Cost-Optimal Code Motion

MAX HAILPERIN
Gustavus Adolphus College

We generalize Knoop et al.'s Lazy Code Motion (LCM) algorithm for partial redundancy elimination so that the generalized version also performs strength reduction. Although Knoop et al. have themselves extended LCM to strength reduction with their Lazy Strength Reduction algorithm, our approach differs substantially from theirs and results in a broader class of candidate expressions, stronger safety guarantees, and the elimination of the potential for performance loss instead of gain. Also, our general framework is not limited to traditional strength reduction, but rather can also handle a wide variety of optimizations in which data-flow information enables the replacement of a computation with a less expensive one. As a simple example, computations can be hoisted to points where they are constant foldable. Another example we sketch is the hoisting of polymorphic operations to points where type analysis provides leverage for optimization. Our general approach consists of placing computations so as to minimize their cost, rather than merely their number. So long as the cost differences between flowgraph nodes obey a certain natural constraint, a cost-optimal code motion transformation that does not unnecessarily prolong the lifetime of temporary variables can be found using techniques completely analogous to LCM. Specifically, the cost differences can be discovered using a wide variety of forward data-flow analyses in a manner which we describe.

Categories and Subject Descriptors: D.3.4 [**Programming Languages**]: Processors—*optimization*

General Terms: Algorithms, Performance

Additional Key Words and Phrases: Code motion, elimination of partial redundancies, strength reduction

1. INTRODUCTION

We will present a general framework for code motion transformations that move code not only to less frequently executed program points, but also to program points where specialized circumstances allow the code to be replaced by less expensive but functionally equivalent code. This is a general framework, rather than a specific transformation, because it can be instantiated with different cost-reducing specialization mechanisms. For the sake of concreteness, however, we will use a single example instantiation for the bulk of the article, briefly sketching some other possibilities in Section 5. Our example transformation uses a simple variant of constant propagation and constant folding to replace multiplications with additions or copies. As we will see, when this is coupled with our code motion framework,

Author's address: Gustavus Adolphus College, 800 West College Avenue, Saint Peter, MN 56082; email: max@gustavus.edu.

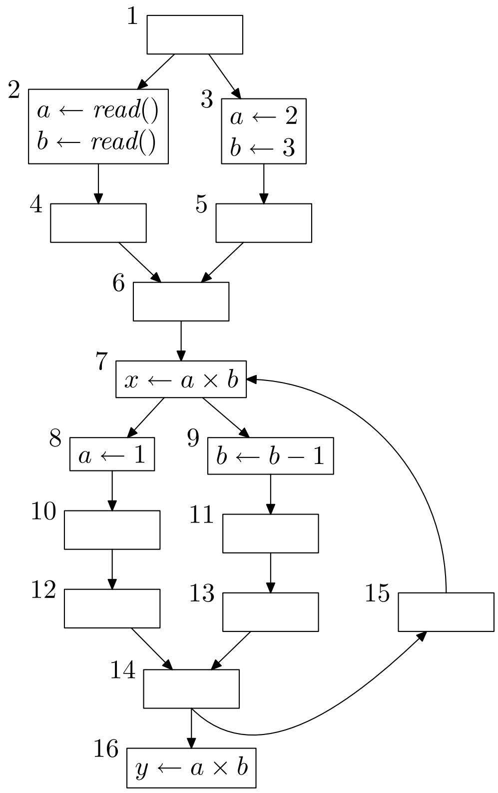

Fig. 1. An example program fragment.

the resulting transformation accomplishes strength reduction, although it does not coincide in scope with any of the existing strength reduction algorithms.

In order to illustrate this example instantiation of our transformation framework, consider the program fragment in Figure 1. Our goal is to transform it into Figure 2. Notice that all the inserted assignments to h (in nodes 4, 5, 12, and 13) have the same effect as an assignment $h \leftarrow a \times b$ would have at the same point. However, by taking advantage of the specialized circumstances of nodes 5, 12, and 13, the multiplication is replaced by a less-expensive operation. This helps explain the nodes into which these assignments were placed. For example, the assignment in node 5 cannot be delayed until node 6 without losing the opportunity for constant folding, because of the joining in of the path from node 4. Similarly, the computations in nodes 12 and 13 cannot be delayed until node 14, because although here each of the two joining paths has a cheap means of computation available, after the join neither of those cheap means would be legal. Of course, there are some other considerations as well. The computation in node 4 is there not because it is cheap, but as an indirect consequence of the placement of a computation in node 5. The computations in nodes 12 and 13 could equally well be in nodes 10 and 11, from a cost perspective, but we prefer to delay them so as not to unnecessarily lengthen lifetime ranges and risk additional register spilling. This actually only applies to

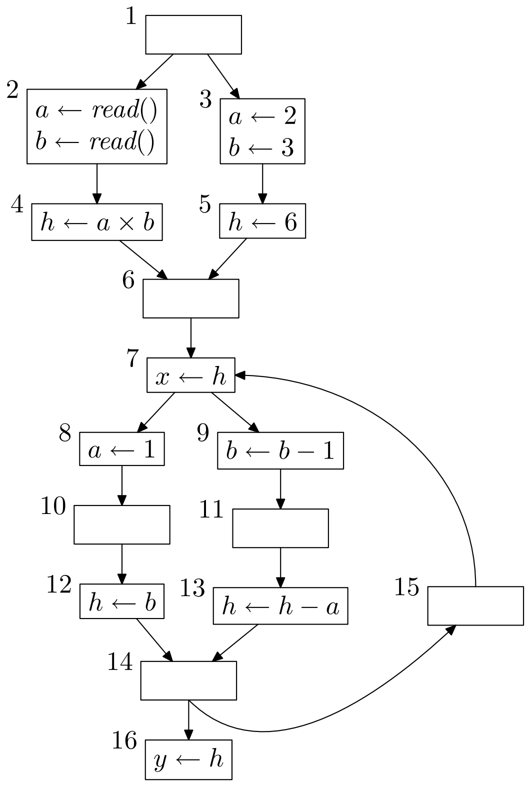

Fig. 2. Transformed version of example program fragment.

the computation in node 12; because node 13's computation is an update, the register pressure would not be any different were the computation done in node 11. For simplicity, however, our algorithm delays all computations, whether update or not. We will return to this example shortly, to show how our algorithm actually produces the desired result.

Our algorithm generalizes that of Knoop et al. [1994]; we assume familiarity with that article, since otherwise we would have to reproduce large portions of it. In referring to lemmas, theorems, etc. from that article, we will prefix the identifying number with "OCM," for *Optimal Code Motion.* We will also use the notations and definitions introduced in that article, as well as one additional standard notation: $[i, j]$ for $\{n \in \mathbb{N} \mid i \leq n \leq j\}$. Knoop et al. [1994] is divided into a "theory" portion, concerning properties of t-refined flowgraphs, and a "practice" portion, concerning how those properties can be calculated as fixed points using the original nonrefined flowgraph. We will present our generalization only in terms of the "theory" portion, assuming t-refined flowgraphs throughout, since the extension to the "practice" portion is quite straightforward.

In keeping with Knoop et al. [1994], we will focus on optimizing a single term; in our example, this is $a \times b$. For notational convenience, this candidate expression will be treated as an implicit parameter of all our functions and predicates, much like the program flow graph being operated on. In our example instantiation of our transformation framework, as in Knoop et al.'s original transformation, there are no interactions between the optimization of different terms of the form $v_1 \times v_2$, where v_1 and v_2 are variables. Thus if multiple terms are optimized, the results are independent of the order in which the terms are handled. However, as with Knoop et al.'s original transformation, this does not remain true if other optimizations, such as copy propagation, are interleaved with the code motions. Nor need this order independence hold for all instantiations of our framework. Therefore, in general it may be necessary to use a heuristic ordering and/or an iterative approach; we do not address these issues in this article.

As Figure 2 illustrated, our transformation consists of replacing each instance of the candidate expression $(a \times b)$ by a new variable (h), and inserting computations that in their respective contexts have the same effect as an assignment of the candidate expression to the variable. In order to keep our framework general, we will concern ourselves only with selecting the insertion points, and not with the question of which computation should be inserted at each selected point. The latter is instead determined by the particular instantiation of our framework, which must provide a function from nodes to statements, *computation*, such that $\mathit{computation}(n)$ is the statement to insert at node n if node n is chosen as an insertion point. This function is as usual implicitly parameterized by the candidate expression and flow graph. For example, given the flowgraph of Figure 1 and the candidate expression $a \times b$, our example instantiation of the transformation framework would have $\mathit{computation}(5)$ equal to $h \leftarrow 6$. (We will later show how this example *computation* function is defined.)

Although the selection of specific computations to insert is outside the scope of our framework, our transformation framework needs to know something about the computations if it is to select insertion points wisely. Namely, it needs to know how expensive a computation would be inserted at each point, were that point selected. Therefore, our framework is instantiated not only with a *computation* function, but also with a *cost* function that specifies a numerical cost for each of the computations. That is, for a node n, the cost of computing $\mathit{computation}(n)$ at node n is given by $\mathit{cost}(n)$. We will take this *cost* function to be another implicit parameter, treating it as a fixed, given function from the flowgraph's nodes to the natural numbers.

The overall transformation algorithm can be viewed conceptually as having three steps:

(1) Compute $\mathit{cost}(n)$ for each node n.

(2) Use these costs and the analysis described in this article to select insertion points. (This requires finding greatest fixed points of several equation systems, as described in the practice section of Knoop et al. [1994].)

