Generating Perfect Reversals of Simple Linear-Codes

GIT-CERCS-TR-03-04

May 20, 2003

Kalyan Perumalla
kalyan@cc.gatech.edu
College of Computing, Georgia Tech
Atlanta, GA 30332-0280

Abstract

Bi-directional execution – executing forward as well as in reverse – is useful in many contexts. However, traditional techniques for bi-directional execution are not scalable, as they require infinite storage in the presence of "destructive" assignments. We present a new approach that eliminates the scalability problem for bi-directional execution of a class of functions called linear codes*, which are sequences of assignments of arbitrary linear expressions to variables. Examples of linear codes include Fibonacci-like sequence generators, and operators such as shift, swap and rotate. We present an algorithm to generate perfect forward-reverse code pair from any given linear code, and show that any linear code can be perfectly inverted despite the presence of destructive assignments and apparent singularities in the input code. While existing techniques require memory size proportional to forward execution length, the code generated by our algorithm uses bounded amount of memory. The memory is proportional only to the number of variables in the given forward code, and is independent of both forward code size and forward execution length.*

1. Introduction

Many applications can benefit from the ability to execute forward as well as backwards. Examples include speculative execution, debugging systems, and optimistic parallel simulation. In such applications, reversal of the effects of a (forward) function/procedure is required at application-defined execution points.

Reversal of a function implies that all the variables in the original input function are restored exactly to the values they held just prior to forward function execution. Perfect reversal implies the memory required at runtime is bounded, and independent of forward/reverse execution length.

However, traditional techniques for enabling bi-directional execution are not scalable. Their inability to scale is partly due to their unbounded storage requirement in the presence of "destructive" assignments that induce information loss during unabated forward execution. Lack of scalability makes bi-directional execution either unappealing or impossible.

Admittedly, efficient reversal of functions in general is quite hard, and perfect reversal is even more difficult or impossible. However, reversibility in truth depends on the beholder's eye. While individual instructions might not be retraced backwards perfectly, some program fragments might be perfectly invertible when considered in aggregate. In other words, while individual instructions might not be easily invertible when considered in isolation, their combined effect could in fact be perfectly invertible. We explore this aggregate-view approach in this paper. We know that great potential exists for perfect reversal of aggregate code fragments, but we do not yet know how to automate it (e.g., tree insert/delete, enqueue/dequeue, etc.). This is a first step in advancing the automation process for perfect reversal or state minimization.

In this paper, we consider a simplified sub-problem. We focus on a class of functions called *linear* codes, which are sequences of assignments of arbitrary linear expressions to variables. It turns out that even the simplified problem of reversing an assignment-sequence (linear control flow) requires a non-trivial algorithm.

We present an approach that overcomes the scalability problem for linear codes. We show that any linear code can be perfectly inverted despite the

presence of destructive assignments and apparent singularities in the input code. We present a novel algorithm to generate perfect forward-reverse code pair from any given linear code. While existing techniques require memory size proportional to forward execution length, the code generated by our algorithm uses bounded amount of memory proportional to the number of variables in the given forward code, and is independent of both forward code size and length of forward execution. The number of assignments and the total computation in the generated reverse code is also bounded, being proportional to the number of variables, and independent of forward/reverse execution length.

We present the algorithm and its operation on illustrative linear codes, including Fibonacci-like sequence generators. Our algorithm can be readily incorporated into a compiler for automated identification, generation and optimization of perfectly reversible linear codes.

2. Background and Motivation

2.1. Related Work

Considerable amount of research has lately focused on theoretical aspects of bi-directional computation [1-4, 7-9, 11, 15-17]. Part of that work was triggered by the discovery of the relationship between reversible computations, quantum computing and power dissipation. Theoretical work exists (turning machine to simulate irreversible over reversible, etc.), but it has not yet been translated to practice [10, 12]. It is shown in [6] that reversible execution of an irreversible program in general necessarily involves overheads in terms of space and time. However, in practice, relatively little is known about actual runtime and memory efficiencies of bi-directional execution of conventional computer applications, such as those written using the C language. Reversibly and efficiently executing any arbitrary conventional application in general remains a challenging problem.

While work exists on reversing functional language programs (e.g., LISP [1]), relatively little is known on *perfectly* reversing procedural language programs. Here, we focus on programs written in procedural languages, such as the C language.

In [17], hardware support for reverse execution is provided at the level of program counter and memory address value logging. Although it is designed to support bi-directional execution of arbitrary programs, it does not actually employ true reverse execution operations, such an inverse operators. Instead, a memory address-value pair checkpointing mechanism is employed.

