Principled strength reduction

*Yanhong A. Liu**
Computer Science Department, Indiana University
201E Lindley Hall, Bloomington, IN 47405, U.S.A.
Email: liu@cs.indiana.edu, Tel: (812)855-4373, Fax: (812)855-4829

Abstract

This paper presents a principled approach for optimizing iterative (or recursive) programs. The approach formulates a loop body as a function f and a change operation $\oplus$, incrementalizes f with respect to $\oplus$, and adopts an incrementalized loop body to form a new loop that is more efficient. Three general optimizations are performed as part of the adoption; they systematically handle initializations, termination conditions, and final return values on exits of loops. These optimizations are either omitted, or done in implicit, limited, or *ad hoc* ways in previous methods.

The new approach generalizes classical loop optimization techniques, notably strength reduction, in optimizing compilers, and it unifies and systematizes various optimization strategies in transformational programming. Such principled strength reduction performs drastic program efficiency improvement via incrementalization and appreciably reduces code size via associated optimizations. We give examples where this approach can systematically produce strength-reduced programs while no previous method can.

Keywords

program optimization, incrementalization, strength reduction, caching intermediate results, maintaining auxiliary information, initialization, termination condition, loop optimization, iteration, recursion

1 INTRODUCTION

Strength reduction (Aho, Sethi & Ullman 1986, Allen 1969, Allen, Cocke & Kennedy 1981, Cocke & Kennedy 1977, Cocke & Schwartz 1970, Grau, Hill & Langmaac 1967, Gries 1971, Steffen, Knoop & Rüthing 1991) is a classical loop optimization technique in optimizing compilers. The idea is to replace certain operations in loop bodies by faster operations. For example, a multiplication involving an induction variable can sometimes be transformed into an addition. Finite differencing (Paige & Schwartz 1977, Paige 1983, Paige

*This work was supported by ONR under grant No. N00014-92-J-1973, NSF under grant No. CCR-9503319, and Indiana University under a junior-faculty start-up grant.

& Koenig 1982) generalizes strength reduction to languages with expressions composed of aggregate operations like set operations. Basically, a set of finite differencing rules are developed to transform aggregate operations in loop bodies into more efficient incremental operations. The essence of these optimizations is **incrementalization** of computations in loop bodies, *i.e.*, computing each iteration based on the result of the previous iteration. Such optimizations are crucial for performance.

This paper presents a principled approach for optimizing iterative (or recursive) programs, by explicitly formulating from them problems of incrementalization, using a general systematic approach to do the incrementalization, adopting the resulting incremental programs to form more efficient iterations (or recursions), and performing associated optimizations. This approach allows us to achieve optimizations such as strength reduction and finite differencing, as well as more drastic efficiency improvements. We call it **principled strength reduction**. There are at least two motivations for it: first, to achieve greater incrementality than allowed by a fixed set of strength-reduction or finite-differencing rules, and second, to provide a general and systematic approach following which new rules can be developed for both existing and new languages.

We have previously proposed a general systematic approach to incrementalization (Liu 1996, Liu, Stoller & Teitelbaum 1996). Given a program f and an input change operation $\oplus$, the approach aims to obtain an **incremental program** that computes $f(x \oplus y)$ efficiently by making use of $f(x)$ (Liu & Teitelbaum 1995b), the intermediate results computed in computing $f(x)$ (Liu & Teitelbaum 1995a), and auxiliary information about $f(x)$ that can be inexpensively maintained (Liu et al. 1996). Since every non-trivial computation proceeds by iteration (or recursion), the approach can be used for achieving efficient computation by computing each iteration using an appropriate incremental program. However, until now, a key issue was unaddressed: Given an iterative (or recursive) program, how exactly to determine the appropriate programs f and operations $\oplus$ on which to apply the incrementalization method, and how to adopt the resulting incremental programs to form new iterations (or recursions) that are more efficient?

This paper addresses this key open issue. The approach includes automatic transformations for formulating the incrementalization problem and for adopting the incremental programs. These transformations cleanly handle the initializations before iterations and the conditions for termination. To take full advantage of the incremental programs, three associated optimizations are performed: **folding initialization** and **replacing termination condition** are based on equality analysis; **minimizing maintained information** uses dependence analysis and is based on the idea that seemingly important values are not necessarily maintained in each iteration. Some of these optimizations are often seen as programmers' tricks, and they are particularly subtle and error-prone when done in *ad hoc* fashions. We, for the first time, unify and

systematize these optimizations, together with incrementalization, to achieve principled strength reduction. Once formulated around the idea of incrementalization, these optimizations and the overall principled strength reduction become simple and clear.

We give examples where our approach can systematically produce strength-reduced programs while no previous method can. These examples are in VLSI design, array processing, *etc.*

This paper is organized as follows. Section 2 prepares preliminaries. Section 3 presents a principled approach to speeding-up iterations using appropriate incremental programs. Section 4 describes optimizations in adopting incremental programs to form more efficient iterations. Section 5 discusses extensions and applications of the approach. Section 6 gives examples. Finally, Section 7 discusses related work and concludes.

2 PRELIMINARIES

Our incrementalization method (Liu 1996, Liu et al. 1996) has been described using a first-order, call-by-value functional programming language. The expressions of the language are given by the following grammar:

$e ::= v$	variable
$\mid\ c(e_1, ..., e_n)$	constructor application
$\mid\ p(e_1, ..., e_n)$	primitive function application
$\mid\ g(e_1, ..., e_n)$	function application
$\mid$ **if** e_1 **then** e_2 **else** e_3	conditional expression
$\mid$ **let** $v = e_1$ **in** e_2	binding expression

In particular, <> denotes a tuple constructor, and primitive functions $1st$, $2nd$, $3rd$, ... select the first, second, third, ... elements, respectively, of a tuple.

A program f is a set of mutually recursive function definitions of the form $g(v_1, ..., v_n) = e$ and a function f that is to be evaluated with some input $x = \langle x_1, ..., x_n \rangle$.* Figure 1 gives some example definitions.

An input change operation $\oplus$ to a program f combines an old input $x = \langle x_1, ..., x_n \rangle$ and a change $y = \langle y_1, ..., y_m \rangle$ to form a new input $x' = \langle x'_1, ..., x'_n \rangle = x \oplus y$, where each x'_i is some function of x_j's and y_k's. For example, an input change operation to the functions *cmp*, *odd*, *even*, *sum*, and *prod* of Figure 1 may be $x' = x \oplus_1 y = cons(y, x)$; an input change operation to the function *update* may be $x' = x \oplus_2 y = \langle n, m, i \rangle \oplus_2 \langle\,\rangle = \langle n, update(n, m, i), i - 1 \rangle$, even using *update* itself. Input change operation is an important notion. It describes how a new input to f differs from an old input, and thus affects how the new output can be computed efficiently using the old output. In

*Note that <> is a tuple constructor in the language, but $\langle\rangle$ is only a tuple notation used in the presentation of the paper.

$cmp(x)$: compare sum of odd and product of even positions

$cmp(x) = sum(odd(x)) \leq prod(even(x))$

$odd(x) =$ **if** $null(x)$ **then** nil **else** $cons(car(x), even(cdr(x)))$

$even(x) =$ **if** $null(x)$ **then** nil **else** $odd(cdr(x))$

$sum(x) =$ **if** $null(x)$ **then** 0 **else** $car(x) + sum(cdr(x))$

$prod(x) =$ **if** $null(x)$ **then** 1 **else** $car(x) * prod(cdr(x))$

$update(n, m, i)$: update m by $\pm 2^i$ according to $n - m^2$

$update(n, m, i) =$ **let** $p = n - m^2$ **in** **if** $p > 0$ **then** $m + 2^i$ **else if** $p < 0$ **then** $m - 2^i$ **else** m

Figure 1 Example function definitions.

an iterative (or recursive) program where the loop body is formulated as a function f, an input change operation should capture how the iterative (or recursive) computation proceeds so that we can determine how one iteration can be computed incrementally using the previous iteration.

We use an asymptotic cost model for measuring time complexity and write $t(f(v_1, ..., v_n))$ to denote the asymptotic time of computing $f(v_1, ..., v_n)$. Of course, maintaining additional information takes extra space. Our primary goal is to improve the asymptotic running time of the incremental computation. We attempt to save space by maintaining only information useful for achieving this.

Given a program f and an operation $\oplus$, we can use the approach in (Liu 1996, Liu et al. 1996) to derive (i) a program $\tilde{f}(x)$ that extends $f(x)$ to return also useful additional information about x, (ii) a program $\tilde{f}'(x, y, \tilde{r})$ that incrementally computes $\tilde{f}(x \oplus y)$ when $\tilde{r} = \tilde{f}(x)$, and (iii) a function Π that projects $f(x)$ out of $\tilde{f}(x)$ and projects $f(x \oplus y)$ out of $\tilde{f}'(x, y, \tilde{r})$.* For the function cmp in Figure 1 and operation $\oplus_1$, the intermediate results $sum(odd(x))$ and $prod(even(x))$ and auxiliary information $sum(even(x))$ and $prod(odd(x))$ are also returned, in components 2, 3, 4, 5, respectively, and the functions $\widetilde{cmp}$ and $\widetilde{cmp}'$ in Figure 2 and the projection function $\Pi = 1st$ are obtained.