(3) At each selected insertion node, n, insert $\mathit{computation}(n)$. Also, replace each original use of the candidate expression by the new variable, h.

However, it is important to recognize that $\mathit{cost}(n)$ can actually be calculated on a demand-driven, sparse basis, rather than being precomputed for each node as the conceptual algorithm suggests.

We will assume that the *cost* function obeys the following two constraints:

$$\forall m, n \in N.\ m \in \mathit{pred}(n) \wedge \mathit{Transp}(m) \wedge \neg \mathit{Comp}(m) \ \Rightarrow\ \mathit{cost}(n) \geq \mathit{cost}(m)$$
$$\forall m, n \in N.\ \mathit{pred}(n) = \{m\} \wedge \mathit{Transp}(m) \wedge \neg \mathit{Comp}(m) \ \Rightarrow\ \mathit{cost}(n) = \mathit{cost}(m)$$

These constraints reflect the connection between cost and degree of specialization. When paths join, any specialized computation that can be used after the join could be used equally well before it, but not necessarily vice versa. This is captured by the inequality in the first constraint. For a concrete example, consider node 6 in Figure 2. The computation $h \leftarrow a \times b$ that could be inserted there would work equally well in nodes 4 and 5, but the specialized computation $h \leftarrow 6$ that works in node 5 will not work in node 6. The second constraint says that when there is no joining of paths—when a node only has a single predecessor—there is no change in specificity of context, and hence the cost should remain fixed. However, all bets are off if the predecessor node contains an assignment to a or b or a computation of $a \times b$; then the cost can change arbitrarily.

In broad strokes, the Lazy Code Motion (LCM) technique of Knoop et al. [1994] consists of first moving all computations as early in the program as possible, in order to achieve the minimal number of computations (computational optimality), and then moving them as much later as is possible without undoing this, in order to minimize the lifetime of temporaries (lifetime optimality). Our Thrifty Code Motion (TCM) technique is based on the same outline of first moving computations as early as possible, then delaying them to a later point. However, it stops the second phase (delaying) earlier if necessary to maintain cost optimality (minimal cost of the computations). This approach is intuitively reasonable, but a skeptical attitude is in order. After all, although the starting and ending points of LCM's delaying are both computationally optimal, in general some of the intermediate points are not. Therefore, stopping the delaying at some intermediate point based on considerations foreign to LCM (such as our cost) will not in general preserve computational optimality. As it happens, when the stopping point is chosen to optimize a cost function that satisfies the above constraints, computational optimality *is* preserved—but that is part of what we need to prove in this article.

Temporarily taking correctness for granted, let us sketch how our transformation takes us from Figure 1 to Figure 2, and how it differs from Knoop et al.'s Busy Code Motion (BCM) and LCM transformations. Using a simple variant of constant propagation and folding, together with some assumptions about hardware-level costs, we can derive the *computation* and *cost* functions shown in Figure 3. (You can easily verify that this cost function obeys the stated constraints.) Next we analyze how early the computations could legally be placed, and determine that the earliest nodes are 4, 5, 10, and 11. Thus BCM would hoist the computations from nodes 7 and 16 to nodes 4, 5, 10, and 11. Not only is this hoisting computationally optimal (minimizing the number of computations), but it is also cost optimal if we insert the computations given by our *computation* function at nodes 4, 5, 10, and 11, rather than blindly inserting $h \leftarrow a \times b$ into each of them. We show this cost optimality of BCM in Section 2. However, this earliest placement unnecessarily prolongs the lifetime ranges of h. The LCM transformation therefore delays the computations as far as possible without introducing redundancy, which in this case would be all the way back to nodes 7 and 16, since the original program had

n	$computation(n)$	$cost(n)$
1	$h \leftarrow a \times b$	4
2	$h \leftarrow a \times b$	4
3	$h \leftarrow a \times b$	4
4	$h \leftarrow a \times b$	4
5	$h \leftarrow 6$	2
6	$h \leftarrow a \times b$	4
7	$h \leftarrow a \times b$	4
8	nop	1
9	nop	1
10	$h \leftarrow b$	2
11	$h \leftarrow h - a$	3
12	$h \leftarrow b$	2
13	$h \leftarrow h - a$	3
14	$h \leftarrow a \times b$	4
15	$h \leftarrow a \times b$	4
16	$h \leftarrow a \times b$	4

Fig. 3. Example *computation* and *cost* functions for Figure 1. The computation *nop* stands for no operation, i.e., no computation is needed to bring h up-to-date.

no redundancy. Unfortunately, this eliminates the opportunities for using more specialized computations. Therefore, our TCM transformation stops sinking the computation not only when redundancy would be introduced, but also when the cost would increase. In particular, since the cost of node 6 is greater than the cost of one of its predecessors (node 5), TCM will not delay into node 6. Therefore, nodes 4 and 5 are recognized as latest nodes for thrifty placement. Similarly, node 14's cost is greater than that of one of its predecessors (in fact, both). So delaying to node 14 is suppressed, and computations are placed in nodes 12 and 13. In Section 3 we show that this minimizes lifetime ranges of the new temporary, h, subject to retaining computational and cost optimality.

After we establish the TCM framework's optimality in Sections 2 and 3, we will turn in Section 4 to the question of how the framework can be instantiated with suitable *computation* and *cost* functions. We will first show how our example functions of Figure 3 are computed using a data-flow analysis. Next, we will show that the example *cost* function satisfies the two constraints. Then we will generalize and show that any *cost* function that results in an analogous manner from a forward data-flow analysis must satisfy our constraints. Hence, any such *cost* function and its companion *computation* function can be used to instantiate our TCM framework. We sketch a couple examples of this broader applicability in Section 5, and discuss our approach's limitations. The primary limitation, inherited from LCM, is the restriction to only inserting *safe* computations, i.e., those that compute values also computed by the unoptimized program. We conclude in Section 6 with a comparison to related work.

2. COST OPTIMALITY

In this section, we show that Knoop et al.'s BCM, i.e., the "earliest possible" placement of computations, optimizes the cost of computations, not just the number of computations. This provides necessary background to the following section, where we show our new code motion transformation, TCM, to also be cost optimal.

However, our first order of business is to formally define cost optimality. We actually provide two definitions, the first more intuitively reasonable and the second more suited to proving TCM's optimality, and show the two definitions to be nearly equivalent.

Our first notion of cost optimality consists of minimizing the total cost, along any finite path through the program, of the points at which the code motion places computations. We will call this notion *total-cost optimality*, reserving the term *cost optimality* for our second definition. The following definitions formalize total-cost optimality, after first pausing to give a convenient notation for the indices along a particular finite path where a particular code motion transformation places computations.

Definition 1. For each code motion transformation $CM \in \mathcal{CM}$ and path $p \in \mathbf{P}[s,e]$, let the set $C_{CM,p} = \{i \mid \mathit{Comp}_{CM}(p_i)\}$.

Definition 2. An admissible code motion transformation $CM \in \mathcal{CM}_{Adm}$ is total-cost-cheaper than another admissible code motion transformation $CM' \in \mathcal{CM}_{Adm}$ if and only if

$$\forall p \in \mathbf{P}[s,e].\ \sum_{i \in C_{CM,p}} \mathit{cost}(p_i) \leq \sum_{i \in C_{CM',p}} \mathit{cost}(p_i).$$

Definition 3. An admissible code motion transformation $CM \in \mathcal{CM}_{Adm}$ is total-cost-optimal if and only if it is total-cost-cheaper than any other admissible code motion transformation. We will call the set of total-cost-optimal code motion transformations $\mathcal{CM}_{TotCostOpt}$.

We next define cost optimality as an alternative to total-cost optimality. The key difference is that we will restrict our attention to those code motion transformations that are not merely admissible, but rather computationally optimal, i.e., which minimize the number of computations done. Since any two computationally optimal code motion transformations perform the same number of computations along a given finite path through the program, it makes sense to individually compare the costs of corresponding pairs of computations. In order to facilitate this comparison, we define a notation for the ith computation point. Using this, we can define a computationally optimal code motion transformation as cost-optimal if it makes each computation no more expensive than the corresponding computation under any other computationally optimal code motion.

Definition 4. For each code motion transformation $CM \in \mathcal{CM}$ and path $p \in \mathbf{P}[s,e]$, let the function $\mathit{compNumb}_{CM,p}(i)$ map the number $i \in [1, |C_{CM,p}|]$, to the node p_j such that $j \in C_{CM,p} \wedge |\{k \in C_{CM,p} \mid k < j\}| = i - 1$. That is, $\mathit{compNumb}_{CM,p}(i)$ is the ith node along the path p for which Comp_{CM} holds.

Definition 5. A code motion transformation $CM \in \mathcal{CM}_{CmpOpt}$ is cheaper than a code motion transformation $CM' \in \mathcal{CM}_{CmpOpt}$ if and only if

$$\forall p \in \mathbf{P}[s,e]\ \forall i \in [1, |C_{CM,p}|] \quad \mathit{cost}(\mathit{compNumb}_{CM,p}(i)) \leq \mathit{cost}(\mathit{compNumb}_{CM',p}(i)).$$

Definition 6. A code motion transformation $CM \in \mathcal{CM}_{CmpOpt}$ is cost-optimal if and only if it is cheaper than any other computationally optimal code motion transformation. We will call the set of cost-optimal code motion transformations $\mathcal{CM}_{CostOpt}$.