Re-execution approach is used in many other systems as well [13]. In fact, most "bi-directional execution" systems that we are aware of are based on checkpoint-and-re-execute paradigm. More recently, in [5], program counters are generated and saved efficiently to enable efficient rollback. Again, rollback is actually realized via re-execution, rather than reverse execution – execution is re-started from the most recently check-pointed state. Admittedly, these approaches are more general-purpose in nature, while our current work is not. However, our goal is to find efficient ways by which check-pointing can be minimized or avoided altogether.

A reverse execution capability is presented in [4], which is based on generating a log of inverse statements and interpreting them in reverse direction during reverse execution. This approach, again, entails memory overheads for even simple code fragments such as linear codes considered here.

It is worth re-iterating that perfect reversal capability is immensely useful in applications such as optimistic/speculative computing. This is especially true considering the relatively high cost of memory accesses as compared to computation (CPU) cost. Thus, while check-pointing-based approaches do get the (reversal) job done, it is important to find alternative means to minimize memory usage for check-pointing.

In [7] it is shown how perfectly reversible code can deliver high performance for efficient optimistic execution. Due to high speed of fine-grained discrete events in parallel simulation applications (such as ATM simulations), check-pointing methods are ill-suited as they accumulate a lot of memory very fast. True reverse execution helps to not only avoid memory overheads, but also to eliminate forward computation overhead for check-pointing. In such applications, it is desirable to employ a compiler-based approach to automatically generate prefect reverse code from user-written simulation models.

2.2. *Example: Fibonacci Sequence*

Consider the functions in Table 1 used to generate Fibonacci-like sequences ($X_n=X_{n-1}+X_{n-2}$):

Table 1: Functions for Fibonacci-like sequence generation. The variables a and b are initialized in init(). Thereafter, they can be updated one-step forward by calling f(), and one step backward by calling f^{-1}(). The init() and f() are supplied by the user. The f^{-1}() needs to be automatically generated by a compiler.

int **a**, **b**;
void **init**()
{
  a = …;
  b = …;
}
	void **f**()
	{
	  int c = a;
	  a = b;
	  b = b + c;
	}
		void **f^{-1}**()
		{
		
		  ?
		
		}

The variables a and b are initialized in init() to 0 and 1 respectively for Fibonacci sequence, or to 2 and 1 for Lucas numbers, and so on. Thereafter, f() is called multiple times to update a and b. Calling f() n times makes a and b move n steps forward in the sequence.

Suppose that, based on the code for f(), we would like to *automatically* generate a complementary function f^{-1}() to move backward in sequence by undoing the effects of calls to f(). In other words, f() modifies a and b, and f^{-1}() restores a and b to their original values. This would provide the capability to move forward as well as backward in the sequence.

In general, we would like an automated method (preferably incorporated into a compiler) to automatically generate inverse functions for such forward functions, so that code can be executed in forward or reverse directions at will.

2.3. *Motivation: Speculative Execution*

The need for (perfect) reversibility arises in the context of any speculative parallel execution system, such as optimistic parallel simulation. Figure 1 shows an example in which processor 1 executes ahead optimistically, invoking f() as a sequence generator (e.g., population model) as it goes along. If/when a remote message with a timestamp in processor 1's past is received, then processor 1 needs to restore its state to the point of inconsistency. This happens quite commonly, in protocols such as Time Warp. A similar problem also manifests itself in speculative processor execution of instruction blocks.

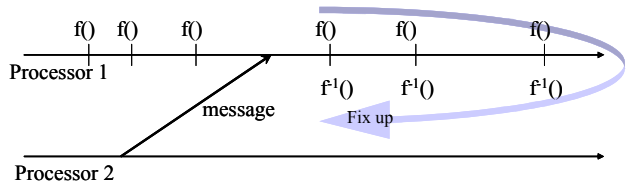


Figure 1: A distributed system example in which an incoming remote message uncovers inconsistency at processor 1 and initiates a fix up to undo erroneous computation.

3. Traditional Solution Approach

One way to automatically generate reverse code is to generate inverse statements corresponding to each forward statement. For example, an increment operation on a variable can be undone with its inverse, namely, a decrement operation. However, *destructive assignment* statements pose a major challenge in the inversion process, since they are not easily invertible. A destructive assignment is one in which a variable is overwritten with a new value from which the overwritten value cannot be easily recovered, when the assignment is considered in isolation.