Using the above functional language, we can directly simulate an imperative language with assignment, sequence, conditional, and loop statements, with functions and procedures, with basic data constructions, such as records, but without arrays or pointers. This paper considers iterative programs written in such an imperative language. Pieces of the programs that need to be incrementalized are translated into the above functional language. For simplicity, only structured programs are considered.

*While $f(x)$ abbreviates $f(x_1, ..., x_n)$, and $f(x \oplus y)$ abbreviates $f(\langle x_1, ..., x_n \rangle \oplus \langle y_1, ..., y_m \rangle)$, $f'(x, y, r)$ abbreviates $f'(x_1, ..., x_n, y_1, ..., y_m, r)$. Note that some of the parameters of f' may be dead and eliminated (Liu & Teitelbaum 1995b).

$cmp(x) = 1st(\widetilde{cmp}(x))$.
For x of length n,
  $\widetilde{cmp}(x)$ takes time $O(n)$;
  $cmp(x)$ takes time $O(n)$.

If $\widetilde{cmp}(x) = \tilde{r}$,
  then $\widetilde{cmp}'(y, \tilde{r}) = \widetilde{cmp}(cons(y, x))$.
For x of length n,
  $\widetilde{cmp}'(y, \tilde{r})$ takes time $O(1)$;
  $\widetilde{cmp}(cons(y, x))$ takes time $O(n)$.

$$
\begin{aligned}
\widetilde{cmp}(x) \quad = \quad & \textbf{let } v_1 = odd(x) \textbf{ in} \\
& \textbf{let } u_1 = sum(v_1) \textbf{ in} \\
& \textbf{let } v_2 = even(x) \textbf{ in} \\
& \textbf{let } u_2 = prod(v_2) \textbf{ in} \\
& < u_1 \leq u_2,\ u_1,\ u_2, \\
& \quad sum(v_2),\ prod(v_1) >
\end{aligned}
$$

$$
\begin{aligned}
\widetilde{cmp}'(y, \tilde{r}) = \quad & < y{+}4th(\tilde{r}) \leq 5th(\tilde{r}), \\
& \quad y{+}4th(\tilde{r}),\ 5th(\tilde{r}), \\
& \quad 2nd(\tilde{r}),\ y{*}3rd(\tilde{r}) >
\end{aligned}
$$

Figure 2 Resulting function definitions.

Example We use the derivation of an efficient binary integer square root algorithm for VLSI design (O'Leary, Leeser, Hickey & Aagaard 1994) as a running example. The initial specification of the algorithm is given in Figure 3(a). Given a binary integer n of l bits, where $n > 0$ (and l is usually 8, 16, ...), it computes the binary integer square root m of n using the non-restoring method (Flores 1963, O'Leary et al. 1994), which is exact for perfect squares and off by at most 1 for other integers. In hardware, multiplications and exponentials are much more expensive than additions and shifts (doublings or halvings), so the goal is to replace the former by the latter. We will obtain the program in Figure 3(b).

```
n := input; m := 2^(l-1);
for i := l - 2 downto 0 do
    p := n - m^2;
    if p > 0 then
        m := m + 2^i
    else if p < 0 then
        m := m - 2^i;
output(m)
```

(a)

```
p := input; v := 0; w := 2^(2*(l-1));
while w ≥ 1 do
    if p > 0 then
        p := p - v - w; v := v/2 + w; w := w/4
    else if p < 0 then
        p := p + v - w; v := v/2 - w; w := w/4
    else v := v/2; w := w/4;
output(v)
```

(b)

Figure 3 Non-restoring binary integer square root example.

3 A PRINCIPLED APPROACH

This section shows how to formulate a problem of incrementalization from an iterative program and how to use a derived incremental program to form a new iterative program. For simplicity, this section considers only single loops; extensions are discussed in Section 5.

3.1 Step 1: Formulating the incrementalization problem

Each loop consists of initialization, termination condition, and loop body, where loop body includes updates to the induction variables. To formulate the incrementalization problem, we regard computations in the loop body as forming a function f, and we regard the update to the input of f, including in particular the update to the induction variables, in each iteration as forming an input change operation $\oplus$. Note that the update to the input of f in each iteration is a result of update of the state by f itself.

Consider a loop statement. Let s be a tupling of the variables that are defined before the loop and are used in either the termination condition or the loop body. Then, any single loop can be transformed into a **while** loop of the form:

$$\begin{array}{ll} s := s_1; & --\text{initialization} \\ \textbf{while } c(s) \textbf{ do} & --\text{termination condition} \\ \quad s := b(s) & --\text{loop body} \end{array} \tag{1}$$

from which we directly formulate a function f and an input change operation $\oplus$:

$$f(x) = b(x) \quad \text{and} \quad x \oplus y = b(x) \tag{2}$$

Example Transforming the **for** loop into a **while** loop, the program in Figure 3(a) is transformed into:

```
[1]   n := input;
[2]   m := 2^(l-1);
[3]   i := l - 2;
[4]   while i ≥ 0 do
[5]       p := n - m^2;
[6]       if p > 0 then                                    (3)
[7]           m := m + 2^i
[8]       else if p < 0 then
[9]           m := m - 2^i;
[10]      i := i - 1;
[11]  output(m)
```

The part of the program between lines 1 and 10 is then transformed into a **while** loop of form (1):

$$\begin{array}{l} < n, m, i > \ := \ < input, 2^{l-1}, l-2 >; \\ \textbf{while } i \geq 0 \textbf{ do} \\ \quad < n, m, i > \ := \ < n, update(n, m, i), i-1 > \end{array} \tag{4}$$

where assignments to the tuple elements are parallel and function *update* is

as defined in Figure 1. From the loop body, we obtain the function f and operation $\oplus$:

$$f(n,m,i) = <n, update(n,m,i), i-1> \text{ and } \langle n,m,i\rangle \oplus \langle\,\rangle = \langle n, update(n,m,i), i-1\rangle \quad (5)$$

3.2 Step 2: Incrementalization

Given a functional program f and an input change operation $\oplus$, using the approach in (Liu 1996, Liu et al. 1996, Liu & Teitelbaum 1995a, Liu & Teitelbaum 1995b), we can derive an incremental program that computes $f(x \oplus y)$ incrementally by using the return value (Liu & Teitelbaum 1995b), the intermediate results (Liu & Teitelbaum 1995a), and certain auxiliary information (Liu et al. 1996) of $f(x)$, *i.e.*, we obtain a program $\tilde{f}$ that computes $f(x)$ and necessary additional information, a program $\tilde{f}'$ that incrementally computes $f(x \oplus y)$ and maintains the additional information, and a constant-time projection function Π that projects the value of f out of $\tilde{f}$ and $\tilde{f}'$. The derived programs $\tilde{f}$ and $\tilde{f}'$ and projection function Π satisfy: if $f(x) = r$, then

$$\Pi(\tilde{f}(x)) = r \quad \text{and} \quad t(\tilde{f}(x)) \leq t(f(x)) \quad (6)$$

and if $f(x \oplus y) = r'$ and $\tilde{f}(x) = \tilde{r}$, then

$$\Pi(\tilde{f}'(x,y,\tilde{r})) = r',\ \tilde{f}'(x,y,\tilde{r}) = \tilde{f}(x \oplus y),\ \text{and } t(\tilde{f}'(x,y,\tilde{r})) \leq t(f(x \oplus y)) \quad (7)$$

i.e., the programs $\tilde{f}$ and $\tilde{f}'$ preserve the semantics and compute asymptotically at least as fast as long as the original program terminates.

For the program f and operation $\oplus$ formulated in (2), the parameter y is not used, *i.e.*, $x \oplus y = x \oplus \langle\rangle$. In addition, the parameter x, if necessary, can always be included in the return value of $\tilde{f}(x)$. Therefore, the corresponding program $\tilde{f}'$ actually uses only the parameter $\tilde{r}$. We obtain specialized forms of (6) and (7): if $f(x) = r$, then

$$\Pi(\tilde{f}(x)) = r \quad \text{and} \quad t(\tilde{f}(x)) \leq t(f(x)) \quad (8)$$

and if $f(x \oplus \langle\rangle) = r'$ and $\tilde{f}(x) = \tilde{r}$, then

$$\Pi(\tilde{f}'(\tilde{r})) = r',\ \tilde{f}'(\tilde{r}) = \tilde{f}(x \oplus \langle\rangle),\ \text{and } t(\tilde{f}'(\tilde{r})) \leq t(f(x \oplus \langle\rangle)) \quad (9)$$

Incrementalization *per se* is not the subject of this paper. Thus, we explain the methods of (Liu 1996, Liu et al. 1996, Liu & Teitelbaum 1995a, Liu & Teitelbaum 1995b) only on the running example.