Comparing our two notions, total-cost optimality and cost optimality, it is not initially apparent that they should essentially coincide. On the one hand, it appears that total-cost optimality is more restrictive because it optimizes over a broader class of code motion transformations. On the other hand, it appears that cost optimality is more restrictive because it constrains the cost of each computation individually rather than only in aggregate. However, we will establish in Theorem 1 below that these issues are essentially illusory. We will build up to that theorem with some generally useful lemmas, which also show BCM in particular to be both cost optimal and total-cost optimal.

LEMMA 1 (CORRESPONDENCE LEMMA). *For any computationally optimal code motion transformation $CM \in \mathcal{CM}_{CmpOpt}$ and any path $p \in \mathbf{P}[s,e]$*

(*1*) *There is a bijection $f_{CM,p} : C_{CM,p} \to \{(j,l) \in \mathbb{N}^2 \mid p[j,l] \in \textit{FU-LtRg}(BCM)\}$. This bijection associates with each $i \in C_{CM,p}$ the unique (j,l) pair in the function's range with $j \le i \le l$. That is, this pair specifies the subpath of p containing p_i that is a first-use lifetime range for BCM.*

(*2*) $\forall k \in [1, |C_{CM,p}|]\ \exists l \in \mathbb{N}$

$$f_{CM,p}(\mathit{compNumb}_{CM,p}(k)) = (\mathit{compNumb}_{BCM,p}(k), l).$$

PROOF. Part (3) of the Busy-Code-Motion Lemma (OCM Lemma 3.12) shows that every computation of *CM* takes place in a first-use lifetime range of *BCM*; since the first-use lifetime ranges are disjoint (by OCM Lemma 3.9), this must be unique. Thus the function $f_{CM,p}$ is well defined, and to complete our proof of part (1) we need only show that it is one-to-one and onto. Part (2) of the Busy-Code-Motion Lemma shows that $f_{CM,p}$ is onto; by the computational optimality of *BCM* (OCM Theorem 3.13), the domain and range of $f_{CM,p}$ are equinumerous, so it must be one-to-one as well. Part (2) of the Correspondence Lemma follows from part (1) by induction on k. □

LEMMA 2 (INCREASING COST LEMMA). *For all paths $p \in \textit{FU-LtRg}(BCM)$, $\mathit{cost}(p_1) \le \mathit{cost}(p_2) \le \cdots \le \mathit{cost}(p_{\lambda_p})$.*

PROOF. From the definition of a first-use lifetime range and the admissibility of *BCM*, it follows that $p \in \textit{FU-LtRg}(BCM) \Rightarrow \mathit{Transp}^{\forall}(p[1,\lambda_p[) \wedge \neg \mathit{Comp}^{\exists}(p[1,\lambda_p[)$. The constraint on the *cost* function then directly provides the result. □

COROLLARY 1. *$BCM \in \mathcal{CM}_{CostOpt}$.*

PROOF. This follows directly from the Correspondence and Increasing Cost Lemmas. □

LEMMA 3. *$BCM \in \mathcal{CM}_{TotCostOpt}$.*

PROOF. Let $CM \in \mathcal{CM}_{Adm}$ and $p \in \mathbf{P}[s,e]$. By OCM Lemma 3.12 parts (1) and (2), there exists a function $f : C_{BCM,p} \rightarrow C_{CM,p}$ with the property that $\forall i \in C_{BCM,p} \exists j.\ i \leq f(i) \leq j \wedge p[i,j] \in \mathit{FU\text{-}LtRg}(BCM)$. By the Increasing Cost Lemma, $\forall i \in C_{BCM,p}.\ \mathit{cost}(p_i) \leq \mathit{cost}(p_{f(i)})$. By OCM Lemma 3.9, f is one-to-one. Therefore, we have

$$\begin{aligned}\sum_{i \in C_{BCM,p}} \mathit{cost}(p_i) &\leq \sum_{i \in C_{BCM,p}} \mathit{cost}(p_{f(i)}) \\ &= \sum_{i \in f(C_{BCM,p})} \mathit{cost}(p_i) \\ &\leq \sum_{i \in C_{CM,p}} \mathit{cost}(p_i)\end{aligned}$$ □

THEOREM 1. *(1)* $\mathcal{CM}_{CostOpt} = \mathcal{CM}_{TotCostOpt} \cap \mathcal{CM}_{CmpOpt}$. *(2) If* $\forall n \in N.\ \mathit{cost}(n) \neq 0$*, then* $\mathcal{CM}_{CostOpt} = \mathcal{CM}_{TotCostOpt}$.

PROOF. By definition, any cost-optimal code motion transformation, CM, is also computationally optimal. Therefore, we can establish one direction of each part of the theorem by showing that CM is also total-cost optimal. We have that CM is cost-optimal; by Corollary 1, so is BCM. Thus they must have the same total cost along any path $p \in \mathbf{P}[s,e]$. Together with BCM's total-cost optimality (Lemma 3), this implies the total-cost optimality of CM.

To establish the reverse direction of part (1), we need to show that total-cost optimality and computational optimality together imply cost optimality. Let the code motion transformation $CM \in \mathcal{CM}_{TotCostOpt} \cap \mathcal{CM}_{CmpOpt}$ and the path $p \in \mathbf{P}[s,e]$. Because CM and BCM are both computationally optimal, their computation points along p are in one-to-one correspondence. Because BCM is cost-optimal (by Corollary 1), each computation point of BCM along p has cost less than or equal to that of the corresponding computation point of CM. If any of these inequalities were strict, the total cost of BCM along p would be less than that of CM, contradicting the total-cost optimality of CM. Therefore, CM must be cost optimal as well.

Finally, to finish part (2), we need to show that total-cost optimality and the absence of zero-cost nodes imply cost optimality. Let the code motion transformation $CM \in \mathcal{CM}_{TotCostOpt}$, and let the path $p \in \mathbf{P}[s,e]$. Define the function f as in the proof of Lemma 3. We have

$$\begin{aligned}\sum_{i \in C_{BCM,p}} \mathit{cost}(p_i) &= \sum_{i \in C_{CM,p}} \mathit{cost}(p_i) \\ &\qquad \text{(by the total-cost optimality of } BCM \text{ and } CM\text{)} \\ &= \sum_{i \in f(C_{BCM,p})} \mathit{cost}(p_i) + \sum_{i \in C_{CM,p} - f(C_{BCM,p})} \mathit{cost}(p_i) \\ &\qquad \text{(splitting the sum)}\end{aligned}$$

$$= \sum_{i \in C_{BCM,p}} cost(p_{f(i)}) + \sum_{i \in C_{CM,p} - f(C_{BCM,p})} cost(p_i)$$

(since f is one-to-one)

$$\geq \sum_{i \in C_{BCM,p}} cost(p_i) + \sum_{i \in C_{CM,p} - f(C_{BCM,p})} cost(p_i)$$

(by the Increasing Cost Lemma)

Therefore, we have $\sum_{i \in C_{CM,p} - f(C_{BCM,p})} cost(p_i) \leq 0$. But we are given that none of the costs is zero, i.e., they are all strictly positive. So, the preceding summation must be over an empty range, i.e., f is onto as well as one-to-one. In other words, *CM* is computationally optimal. Part (1) of the theorem then lets us conclude that it is cost optimal as well. □

Thus our original notion of total-cost optimality nearly coincides with cost optimality. The cost-optimal code motion transformations are total-cost optimal as well, so restricting ourselves to computationally optimal transformations does not exact a price in the total cost that can be achieved. In fact, not only are no strictly total-cost cheaper transformations excluded, but moreover if there are no zero-cost nodes, then no transformations at all are excluded, since all the total-cost-optimal transformations are cost optimal as well. Having seen this, we will restrict our attention to cost-optimal transformations and proceed to the problem of minimizing lifetime ranges.

3. MINIMIZING LIFETIME RANGES

Knoop et al. [1994] define their LCM algorithm, which minimizes lifetime ranges while retaining computational optimality, in terms of two predicates, *Delayed* and *Latest*. We can analogously define our TCM in terms of *TDelayed* and *TLatest*, where the "T" stands for "Thrifty." The difference is that we are only willing to delay so long as the cost of the computation does not go up. Recall that in stating these definitions, we are going to treat the *cost* function as fixed, when in fact it is an implicit parameter. Thus we appear to be defining TCM as a single transformation, when in fact it is a general framework or family of transformations.

Definition 7. The predicate $TDelayed(n)$, for $n \in N$, is defined by

$$TDelayed(n) \Leftrightarrow$$
$$\forall p \in \mathbf{P}[s, n]\ \exists i \leq \lambda_p.\ Earliest(p_i) \wedge \neg Comp^{\exists}(p[i, \lambda_p[) \wedge cost(n) = cost(p_i).$$

It is worth observing that the the i which this definition asserts exists must in fact be unique, by OCM Lemma 3.9, and that the same applies to the definition of *Delayed* as well.

For practical computation of *TDelayed* as a fixed point, it is useful to note that by the Increasing Cost Lemma, $cost(p_i) = cost(p_{i+1}) = \cdots = cost(n)$. Therefore, *TDelayed* is the greatest fixed point of

$$TDelayed(n) \Leftrightarrow$$
$$Earliest(n) \vee n \neq s \wedge \bigwedge_{m \in pred(n)} TDelayed(m) \wedge \neg Comp(m) \wedge cost(m) = cost(n).$$

For example, you can use this formulation to verify that in Figure 1, nodes 4, 5, 10, 11, 12, and 13 are *TDelayed*.

Just as only *Delayed* nodes can be computation points after a computationally optimal code motion transformation, we can show that only *TDelayed* nodes can be computation points after a cost-optimal code motion transformation.

LEMMA 4. $\forall CM \in \mathcal{CM}_{CostOpt}\ \forall n \in N.\ Comp_{CM}(n) \Rightarrow TDelayed(n)$.