Nevertheless, there exists an automated method by which any destructive assignment can in fact be undone, albeit, at the cost of additional memory space used to hold the overwritten value. A copy of the modified value is saved before modification during forward execution, and the saved value is restored during reverse execution. Since the saving and restoring operations follow a last-in-first-out (LIFO) discipline, a runtime stack is used to hold the saved values. In the forward code, an assignment of the form lvalue=rvalue, is made reversible by pushing a copy of the current value of lvalue onto a runtime stack. In the reverse code, the assignment is inverted by restoring lvalue to the saved value popped from top of the stack.

Table 2: Illustration of stack-based approach to reversal of the Fibonacci sequence generator.

Original Forward	Stack-based Reversal Approach	
	Forward	Reverse

void **f**()
{
int c = a;
a = b;
b = b + c;
}
	void **f**()
	{
	int c = a;
	PUSH(a);
	a = b;
	PUSH(b);
	b = b + c;
	}
		void **f**$^{-1}$()
		{
		POP(b);
		POP(a);
		}

The stack-based approach is easy to automate, and can be generated by a compiler, as demonstrated in [14]. Table 2 shows the stack-based approach used to generate the modified forward code, and its corresponding reverse code for the Fibonacci example. A PUSH(v) instruction pushes the value of v onto the top of the stack, while POP(v) pops the top value of the stack and assigns the popped value to v. Note in the example that the old value of the temporary variable c is irrelevant, and hence not saved.

3.1. Memory Scalability

A major drawback of the stack-based approach towards reversal is that the size of stack can become quite large, growing proportionally with the length of consecutive forward execution sequence. For example, in the Fibonacci sequence function, the size of the stack equals two integers times the number of consecutive calls to f(). This is because two values are pushed on to the stack for each f() invocation. The stack values cannot be discarded because they are needed for reversal in case $f^{-1}()$ is called any time in the future.

Clearly, an alternative approach is needed that eliminates the memory overhead of the stack-based approach. But, how can a compiler automatically detect an alternative perfect reversal solution in the presence of individual destructive statements? The solution lies in viewing the statements as a group rather than viewing them separately in isolation of each other. The group of statements must be analyzed to uncover the underlying relation among the modified variables, and then their inter-relationships should be exploited towards realizing perfect reversal.

4. Algorithm

4.1. Generalization: Linear Codes

We now consider a slightly more general problem, of which the Fibonacci sequence generator example is a special case. We consider any sequence of assignments of linear expressions to variables, which we call *linear codes*. In addition to sequence generators such as Fibonacci, linear codes encompass several common operations such as swap, circular/destructive shift, rotate, etc.

4.2. Rationale

The main idea behind our algorithm is that the linear sequence of assignments can be analyzed and its net effect can be represented as a matrix product operation. Let the old values of the variables be represented as a column vector V'. Let the operations of linear code assignments be represented as a matrix of constants W. Then, the matrix product WV' gives a new column vector corresponding the new values of the variables V= WV'. This indicates that we only need to multiply both sides by the inverse of W. The resulting equation $W^{-1}V= V'$ delivers the old values in terms of current values. Thus, all we would need to do to recover old values from current values is to apply the inverse of W. The meat of the algorithm is then concerned with addressing singularities, when W (as obtained from user-written code) is non-invertible.

4.3. Algorithm Outline

This algorithm is focused on a single code fragment of contiguous sequence of assignments. The algorithm consists of the following sequence of steps:

1. Preprocess forward code.
2. Obtain matrix representation.
3. Iteratively eliminate matrix singularity:
    a. Row elimination
    b. Column elimination.
4. Invert matrix.
5. Generate optimized reverse code.

4.4. Definitions and Notation

We will use L(…) to denote any linear expression of variables and constants.

For any variable v, denote by v' the value of that variable just prior to the first modification during an invocation by the forward function.

4.5. Input Function

Variables can be assigned multiple times or not at all. Local variables can also be used. However, no jump instructions, branch statements, loops or recursion are allowed – the function must consist of only a single sequence of assignment statements.

For any sequence of linear expressions, we generate equivalent forward and reverse code such that the reverse function exactly restores the values of variables changed by the forward function. This is achieved with memory space whose size is independent of the number of invocations (either consecutive or mixed) of the forward and reverse functions.

4.6. Preprocessing

Input: The input is a sequence of assignment statements, $F^0 \equiv \{v_i=L^0_k(1,v_1,\ldots,v_n)\}$, where n is the number of variables, and $1\le i\le n$, and $0\le k\le |F^0|$.

Note that each v_i could appear as the left hand side (LHS) of zero or more assignment statements, and hence the number of assignments can be larger or smaller than n.

Preprocess: Convert the input function to an equivalent function (with possibly greater or lesser number of lines), such that:

(a) Each variable appears as the LHS of exactly one assignment

(b) All right hand side (RHS) expressions are rewritten equivalently in terms of values held by each variable immediately prior to the first assignment statement in the function.