Example For the function f in (5), components 1 and 3 of its return value are trivially updated by the operation $\oplus$ in (5). Thus, we consider only the function *update* in component 2. Incrementalizing *update* under $\oplus$, as done in (Liu et al. 1996), we obtain the functions $\widetilde{update}$ and $\widetilde{update}'$, explained below, and the projection function $\Pi = 1st$.

$$
\begin{array}{l}
\widetilde{update}(n, m, i) = \ \textbf{let}\ p = n - m^2\ \textbf{in} \\
\qquad\qquad \textbf{if}\ p > 0\ \textbf{then} \\
\qquad\qquad\quad \textbf{let}\ u = 2^i\ \textbf{in}\ < m + u,\ p,\ u,\ 2{*}m{*}u,\ u^2 > \\
\qquad\qquad \textbf{else if}\ p < 0\ \textbf{then} \\
\qquad\qquad\quad \textbf{let}\ u = 2^i\ \textbf{in}\ < m - u,\ p,\ u,\ 2{*}m{*}u,\ u^2 > \\
\qquad\qquad \textbf{else}\ < m,\ 0 >
\end{array}
\tag{10}
$$

$$
\begin{array}{l}
\widetilde{update}'(\tilde{r_1}) = \ \textbf{if}\ \tilde{r_1}.p > 0\ \textbf{then} \\
\qquad\qquad \textbf{let}\ p = \tilde{r_1}.p - \tilde{r_1}.v - \tilde{r_1}.w\ \textbf{in} \\
\qquad\qquad \textbf{if}\ p > 0\ \textbf{then} \\
\qquad\qquad\quad \textbf{let}\ u = \tilde{r_1}.u/2\ \textbf{in}\ < \tilde{r_1}.m + u,\ p,\ u,\ \tilde{r_1}.v/2 + \tilde{r_1}.w,\ \tilde{r_1}.w/4 > \\
\qquad\qquad \textbf{else if}\ p < 0\ \textbf{then} \\
\qquad\qquad\quad \textbf{let}\ u = \tilde{r_1}.u/2\ \textbf{in}\ < \tilde{r_1}.m - u,\ p,\ u,\ \tilde{r_1}.v/2 + \tilde{r_1}.w,\ \tilde{r_1}.w/4 > \\
\qquad\qquad \textbf{else}\ < \tilde{r_1}.m,\ 0 > \\
\qquad\quad \textbf{else if}\ \tilde{r_1}.p < 0\ \textbf{then} \\
\qquad\qquad \textbf{let}\ p = \tilde{r_1}.p + \tilde{r_1}.v - \tilde{r_1}.w\ \textbf{in} \\
\qquad\qquad \textbf{if}\ p > 0\ \textbf{then} \\
\qquad\qquad\quad \textbf{let}\ u = \tilde{r_1}.u/2\ \textbf{in}\ < \tilde{r_1}.m + u,\ p,\ u,\ \tilde{r_1}.v/2 - \tilde{r_1}.w,\ \tilde{r_1}.w/4 > \\
\qquad\qquad \textbf{else if}\ p < 0\ \textbf{then} \\
\qquad\qquad\quad \textbf{let}\ u = \tilde{r_1}.u/2\ \textbf{in}\ < \tilde{r_1}.m - u,\ p,\ u,\ \tilde{r_1}.v/2 - \tilde{r_1}.w,\ \tilde{r_1}.w/4 > \\
\qquad\qquad \textbf{else}\ < \tilde{r_1}.m,\ 0 > \\
\qquad\quad \textbf{else}\ < \tilde{r_1}.m,\ 0 >
\end{array}
\tag{11}
$$

Function $\widetilde{update}(n, m, i)$ extends $update(n, m, i)$ to return also the intermediate results p and u and auxiliary information $2{*}m{*}u$ and u^2.* The intermediate results are directly computed in $update(n, m, i)$. The auxiliary information is obtained by unfolding $update(\langle n, m, i\rangle \oplus \langle\rangle)$ and analyzing the resulting expression. In particular, *update* computes an intermediate result $n - m^2$, and $\oplus$ updates m to be $m \pm u$ (+ or − depending on the condition in *update* in $\oplus$); thus, to compute $n - (m \pm u)^2$, which equals $n - m^2 \mp 2{*}m{*}u - u^2$, two expensive pieces of auxiliary information $2{*}m{*}u$ and u^2 are maintained, in additional to the intermediate result $n - m^2$. Of course the equality above depends on properties of $+$, $-$, and $*$. The analyses and transformations in our incrementalization methods (Liu 1996, Liu et al. 1996, Liu & Teitelbaum 1995a, Liu & Teitelbaum 1995b) first exploit all such intermediate results and auxiliary information and then prune out the ones not used in the incremental computation. For example, the value of m^2 is also an intermediate result, but it is not used separately, and thus is not maintained.

*Such additional values are returned only in branches where they are computed; in other branches, they could be denoted using a placeholder _, which can be safely eliminated when they are at the rightmost positions of a tuple (Liu et al. 1996, Liu & Teitelbaum 1995a).

Function $\widetilde{update}'(\tilde{r_1})$ uses the extended return value of $\widetilde{update}(n,m,i)$ to compute $\widetilde{update}(\langle n,m,i\rangle \oplus \langle\rangle)$. For readability, we name the five components of the return tuple as m, p, u, v, w, respectively, and use these names rather than the corresponding selectors $1st$, $2nd$, $3rd$, *etc.* Under each of the three different conditions on p in $update(n,m,i)$ in $\langle n,m,i\rangle \oplus \langle\rangle$, m is updated differently and is used to compute the new value of p; under each of the three different conditions on the new value of p, the new return tuple is maintained differently. For example, in the first branch of (11), where $\tilde{r_1}.p > 0$ and $p > 0$, the fourth component is $2*m'*u'$, where $m' = m+u$ and $u' = u/2$, and is transformed into $2*(m+u)*(u/2) = 2*m*u/2 + 2*u^2/2 = \tilde{r_1}.v/2 + \tilde{r_1}.w$.

Now, going back to f, we obtain

$$\begin{array}{l}\tilde{f}(n,m,i) = <n, \widetilde{update}(n,m,i), i-1>,\ \ \tilde{f}'(n,\tilde{r_1},i) = <n, \widetilde{update}'(\tilde{r_1}), i-1>, \text{ and} \\ \Pi(<n,\tilde{r_1},i>) = \ \ <n, 1st(\tilde{r_1}), i>.\end{array} \tag{12}$$

Obviously, if we use $\tilde{f}'$ instead of f to form a new loop body, then all multiplications and exponentials in the looping part are replaced by additions and shifts.

3.3 Step 3: Adopting the incrementalized program

With the derived programs $\tilde{f}$ and $\tilde{f}'$ and projection Π, we can obtain a new loop that computes its iteration using the incremental program $\tilde{f}'$. To use notations corresponding to those used in the original program (1), let

$$\tilde{b}(s) = \tilde{f}(s) \quad \text{and} \quad \tilde{b}'(\tilde{s}) = \tilde{f}'(\tilde{s}). \tag{13}$$

We transform the original program (1) into

```
s := s_1;
if c(s) then
  s̃ := b̃(s);
  while c(Π(s̃)) do
    s̃ := b̃'(s̃);
  s := Π(s̃)
```
(14)

where in the first "iteration", the extended function $\tilde{b}$ computes the additional information while computing the result of b.

Theorem 1 *If the program in (1) terminates in a state s, then the program in (14) terminates in the same state s and is asymptotically at least as fast.*

Proof: It follows from the equations in (2) and (13) and the correctness results about the functions $\tilde{f}$, $\tilde{f}'$, and Π in (8) and (9). □

To implement the assignments in (14), new variables are needed to store the additional information in $\tilde{s}$ over s. Moreover, since the assignments to $\tilde{s}$ are parallel, additional temporary variables may be needed to make the assignments sequential. A naive algorithm could create a temporary variable v_{tmp} for each v in $\tilde{s}$, assign all the new values of v's to v_{tmp}'s, and then copy all the values of v_{tmp}'s back to v's. Yet, the number of additional copy statements and the number of additional temporary variables can sometimes be reduced substantially by carefully ordering the assignment statements.