PROOF. By OCM Lemma 3.16(3), we know $Comp_{CM}(n)$ implies that $\forall p \in \mathbf{P}[s,n]\ \exists i \leq \lambda_p.\ Earliest(p_i) \wedge \neg Comp^{\exists}(p[i,\lambda_p[)$. It remains to show that $cost(n) = cost(p_i)$. Arbitrarily extend p to some $p' \in \mathbf{P}[s,e]$. By the Correspondence Lemma, there is some k for which $p_i = compNumb_{BCM,p'}(k)$ and $n = compNumb_{CM,p'}(k)$. By Corollary 1, BCM is cost optimal, and we are given that CM is also cost optimal. Thus we have $cost(p_i) = cost(n)$. □

Definition 8. The predicate $TLatest(n)$, for $n \in N$, is defined by

$$TLatest(n) \Leftrightarrow TDelayed(n) \wedge \left(Comp(n) \vee \bigvee_{m \in succ(n)} \neg TDelayed(m) \right).$$

Next we can prove that *TLatest* truly deserves its name: within each lifetime range of BCM, there is a unique *TLatest* node, which is the last *TDelayed* node in the range.

LEMMA 5. (*1*) $\forall p \in LtRg(BCM)\ \exists i \leq \lambda_p.\ TLatest(p_i)$. (*2*) $\forall p \in LtRg(BCM)\ \forall i \leq \lambda_p.\ TLatest(p_i) \Rightarrow \neg TDelayed^{\exists}(p]i,\lambda_p])$.

PROOF. The proof of part (1) directly parallels that of OCM Lemma 3.17's part (1).

The proof of part (2) requires a bit more care. From the definitions of *TLatest* and *TDelayed*, we have $TLatest(p_i) \Rightarrow TDelayed(p_i) \Rightarrow Delayed(p_i)$. Thus in the case that $Comp(p_i)$ holds, we have $Latest(p_i)$, and hence by OCM Lemma 3.17(2), $\neg Delayed^{\exists}(p]i,\lambda_p])$. From the definition of *TDelayed*, this in turn implies $\neg TDelayed^{\exists}(p]i,\lambda_p])$, our desired result.

The remaining case is for $\neg Comp(p_i)$. Then there must exist a successor, m, of p_i for which we have $\neg TDelayed(m)$. If we moreover have $\neg Delayed(m)$, then as above we have $Latest(p_i)$, so again the result follows from OCM Lemma 3.17(2). Thus we need only still address the case of $\neg Comp(p_i) \wedge \neg TDelayed(m) \wedge Delayed(m)$. Here we have $cost(p_1) = cost(p_i) \neq cost(m)$. But note we not only have $\neg Comp(p_i)$, but also by the admissibility of BCM have $Transp(p_i)$. Therefore, by the constraint on the *cost* function, we conclude that m must have a predecessor other than p_i, and hence by the Control Flow Lemma (OCM Lemma 2.1), m must be p_i's only successor, i.e., $m = p_{i+1}$. Relying again on our constraint on the *cost* function, we can specifically state that $cost(p_{i+1}) > cost(p_1)$. It remains to establish our result from this.

Let j be the smallest integer for which $Comp(p_j)$, i.e., $p[1,j] \in FU\text{-}LtRg(BCM)$. By the Increasing Cost Lemma, for any index k with $i < k \leq j$ we have $cost(p_k) \geq cost(p_{i+1}) > cost(p_1)$, so we have $\neg TDelayed^{\exists}(p]i,j])$. Now consider $p]j,\lambda_p]$. In this remaining portion of the path, we can again follow the proof of OCM Lemma

3.17(2). By the definition of $LtRg(BCM)$, we have $\neg Earliest^{\exists}(p]j, \lambda_p])$, which together with $Comp(p_j)$ implies $\neg Delayed^{\exists}(p]j, \lambda_p])$ and hence the remaining portion of our result, $\neg TDelayed^{\exists}(p]j, \lambda_p])$. □

The above properties of *TDelayed* and *TLatest*, which parallel those of *Delayed* and *Latest*, are the keys to showing that our TCM algorithm enjoys properties analogous to those of LCM: it retains cost-optimality (as LCM retains computational optimality) while minimizing lifetime ranges. We show this below.

Definition 9. The Thrifty Code Motion (TCM) transformation is defined by the $Insert_{TCM}$ and $Repl_{TCM}$ predicates:

$$\begin{aligned} Insert_{TCM}(n) &\Leftrightarrow TLatest(n) \\ Repl_{TCM}(n) &\Leftrightarrow Comp(n) \end{aligned}$$

Definition 10. A cost-optimal code motion transformation $CM \in \mathcal{CM}_{CostOpt}$ is cost-almost-lifetime-optimal if and only if

$$\forall p \in LtRg(CM).\ \lambda_p \geq 2 \Rightarrow \forall CM' \in \mathcal{CM}_{CostOpt} \exists q \in LtRg(CM').\ p \sqsubseteq q.$$

(This is identical to the definition of almost-lifetime-optimal, but with $\mathcal{CM}_{CostOpt}$ substituted for $\mathcal{CM}_{CmpOpt}$.) We will call the set of cost-almost-lifetime-optimal code motion transformations $\mathcal{CM}_{CostALtOpt}$.

THEOREM 2. $TCM \in \mathcal{CM}_{CostALtOpt}$.

PROOF. We will show this in four stages:

(1) $TCM \in \mathcal{CM}_{Adm}$.
(2) $TCM \in \mathcal{CM}_{CmpOpt}$.
(3) $TCM \in \mathcal{CM}_{CostOpt}$.
(4) $TCM \in \mathcal{CM}_{CostALtOpt}$.

The proof of parts (1) and (2) directly parallel those of parts (1) and (2) of OCM Theorem 3.18 (the Almost-Lazy-Code-Motion Theorem). Having established computational optimality, cost optimality (part (3)) directly follows from the cost optimality of *BCM* (Corollary 1) by way of the Correspondence Lemma and the definition of *TDelayed*. The proof of part (4) once again parallels that of the corresponding part of OCM Theorem 3.18, part (3). □

As an example of *TLatest*, you can verify that in Figure 1, nodes 4, 5, 12, and 13 are *TLatest*. This accounts for the insertion of computations into those nodes in Figure 2.

Note that it is also possible to do an “isolation” analysis and so refine TCM to be cost-lifetime-optimal (defined by analogy with lifetime optimal), rather than just cost-almost-lifetime-optimal, i.e., by the elimination of single-node lifetime ranges. However, we choose not to do this because some of the low-cost computation points may gain their cost savings by updating the previous value of the variable h. In particular, this is the case for strength reduction. Thus an assignment to h that

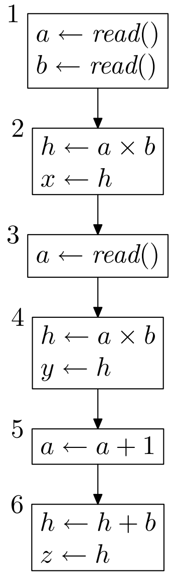

Fig. 4. The definition of h in node 4 is not really isolated.

appears to be isolated may actually be needed for a later update. For example, in the simplified program of Figure 4, all the definitions of h are isolated by the standard of LCM's isolation analysis, but our optimization necessitates retaining the one in node 4. A more sophisticated isolation analysis can of course be done that accounts for these uses of h that occur in update computations, rather than just the uses that replace original computations. A Chaitin-style register allocator [Chaitin et al. 1981; Chaitin 1982] will automatically do this, by detecting the isolated definitions of h as independent live ranges and coalescing each with the variable into which it is copied. We choose to rely on such a coalescing allocator, rather than doing the analysis here.

There is another important observation regarding (almost-)lifetime optimality and the use of update computations of h, that is, inserted computations that themselves make use of h. Since we took our definition of lifetime ranges from Knoop et al. [1994], it completely ignores these uses that occur in inserted computations. Thus, if TCM is faced with a situation in which there are two cost-optimal choices for where to insert a computation, it will always choose the later one, even if at that point an update computation needs to be inserted to achieve the optimal cost while at the earlier position a nonupdate computation could have achieved the same cost. Under this circumstance, the choice of the later insertion point actually results in a a *longer* period over which h is contributing to register pressure. See Figure 5 for an example. This problem arises because the update assignment serves not only to initiate a new lifetime range, but also to extend the previous range, since it is an additional use point. Thus a register is demanded for h all the way from its most recently preceding nonupdate computation. This flaw of the TCM algorithm can be avoided, however, by simply never using a *cost* function that assigns the same cost to a circumstance in which updating is necessary and one where a nonupdat-

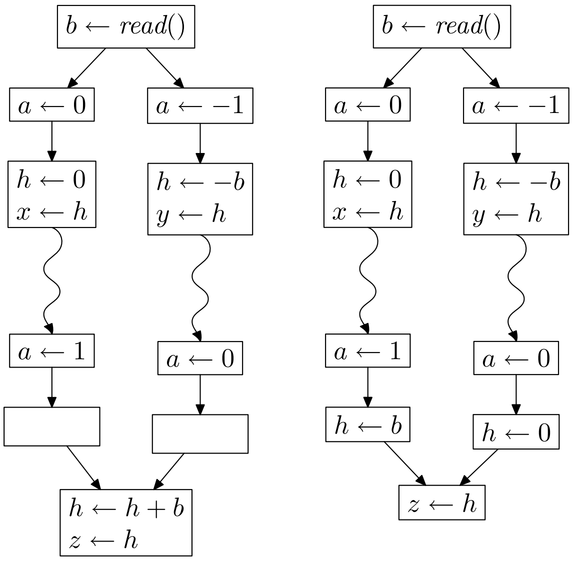

Fig. 5. A late update computation can cause more register pressure than an earlier nonupdate computation. These are two alternative optimized versions; the original (not shown) has no assignments to h and has $a \times b$ in place of each use of h. The wavy arrows represent potentially large regions of uninteresting code, in which h is live in the left-hand version but not in the right-hand version.

ing computation can achieve the same efficiency. That is, in the design of the *cost* function, the necessity of using updating can serve as a "tie-breaker" between otherwise equally attractive options. As an example, suppose that were it not for this consideration, we would assign a cost of $cost'(n)$ to the cheapest way we know to compute the term t at node n on our target hardware. Moreover, let $update(n)$ be 0 if there is some way to compute t this cheaply at n without using the old value of h, and 1 otherwise. Then we could define $cost(n) = 2 \cdot cost'(n) + update(n)$. This avoids the register pressure problem, because when the joining of paths causes an update computation to be necessary (as in Figure 5), $cost(n)$ will be higher after the join than it is before, even if $cost'(n)$ remains constant. Thus TCM will choose the nodes before the join as the insertion points.