Algorithm:

1. Temporarily, treat local variables as global.
2. For each variable v_i, add the assignment $v_i=v'_i$ to the top of the function, i.e., $F^0 \leftarrow \{v_i=v'_i\} \cup F^0$.
3. For each assignment $v_i=L^0_k(1,v_1,\ldots,v_n)$, replace it with $v_i=L^0_k(1,X(v_1),\ldots,X(v_n))$, where X(v) is the RHS expression of the most recent assignment to v.
4. For each local variable v_i, delete all assignments to v_i (at this point, no global variable will have dependency on any local variable).
5. For each v_i, delete all except the last assignment to v_i.

Output: $F=\{v_i=L_i(1,v'_1,\ldots,v'_n)\}$, $1<=i<=n$.

Note that F^0 is a sequence, but F is a set (order of statements is important in F^0, but not important in F).

4.7. Matrix Representation

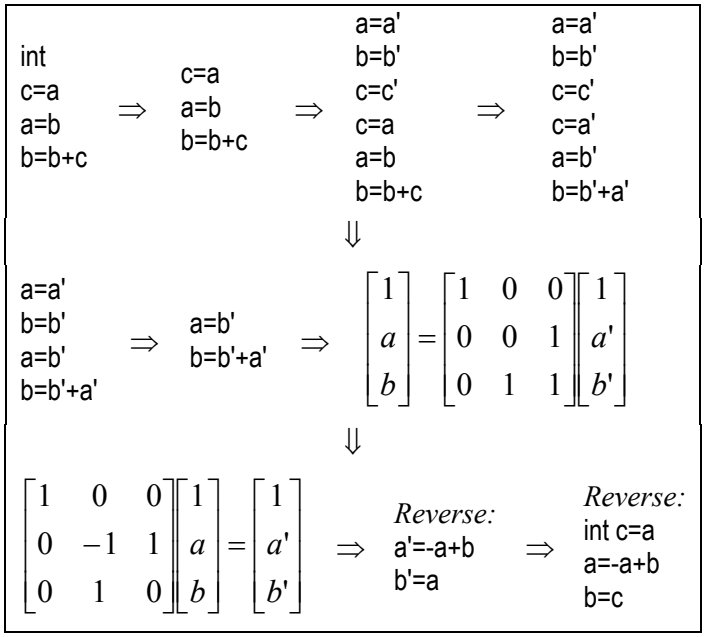
After preprocessing, let $L_i=w_{i0}+\Sigma^n_{j=1}w_{ij}v'_j$, $1\le i\le n$. Then F can be written as:

$$\begin{bmatrix} 1 \\ v_1 \\ \vdots \\ v_n \end{bmatrix} = \begin{bmatrix} 1 & 0 & \cdots & 0 \\ w_{10} & w_{11} & \cdots & w_{1n} \\ \vdots & \vdots & \cdots & \vdots \\ w_{n0} & w_{n0} & \cdots & w_{nn} \end{bmatrix} \begin{bmatrix} 1 \\ v'_1 \\ \vdots \\ v'_n \end{bmatrix}$$

We will represent it equivalently as V=WV'.

If W is non-singular, then it is easy to recover V' by multiplying both sides of the equation by the inverse matrix of W: $W^{-1}V=V'$.

Example: Consider the Fibonacci sequence generator f() given in Table 1. The following shows the series of transformations performed by each step of the algorithm:



Thus, the reverse code is generated to be the following:

void **f**()	void **f**$^{-1}$()
{	{
int c = a;	int c = a;
a = b;	a = -a + b;
b = b + c;	b = c;
}	}

5. Eliminating Singularity

When the matrix W is invertible, as in the preceding example, it is clear that it is straightforward to recover the old values V' of the variables V. But what if W turns out to be singular, with no inverse? We now address that case.

When W is singular, we know from linear algebra that at least one of the following two possibilities holds:

1. One of the rows can be expressed as a linear combination of the rest of the rows.
2. One of the columns can be expressed as a linear combination of the rest of the columns.

Consider the following simple linear code using two variables a and b:

Singularity Example:

Forward: a=a+b; b=2a;

After preprocessing: a=a'+b'; b=2a'+2b';

The resultant matrix relation is given by:

$$\begin{bmatrix}1&0&0\\0&1&1\\0&2&2\end{bmatrix}\begin{bmatrix}1\\a'\\b'\end{bmatrix}=\begin{bmatrix}1\\a\\b\end{bmatrix}$$

Clearly, the matrix is non-invertible, because of linear relationship between the last two rows (or columns). So, it is not possible to simply multiply the two sides of the equations by the matrix's inverse to get the reverse code. Does this imply reverse code doesn't exist for this forward code? No. In fact, the following is one of the possible reverse codes that can invert the given forward code:

Reverse: a'=a/3; b'=2a';

So, how do we obtain the inverse code for forward codes whose matrices cannot be inverted? The solution is to eliminate simple linear combination relationships among the variables or sub-expressions. In the preceding example, the RHS expressions of both the variables a and b are simply linearly related to each other, which makes the resultant matrix non-invertible.

