Code-Size Sensitive Partial Redundancy Elimination

Oliver Rüthing, Jens Knoop, and Bernhard Steffen

Universität Dortmund, Baroper Str. 301, D-44221 Dortmund, Germany
{ruething,knoop,steffen}@ls5.cs.uni-dortmund.de

Abstract. Program optimization focuses usually on improving the run-time efficiency of a program. Its impact on the code size is typically not considered a concern. In fact, classical optimizations often cause code replication without providing any means allowing a user to control this. This limits their adequacy for applications, where code size is critical, too, like *embedded systems* or *smart cards*. In this article, we demonstrate this by means of *partial redundancy elimination* (*PRE*), one of the most powerful und widespread optimizations in contemporay compilers, which intuitively aims at avoiding multiple computations of a value at run-time. By modularly extending the classical PRE-approaches we develop a family of *code-size sensitive* PRE-transformations, whose members in addition to the two traditional goals of PRE (1) reducing the number of computations and (2) avoiding unnecessary register pressure, are unique for taking also (3) *code size* as a third optimization goal into account. Each of them optimally captures a predefined choice of priority between these three goals. The flexibility and aptitude of these techniques for size-critical applications is demonstrated by various examples.

1 Motivation

Partial redundancy elimination (*PRE*), also known as *code motion* (*CM*), is one of the most important and widespread optimizations in compilers (cf. [8]). Intuitively, it aims at avoiding unnecessary recomputations of values at run-time. Technically, this is achieved by storing the value of computations for later reuse in temporaries. Traditionally, PRE-techniques focus on reducing the number of computations performed at run-time to a minimum while keeping the lifetimes of introduced temporaries as small as possible in order to avoid unnecessary register pressure. State-of-the-art PRE-techniques achieve these goals even optimally, however, they are not space-sensitive: They can cause code replication without providing any means allowing a user to control this (cf. Figure 1 for illustration). This limits their adequacy for applications where code size is crucial, too, like embedded systems or smart cards.

In this article, which is based on the presentation of [10], we show how to add *code size* as a third optimization goal to partial redundancy elimination in addition to the two more classical goals of computation costs and register pressure. We arrive at a family of *sparse* code motion algorithms coming as modular extensions of the algorithms for *busy* and *lazy code motion* (cf. [3, 4]). Each algorithm of this family optimally captures a predefined choice of priority between these three optimization goals, e.g., the construction of code-size optimal programs of at least the same efficiency as the original program, or of computationally optimal programs of minimal code size, each with lowest register pressure. These algorithms are well-suited for size-critical application areas like smart cards and embedded systems, as they provide a handle to control the code replication problem of classical code motion techniques. We believe that our systematic, priority-based treatment of trade-offs between optimization goals may substantially decrease development costs of size-critical applications: Users may "play" with the priorities until the algorithm automatically delivers a satisfactory solution.

In the presentation here, we primarily focus on the intuition and the algorithmic essence of our approach to code-size sensitive PRE, which we illustrate by various examples. Proofs omitted, however, can be found in [10].

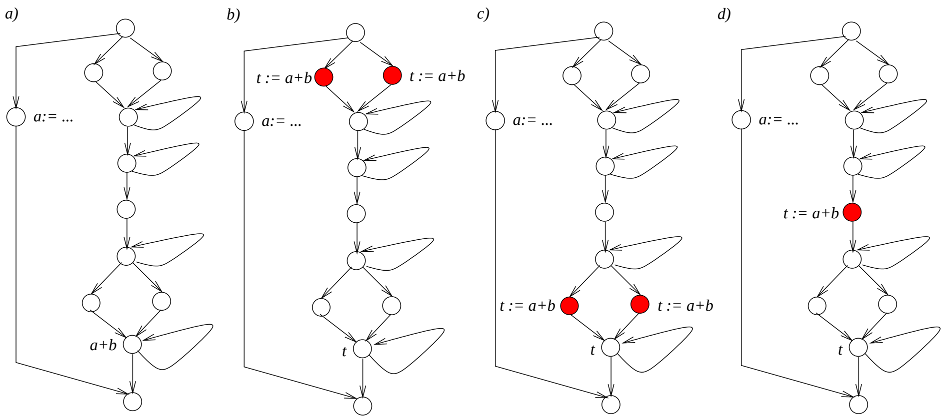


Fig. 1. (a) A program containing a loop invariant computation of $a+b$. (b) Computationally optimal (*busy code motion*), and (c) computationally and lifetime optimal code motion (*lazy code motion*) lead in this example both to code replication. (d) Computationally optimal code motion transformation being free of code replication.