4. ORIGIN OF THE COST AND COMPUTATION FUNCTIONS

As we have repeatedly stressed, the TCM framework is independent of the *cost* and *computation* functions, so long as the *cost* function meets the two stated constraints. However, in this section we will show one way in which such functions can plausibly arise. In particular, we will examine in detail the functions used in our running example, and then show that any functions arising in an analogous manner will also satisfy the constraints.

Our approach consists of first using a data flow analysis to assign each node, n, a set of computations, S_n, that would be legal alternatives to insert at that node. Then we can use a hardware-specific model of the cost of each computation to define $computation(n)$ as a minimal cost element of S_n and $cost(n)$ as its cost. (Of course,

computation	cost
nop	1
$h \leftarrow \pm a$	2
$h \leftarrow \pm b$	2
$h \leftarrow const$	2
$h \leftarrow h \pm a$	3
$h \leftarrow h \pm b$	3
$h \leftarrow a \times b$	4

Fig. 6. Example hardware-specific assignment of costs to computations.

an actual implementation need not construct the full set to determine a minimal cost element.) In the example of Figures 1 and 2, we have $S_5 = \{h \leftarrow 6,\, h \leftarrow a \times b\}$ and $S_6 = \{h \leftarrow a \times b\}$, for instance. Note that $S_6 \subseteq S_5$. Thus, no matter what hardware-specific costs are assigned to the computations, a minimal cost element of S_6 will be no more expensive than a minimal cost element of S_5. This illustrates a general benefit of our approach: we will be able to show that the *cost* function satisfies its constraints purely by reference to the properties of the sets S_n, without regard to the hardware-specific cost assignment. Thus the cost assignment is quite arbitrary, other than that it should break ties in favor of nonupdate computations, in order to reduce register pressure, as discussed earlier. To construct the example functions in Figure 3, we used the costs shown in Figure 6.

Having seen that the sets S_n are the crux of the matter, we can turn to their computation using data flow analysis. The analysis used for our example is a variant of constant propagation and folding. Because the sets S_n are not information rich enough to directly serve as the basis of the data flow analysis, we perform the analysis using some more detailed information, $\mathit{info}(n)$, and then translate to S_n afterward. Specifically, we associate with each node four pieces of information, which together determine S_n. Two are the values of a and b, if each is known to have a constant value at that node. For example, at node 10 we know that $a = 1$, but do not know a constant value for b. The other two pieces of information are the changes in a and b since the most recent computation of $a \times b$. Again, we can either have a constant value or an indication (which we write $\top$) that no constant value is known. For example, at node 10, $\Delta_b = 0$, but $\Delta_a = \top$. We summarize the four pieces of information at node n as a four-tuple, $\mathit{info}(n) = (a, \Delta_a, b, \Delta_b)$. For example, $\mathit{info}(10) = (1, \top, \top, 0)$ and $\mathit{info}(11) = (\top, 0, \top, -1)$. Formally, we define a complete lattice **FlatInt** that contains $\perp$, the integers, and $\top$, with $\perp \sqsubseteq k \sqsubseteq \top$ for any integer k, and with any distinct integers incomparable. Then the function *info* maps flow graph nodes to elements of the lattice $\mathbf{FlatInt}^4$. We will use $\mathbf{L}$ as a name for $\mathbf{FlatInt}^4$.

As is usual in data flow analysis, we will associate with each node, n, a flow function $f_n : \mathbf{L} \rightarrow \mathbf{L}$, and will use these flow functions to define the *info* function. The flow functions themselves can be readily derived from our intended semantics of the tuples $(a, \Delta_a, b, \Delta_b)$. In doing so, we need to extend the arithmetic operators $+$ and $-$ to operate on **FlatInt**. We use $\oplus$ and $\ominus$ for the extended versions, and have $\forall x \in \mathbf{FlatInt}.\ \top \oplus x = x \oplus \top = \top$, and $\forall x \in \mathbf{FlatInt}.\ x \neq \top \Rightarrow \perp \oplus x = x \oplus \perp = \perp$. (And similarly for $\ominus$.) If node n contains only a single three-address instruction, we can define f_n as follows. We only show assignments to a and x; assignments to

n	$f_n(a, \Delta_a, b, \Delta_b)$
1	$(a, \Delta_a, b, \Delta_b)$
2	$(\top, \top, \top, \top)$
3	$(2, \Delta_a \ominus a \oplus 2, 3, \Delta_b \ominus b \oplus 3)$
4	$(a, \Delta_a, b, \Delta_b)$
5	$(a, \Delta_a, b, \Delta_b)$
6	$(a, \Delta_a, b, \Delta_b)$
7	$(a, 0, b, 0)$
8	$(1, \Delta_a \ominus a \oplus 1, b, \Delta_b)$
9	$(a, \Delta_a, b \ominus 1, \Delta_b \ominus 1)$
10	$(a, \Delta_a, b, \Delta_b)$
11	$(a, \Delta_a, b, \Delta_b)$
12	$(a, \Delta_a, b, \Delta_b)$
13	$(a, \Delta_a, b, \Delta_b)$
14	$(a, \Delta_a, b, \Delta_b)$
15	$(a, \Delta_a, b, \Delta_b)$
16	$(a, 0, b, 0)$

Fig. 7. Flow functions for example analysis of Figure 1.

b should be handled symmetrically to a, and all other variables should be treated as x.

$$f_n(a, \Delta_a, b, \Delta_b) = \begin{cases} (a, \Delta_a, b, \Delta_b) & \text{if } \mathit{Transp}(n) \wedge \neg \mathit{Comp}(n) \\ (\mathit{const}, \Delta_a \ominus a \oplus \mathit{const}, b, \Delta_b) & \text{if } n \text{ is } a \leftarrow \mathit{const} \\ (a \oplus \mathit{const}, \Delta_a \oplus \mathit{const}, b, \Delta_b) & \text{if } n \text{ is } a \leftarrow a + \mathit{const} \\ (\top, \top, b, 0) & \text{if } n \text{ is } a \leftarrow a \times b \\ (\top, \top, b, \Delta_b) & \text{if } n \text{ is } a \leftarrow \mathit{anything\ else} \\ (a, 0, b, 0) & \text{if } n \text{ is } x \leftarrow a \times b. \end{cases}$$

If the node n contains more than one instruction, we can construct f_n from the above basis by composition. For instance, Figure 7 shows the flow functions for our running example.

Having defined our flow functions, we next examine how we use them to solve for *info*. Our convention for the ordering relation on $\mathbf{L}$ is that $x \sqsubseteq y$ means y describes a situation at least as general as x. As such, we will use the join or least-upper-bound operator, $\sqcup$, as our confluence operator to merge the information on converging paths. We consider two different ways in which the function $\mathit{info}(n)$ can be defined: by taking a join over all paths leading up to the node n, or as a least-fixed-point solution to equations relating the information at each node to that at its predecessors. It is well known [Kildall 1973] that these produce the same result when the flow functions are distributive, rather than merely monotonic. However, our example flow functions are not all distributive (specifically, f_3 and f_8 are not), and we speculate that this may also be true for many other analyses to which our general framework might profitably be applied. Therefore, we will show that either solution leads to a cost function obeying our necessary constraint. Thus if a means exists to find the join over all paths, its additional precision can safely be used. In the more normal case that only the least fixed point is computable, that will also be safe. It so happens that although our example analysis is not distributive, the two solutions coincide for the particular flowgraph we show. Thus the $\mathit{info}(n)$ column of Figure 8 can be correctly read as either the join over all paths or the least fixed

n	$info(n)$
1	$(\top, \top, \top, \top)$
2	$(\top, \top, \top, \top)$
3	$(\top, \top, \top, \top)$
4	$(\top, \top, \top, \top)$
5	$(2, \top, 3, \top)$
6	$(\top, \top, \top, \top)$
7	$(\top, \top, \top, \top)$
8	$(\top, 0, \top, 0)$
9	$(\top, 0, \top, 0)$
10	$(1, \top, \top, 0)$
11	$(\top, 0, \top, -1)$
12	$(1, \top, \top, 0)$
13	$(\top, 0, \top, -1)$
14	$(\top, \top, \top, \top)$
15	$(\top, \top, \top, \top)$
16	$(\top, \top, \top, \top)$

Fig. 8. The *info* function for our running example.

point.