We now address both the possibilities for singularity – dependencies across rows and columns.

Note: If there is a choice between row elimination and column elimination, it appears to be better to perform row elimination first, because (1) it is easier to perform (2) it does not add any new variables, thereby minimizing memory usage (3) the column dependency might in fact disappear after row elimination.

5.1. Row Elimination

Suppose the singularity can be attributed to linear dependency across rows. Let W be represented as a column matrix of rows R_i, $0 \le i \le n$, as follows:

$$W = \begin{bmatrix} R_0 \\ \vdots \\ R_n \end{bmatrix}$$

If W is singular due to linear dependency across rows, then any one of its rows can be expressed as

linear combination of the rest of the rows. Except for the first row R_0 (which represents the constant component of all L_i), pick any row R_r, $0<r\leq n$ such that $R_r=L^r(R_i, 0\leq i\leq n, i\neq r)$, for some linear function L_r on the rest of the row matrices R_i.

It is then clear that v_r can be expressed as a linear combination of all variables except itself. This permits us to eliminate v_r from the RHS of all variables (including v_r). Thus, F can be rewritten as: $F=\{v_i=L_i(1,v'_1,\ldots,L^r(1,v_j,\ 1\leq j\leq n, j\neq r),\ldots,v'_n)\},\ 1<=i<=n$. In other words, v_r is replaced by its equivalent expression that does not contain v_r or v'_r.

The resultant new matrix becomes:

$$\begin{bmatrix} 1 \\ v_1 \\ \vdots \\ v_r \\ \vdots \\ v_n \end{bmatrix} = \begin{bmatrix} 1 & 0 & \cdots & 0 & \cdots & 0 \\ w_{10} & w_{11} & \cdots & 0 & \cdots & w_{1n} \\ \vdots & \vdots & \cdots & \vdots & \cdots & \vdots \\ w_{r0} & w_{r1} & \cdots & 0 & \cdots & w_{rn} \\ \vdots & \vdots & \cdots & \vdots & \cdots & \vdots \\ w_{n0} & w_{n1} & \cdots & 0 & \cdots & w_{nn} \end{bmatrix} \begin{bmatrix} 1 \\ v'_1 \\ \vdots \\ v'_r \\ \vdots \\ v'_n \end{bmatrix}$$

Notice that the $(r+1)^{th}$ column becomes zero. Since v_r becomes a trivial variable that can always be easily recovered by evaluating its RHS expression on the rest of the variables, we now remove v_r from consideration for reversal. The $(r+1)^{th}$ row and $(r+1)^{th}$ column are removed from W, and v_r and v'_r are removed from V and V' respectively. During reverse code generation phase, a corresponding assignment statement is generated to recreate the v'_r based on the recovered values of the rest of the variables.

In the preceding singularity example, the second row can be expressed as two times the first row:

a=a'+b'; b=2a; $\Rightarrow$ b'=2a' $\Rightarrow$ a=a'+2a' $\Rightarrow$ a=3a'

$\Rightarrow$ a'=a/3; b'=2a'

Thus, the reverse code is given as specified previously:

Reverse: a'=a/3; b'=2a';.

5.2. Column Elimination

A more challenging case arises when one of the columns equals a linear combination of the rest of the columns. Column dependency arises due the existence of the same common sub-expressions in all rows. This is a slightly harder case, since we cannot simply eliminate any variables, but instead need to isolate sub-expressions that are common across rows.

Let W be represented as a row matrix of columns C_j, $0\leq j\leq n$, as follows:

$$W = \begin{bmatrix} C_0 & \cdots & C_n \end{bmatrix}$$

If W is singular due to linear dependency across columns, then any one of its columns can be expressed as linear combination of the rest of the columns. Pick any column C_c, $0<c\leq n$ such that $C_c=L^c(C_j, 0\leq j\leq n, j\neq c)$, for some linear function L^c on the rest of the column matrices C_j. Note that we avoid choosing the first column C_0 as C_c, since it corresponds to the constants in the linear expressions.