The problem of minimizing the number S of additional copy statements is NP-complete (Garey & Johnson 1979, Sethi 1973), but good heuristics exist (Sethi 1973). We can use an algorithm that easily decides an appropriate order of assignments when no additional temporary variables or copy statements are needed and reduces the problem to the problem of minimizing S otherwise. Basically, it builds a directed graph G representing the dependencies in the given assignments, solves the assignment problem for each of the strongly connected components (SCC) of G, and puts the resulting assignments for each of the SCCs together according to a topological order of the SCCs. Thus, if each SCC contains a single node, then no additional temporary variables or copy statements are needed, and a topological order of all the nodes determines an appropriate order of the sequential assignments.

Example For the running example, we have

$$\tilde{b}(n,m,i) = \ <n, \widetilde{update}(n,m,i), i-1> \quad \text{and} \quad \tilde{b}'(n,\tilde{r_1},i) = \ <n, \widetilde{update}'(\tilde{r_1}), i-1> . \quad (15)$$

The termination condition is still $i \geq 0$. We can obtain a new program of form (14) using these resulting functions.

The orders of the return components in the parallel assignments using $\tilde{b}$ and $\tilde{b}'$, respectively, are also topological orders according to the dependencies in the assignments, with the exception that, in the assignments using $\tilde{b}$, the first return component of $\widetilde{update}$ should be assigned last. Thus, using the algorithm above, assignments to all the components are sequentialized without using additional temporary variables or copy statements. We simply use p and u for the intermediate results in components 2 and 3, respectively, and v and w for the auxiliary information in components 4 and 5, respectively.

4 OPTIMIZING THE INCREMENTALIZED LOOP

This section discusses three associated optimizations that further improve the strength-reduced programs: folding the initialization, replacing the termination condition, and minimizing the maintained information. These optimizations benefit from the use of incrementalization.

4.1 Step 4: Folding the initialization

Additional information is often used by $\tilde{b}'$ for the incremental computation of a loop body. In (14), that information is initialized by $\tilde{b}$ before entering the loop. The code $\tilde{b}$ for initialization may often be saved by folding it into the loop body $\tilde{b}'$.

Given the initial state s_1, loop body b, and termination condition c as in (1), we look for a state s_0 such that $b(s_0) = s_1$ and $c(s_0) = \mathit{true}$. If b has an inverse b^{-1}, then a candidate state for s_0 is $b^{-1}(s_1)$. Note that s_0 is not necessarily unique and does not need to be. With such a state s_0, the original program (1) can be transformed into

$$\begin{array}{l} s := s_0; \\ \textbf{while } c(s) \textbf{ do} \\ \quad s := b(s) \end{array} \tag{16}$$

and thus the resulting program (14) can be transformed into

$$\begin{array}{l} \tilde{s} := \tilde{b}(s_0); \\ \textbf{while } c(\Pi(\tilde{s})) \textbf{ do} \\ \quad \tilde{s} := \tilde{b}'(\tilde{s}); \\ s := \Pi(\tilde{s}) \end{array} \tag{17}$$

It is easy to see that, in the resulting program (17), the part of the initial state $\tilde{s}$ that corresponds to s just equals s_1. Thus, the initialization in (17) needs to compute only the part of $\tilde{s}$ that corresponds to the additional information for s_1. In general, we look for a state $\tilde{s_1}$ that extends s_1 with appropriate additional information, *i.e.*, $\Pi(\tilde{s_1}) = s_1$ and, if $\tilde{b}(s_1) = \tilde{s}$, then $\tilde{b}'(\tilde{s_1}) = \tilde{s}$, and that can be initialized asymptotically as fast as s_1. Then, the computation of $\tilde{b}$ in (14) can be replaced with $\tilde{b}'$, which can be folded into the loop body of (14), yielding the simplified program:

$$\begin{array}{l} \tilde{s} := \tilde{s_1}; \\ \textbf{while } c(\Pi(\tilde{s})) \textbf{ do} \\ \quad \tilde{s} := \tilde{b}'(\tilde{s}); \\ s := \Pi(\tilde{s}) \end{array} \tag{18}$$

The correctness of program (17) follows from Theorem 1 and properties of s_0; the correctness of program (18) then follows from properties of s_1.

Because of their neat effect in reducing code size, such optimizations are often performed by programmers. Albeit seemingly simple, they are prone to errors when done in an *ad hoc* fashion. Our method can find a state $\tilde{s_1}$ by inspecting the original program (1) (finding s_0) and deducing the corresponding additional information for the incrementalized program (computing

$\tilde{b}(s_0)$), rather than directly inspecting the resulting program (14), which is often more complicated.

Similar optimizations for the preprocessing code are discussed by Cai and Paige (Cai & Paige 1988/89), whose programs are written in a very-high-level language based on sets. While they can obtain simplified programs by exploiting properties of sets, our optimizations apply also to programs written in lower level languages.

It is worth noting that folding the initialization may increase the absolute running time of a program, but by at most a constant amount. So the asymptotic running time here is not increased, but there can be a trade-off between the code size and the absolute running time of a program. To gain some possible speedup, an optimizing compiler may prefer to unfold the first iteration of program (18). However, the actual running time also depends on the machine cache behavior, which is partly affected by the code size. Also worth noting is that, for a particular machine, the initial values of certain variables may be the default values on the machine and their initialization can be saved. Our method is independent of particular machines.

Example For the running example, the initial state s_1 is $<n, m, i> \;=\; < input, 2^{l-1}, l-2>$. We find a preceding state

$$s_0 = \; < input, 0, l-1 >$$

such that $b(s_0) = s_1$ and $c(s_0) = true$. Computing the initial state $\tilde{s_1} = \tilde{b}(s_0)$, where $\tilde{b}$ is as in (15), we obtain

$$\tilde{s_1} = \; < input, <2^{l-1}, \underline{input}, 2^{l-1}, \underline{0}, \underline{2^{2*(l-1)}}>, l-1> . \tag{19}$$

We will see in the next two subsections that the components not underlined are redundant and will be eliminated by appropriate optimizations in Steps 5 and 6. Thus, in the final program in Figure 3(b), only the underlined components, which correspond to variables p, v, and w, are initialized.

This optimization would have saved a level in the 2-level proofs in (O'Leary et al. 1994). Also, while we obtain this optimization based on the original program (1), they discover the transformation from the resulting program (14), which is more complicated and harder to reason about.

4.2 Step 5: Replacing the termination condition

Certain values in the state s are maintained by a loop solely for testing the termination condition. This becomes unnecessary if the termination condition can be replaced by an equivalent test using other values, in particular, the

values maintained in the additional information in $\tilde{s}$. The new condition $\tilde{c}$ must satisfy

$$\tilde{c}(\tilde{s}) = c(s) \quad \text{and} \quad t(\tilde{c}(\tilde{s})) \leq t(c(s)). \tag{20}$$

Thus, such a condition $\tilde{c}$ can be obtained based on the original condition c and the relationship between $\tilde{s}$ and s, which is encoded in the function $\tilde{b}$.

For additional information in $\tilde{s}$ that is computed and maintained only in some but not all branches of $\tilde{b}$ and $\tilde{b}'$, if it can be inexpensively computed and maintained in those other branches as well, then we simply add the computation and maintenance of such information to those other branches. This allows such information to be used in the termination condition regardless of which branches are taken in any iterations.

Since the original termination condition often uses values of induction variables, this optimization generalizes the elimination of induction variables in classical strength reduction (Aho et al. 1986, Allen et al. 1981, Cocke & Kennedy 1977) to the elimination of all replaceable variables.

Example Consider the running example. It is obvious that the variable i is maintained only to test the termination condition, since i is used in the new loop body $\tilde{b}'$ in (15) only for updating itself.

Consider the additional information computed in the loop body, as shown in functions $\widetilde{update}$ in (10) and $\widetilde{update}'$ in (11). In particular, u, $2*m*u$, and u^2 are computed in the first two branches of every conditional but not in the third. Yet they can be inexpensively maintained in each of the third branches as well. We obtain the functions $\widetilde{update}_1$ and $\widetilde{update}_1'$ below:

$$\begin{aligned}
\widetilde{update}_1(n,m,i) = \ & \textbf{let } p = n - m^2 \textbf{ in} \\
& \textbf{if } p > 0 \textbf{ then} \\
& \quad \textbf{let } u = 2^i \textbf{ in } < m+u, p, u, 2*m*u, u^2 > \\
& \textbf{else if } p < 0 \textbf{ then} \\
& \quad \textbf{let } u = 2^i \textbf{ in } < m-u, p, u, 2*m*u, u^2 > \\
& \textbf{else let } u = 2^i \textbf{ in } < m, 0, u, 2*m*u, u^2 >
\end{aligned} \tag{21}$$