In order to discuss joins over all paths (even when there are infinitely many), we will rely on the lattice being complete. In order to discuss the least fixed point, we will rely on the flow functions being monotonic. When considering other analyses beyond our initial example, one can settle for weaker conditions on the lattice and flow functions by omitting either our results about joins over all paths or our results about least fixed points.

Either way we solve the forward data flow analysis problem, we will take as given the value of $info(s)$, which we will call *Init*. For our sample analysis, we use $Init = (\top, \top, \top, \top)$. To define the join over all paths, we extend the flow functions from individual nodes to paths by composition. That is, for any path p, $f_p = f_{p_{\lambda_p}} \circ \cdots \circ f_{p_2} \circ f_{p_1}$. Then our join over paths definition is

$$\forall n \in N.\ info(n) = \bigsqcup_{p \in \mathbf{P}[s,n]} f_{p[1,\lambda_p[}(Init).$$

Our least-fixed-point definition is that we take *info* as the least fixed point of the system

$$info(n) = \begin{cases} Init & n = s \\ \bigsqcup_{m \in pred(n)} f_m(info(m)) & n \neq s. \end{cases}$$

Once $info(n)$ is computed using either definition, it can be abstracted to S_n, the set of computations that might be inserted into n. In the following, c_a and c_b are arbitrary constants, and $(c_a \times c_b)$ should be interpreted as the constant formed by the compile-time multiplication of c_a and c_b. S_n is the minimal set such that

$$\begin{aligned}
info(n) \sqsubseteq (\top, 0, \top, 0) &\Rightarrow nop \in S_n \\
info(n) \sqsubseteq (0, \top, \top, \top) &\Rightarrow h \leftarrow 0 \in S_n \\
info(n) \sqsubseteq (\top, \top, 0, \top) &\Rightarrow h \leftarrow 0 \in S_n \\
info(n) \sqsubseteq (\pm 1, \top, \top, \top) &\Rightarrow h \leftarrow \pm b \in S_n \\
info(n) \sqsubseteq (\top, \top, \pm 1, \top) &\Rightarrow h \leftarrow \pm a \in S_n
\end{aligned}$$

$$
\begin{aligned}
\mathit{info}(n) \sqsubseteq (c_a, \top, c_b, \top) &\Rightarrow h \leftarrow (c_a \times c_b) \in S_n \\
\mathit{info}(n) \sqsubseteq (\top, \pm 1, \top, 0) &\Rightarrow h \leftarrow h \pm b \in S_n \\
\mathit{info}(n) \sqsubseteq (\top, 0, \top, \pm 1) &\Rightarrow h \leftarrow h \pm a \in S_n \\
\mathit{info}(n) \sqsubseteq (\top, c_a, c_b, 0) &\Rightarrow h \leftarrow h + (c_a \times c_b) \in S_n \\
\mathit{info}(n) \sqsubseteq (c_a, 0, \top, c_b) &\Rightarrow h \leftarrow h + (c_a \times c_b) \in S_n \\
\mathit{info}(n) \sqsubseteq (\top, \top, \top, \top) &\Rightarrow h \leftarrow a \times b \in S_n.
\end{aligned}
$$

We have now completely shown how our example *cost* and *computation* functions are defined. First the data flow analysis yields $\mathit{info}(n)$. Then this is mapped to S_n as shown above. Finally a minimal cost element of S_n is chosen as $\mathit{computation}(n)$, and $\mathit{cost}(n)$ is its cost. Having seen this three-stage process, we next turn to the question of how it ensures that the *cost* functions satisfies the two necessary constraints. We start with the key observation that the *info* function has properties analogous to those we need for *cost*.

LEMMA 6. *Using the least-fixed-point definition of the info function,*

$$
\begin{aligned}
\forall m, n \in N.\ m \in \mathit{pred}(n) \wedge \mathit{Transp}(m) \wedge \neg \mathit{Comp}(m) &\Rightarrow \mathit{info}(n) \sqsupseteq \mathit{info}(m) \\
\forall m, n \in N.\ \mathit{pred}(n) = \{m\} \wedge \mathit{Transp}(m) \wedge \neg \mathit{Comp}(m) &\Rightarrow \mathit{info}(n) = \mathit{info}(m).
\end{aligned}
$$

PROOF. Suppose we have $\mathit{Transp}(m) \wedge \neg \mathit{Comp}(m)$. Recall from the construction of the flow functions that f_m is then the identity function. Therefore,

$$
\begin{aligned}
\mathit{info}(n) &= \bigsqcup_{m' \in \mathit{pred}(n)} f_{m'}(\mathit{info}(m')) \\
&\sqsupseteq f_m(\mathit{info}(m)) \\
&= \mathit{info}(m).
\end{aligned}
$$

When m is the only predecessor of n, i.e., the only value for m' above, we can replace the $\sqsupseteq$ by $=$. $\square$

LEMMA 7. *Using the join over paths definition of the info function,*

$$
\begin{aligned}
\forall m, n \in N.\ m \in \mathit{pred}(n) \wedge \mathit{Transp}(m) \wedge \neg \mathit{Comp}(m) &\Rightarrow \mathit{info}(n) \sqsupseteq \mathit{info}(m) \\
\forall m, n \in N.\ \mathit{pred}(n) = \{m\} \wedge \mathit{Transp}(m) \wedge \neg \mathit{Comp}(m) &\Rightarrow \mathit{info}(n) = \mathit{info}(m).
\end{aligned}
$$

PROOF. By the definition of *info* and the fact that f_m is the identity, we have

$$
\begin{aligned}
\mathit{info}(n) &= \bigsqcup_{p \in \mathbf{P}[s,n]} f_{p[1,\lambda_p[}(\mathit{Init}) \\
&= \bigsqcup_{m' \in \mathit{pred}(n)} \bigsqcup_{p \in \mathbf{P}[s,m']} f_{m'}(f_{p[1,\lambda_p[}(\mathit{Init})) \\
&\sqsupseteq \bigsqcup_{p \in \mathbf{P}[s,m]} f_m(f_{p[1,\lambda_p[}(\mathit{Init})) \\
&= \bigsqcup_{p \in \mathbf{P}[s,m]} f_{p[1,\lambda_p[}(\mathit{Init}) \\
&= \mathit{info}(m).
\end{aligned}
$$

In the case where m is the only predecessor of n, the $\sqsupseteq$ in the above reasoning can be replaced by $=$. $\square$

THEOREM 3. *If $cost(n)$ is defined in terms of S_n, and thus in terms of $info(n)$, as described above, we have*

$$\forall m, n \in N.\ m \in pred(n) \wedge Transp(m) \wedge \neg Comp(m) \ \Rightarrow\ cost(n) \geq cost(m)$$
$$\forall m, n \in N.\ pred(n) = \{m\} \wedge Transp(m) \wedge \neg Comp(m) \ \Rightarrow\ cost(n) = cost(m).$$

PROOF. By Lemma 6 or 7, we have $info(n) \sqsupseteq info(m)$, with equality when n has no predecessors other than m. From the construction of the sets, S, this implies $S_n \subseteq S_m$, again with equality when there are no other predecessors. Thus $computation(n) \in S_m$, and since $computation(m)$ is chosen as a minimal-cost element of S_m, we must have $cost(n) \geq cost(m)$. When there is only one predecessor, $S_n = S_m$, and so clearly their minimal-cost elements must be of equal cost. $\square$

We have now finished discussing our example functions and can turn to the more general question of how other data flow analyses can analogously give rise to *cost* functions that satisfy the constraints. The proofs above relied only on quite general properties (such as monotonicity), with one notable exception. Namely, our example data flow analysis had the property that $Transp(n) \wedge \neg Comp(n) \Rightarrow f_n(x) = x$. This was relied upon in both Lemmas 6 and 7, and does not hold for many more general data flow analyses. For example, if the *info* at each node were expanded to track the values of other variables, the above condition would not hold. However, by adding one additional step to our process, we can substantially relax this restriction.

Namely, rather than directly defining S_n from $info(n)$, we can interpose an abstraction function, *relevance*, mapping $\mathbf{L}$ to another lattice $\mathbf{L}'$, and define S_n in a monotonic fashion from $relevance(info(n))$. Our restriction is now that

$$\forall n \in N.\ Transp(n) \wedge \neg Comp(n) \Rightarrow relevance \circ f_n = relevance.$$

We can informally say that f_n must be transparent to the relevant portion of the data flow information, rather than to all of it. Provided *relevance* is monotonic, a result analogous to Lemma 6 can be proved, but with $relevance(info(m))$ and $relevance(info(n))$ in place of $info(m)$ and $info(n)$. If we further restrict *relevance* to be distributive, Lemma 7 can similarly be generalized. Therefore, a result like Theorem 3 still holds. This provides the key to TCM's more general applicability—an area we touch on in the next section.

5. LIMITATIONS AND POTENTIAL

This section attempts to delimit more clearly the applicability of the TCM technique by both making more explicit some of its limitations, which previously were mentioned only in passing, and sketching two examples of additional potential applications, rather different from the running example used up until now.

We will take up the limitations first—in a spirit of realism, not pessimism. There are three principle limitations: the insistence on safety, the restriction on the cost function, and the absence of a mechanism for transforming nested expressions in ways made profitable by later linear function test replacement and dead-code elimination.