2 Preliminaries

Flow graphs. We consider *directed flow graphs* $G=(N,E,\mathbf{s},\mathbf{e})$ with node set N and edge set E, where nodes represent elementary statements, edges the nondeterministic branching structure, and $\mathbf{s}$ and $\mathbf{e}$ the unique *start node* and *end node* of G, which are assumed to be free of any predecessors and successors, respectively. Additionally, $succ(n)$ and $pred(n)$ denote the set of all immediate successors and predecessors of a node n. A *finite path* in G is a sequence $\langle n_1,\dots,n_k\rangle$ of edges such that $(n_i,n_{i+1})\in E$ for $i\in\{1,\dots,k-1\}$. Every node $n\in N$ is assumed to lie on a path from $\mathbf{s}$ to $\mathbf{e}$.

As in [3] we assume that every edge leading from a branch node to a join node is split by inserting a synthetic node representing the empty statement skip. While splitting these so-called *critical edges* keeps the code motion process simple and more powerful (cf. [3]), the following assumption on node splitting is only for technical convenience. W. l. o. g. we assume that every node n is replaced by two copies, called *entry* and *exit* node n_N and n_X of n which are connected by an edge (n_N, n_X). Node n_N inherits the statement associated with n and n's predecessors, while n_X inherits n's successors and is associated with skip.[1] Assuming this "normal" form of a flow graph allows us to restrict ourselves to entry-placements, which simplifies the reasoning without restricting generality.

A predicate pr on n, i.e., $pr : N \to \mathcal{B}$, induces a set $S =_{df} \{n \in N \mid pr(n)\}$. On the other hand, a set $S \subseteq N$ induces a characteristic predicate on N defined by: $pr(n) \Longleftrightarrow_{df} n \in S$. We will make use of this duality throughout this article and liberally identify subsets of N and their corresponding characteristic predicates on N. Moreover, any function f with domain N is naturally extended to sets $M \subseteq N$ by defining: $f(M) =_{df} \bigcup_{n \in M} f(n)$.

Bipartite graphs. We consider undirected graphs (V, E) with vertices V and edges $\mathsf{E} \subseteq \mathfrak{P}_2(\mathsf{V})$.[2] The *neighbours* $\Gamma(\mathsf{v})$ of a vertex $\mathsf{v} \in \mathsf{V}$ are defined by $\Gamma(\mathsf{v}) =_{df} \{\mathsf{w} \mid \{\mathsf{v}, \mathsf{w}\} \in \mathsf{E}\}$. (V, E) is called *bipartite*, if there are two disjoint sets of vertices S and T such that $\mathsf{V} = \mathsf{S} \,\dot{\cup}\, \mathsf{T}$ and $\mathsf{e} \cap \mathsf{S} \neq \emptyset \ \wedge\ \mathsf{e} \cap \mathsf{T} \neq \emptyset$ for every edge e in E, where $\dot{\cup}$ denotes the disjoint union. For convenience, bipartite graphs will often be given in a notation $(\mathsf{S} \,\dot{\cup}\, \mathsf{T}, \mathsf{E})$, which already reflects the bipartition on the set of vertices. Moreover, we usually view S as the lower layer and T as the upper layer of the bipartite graph.