In the linear function L^c, consider only those columns, C_j, whose coefficients are non-zero. In other words, $C_c=\Sigma^{nc}_{k=1}x_kC_{jk}$, for some constants x_k and some column numbers j_k, $1\leq k\leq n_c$, and let denote c by j_0.

In the preceding singularity example, for illustration, let us apply column elimination instead of row elimination. Since a'+b' is the common sub-expression, we instantiate a new variable, c=a+b:

a=(a'+b'); b=2(a'+b'); $\Rightarrow$

a=(a'+b'); b=2(a'+b'); c=a+b; $\Rightarrow$

a=c'; b=2c'; c=a+b; $\Rightarrow$

a=c'; b=2c'; c=c'+2c'; $\Rightarrow$

a=c'; b=2c'; c=3c'; $\Rightarrow$

a'=c''; b'=2c''; c'=c/3; c''=c'/3;$\Rightarrow$

c'=c/3; a'=c/9; b'=2c/9;

Thus, an equivalent, alternative forward-reverse code pair is given as:

Initial: $a=a_0$; $b=b_0$; $c=a_0+b_0$;

Forward: a=c; b=2c; c=3c;

Reverse: a=c/9; b=2c/9; c=c/3;

Guards are, of course, needed to deal with boundary condition (for initial values).

Note the use of c'' in the derivation. Even though we can compute c', we still need to recover the user's original values of a' and b'. Since c'=a'+b' gives only one equation with two unknowns, we cannot recover a' and b' from c' alone. Computing c'' and re-evaluating a' and b' from c'' solves the problem.

Another Example:

Consider the following linear code with four variables, obtained after the preprocessing phase:

$$v_1 = w_{10} + w_{11}v'_1 + w_{12}v'_2 + w_{13}v'_3 + w_{14}v'_4$$
$$v_2 = w_{20} + w_{21}v'_1 + w_{22}v'_2 + w_{23}v'_3 + w_{24}v'_4$$
$$v_3 = w_{30} + w_{31}v'_1 + w_{32}v'_2 + w_{33}v'_3 + w_{34}v'_4$$
$$v_4 = w_{40} + w_{41}v'_1 + w_{42}v'_2 + w_{43}v'_3 + w_{44}v'_4$$

The corresponding matrix form V=WV' is as follows:

$$\begin{bmatrix} 1 \\ v_1 \\ v_2 \\ v_3 \\ v_4 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ w_{10} & w_{11} & w_{12} & w_{13} & w_{14} \\ w_{20} & w_{21} & w_{22} & w_{23} & w_{24} \\ w_{30} & w_{31} & w_{32} & w_{33} & w_{34} \\ w_{40} & w_{41} & w_{42} & w_{43} & w_{44} \end{bmatrix} \begin{bmatrix} 1 \\ v'_1 \\ v'_2 \\ v'_3 \\ v'_4 \end{bmatrix}$$

Suppose the matrix W is singular due to linear dependency among some of its columns, say among columns 1, 2 and 3: $C_1 = x_1C_2 + x_2C_3$, i.e.,

$$\begin{bmatrix} 0 \\ w_{11} \\ w_{21} \\ w_{31} \\ w_{41} \end{bmatrix} = x_1 \begin{bmatrix} 0 \\ w_{12} \\ w_{22} \\ w_{32} \\ w_{42} \end{bmatrix} + x_2 \begin{bmatrix} 0 \\ w_{13} \\ w_{23} \\ w_{33} \\ w_{43} \end{bmatrix}.$$

This implies that we can rewrite the RHS of each v_i by regrouping the sub-expressions differently:

$$v_1 = w_{10} + (x_1w_{12} + x_2w_{13})v'_1 + w_{12}v'_2 + w_{13}v'_3 + w_{14}v'_4$$
$$v_2 = w_{20} + (x_1w_{22} + x_2w_{23})v'_1 + w_{22}v'_2 + w_{23}v'_3 + w_{24}v'_4$$
$$v_3 = w_{30} + (x_1w_{32} + x_2w_{33})v'_1 + w_{32}v'_2 + w_{33}v'_3 + w_{34}v'_4$$
$$v_4 = w_{40} + (x_1w_{42} + x_2w_{43})v'_1 + w_{42}v'_2 + w_{43}v'_3 + w_{44}v'_4$$

The same can be rewritten with a different grouping of the variables as follows by grouping the columns reflecting the linear dependency of C_1, C_2 and C_3:

$$v_1 = w_{10} + w_{12}(x_1v'_1 + v'_2) + w_{13}(x_2v'_1 + v'_3) + w_{14}v'_4$$
$$v_2 = w_{20} + w_{22}(x_1v'_1 + v'_2) + w_{23}(x_2v'_1 + v'_3) + w_{24}v'_4$$
$$v_3 = w_{30} + w_{32}(x_1v'_1 + v'_2) + w_{33}(x_2v'_1 + v'_3) + w_{34}v'_4$$
$$v_4 = w_{40} + w_{42}(x_1v'_1 + v'_2) + w_{43}(x_2v'_1 + v'_3) + w_{44}v'_4$$

We now add n_c number of *new* variables, v_{n+1}... v_{n+nc}, into the representation, such that $v_{n+k} = x_kv_{j0} + v_{jk}$, $1 \le k \le n_c$.