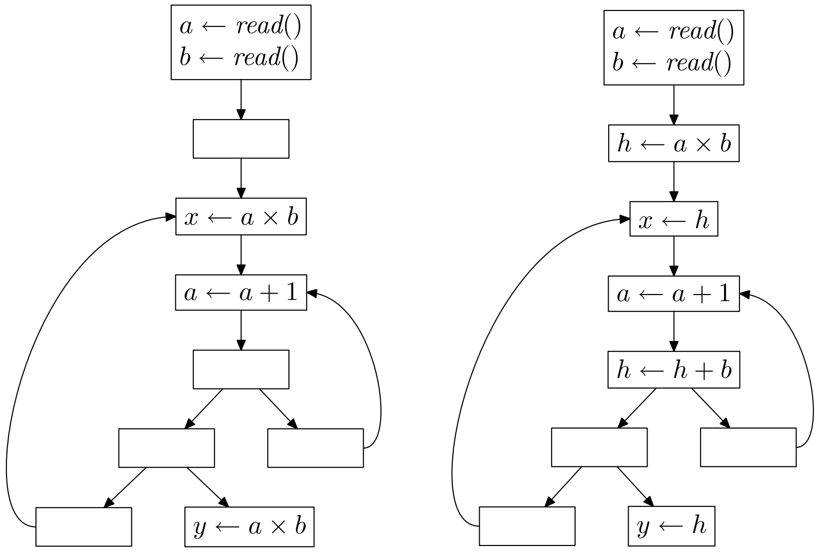

Fig. 9. A questionable replacement of an outer-loop multiplication by an inner-loop addition.

When we say our insertion points are all safe, we mean that no inserted computation calculates a value not calculated by the unoptimized program. One consequence of our insistence on safety is that we never can chain together multiple update computations without an intervening use. For example, consider a loop body containing the multiplication $i \times 10$, then a conditional increment of i, and then a conditional decrement of i. On a machine where multiplication is more than twice as expensive as addition, the loop would run faster if it were strength reduced, coupling each of the conditional updates to i with a corresponding update to a temporary holding $i \times 10$. However, TCM does not consider this option, as it is unsafe. Note that "safety" is not merely a technical term here: a spurious overflow could arise in the scenario described above. On the other hand, the overflow could in practice be engineered around, leaving TCM at a disadvantage relative to other strength reduction algorithms.

This also allows us to gain more insight into the superficially surprising result that cost optimality and total cost optimality nearly coincide (Theorem 1). As the above example shows, it *can* be profitable to replace a single expensive computation with two inexpensive computations. However, doing so requires the unsafe chaining of updates, which we had implicitly ruled out in the statement of the theorem.

On the other hand, if we were to drop the requirement of safety and allow multiple updates, it turns out that there is in general no total-cost-optimal version of a program, independent of the path taken through the program. One version can favor one path through the program while another favors another path, with none being optimal for all paths. For example, in Figure 9, either version of the program can have lower total cost, depending on the relative frequencies of the inner and outer back edges.

Our safety restriction has other consequences as well, beyond ruling out multiple updates. In the introductory example, the use of $a \times b$ that occurs after the loop exit is essential to allow safe strength reduction. This is not as much of a limitation as it may initially appear, however, in that it only affects situations (like the example) in which a redefinition of a or b in the loop body is under conditional control. The more straightforward case of an unconditional assignment in the loop body can be safely handled with an update in the synthetic node on the loop's back edge.

Similarly, the insistence on safety means that we can only handle loops that first use $a \times b$, then afterward redefine a and/or b. (Unless, that is, another use of $a \times b$ occurs before loop entry.) A loop that did the redefinition before the use would only be strength reducible if we allowed an unsafe computation in the loop prologue. Luckily, modern programming style often favors use before redefinition.

Moving on to the second principal limitation of TCM, we have to be aware that although the cost function can arise from many analyses, it is not fully general. Consider, for example, a cost function that models the scheduling of hardware resources. Such a cost function might assign a lower cost to an earlier node even without joining of paths, uses of the optimized term, or definitions of its variables. For example, it could be less expensive to do the computation earlier not because a specialized version can be substituted, but because the latency can be fully overlapped with other computations before the result is needed. Important as such scheduling considerations are, they are completely outside of TCM's scope. The cost of a computation is allowed to depend on the specificity of information available about the operands, but not on such "extrinsic" factors as hardware scheduling.

Also, note that hoisting a computation only increases the specificity of context if we are considering the precontext, or history, of the computation. Sinking a computation increases the specificity of the postcontext, or future. Either form of specificity can be relevant to cost. For example, by sinking a computation into a region where one of its operands becomes dead, we might allow a less expensive two-address instruction to be used. This opportunity would not be exploitable using TCM. Instead, it appears that an analogous extension to partial dead-code elimination might be useful.

Finally, in addition to these general limitations of the TCM framework, there are some limitations specific to the instantiation of TCM we used as our running example. For instance, our analysis cannot track a constant through variables other than those being multiplied. As a second example, our transformation cannot take advantage of an increment of a by 2 if b is nonconstant, even if a shift and an add are cheaper than a multiplication, or if the value $2b$ happens to be available.

Having commented on TCM's limitations, we can now provide some positive evidence of its applicability, in particular two examples supporting our claim that TCM has potential outside of the normal strength reduction area. Our first example is the compilation of polymorphic operations. A type (or class) analysis can serve as the basis for the cost function: the operation is cheaper at program points where specific type information is deducible. For example, consider the following code in an object-oriented language like Java:

```
Foo x = getSomeFoo();   // could be any subclass of Foo
while(x.someMethod())   // have to find which class's someMethod
  x = new Foo(x);       // this makes a Foo, no subclass
```

On entry to this loop, it is not known what specific class of object `x` refers to—it

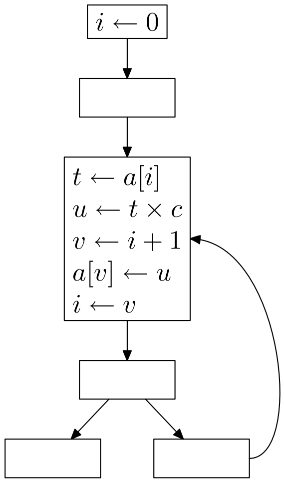

Fig. 10. A program with opportunity for scalar replacement.

could be any subclass of Foo. As such, the concrete method to be invoked needs to be dynamically selected. However, on subsequent iterations a cheaper static binding to a specific method is possible. Assuming the compiler's intermediate representation separates method lookup from method invocation, TCM could be used to hoist one copy of the method lookup code above the loop[1] and another copy to the end of the loop body. The one at the end of the loop body could then be "strength reduced" to a static binding.

One of the anonymous referees observed that some simple cases of scalar replacement might also be expressible in the TCM framework. Consider for example the program in Figure 10. Performing BCM on the expression $a[i]$ produces the flowgraph shown on the left in Figure 11. It would be plausible for a specialization mechanism to then replace the two hoisted instances of $a[i]$ as shown on the right in that figure. Thus TCM, armed with suitable *computation* and *cost* functions embodying the simplification mechanism, would produce this result. (In this example, TCM's lifetime reduction compared with BCM does not come into play, but in general it would.)

6. RELATED WORK

There are two fundamentally distinct approaches to integrating strength reduction into partial redundancy elimination. One approach (exemplified by our work) treats all computations that have the same net effect (at some point in the flowgraph) as equivalent. In our running example, we have a number of computations equivalent to $h \leftarrow a \times b$. In this approach, we use the techniques of partial redundancy elimination to select in a unified manner the program points that should contain some computation (of this equivalence class), whether it be a high-cost compu-

[1]In a language like Java which requires an exception to be raised at method invocation time if the initial `x` is a null reference, some additional care will be required with the hoisted method lookup to make sure the exception is not prematurely raised.

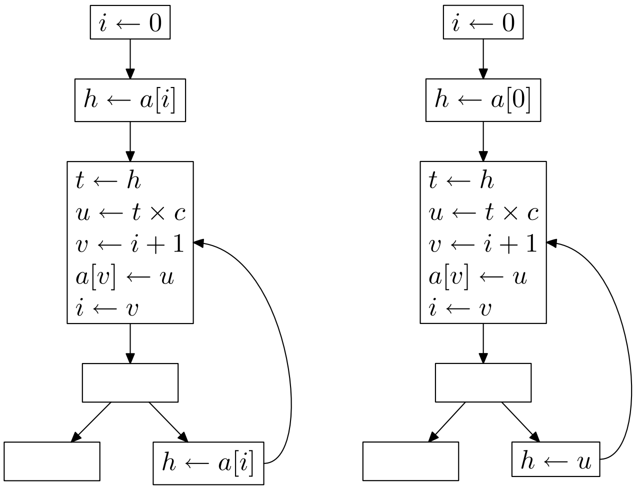

Fig. 11. Code motion and simplification achieves scalar replacement.

tation or a low-cost one. However, we "bias" the partial redundancy elimination in favor of choosing those computation points at which low-cost members of the equivalence class exist. At each chosen point, we then use the lowest-cost equivalent computation.

In the alternative approach to integrating strength reduction with partial redundancy elimination, the low-cost update computations are initially ignored. That is, nodes at which an update computation would suffice are treated as though they were transparent. Given this simplification, normal ("unbiased") partial redundancy elimination is used to choose only the full-cost computation points. The necessary update computation points are then added in afterward.

The majority of prior work on integrating strength reduction with partial redundancy elimination has fallen into this latter category, unlike the approach presented in this article. The earliest work is that of Joshi and Dhamdhere [1982], which was followed by Dhamdhere [1989]. The current state-of-the-art versions are those of Knoop et al. [1993], Dhaneshwar and Dhamdhere [1995], and Kennedy et al. [1998].