$$\begin{aligned}
\widetilde{update}_1'(\tilde{r_1}) = \ & \textbf{if } \tilde{r_1}.p > 0 \textbf{ then} \\
& \quad \textbf{let } p = \tilde{r_1}.p - \tilde{r_1}.v - \tilde{r_1}.w \textbf{ in} \\
& \quad \textbf{if } p > 0 \textbf{ then} \\
& \quad\quad \textbf{let } u = \tilde{r_1}.u/2 \textbf{ in } < \tilde{r_1}.m + u,\ p,\ u,\ \tilde{r_1}.v/2 + \tilde{r_1}.w,\ \tilde{r_1}.w/4 > \\
& \quad \textbf{else if } p < 0 \textbf{ then} \\
& \quad\quad \textbf{let } u = \tilde{r_1}.u/2 \textbf{ in } < \tilde{r_1}.m - u,\ p,\ u,\ \tilde{r_1}.v/2 + \tilde{r_1}.w,\ \tilde{r_1}.w/4 > \\
& \quad \textbf{else let } u = \tilde{r_1}.u/2 \textbf{ in } < \tilde{r_1}.m,\ 0,\ u,\ \tilde{r_1}.v/2 + \tilde{r_1}.w,\ \tilde{r_1}.w/4 > \\
& \textbf{else if } \tilde{r_1}.p < 0 \textbf{ then} \\
& \quad \textbf{let } p = \tilde{r_1}.p + \tilde{r_1}.v - \tilde{r_1}.w \textbf{ in} \\
& \quad \textbf{if } p > 0 \textbf{ then} \\
& \quad\quad \textbf{let } u = \tilde{r_1}.u/2 \textbf{ in } < \tilde{r_1}.m + u,\ p,\ u,\ \tilde{r_1}.v/2 - \tilde{r_1}.w,\ \tilde{r_1}.w/4 > \\
& \quad \textbf{else if } p < 0 \textbf{ then} \\
& \quad\quad \textbf{let } u = \tilde{r_1}.u/2 \textbf{ in } < \tilde{r_1}.m - u,\ p,\ u,\ \tilde{r_1}.v/2 - \tilde{r_1}.w,\ \tilde{r_1}.w/4 > \\
& \quad \textbf{else let } u = \tilde{r_1}.u/2 \textbf{ in } < \tilde{r_1}.m,\ 0,\ u,\ \tilde{r_1}.v/2 - \tilde{r_1}.w,\ \tilde{r_1}.w/4 > \\
& \textbf{else let } u = \tilde{r_1}.u/2 \textbf{ in } < \tilde{r_1}.m,\ 0,\ u,\ \tilde{r_1}.v/2,\ \tilde{r_1}.w/4 >
\end{aligned} \tag{22}$$

The termination condition tests $i' \geq 0$ after assignments of $u = 2^i$, $w = u^2 = 2^{2*i}$, and $i' = i - 1$ by the function $\tilde{b}$. We have

$$i' \geq 0 \quad \text{iff} \quad i \geq 1 \quad \text{iff} \quad u \geq 2 \quad \text{iff} \quad u^2 \geq 4 \quad \text{iff} \quad w \geq 4. \tag{23}$$

Therefore, the test on the induction variable i can be replaced by a test on u or w; the value of i becomes redundant and can be eliminated. We will see the value of u is not needed for any other purpose but the value of w is. Thus, in the final program in Figure 3(b), the test on w is used in the termination condition. Due to the folding done at the end of Step 6, which will be explained in the next subsection, the test is $w \geq 1$ instead of $w \geq 4$.

In the design work in (O'Leary et al. 1994), the termination condition is not handled formally, and elimination of the induction variable i is deferred to transformations in the hardware implementation. Our method yields a cleaner strength-reduced version that maps into hardware more directly.

4.3 Step 6: Minimizing the maintained information

The entire final state is not necessarily needed on exit of the loop. Using a backward dependence analysis, starting from the "output" or "export" statements of the original program, we can determine the part that is needed. We denote that part as s_{exit}. For the running example, s_{exit} is m.

The main observation is that, even though the value of s_{exit} must be available on exit of the loop, it is not necessarily needed in each iteration. If the value of s_{exit} can be retrieved, on exit of the loop, from other values, in particular, the values maintained in the additional information in $\tilde{s}$, and if maintaining these other values does not need s_{exit}, then the value of s_{exit} does not need to be maintained by the loop. The power of this novel optimization goes beyond any direct dependence analysis; it is enabled only because the addition information for the incrementalization is explicitly maintained by the loop and the relationship between such information and the state of the original program is explicitly recorded. Also note that this optimization does not move any code out of the loop; it removes the unnecessary code. As in the previous subsection, appropriate additional information computed only in some but not all branches is added to those other branches as well.

The optimization is composed of three sub-steps. We first analyze the program $\tilde{b}$ to trace how the value of s_{exit} can be retrieved from what pieces of additional information in $\tilde{s}$ without the need to maintain s_{exit}. Once such pieces s_{retr} of additional information are found, we analyze $\tilde{b}'$ to determine all the values that are needed in computing s_{retr}, using a backward dependence analysis. We then prune computations not needed for computing s_{retr}, including in particular the computation of s_{exit}, out of $\tilde{b}$ and $\tilde{b}'$, and adjust the projection function Π accordingly. By pruning in the backward dependency

graph, this optimization always produces programs that compute faster and use less space. This optimization can be surprisingly powerful, since seemingly important values can be completely eliminated from the loop.

In general, we can retrieve the final value from additional information maintained in the loop only if the loop body is executed at least once. So, a conditional is needed. However, if the loop is obtained after folding the initialization, then the additional information is appropriately initialized even if the loop body is not executed and, thus, we can omit the conditional. A last associated optimization is to fold the statement for retrieval into the loop body, if possible. This can be determined by testing the equality between the retrieved state and the state following the final state of the loop. In other words, such folding can be done if the final return value can be computed using one more iteration based on the maintained additional information.

The backward dependence analysis needed for this optimization can use the one developed in (Liu & Teitelbaum 1995a).

Example For the running example, the value that is needed on exit of the loop, *i.e.*, the value of m, is the first component computed by $\widetilde{update}_1$. Analyze (21). This value depends on (i) the value of p, (ii) the value of m in the previous computation, and (iii) the value of u. Since, right after the exit of the loop, we have the "opposite" conditions of (23), *i.e.*, $i' = -1$ iff $i = 0$ iff $u = 1$, we know that (ii) is retrievable from $v = 2 * m * u$ by $m = v/2$ and (iii) is 1. The retrieval is:

$$\textbf{if } p > 0 \textbf{ then } m := v/2 + 1 \textbf{ else if } p < 0 \textbf{ then } m := v/2 - 1 \textbf{ else } m := v/2 \tag{24}$$

Analyzing dependencies for the function $\widetilde{update}_1'$ in (22), we obtain all components in the maintained state that are needed to compute components p and v. They are components p, v, and w. Since the original return value in m, the first component, is not needed anymore, different branches induced by conditional tests on p can be merged. Pruning the functions $\widetilde{update}_1$ and $\widetilde{update}_1'$ yields:

$$\begin{aligned} \widetilde{update}_2(n, m, i) = \quad & \textbf{let } p = n - m^2 \textbf{ in} \\ & \textbf{let } u = 2^i \textbf{ in } < p, 2*m*u, u^2 > \end{aligned} \tag{25}$$

$$\begin{aligned} \widetilde{update}_2'(\tilde{r_2}) = \quad & \textbf{if } \tilde{r_2}.p > 0 \textbf{ then} \\ & \quad < \tilde{r_2}.p - \tilde{r_2}.v - \tilde{r_2}.w,\ \tilde{r_2}.v/2 + \tilde{r_2}.w,\ \tilde{r_2}.w/4 > \\ & \textbf{else if } \tilde{r_2}.p < 0 \textbf{ then} \\ & \quad < \tilde{r_2}.p + \tilde{r_2}.v - \tilde{r_2}.w,\ \tilde{r_2}.v/2 - \tilde{r_2}.w,\ \tilde{r_2}.w/4 > \\ & \textbf{else } < \tilde{r_2}.p,\ \tilde{r_2}.v/2,\ \tilde{r_2}.w/4 > \end{aligned} \tag{26}$$

The three components computed by $\widetilde{update}_2$ correspond to variables p, v, and w. In the final program in Figure 3(b), they are computed incrementally in the loop body as in the function $\widetilde{update}_2'$.

We fold the retrieval (24) into the loop body by finally retrieving the value of m from v and changing the termination condition from $w \geq 4$ to $w \geq 1$ (since w is updated by $w/4$, $w \geq 4$ equals $w/4 \geq 1$). We obtain the final program in Figure 3(b).

Similar optimization is needed in (O'Leary et al. 1994) but is done implicitly; it is never said explicitly or proved formally how the desired value is finally returned. This is another subtle and error-prone optimization.

5 EXTENSION AND DISCUSSION

So far, we have limited ourselves to considering only iterative programs with only single loops. This section discusses extensions and other issues about applying the approach.