In this example, $n_c = 2$, so we add $v_5 = x_1v_1 + v_2$, and $v_6 = x_2v_1 + v_3$. This implies $v'_5 = x_1v'_1 + v'_2$, and $v_6 = x_2v'_1 + v'_3$. We can now use the newly added variables to eliminate v'_1, v'_2 and v'_3 from the RHS of v_1, v_2 and v_3, by rewriting them in terms of v'_5 and v'_6.

$$v_1 = w_{10} + w_{14}v'_4 + w_{12}v'_5 + w_{13}v'_6$$
$$v_2 = w_{20} + w_{24}v'_4 + w_{22}v'_5 + w_{23}v'_6$$
$$v_3 = w_{30} + w_{34}v'_4 + w_{32}v'_5 + w_{33}v'_6$$
$$v_4 = w_{40} + w_{44}v'_4 + w_{42}v'_5 + w_{43}v'_6$$
$$v_5 = x_1v_1 + v_2$$
$$v_6 = x_2v_1 + v_3$$

We can now rewrite the same to eliminate v_1, v_2 and v_3 from the RHS of v_5 and v_6 by substituting them with their corresponding RHS's, giving the following:

$$v_1 = w_{10} + w_{14}v'_4 + w_{12}v'_5 + w_{13}v'_6$$
$$v_2 = w_{20} + w_{24}v'_4 + w_{22}v'_5 + w_{23}v'_6$$
$$v_3 = w_{30} + w_{34}v'_4 + w_{32}v'_5 + w_{33}v'_6$$
$$v_4 = w_{40} + w_{44}v'_4 + w_{42}v'_5 + w_{43}v'_6$$
$$v_5 = (x_1w_{10} + w_{20}) + (x_1w_{14} + w_{24})v'_4 + (x_1w_{12} + w_{22})v'_5 + (x_1w_{13} + w_{23})v'_6$$
$$v_6 = (x_2w_{10} + w_{20}) + (x_2w_{14} + w_{34})v'_4 + (x_2w_{12} + w_{32})v'_5 + (x_2w_{13} + w_{33})v'_6$$

In general, the RHS of all v_{jk}, $0 \le k \le n_c$, can be rewritten only in terms of v'_i, $1 \le i \le n+n_c$, and $i \ne j_k$, thereby eliminating v_{jk} from the RHS of *all* variables.

5.3. Termination

Each step of column elimination or row elimination removes at least one source of singularity due to the linear dependency across columns. Since at each iteration, the size of the matrix is reduced by at least one (one row reduced by row elimination, or one column reduced by column elimination), the algorithm is guaranteed to terminate.

5.4. Guards for Initial Values

Initial values of the variables can be easily protected by generating "guard" conditions in the code, such that reverse execution does not go backwards beyond the valid forward-reverse code relationship. Note that this might require protecting between 0 and *n* initial conditions, where *n* is the number of variables in the original user-specified forward code.

6. Fast-backwards

A natural and simple extension of the matrix approach is in reversing more than one step backwards at a time. While normal reverse execution mode might entail reversing one forward invocation at a time, it is also possible to efficiently jump s steps backwards in one step. Since $V = WV'$, $V = W^s V'^s$, $(W^s)^{-1} V = V'^s$.

In the Fibonacci generator example, we can jump s=2 steps backwards at a time, using the following deductions:

$$\begin{bmatrix} 1 \\ a \\ b \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 1 & 1 \end{bmatrix} \begin{bmatrix} 1 \\ a' \\ b' \end{bmatrix}$$

$$\Rightarrow \begin{bmatrix} 1 \\ a \\ b \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 1 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 1 & 1 \end{bmatrix} \begin{bmatrix} 1 \\ a'' \\ b'' \end{bmatrix}$$

$$\Rightarrow \begin{bmatrix} 1 \\ a \\ b \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 1 \\ 0 & 1 & 2 \end{bmatrix} \begin{bmatrix} 1 \\ a'' \\ b'' \end{bmatrix}$$

$$\Rightarrow \begin{bmatrix} 1 & 0 & 0 \\ 0 & 2 & -1 \\ 0 & -1 & 1 \end{bmatrix} \begin{bmatrix} 1 \\ a \\ b \end{bmatrix} = \begin{bmatrix} 1 \\ a'' \\ b'' \end{bmatrix}$$