All of these variants differ from our approach in that they may insert "unsafe" computations, i.e., computations of values that were not computed by the unoptimized program. This can cause problems in some circumstances, for example when an overflow exception can result. On the other hand, it may be possible to avoid these adverse consequences, for example by using instructions that do not signal exceptions. If such precautions suffice, the ability to insert unsafe computations can open up some optimization opportunities that are closed to our rigidly safe approach, as discussed in the previous section.

The algorithm of Knoop et al. and those of Dhamdhere and his collaborators also differ from our approach in being restricted to a narrower class of candidate computations. For this reason, none of them would be able to handle our introductory example. In particular, all except the Strength Reduction of Large Expressions (SRLE) algorithm of Dhaneshwar and Dhamdhere are limited to multiplications of a variable by a constant. This is extended in SRLE to larger expressions, but with one operand of each multiplication still required to be a constant. By contrast, Kennedy et al.'s algorithm covers a rather broad class of candidates. As published, it still does not suffice for our introductory example, but a simple extension would allow it to handle the loop portion of our example.

More fundamentally, this approach inextricably combines the notions of "less-expensive computation" and "update computation." As such, it cannot be adapted to goals outside the traditional ambit of strength reduction, such as hoisting computations to points where they are constant foldable. (Our introductory example also included an instance of this optimization.)

Finally, all of these algorithms (unlike ours) can make some programs slower rather than faster. Of the three state-of-the-art algorithms, those of Knoop et al. and Kennedy et al. have a more severe problem in this regard than does that of Dhaneshwar and Dhamdhere, in that they can cause unboundedly large slowdowns, by replacing a multiplication in an outer loop by an addition in an inner loop.[2] Consider for example the transformation shown in Figure 9. Because the update computation $h \leftarrow h + b$ in this example is unsafe, in the sense that it can compute values that would not otherwise be computed, our TCM transformation would not fall into this trap.

Of the above papers focusing on the integration of strength reduction with partial redundancy elimination, Kennedy et al.'s is unique in that it is based on a novel algorithm (SSAPRE) for partial redundancy elimination in static single-assignment (SSA) form [Chow et al. 1997]. We suspect that SSAPRE could equally well be extended in the manner of our TCM, though proving this remains as future work.

Turning to our own approach, in which partial redundancy elimination is used to place all computations, not just the full-cost ones, the prior work is decidedly sparser. An early letter by Dhamdhere [1979] seems to suggest the general approach of first placing computations, then replacing some by less expensive versions. The only paper to flesh this out in a partial redundancy elimination context is the "semantic" approach of Knoop and Steffen [1991]. This differs from our work in two important regards. First, a very powerful and expensive analysis is used to determine semantically equivalent computations, while we suggest more limited means of determining equivalences. Second, no provision is made for minimizing the lifetimes of temporaries, unlike in TCM.

As we remarked in the previous section, if one abandons TCM's insistence on safety, it is no longer possible in general to simultaneously optimize each path through the program. There is a body of work on partial redundancy elimination that follows up on precisely this observation, by placing computations that are "speculative" (or in our more prejudicial terminology, unsafe), and using statistical information about execution frequencies to optimize over the frequency-weighted collection of paths. This approach is exemplified by Bodík et al. [1998], Gupta et al. [1998], and Horspool and Ho [1997].

Another of TCM's limitations mentioned in the previous section is that the restrictions on the *cost* function make it unsuitable for modeling hardware resource

[2] At least, the published version of Kennedy et al.'s algorithm has this problem. Fred Chow indicated in private correspondence that in implementing the algorithm, the SGI group made a simple modification that avoids this problem.

availability. For an alternative approach to partial redundancy elimination that does incorporate resource availability, see Gupta et al. [1997].

Rüthing's FCM algorithm [1998] is another partial redundancy elimination algorithm that, like TCM, stops sinking computations in between the earliest point of BCM and the latest point of LCM. In FCM's case, the objective is to globally optimize register pressure in the face of large (i.e., nested) expressions.

Finally, it is worth comparing our approach briefly to other strength reduction algorithms that are not based on partial redundancy elimination: the classic algorithms of Allen [1969], Cocke and Kennedy [1977], and Allen et al. [1981], and the SSA-based approach of Cooper et al. [1995]. These approaches focus on the notions of induction variables and region constants. This causes some reduction in generality, with the extent of the reduction dependent on the specific definitions used by the particular strength reduction algorithm. For example, the strength reduction aspect of our introductory example could be handled by Allen et al. [1981] but not by Cooper et al. [1995], Allen [1969], or Cocke and Kennedy [1977]. A minor variation on the example would move it out of the range of Allen et al. [1981] as well, however, while leaving our own algorithm effective. Also, like most of the algorithms based on partial redundancy elimination, these strength reduction algorithms may introduce unsafe computations and cause performance loss, rather than gain.

ACKNOWLEDGMENTS

This article benefited from conversations with John Engebretson, and from his work on the proof of Theorem 1. Fred Chow and the anonymous referees provided extensive constructive criticism of an earlier draft, which was very helpful in the revision process.

REFERENCES

Allen, F. E. 1969. Program optimization. In *Annual Review in Automatic Programming, Vol. 5.* Pergamon Press.

Allen, F. E., Cocke, J., and Kennedy, K. 1981. Reduction of operator strength. In *Program Flow Analysis: Theory and Applications*, S. S. Muchnick and N. D. Jones, Eds. Prentice-Hall, Chapter 3, 79–101.

Bodík, R., Gupta, R., and Soffa, M. L. 1998. Complete removal of redundant expressions. *SIGPLAN Not. 33,* 5 (May), 1–14. Proceedings of the 1998 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI).

Chaitin, G. J. 1982. Register allocation and spilling via graph coloring. In *Proceedings of the ACM SIGPLAN 82 Symposium on Compiler Construction.* Association for Computing Machinery, 98–105.

Chaitin, G. J., Auslander, M. A., Chandra, A. K., Cocke, J., Hopkins, M. E., and Markstein, P. W. 1981. Register allocation via coloring. *Comput. Lang. 6*, 47–57.

Chow, F., Chan, S., Kennedy, R., Liu, S.-M., Lo, R., and Tu, P. 1997. A new algorithm for partial redundancy elimination based on SSA form. *SIGPLAN Not. 32,* 5 (May), 273–286. Proceedings of the 1997 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI).

Cocke, J. and Kennedy, K. 1977. An algorithm for reduction of operator strength. *Commun. ACM 20,* 11 (Nov.), 850–856.

Cooper, K., Simpson, T., and Vick, C. 1995. Operator strength reduction. Tech. Rep. CRPC-TR95635-S, Center for Research on Parallel Computation, Rice University. Oct.

DHAMDHERE, D. M. 1979. On an algorithm for operator strength reduction. *Commun. ACM 22,* 5 (May), 311–312.

DHAMDHERE, D. M. 1989. A new algorithm for composite hoisting and strength reduction optimisation. *Int. J. Comput. Math. 27*, 1–14, 31–32.

DHANESHWAR, V. M. AND DHAMDHERE, D. M. 1995. Strength reduction of large expressions. *J. Program. Lang. 3*, 95–120.

GUPTA, R., BERSON, D. A., AND FANG, J. Z. 1997. Resource-sensitive profile-directed data flow analysis for code optimization. In *Proceedings of the 30th Annual IEEE/ACM Symposium on Microarchitecture*. 358–368.

GUPTA, R., BERSON, D. A., AND FANG, J. Z. 1998. Path profile guided partial redundancy elimination using speculation. In *Proceedings of the IEEE International Conference on Computer Languages*. 230–239.

HORSPOOL, R. N. AND HO, H. C. 1997. Partial redundancy elimination driven by a cost-benefit analysis. In *Proceedings of the 8th Israeli Conference on Computer Systems and Software Engineering*. 111–118.

JOSHI, S. M. AND DHAMDHERE, D. M. 1982. A composite hosting-strength reduction transformation for global program optimization. *Int. J. Comput. Math. 11*, 21–41, 111–126.

KENNEDY, R., CHOW, F., DAHL, P., LIU, S.-M., LO, R., AND STREICH, M. 1998. Strength reduction via SSAPRE. In *International Conference on Compiler Construction*. Lecture Notes in Computer Science, vol. 1383. Springer-Verlag, 144–158.

KILDALL, G. A. 1973. A unified approach to global program optimization. In *Conference Record of the ACM SIGACT/SIGPLAN Symposium on Principles of Programming Languages*. ACM, New York, 194–206.

KNOOP, J. AND STEFFEN, B. 1991. Unifying strength reduction and semantic code motion. Tech. Rep. 91-28, Lehrstuhl für Informatik II, RWTH Aachen. This paper is an extended and revised version of the second part of Steffen et al. [1991].

KNOOP, J., RÜTHING, O., AND STEFFEN, B. 1993. Lazy strength reduction. *J. Program. Lang. 1,* 1, 71–91.

KNOOP, J., RÜTHING, O., AND STEFFEN, B. 1994. Optimal code motion: Theory and practice. *ACM Trans. Program. Lang. Syst. 16,* 4 (July), 1117–1155.

RÜTHING, O. 1998. Optimal code motion in the presence of large expressions. In *Proceedings of the IEEE International Conference on Computer Languages*. IEEE, New York, 215–226.

STEFFEN, B., KNOOP, J., AND RÜTHING, O. 1991. Efficient code motion and an adaption to strength reduction. In *Proceedings of the 4th International Joint Conference on Theory and Practice of Software Development*. Lecture Notes in Computer Science, vol. 494. Springer-Verlag, 394–415.

Received March 1998; accepted May 1998