5.1 Multiple iterations

In a structured program, multiple loops are either disjoint or nested. If the loops are disjoint, then we incrementalize each one separately. If the loops are nested, we handle the innermost loop first. Such an approach is based on the the assumption that, in common programming style, the inner iterations form a piece of computation whose states are updated more continuously following one another. Of course, given programs could be arbitrary. We may need to develop special program analysis to recognize such continuity and perform loop interchanging transformations, similar to those used for enhancing parallelism and data locality (Banerjee 1990, Wolf & Lam 1991a, Wolf & Lam 1991b). Section 6 gives examples with nested loops.

5.2 Recursive programs

For a recursive program, determining an input change operation that corresponds to how the recursion proceeds is not always simple.

If the recursion is linear, not necessarily being a tail recursion, then it can be handled in a similar way as iteration, since it straightforwardly corresponds to an iteration: the base case corresponds to the initialization; the condition that distinguishes base case from recursive case corresponds to the termination condition; and the recursive case corresponds to the loop body.

If the recursion is not linear, then there is no direct correspondence of it to an iteration. In fact, it is known that the **while** scheme is equivalent to flow chart, which is strictly less expressive than the recursive scheme (Greibach 1975). Simple heuristics exist for recognizing how recursions proceed on the argument, e.g., for an integer argument, a change operation may be $x' = x+1$; for a list argument, a change operation may be $x' = cons(y, x)$. Section 6 gives examples with non-linear recursions.

5.3 Applying the approach

Steps 1 and 3 can be fully automated. They are simply transformations between a simple functional language and an imperative language that uses the corresponding constructs. Of course, how to handle more complicated program constructs needs to be further studied. Steps 2 and 4-6 are systematic but are parameterized with equality reasonings and dependency analyses. These analyses may exploit various properties of the program constructs. Thus, their degrees of automation depend on the power we require from such analyses.

In general, the approach has a spectrum of applications. First, by limiting the analyses to use fully automatable techniques, it can be used in optimizing compilers. Second, through interacting with the user or a theorem prover, it can be used for semi-automatic transformational programming. Third, used off-line on paper, it supports a general methodology for systematic program efficiency improvement, which is one of the most important issues in program development and maintenance. A prototype implementation for semi-automatic use is under development (Liu 1995). To scale up the method for application to larger problems, we can select only expensive subcomputations in an iteration to be strength reduced.

5.4 How good is the resulting program

Even though it is not always possible to analyze the cost of executing an arbitrary program, it is important to guarantee that a transformed program P' is at least as efficient as the original program P. Our method guarantees that P' is asymptotically at least as fast as P. Further study is needed to guarantee that P' is in practice at least as fast as P or that P' is (asymptotically) faster than P. Further study is also needed on space efficiency. For the square root example, since, in hardware, multiplications and exponentials are much (asymptotically, in a sense) slower than additions and shifts, replacing the former by a few of the latter indeed results in a much faster program. Also, examining Figure 3, the original program (a) uses five units of space (n, m, l, i, p), but the new program (b) uses only four (p, v, w, l).

As with usual compiler optimization techniques, the effectiveness of our techniques depends on the original programs. Section 6 illustrates this. In any case, it is the incrementalization that can give the drastic speedup. The associated optimizations give only linear speedup, but they can save much space and appreciably reduce code size. For the square root example, incrementalization replaces multiplications and exponentials with additions and shifts; associated optimizations reduce the eight units of space (n, m, l, i, p, u, v, w) used in (14) to four (p, v, w, l) in (18), and reduce the size of code in (14), which uses (10) and (11), to (18), which uses (26).

6 EXAMPLES

This section gives more examples, including some with nested iterations and some with non-linear recursions. In particular, we show that different ways of writing the original programs result in different optimized programs. We also show that the general principles underlying our approach apply to programs that use arrays, though detailed analyses and transformations for arrays are worked out elsewhere (Liu & Stoller 1997).

6.1 Non-restoring binary integer square root

The running example is taken from VLSI circuit design (O'Leary et al. 1994), which transforms the original specification into a strength reduced version and further into a hardware implementation. The strength-reduced program was manually discovered and proved correct using Nuprl (Constable et al. 1986).

As discussed above, the optimizations used in (O'Leary et al. 1994) either incurred extra levels of proofs or were not handled formally. Another drawback is that there was no formal treatment of cost. As mentioned in (Liu et al. 1996), the final program in (O'Leary et al. 1994) contains an unnecessary shift.

We have showed through the presentation how our method is used to systematically derive a strength-reduced program, which automates and simplifies the VLSI circuit design process. Many similar programs, such as various versions of real/integer division/square-root algorithms (Dershowitz 1983), can also be derived using our method.

6.2 Minimum-sum section problem

This example is taken from (Gries 1984). Given an array $a[1..n]$ of numbers, where $n \geq 1$. A minimum-sum section of a is a non-empty sequence of adjacent elements whose sum is a minimum. A naive algorithm takes $O(n^3)$ time to compute such a minimum. Some ways of writing the loops enable easy improvement to $O(n^2)$, while others enable easy improvement to $O(n)$.

From the $O(n^3)$ time program on the left, we obtain the program on the right that takes $O(n^2)$ time:

```
min := a[1];
for i := 1 to n do
  for j := i to n do
    sum := 0;
    for k := i to j do
      sum := sum + a[k];
    min := min(min, sum)
```

```
min := a[1];
for i := 1 to n do
  sum1 := 0;
  for j := i to n do
    sum1 := sum1 + a[j];
    min := min(min, sum1)
```

(27)

The transformation proceeds as follow. First, consider the innermost loop L_k

only. Since each iteration of L_k adds a new number, L_k remains unchanged. Next, consider the middle loop L_j, which contains the loop L_k. Since each iteration of L_j first repeats computation in the previous iteration, L_j is incrementalized: the value of sum is maintained in a variable sum_1 after each iteration by adding only a new number; sum_1 is initialized to 0 (obtained with the optimization of folding initialization) before L_j starts; and the loop L_k is eliminated. Finally, consider the outermost loop L_i, which now contains the new middle loop L_j. Unfortunately, L_i can not be incrementalized since its iteration goes destructively, *i.e.*, as i increases, the interval from i to n decreases. This suggests that, if j loops from i down to 1 instead of to n, we may be able to incrementalize L_i. This is indeed the case.*

From the $O(n^3)$ time program on the left, which is the same as the one on the left of (27) except that j goes from i down to 1 instead of to n, we obtain the program on the right that takes $O(n)$ time:

```
min := a[1];
for i := 1 to n do
  for j := i downto 1 do
    sum := 0;
    for k := i to j do
      sum := sum + a[k]
    min := min(min, sum);
```

```
min := a[1];
for i := 1 to n do
  sum1 := 0;
  for j := i downto 1 do
    sum1 := sum1 + a[j];
    min := min(min, sum1)
```

```
min := a[1];
min1 := 0;
for i := 1 to n do
  min1 := min(min1 + a[i], a[i]);
  min := min(min, min1)
```

(28)

The transformations on the innermost and the middle loops are similar to those for (27), and we first obtain the program in the middle of (28), which is the same as the one on the right of (27) except that j goes from i to 1 instead of to n. Next, consider the outermost loop L_i, which now contains the new middle loop L_j. Using properties of min:

$$\begin{aligned} \min\{\Sigma_{k=j}^{i+1} a[k] \mid j = i+1..1\} &= \min\{\min\{\Sigma_{k=j}^{i+1} a[k] \mid j = i..1\},\ a[i+1]\} \\ &= \min\{\min\{\Sigma_{k=j}^{i} a[k] \mid j = i..1\} + a[i+1],\ a[i+1]\}, \end{aligned}$$

L_i is incrementalized: the value of min is maintained in a variable min_1 after each iteration using the above equation; min_1 is initialized to 0 (obtained with the optimization of folding initialization) before L_i starts; and the loop L_j is eliminated.

We have studied automatic techniques for incrementalizing array computations (Liu & Stoller 1997), and they can be applied in obtaining the program on the right of (27) from that on the left and obtaining the program in the middle of (28) from that on the left. To obtain the program on the right of (28) automatically, we need to extend our techniques to identify the functionality of min over a loop and use the equation above.

---

*We are mainly showing that different ways of writing the original programs end up in different optimized programs, not that how one way can be transformed into another that has a better optimized program. The latter is a problem that needs further study.