$\Rightarrow$ a"=2a-b; b"= -a + b;

Thus, the reverse code to jump backwards two steps at a time, $f^{-2}()$, is given as.

void **f**()	void **f**$^{-1}$()	void **f**$^{-2}$()
{	{	{
int c = a;	int c = a;	int c = a;
a = b;	a = -a + b;	a = 2a - b;
b = b + c;	b = c;	b = -c+ b;
}	}	}

7. Conclusions and Future Work

We presented an algorithm that works around the destructive assignment problem for perfect reverse execution, thereby eliminating memory overheads even in infinite speculative execution. In view of the current gross speed differential between CPU versus memory systems, we believe that our perfect reverse execution is preferable to check-pointing based approaches. We are working on extending this approach further to more general program constructs. In general, programs make use of a plethora of operations which have perfect inverses (e.g., enqueue/dequeue, insert/delete, etc.) We are investigating automated detection and perfect reversal of the same, and combining them with the linear code algorithm presented here. Although we illustrated the algorithm with the Fibonacci example, it should be noted that the algorithm is valid on a wider set of input codes, namely, all linear codes, including those whose corresponding matrices are singular.

As another application domain, we are investigating the use of our linear code algorithm in speculative microprocessor execution. Since certain blocks of register instructions/operations can be expressed as linear codes, it should be possible to exploit their perfect reversibility on the fly.

8. References

[1] H. Baker, "NReversal of Fortune -- The Thermodynamics of Garbage Collection," presented at International Workshop on Memory Management, 1992.

[2] C. Bennet, "Thermodynamics of Computation," *International Journal of Physics*, vol. 21, pp. 905-940, 1982.

[3] P. Bishop, "Using Reversible Computing to Achieve Fail-Safety," presented at ISSRE-97, 1997.

[4] B. Biswas and R. Mall, "Reverse Execution of Programs," in *ACM SIGPLAN Notices*, vol. 34, 1999, pp. 61-69.

[5] B. Boothe, "Efficient Algorithms for Bidirectional Debugging," presented at Programming Language Design and Implementation, Vancouver, British Columbia, Canada, 2000.

[6] H. Buhrman, J. Tromp, and P. Vitanyi, "Time and Space Bounds for Reversible Simulation," *Journal of Physics A: Mathematical and General*, vol. 34, pp. 6821-6830, 2001.

[7] C. Carothers, K. S. Perumalla, and R. M.

Fujimoto, "Efficient Optimistic Parallel Simulations using Reverse Computation," *ACM Transactions on Modeling and Computer Simulation*, vol. 9, 1999.

[8] W. Chen and J. Udding, "Program Inversion: More than Fun!," *Science of Computer Programming*, vol. 15, pp. 1-13, 1990.

[9] D. Eppstein, "A Heuristic Approach to Program Inversion," presented at Internationl Joint Conference on Artificial Intelligence, 1985.

[10] E. Fredkin and T. Toffoli, "Conservative Logic," *International Journal of Theoretical Physics*, vol. 21, pp. 905-940, 1982.

[11] C. N. Ip and D. L. Dill, "State Reduction Using Reversible Rules," presented at 33rd Design Automation Conference, 1996.

[12] M. Li and P. Vitanyi, "Reversible Simulation of Irreversible Computation," presented at IEEE Conference on Computational Complexity (CCC), 1996.

[13] R. H. B. Netzer and M. H. Weaver, "Optimal Tracing and Incremental Re-execution for Debugging Long-Running Programs," presented at Programming Language Design and Implementation, 1994.

[14] K. S. Perumalla and R. M. Fujimoto, "Source Code Transformations for Efficient Reversibility," College of Computing, Georgia Institute of Technology, Atlanta, Technical Report GIT-CC-99-21, 1999/09/01 1999.

[15] B. J. Ross, "Running Programs Backwards: the Logical Inversion of Imperative Computation," Dept. of Computer Science, Brock University, St. Catharines, Ontario, Technical Report CS-94-03, 1994/01/01 1994.

[16] B. J. Ross, "Running Programs Backwards: the Logical Inversion of Imperative Computation," *Journal of Formal Aspects of Computing*, vol. 9, pp. 331-348, 1997.

[17] R. Sosic, "History Cache: Hardware Support for Reverse Execution," in *Computer Architecture News*, vol. 22, 1994, pp. 11-18.