Tight sets. Let $(\mathsf{S} \,\dot{\cup}\, \mathsf{T}, \mathsf{E})$ be a bipartite graph and $\mathsf{S}' \subseteq \mathsf{S}$. The so-called *deficiency* of S', in symbols $\mathrm{defic}(\mathsf{S}')$, is defined by: $\mathrm{defic}(\mathsf{S}') =_{df} |\mathsf{S}'| - |\Gamma(\mathsf{S}')|$. If S' is of maximum deficiency among all subsets of S, then it is called a *tight set* (wrt S). Note that $\mathrm{defic}(\mathsf{S}') \geq 0$, if S' is a tight set, since $\mathrm{defic}(\emptyset) = 0$. Tight sets can be efficiently computed by means of matching based techniques as recalled later in this section.

Matchings. A set of edges $\mathsf{M} \subseteq \mathsf{E}$ is a *matching*, if $\mathsf{e}_1 \cap \mathsf{e}_2 = \emptyset$ for different members $\mathsf{e}_1, \mathsf{e}_2$ of M. A vertex v is *matched* by M, if $\mathsf{v} \in \mathsf{e}$ for some $\mathsf{e} \in \mathsf{M}$. M is a *maximum matching*, if $|\mathsf{M}| \geq |\mathsf{M}'|$ for any matching $\mathsf{M}' \subseteq \mathsf{E}$. Maximum matchings can efficiently be computed using techniques based on the construction of *augmenting paths*. These are paths between two unmatched vertices which are *alternating*, i.e., whose edges alternate between those which are part of a matching and those which are not. A straightforward algorithm with worst case time complexity $\mathcal{O}(|\mathsf{V}|\,|\mathsf{E}|)$, $\mathsf{V} =_{df} \mathsf{S} \,\dot{\cup}\, \mathsf{T}$, has been given in [1], a more sophisticated algorithm of complexity $\mathcal{O}(|\mathsf{V}|^{\frac{1}{2}}\,|\mathsf{E}|)$ in [2].

Computing tight sets. Given a maximum matching, tight sets can easily be computed by an iterative procedure. E.g., Algorithm 1, which evolves as a side-

[1] Throughout this article we suppress the node splitting in the figures in order to keep them as small as possible. For the examples shown, the node splitting is not relevant.

[2] $\mathfrak{P}_2(\mathsf{V})$ denotes the set of all two-elementary subsets of V. Hence, an edge of an undirected graph is a subset like $\{\mathsf{v}, \mathsf{w}\}$ with $\mathsf{v}, \mathsf{w} \in \mathsf{V}$.

product of the Galai-Edmonds decomposition of a bipartite graph [7], computes the largest tight set of a bipartite graph by successively removing vertices from an initial approximation.

Algorithm 1 (Computing largest tight sets).

Input: *Bipartite graph* $(\mathsf{S} \dot\cup \mathsf{T}, \mathsf{E})$*, maximum matching* M.
Output: *Largest tight set* $\mathcal{T}_L(\mathsf{S}) \subseteq \mathsf{S}$.

$\mathsf{S_M} := \mathsf{S};\ \mathsf{D} := \{\mathsf{t} \in \mathsf{T} \mid \mathsf{t}$ *is unmatched*$\}$;
`WHILE` $\mathsf{D} \neq \emptyset$ `DO`
    choose some $\mathsf{x} \in \mathsf{D}$; $\mathsf{D} := \mathsf{D} \setminus \{\mathsf{x}\}$;
    `IF` $\mathsf{x} \in \mathsf{S}$ `THEN` $\mathsf{S_M} := \mathsf{S_M} \setminus \{\mathsf{x}\};\ \mathsf{D} := \mathsf{D}\ \cup\ \{\mathsf{y} \mid \{\mathsf{x},\mathsf{y}\} \in \mathsf{M}\}$
            `ELSE` $\mathsf{D} := \mathsf{D}\ \cup\ (\Gamma(\mathsf{x}) \cap \mathsf{S_M})$ `FI OD`;
$\mathcal{T}_L(\mathsf{S}) := \mathsf{S_M}$

The function of Algorithm 1 is quite simple: Starting from an upper initial approximation of $\mathsf{S_M}$ those vertices which can be reached via an alternating path originating at an unmatched vertex of T are removed. Intuitively, this ensures that all detracted S-vertices are matched, as otherwise this would establish an augmenting path contradicting the maximality of M. Hence, the set of removed S-vertices is of negative deficiency. Formally, we have [9, 10]:

Theorem 1. *Algorithm 1 terminates with* $\mathcal{T}_L(\mathsf{S})$ *being the largest tight set, i.e., (1)* $\mathcal{T}_L(\mathsf{S})$ *contains any other tight set, and (2)* $\mathcal{T}_L(\mathsf{S})$ *is tight.*

It is easy to see that the complexity of Algorithm 1 is predominated by the process of determining a maximum matching. Actually, this also holds for the overall complexity of our algorithm for code-size sensitive code motion (cf. [10]).

Smallest tight sets can be computed in quite a similar fashion as has been shown in [9]. Intuitively, this dual algorithm, which can be found in [10], too, works by successively adding vertices until a fixed point is reached. In our approach, it will be used in order to take lifetimes of temporaries into account.

3 Code Motion

Given a term t, also called a *code motion candidate*, code motion can conceptually be considered a two-step transformation. First, inserting statements of the form $\mathbf{h} := t$ at some program points, where $\mathbf{h}$ is a fresh temporary associated with t. Second, replacing some of the original computations of t by $\mathbf{h}$. A code motion transformation is called *admissible*, if it is semantics and performance preserving.

It is well-known that under this admissibility constraint, computationally optimal results can be obtained by placing computations *as early as possible* in a program. This is known as the "earliestness" principle realized by *busy code motion* (cf. [3, 4]). Computationally and lifetime optimal results can be achieved by placing computations *as early as necessary* (in order to achieve computational optimality), *but as late as possible* (in order to keep lifetimes of temporaries as small as possible). This is known as the "latestness principle," which has first been realized by *lazy code motion* (cf. [3, 4]). As illustrated in Section 1, none of these placing strategies (nor any other placing strategy realized by a code motion transformation proposed so far) is space-sensitive. In contrast, their impact on the code size of a program is generally unpredictable. In the following section, we will show how to modularly extend the busy and lazy code motion transformations in order to arrive at a family of code-size sensitive code motion transformations.

4 Code-Size Sensitive Code Motion

Intuitively, the problem of code-size sensitive code motion can be considered a trade-off problem, where the original computations of the program are to be traded against newly inserted ones such that the resulting program is semantics and performance preserving, and code-size optimal. Here, we will show how this trade-off problem can essentially be reduced to the computation of tight sets on bipartite graphs. Conceptually important is the notion of *down-safety regions*. They allow a precise characterization of admissible code motion transformations.

Down-Safety Regions. Intuitively, a *down-safety region* is a set of down-safe, but not up-safe program points satisfying some additional closure constraints.[3] Their definition relies on the notion of a *down-safety closure* $\rho(n)$, $n \in \mathit{DnSafe} \setminus \mathit{UpSafe}$. This is defined as the smallest set of nodes satisfying the following three properties, where the predicate *Comp* is enjoyed by nodes containing an occurrence of the code motion candidate under consideration:

1. $n \in \rho(n)$
2. *Closedness wrt successors*: $\forall\, m \in \rho(n) \setminus \mathit{Comp}.\ \mathit{succ}(m) \subseteq \rho(n)$
3. *Homogeneity*: $\forall\, m \in \rho(n).\ \mathit{pred}(m) \cap \rho(n) \neq \emptyset \ \Rightarrow\ \mathit{pred}(m) \setminus \mathit{UpSafe} \subseteq \rho(n)$

Intuitively, down-safety closures of nodes allow us to characterize the set of nodes to be considered during code motion. *Down-safety regions* are essentially sets of down-safe program points being closed under ρ. Formally, a set of nodes $\mathcal{R} \subseteq N$ is a *down-safety region* if and only if it meets the following two constraints:

1. $\mathit{Comp} \setminus \mathit{UpSafe} \subseteq \mathcal{R} \ \subseteq\ \mathit{DnSafe} \setminus \mathit{UpSafe}$ 2. $\rho(\mathcal{R}) \ =\ \mathcal{R}$

In the following, we abbreviate $\mathit{Comp} \setminus \mathit{UpSafe}$ by *RelComp*. This is motivated by the fact that only computations which are not totally redundant, i.e., which are not up-safe, are relevant for the placing strategy. Totally redundant ones can always be eliminated. Note that *RelComp* defines the smallest down-safety region, while $\mathit{DnSafe} \setminus \mathit{UpSafe}$ the largest one.

