(A,B) \in \{(r,l) \mid r = \delta_l(A,B) \text{ and } l = \langle AB\rangle\} \qquad (2.1)$$
$$\forall(r,l),\ r = \tau \ \Leftrightarrow\ l = \varnothing_P \qquad (2.2)$$

$$\Delta(A,B) +_\Delta \Delta(C,D) = \begin{cases} ((\delta(A,B) + \delta(C,D)), \langle P\rangle) & \text{if } \varnothing_P \neq \langle P\rangle \\ (\tau, \varnothing_P) & \text{otherwise} \end{cases} \qquad \text{[sum rule] } (2.3)$$

$$\Delta(A,B) \geq_\Delta \Delta(C,D) \ \Leftrightarrow\ \delta_p(A,B) \geq \delta_p(C,D) \qquad \text{[partial order] } (2.4)$$

A.3. Defining a path metric space

I want to characterise nodes as points in a path metric space, (**M**, Δ,δ). The main motivation for (**M**, Δ, δ) is to be able to assess the position (or displacement) of an instruction code on an execution path. In this way, a debugger may be able to identify the correct code even after multiple displacements. Thus, things are very much like in a conventional metric, except that we take the precaution of identifying the paths used, or mapping prior measurements [or assumptions] onto the same path.

Definition 4: A path metric space is the tuple (**M**,Δ,δ) where $\delta = \{\delta_e \mid e \text{ are paths in the graph}\}$ and Δ is the path metric function. δ_e's are identically defined distance functions, but applied to separate paths in the graph. Let $A, B, C, D \in \mathbf{M}$. Let **P** be the set of paths on **M** and **R** the set of real numbers. Δ satisfies the following for all paths.

$$\Delta(A, B) \in \{(r, l) \mid r \in \mathbf{R},\ r \geq 0, \text{ and } l \in \mathbf{P}\ \} \qquad (3.1)$$
$$\Delta(A, B) = (0, l) \ \Leftrightarrow\ A \equiv B \qquad (3.2)$$
$$\Delta(A, B) = \begin{cases} (\delta_e(A,B), \langle AB\rangle) \text{ if } \alpha(\langle AB\rangle) = e \text{ and } e \neq \varnothing_P \\ (\tau, \varnothing_P) \qquad \text{otherwise} \end{cases} \qquad (3.3)$$

$$\Delta(A, B) +_\Delta \Delta(B, C) \geq_\Delta \Delta(A, C) \qquad (3.4)$$

Note that r depends on $\langle e \rangle$ and hence Δ_l since the value of r is obtained from measurements along $\langle e \rangle$. Also, from Eqn. 3.4 and the acyclic nature of G(M), relational comparisons are preserved across paths in a given G(M).

Proposition 1: (M,Δ,δ) is a path metric space, given δ and Δ in Definitions 3 and 4.

Proof: The δ_e's are identical to some distance function δ_0, as explained earlier. One has to show that Δ satisfies equations 3.1 to 3.4. Equations 3.1 to 3.3 are trivial to prove. Only Equation 5.4 needs proving.

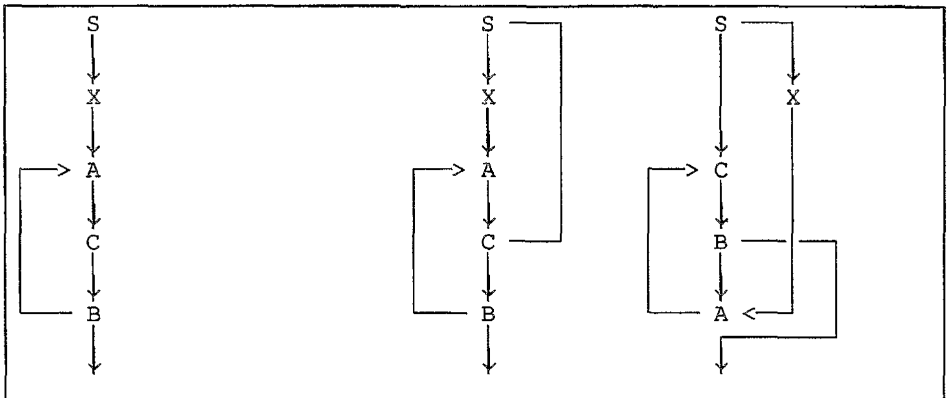
If $\alpha(\langle ABC \rangle) \neq \varnothing_P$ then the inequality holds, given that each r_i, $i = AB, BC, AC$ is obtained using δ, a metric function, on the path $\alpha(\langle ABC \rangle)$. If $\alpha(\langle ABC \rangle) = \varnothing_P$ then there is no path from A to C through B. Therefore, $\Delta(A,B)$ or $\Delta(B,C)$ or both are equal to $(\tau, \varnothing_P)$. From Δ-addition and given that $\tau \geq \delta(\langle AC \rangle)$, the inequality holds. []

A.4. Allowing for cyclic graphs

In a cyclic graph, the distance between two nodes, A and B, which lie on a cycle may be measured from A to B or from B to A. This is undesirable because the relative execution order of nodes is no longer unique. Thus, in [Figure 3(ii)], the distance $\langle AB \rangle$ is either 2 via $\langle ACB \rangle$ or 1 through $\langle BA \rangle$. Besides, paths may become infinite, through loop unravelling, and hence invalidate δ. Some restrictions thus become necessary, and their nature depends on whether the graph is a reducible flow graph or not.

A reducible flow graph (RFG), G, is one whose forward edges form an acyclic graph in which every node can be reached from the initial [*i.e.* start] node of G. Also, its back edges consist only of edges whose head dominates their tail; that is, all computations that reach each tail must first pass through its head (See Aho *et al.* [1]).

Figure 3 – (i) Reducible and (ii) Non-Reducible Flow-Graphs



Two valid paths between A and B
(i)

Two valid paths into loop from Start Node
(ii)

If measurements on two points, A and B, are such that A precedes B in *any* computational sequence from the start node (*i.e.* A dominates B), then Δ can be used on reducible flow graphs (RFGs). Clearly, since all nodes are reached from the start node via acyclic paths, a RFG may be perceived as an acyclic graph obtained by notionally severing loops at their back edges.

Unfortunately, in the presence of loops, relative code movements may become ambiguous or invalidated in a debugging context. Consider ? (i). In each iteration of the loop, the instruction, A, is executed before B. However, B is seen as executed before A *across* iterations (*i.e.* across back edges). This is the case if a debugger stopped the program at C but the user expects execution to reach A through $\langle CBA \rangle$. This may be corrected for in a debugger by knowing the "iteration number" in which the instructions are found. Such additional information from the computational procedure used is termed a *determinant* (*c.f.* Shu [7]). It may be seen as an attribute of the path.

Non-reducible flow graphs (NRFGs) typically contain multi-entry loop(s) [? (ii)]. The current loop entry may be used as a determinant in any measurements. Potentially, this implies n measurements for an n-entry loop! Fortunately, NRFGs are rare — even in unstructured programs. They are rarer still, because of structured programming. Besides, many conventional optimisations will not take place in their presence.