6.3 Fibonacci function and path sequence problem

The Fibonacci function fib is improved from an exponential time program to a linear time program $fib(x) = 1st(\widetilde{fib}(x))$ in (Liu & Teitelbaum 1995a), by considering fib as a function f, $x' = x + 1$ as an operation $\oplus$, and using the derived constant time function f' to form a new recursion:

$$\begin{array}{l} fib(x) = \textbf{if } x \leq 1 \textbf{ then } 1 \\ \qquad\qquad \textbf{else } fib(x-1) + fib(x-2) \end{array} \qquad \begin{array}{l} \widetilde{fib}(x) = \textbf{ if } x \leq 1 \textbf{ then } < 1 > \\ \qquad\qquad \textbf{else if } x = 2 \textbf{ then } < 2, 1 > \\ \qquad\qquad \textbf{else let } \tilde{r} = \widetilde{fib}(x-1) \textbf{ in} \\ \qquad\qquad\qquad < 1st(\tilde{r}) + 2nd(\tilde{r}),\ 1st(\tilde{r}) > \end{array} \tag{29}$$

The resulting program $\widetilde{fib}$ had an additional conditional that seemed unnecessary but it was not clear how it could be eliminated as a result of systematic procedure. Such elimination falls out of the optimizations in this paper. In particular, folding the initialization by maintaining an additional value 1 in the branch where $x \leq 1$ and folding the branch where $x = 2$ into the recursive case, we obtain a simpler program:

$$\begin{array}{l} \widetilde{fib}(x) = \quad \textbf{if } x \leq 1 \textbf{ then } < 1, 1 > \\ \qquad\qquad\quad \textbf{else let } \tilde{r} = \widetilde{fib}(x-1) \textbf{ in} \\ \qquad\qquad\qquad < 1st(\tilde{r}) + 2nd(\tilde{r}),\ 1st(\tilde{r}) > \end{array} \tag{30}$$

Bird's path sequence problem generalizes Dijkstra's longest up sequence problem and the longest common subsequence problem (Bird 1984). Given a directed acyclic graph, as a predicate arc, and a string l whose elements are vertices in the graph, the function llp below computes the length of the longest subsequence in l that forms a path in the graph (Bird 1984):

$$\begin{array}{l} llp(l) = \\ \textbf{if } null(l) \textbf{ then } 0 \\ \textbf{else } \max(llp(cdr(l)), 1 + g(car(l), cdr(l))) \end{array} \qquad \begin{array}{l} g(n, l) = \\ \textbf{if } null(l) \textbf{ then } 0 \\ \textbf{else if } \mathrm{arc}(n, car(l)) \textbf{ then} \\ \qquad \max(g(n, cdr(l)), 1 + g(car(l), cdr(l))) \\ \textbf{else } g(n, cdr(l)) \end{array} \tag{31}$$

This program is improved from exponential time to square time in (Liu et al. 1996) (and also incorrectly in (Bird 1984), but corrected in (Bird 1985)), and the resulting program is $llp(l) = 1st(\widetilde{llp}(l))$, where $\widetilde{llp}$ is

$$\begin{array}{l} \widetilde{llp}(l) = \\ \textbf{if } null(l) \textbf{ then } <0> \\ \textbf{else if } null(cdr(l)) \textbf{ then } <1, <0>> \\ \textbf{else let } \tilde{r} = \widetilde{llp}(cdr(l)) \textbf{ in} \\ \qquad \textbf{let } v_2 = \tilde{g}'(car(l), cdr(l), 2nd(\tilde{r})) \textbf{ in} \\ \qquad <\max(1st(\tilde{r}), 1 + 1st(v_2)),\ v_2> \end{array} \qquad \begin{array}{l} \tilde{g}'(i, l, \tilde{r_1}) = \\ \textbf{if } null(cdr(l)) \textbf{ then} \\ \qquad \textbf{if } \mathrm{arc}(i, car(l)) \textbf{ then } <1, <0>> \\ \qquad \textbf{else } <0, <0>> \\ \textbf{else let } v_1 = \tilde{g}'(i, cdr(l), 2nd(\tilde{r_1})) \textbf{ in} \\ \qquad \textbf{if } \mathrm{arc}(i, car(l)) \textbf{ then} \\ \qquad\qquad <\max(1st(v_1), 1 + 1st(\tilde{r_1})),\ \tilde{r_1}> \\ \qquad \textbf{else } <1st(v_1),\ \tilde{r_1}> \end{array} \tag{32}$$

Optimizing this resulting program by folding the initialization for $\widetilde{llp}$ and then $\tilde{g}$, *i.e.*, maintaining an additional value $<>$ in the branch where $null(l)$ is true and folding the branch where $null(cdr(l))$ is true into the recursive case, we obtain a simpler program:

$$
\begin{array}{l}
\widetilde{llp}(l) = \\
\textbf{if } null(l) \textbf{ then } <0, <>> \\
\textbf{else let } \tilde{r} = \widetilde{llp}(cdr(l)) \textbf{ in} \\
\quad \textbf{let } v_2 = \tilde{g}'(car(l), cdr(l), 2nd(\tilde{r})) \textbf{ in} \\
\quad <\max(1st(\tilde{r}), 1{+}1st(v_2)),\ v_2>
\end{array}
\qquad
\begin{array}{l}
\tilde{g}'(i, l, \tilde{r_1}) = \\
\textbf{if } null(l) \textbf{ then } <0, <>> \\
\textbf{else let } v_1 = \tilde{g}'(i, cdr(l), 2nd(\tilde{r_1})) \textbf{ in} \\
\quad \textbf{if } \mathrm{arc}(i, car(l)) \textbf{ then} \\
\qquad <\max(1st(v_1), 1{+}1st(\tilde{r_1})),\ \tilde{r_1}> \\
\quad \textbf{else } <1st(v_1),\ \tilde{r_1}>
\end{array}
\tag{33}
$$

7 RELATED WORK AND CONCLUSION

Strength reduction (Allen et al. 1981, Cocke & Kennedy 1977) is a classical compiler optimization technique that can be traced back to recursive address calculation for early ALGOL 60 compilers (Grau et al. 1967, Gries 1971). As discussed in (Steffen et al. 1991), it is syntactic (ignoring semantic equivalences between syntactically different terms), locally updating (thus not guaranteeing safety or speedup), and structurally restricted (only working on induction variables and region constants). **Composite hoisting-strength reduction** (Joshi & Dhamdhere 1982a, Joshi & Dhamdhere 1982b) is also syntactic and locally updating. Although it can handle more program terms, these terms are still of limited structures. Our method is general; it exploits program semantics to reduce computation strength, utilizes program analyses to guarantee correctness and efficiency, and is not limited to particular term structures. In fact, we can reduce the computation strength for a loop body as a whole. In particular, eliminating induction variables (Allen et al. 1981) is a special case of one of our optimizations.

Optimal code motion (Steffen, Knoop & Rüthing 1990) is a principled method for optimal placement of computations within a program with respect to the Herbrand interpretation. It is adopted for strength reduction by exploring the additional availability obtained from properties, such as distributivity, of numeric operators (Steffen et al. 1991), and it improves over conventional methods. Our method is also a principled approach, based on the idea of incrementalization. It exploits properties of more primitive operators, data structures, and conditionals, and thus is a more comprehensive exploration of availability. In fact, their method would not perform any strength reduction on the square root example (Knoop 1994). Of course, the complexity of our algorithm is larger. We plan to further study the complexity issues.

Inductively computable constructs in very-high-level languages (Fong 1977, Fong 1979, Fong & Ullman 1976) generalize conventional strength reduction and the elimination of induction variables to set-based languages. **Finite differencing** (Paige & Schwartz 1977, Paige 1983, Paige & Koenig 1982) and **fixed point recomputation** (Cai & Paige 1988/89) systematically

reduce strength of programs that use fixed point iteration and set-theoretic notations as the initial program specification. These techniques do not handle function abstractions, conditionals, or data types other than sets, as we do. In general, they apply only to programs written in very-high-level languages like SETL; our method applies also to lower-level languages.

Maintaining and strengthening loop invariants has been advocated by Dijkstra, Gries, and others (Dijkstra 1976, Gries 1981, Gries 1984, Reynolds 1981) for almost two decades as a standard strategy for developing loops. As discussed in a previous paper (Liu et al. 1996), its underlying principle is essentially incrementalization. But their work stresses mental tools for programming, rather than mechanical assistance, so no systematic procedures were proposed for automatic or semi-automatic uses.

Transforming recursive functions in CIP (Bauer, Möller, Partsch & Pepper 1989, Broy 1984, Partsch 1990) uses a collection of optimization strategies, including memoization, tabulation, relocation, precomputation, differencing, *etc.* They are essentially all subsumed by principled strength reduction, which is one method that is composed of step-by-step analyses and transformations, rather than a collection of strategies that needs to be applied by prudent judgment, and thus is more unified and more systematic. Other work on transformational programming for improving program efficiency, including the **extension technique** (Dershowitz 1983), the **promotion and accumulation strategies** (Bird 1984, Bird 1985), and finite differencing of functional programs in KIDS (Smith 1990), can also be further automated with principled strength reduction.

Principled strength reduction improves over previous approaches for program efficiency improvement. It systematically handles program constructs and operations that were not handled systematically before. Also, it systematically handles initializations and termination conditions, which are often particularly error-prone. Our three optimizations are either omitted, or done in implicit, limited, or *ad hoc* ways in previous methods. This unified approach also opens up a number of directions for further study. Its potential application is widespread.

REFERENCES

Aho, A. V., Sethi, R. & Ullman, J. D. (1986). *Compilers, Principles, Techniques, and Tools*, Addison-Wesley, Reading, Massachusetts.

Allen, F. E. (1969). Program optimization, *Annual Review of Automatic Programming*, Vol. 5, Pergamon Press, New York, pp. 239–307.

Allen, F. E., Cocke, J. & Kennedy, K. (1981). Reduction of operator strength, *in* S. S. Muchnick & N. D. Jones (eds), *Program Flow Analysis*, Prentice-Hall, Englewood Cliffs, New Jersey, chapter 3, pp. 79–101.

Banerjee, U. (1990). Unimodular transformations of double loops, *Proceedings of the Workshop on Advances in Languages and Compilers for Parallel Processing*, pp. 192–219.

Bauer, F. L., Möller, B., Partsch, H. & Pepper, P. (1989). Formal program construction by transformations—Computer-aided, intuition-guided programming, *IEEE Transactions on Software Engineering* **15**(2): 165–180.

Bird, R. S. (1984). The promotion and accumulation strategies in transformational programming, *ACM Transactions on Programming Languages and Systems* **6**(4): 487–504.

Bird, R. S. (1985). Addendum: The promotion and accumulation strategies in transformational programming, *ACM Transactions on Programming Languages and Systems* **7**(3): 490–492.

Broy, M. (1984). Algebraic methods for program construction: The project CIP, *in* P. Pepper (ed.), *Program Transformation and Programming Environments*, Springer-Verlag, Berlin, pp. 199–222.

Cai, J. & Paige, R. (1988/89). Program derivation by fixed point computation, *Science of Computer Programming* **11**: 197–261.

Cocke, J. & Kennedy, K. (1977). An algorithm for reduction of operator strength, *Communications of the ACM* **20**(11): 850–856.

Cocke, J. & Schwartz, J. T. (1970). Programming Languages and Their Compilers; Preliminary Notes, *Technical report*, Courant Institute of Mathematical Sciences, New York University.

Constable, R. L. et al. (1986). *Implementing Mathematics with the Nuprl Proof Development System*, Prentice-Hall, Englewood Cliffs, New Jersey.

Dershowitz, N. (1983). *The Evolution of Programs*, Vol. 5 of *Progress in Computer Science*, Birkhäuser, Boston.

Dijkstra, E. W. (1976). *A Discipline of Programming*, Prentice-Hall Series in Automatic Computation, Prentice-Hall, Englewood Cliffs, New Jersey.

Flores, I. (1963). *The Logic of Computer Arithmetic*, Prentice-Hall, Englewood Cliffs, New Jersey.

Fong, A. C. (1977). Generalized common subexpressions in very high level languages, *Conference Record of the 4th Annual ACM Symposium on POPL*, Los Angeles, California, pp. 48–57.

Fong, A. C. (1979). Inductively computable constructs in very high level languages, *Conference Record of the 6th Annual ACM Symposium on POPL*, San Antonio, Texas, pp. 21–28.

Fong, A. C. & Ullman, J. D. (1976). Inductive variables in very high level languages, *Conference Record of the 3rd Annual ACM Symposium on POPL*, Atlanta, Georgia, pp. 104–112.

Garey, M. R. & Johnson, D. S. (1979). *Computers and Intractability: A Guid to the Theory of NP-Completeness*, W. H. Freeman and Company, New York.

Grau, A. A., Hill, U. & Langmaac, H. (1967). *Translation of ALGOL 60*, Vol. 1 of *Handbook for automatic computation*, Springer, Berlin.

Greibach, S. A. (1975). *Theory of Program Structures: Schemes, Semantics, Verification*, Vol. 36 of *Lecture Notes in Computer Science*, Springer-Verlag, Berlin.

Gries, D. (1971). *Compiler Construction for Digital Computers*, John Wiley & Sons, New York.

Gries, D. (1981). *The Science of Programming*, Springer-Verlag, New York.

Gries, D. (1984). A note on a standard strategy for developing loop invariants and loops, *Science of Computer Programming* **2**: 207–214.

Joshi, S. M. & Dhamdhere, D. M. (1982a). A composite hoisting-strength reuction transformation for global program optimization—part I, *International Journal of Computer Mathematics* **11**: 21–41.

Joshi, S. M. & Dhamdhere, D. M. (1982b). A composite hoisting-strength reuction transformation for global program optimization—part II, *International Journal of Computer Mathematics* **11**: 111–126.

Knoop, J. (1994). Private comminication.

Liu, Y. A. (1995). CACHET: An interactive, incremental-attribution-based program transformation system for deriving incremental programs, *Proceedings of the 10th Knowledge-Based Software Engineering Conference*, IEEE Computer Society Press, Boston, Massachusetts, pp. 19–26.

Liu, Y. A. (1996). *Incremental Computation: A Semantics-Based Systematic Transformational Approach*, PhD thesis, Department of Computer Science, Cornell University, Ithaca, New York.

Liu, Y. A. & Stoller, S. D. (1997). Loop optimization for aggregate array computations, *Technical Report TR 477*, Computer Science Department, Indiana University, Bloomington, Indiana.

Liu, Y. A., Stoller, S. D. & Teitelbaum, T. (1996). Discovering auxiliary information for incremental computation, *Conference Record of the 23rd Annual ACM Symposium on POPL*, St. Petersburg Beach, Florida, pp. 157–170.

Liu, Y. A. & Teitelbaum, T. (1995a). Caching intermediate results for program improvement, *Proceedings of the ACM SIGPLAN Symposium on PEPM*, La Jolla, California, pp. 190–201.

Liu, Y. A. & Teitelbaum, T. (1995b). Systematic derivation of incremental programs, *Science of Computer Programming* **24**(1): 1–39.

O'Leary, J., Leeser, M., Hickey, J. & Aagaard, M. (1994). Non-restoring integer square root: A case study in design by principled optimization, *in* R. Kumar & T. Kropf (eds), *Proceedings of the 2nd International Conference on Theorem Provers in Circuit Design: Theory, Practice, and Experience*, Vol. 901 of *Lecture Notes in Computer Science*, Springer-Verlag, Berlin, pp. 52–71.

Paige, B. & Schwartz, J. T. (1977). Expression continuity and the formal dif-

ferentiation of algorithms, *Conference Record of the 4th Annual ACM Symposium on POPL*, pp. 58–71.

Paige, R. (1983). Transformational programming—Applications to algorithms and systems, *Conference Record of the 10th Annual ACM Symposium on POPL*, pp. 73–87.

Paige, R. & Koenig, S. (1982). Finite differencing of computable expressions, *ACM Transactions on Programming Languages and Systems* **4**(3): 402–454.

Partsch, H. A. (1990). *Specification and Transformation of Programs—A Formal Approach to Software Development*, Springer-Verlag, Berlin.

Reynolds, J. C. (1981). *The Craft of Programming*, Prentice-Hall, Englewood Cliffs, New Jersey.

Sethi, R. (1973). A note on implementing parallel assignment instructions, *Information Processing Letter* **2**: 91–95.

Smith, D. R. (1990). KIDS: A semiautomatic program development system, *IEEE Transactions on Software Engineering* **16**(9): 1024–1043.

Steffen, B., Knoop, J. & Rüthing, O. (1990). The value flow graph: A program representation for optimal program transformation, *Proceedings of the 3rd ESOP*, Vol. 432 of *Lecture Notes in Computer Science*, Springer-Verlag, Berlin, pp. 389–405.

Steffen, B., Knoop, J. & Rüthing, O. (1991). Efficient code motion and an adaption to strength reduction, *Proceedings of the 4th International Joint Conference on TAPSOFT*, Vol. 494 of *Lecture Notes in Computer Science*, Springer-Verlag, Berlin, pp. 394–415.

Wolf, M. & Lam, M. (1991a). A data locality optimizing algorithm, *Proceedings of the ACM SIGPLAN '91 Conference on PLDI*, pp. 30–44.

Wolf, M. & Lam, M. (1991b). A loop transformation theory and an algorithm to maximize parallelism, *IEEE Transactions on Parallel and Distributed Systems* .

BIOGRAPHY

Y. Annie Liu is assistant professor of computer science at Indiana University in Bloomington. She received a BS from Peking University (1987), an ME from Tsinghua university (1988), and an MS and a PhD from Cornell University (1992, 1996), all in computer science. She was a post-doctoral associate at Cornell University from 1995 to 1996. Liu's primary research interests are in the areas of programming languages, compilers, and software systems. She is particularly interested in general and systematic approaches to improving the efficiency of computations. Liu has strong other interests in database management, document processing, information management, and distributed computing.