Down-safety regions are important because they allow a precise characterization of semantics and performance preserving code motion transformations.

[3] Down-safety and up-safety are also known as *very busyness* and *availability* (cf. [4]).

Given a down-safety region $\mathcal{R}$, its *earliestness-frontier* is intuitively the set of its entry points, its *body* are its remaining nodes (cf. Figure 2 for illustration). Formally: $\mathit{Earliest}_{\mathcal{R}}(n) \Longleftrightarrow_{df} n \in \mathcal{R} \wedge \big((n = \mathbf{s}) \vee \exists\, m \in \mathit{pred}(n).\ \neg \mathit{Transp}(m) \vee m \notin \mathcal{R} \cup \mathit{UpSafe}\big)$. With this definition, earliestness frontier and body of a down-safety region $\mathcal{R}$ are defined by: $\mathit{EarliestFrontier}_{\mathcal{R}} =_{df} \mathit{Earliest}_{\mathcal{R}}$ and $\mathit{Body}_{\mathcal{R}} =_{df} \mathcal{R} \backslash \mathit{EarliestFrontier}_{\mathcal{R}}$.

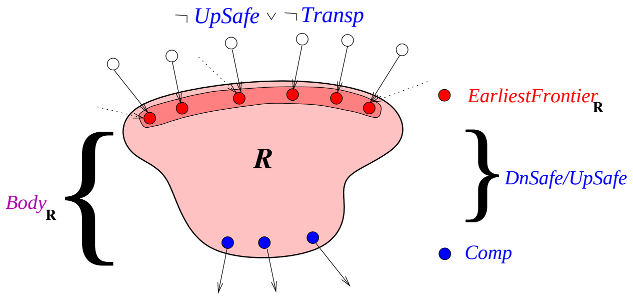


Fig. 2. Illustrating earliestness frontier and body of a down-safety region $\mathcal{R}$.

Theorem 2 (Semantics and Performance Preserving Code Motion). *A code motion transformation CM (replacing all original computations) is semantics and performance preserving if and only if there is a down-safety region $\mathcal{R}$ such that the insertion points of CM are given by the earliestness frontier of $\mathcal{R}$, i.e., $\mathit{Insert}_{CM} = \mathit{Earliest}_{\mathcal{R}}$.*

4.1 Constructing the Bipartite Graph

In this section we show how to model our code-minimization problem in terms of a bipartite graph. Intuitively, it models all possibilities of constructing a down-safety region. Formally, this bipartite graph $\mathcal{B}_{\mathsf{DS}} = (\mathsf{S}_{\mathsf{DS}} \,\dot{\cup}\, \mathsf{T}_{\mathsf{DS}}, \mathsf{E}_{\mathsf{DS}})$ is constructed as follows, where *Earliest* denotes the earliest safe program points, i.e., the insertion points of the busy code motion transformation (cf. [3, 4]):

1. For each node in $\mathit{DnSafe} \setminus (\mathit{UpSafe} \cup \mathit{Earliest})$ there is a corresponding (new) vertex in S_{DS}.[4]
2. For each vertex in $\mathit{DnSafe} \setminus (\mathit{UpSafe} \cup \mathit{Comp})$ there is a corresponding (new) vertex in T_{DS}.[4]
3. For each vertex $n \in \mathsf{S}_{\mathsf{DS}}$ and $m \in \mathsf{T}_{\mathsf{DS}}$: $\{n, m\} \in \mathsf{E}_{\mathsf{DS}} \Longleftrightarrow_{df} m \in \rho(\mathit{pred}(n))$.

S_{DS} and T_{DS} define the lower and upper layer of $\mathcal{B}_{\mathsf{DS}}$. Intuitively, nodes of S_{DS} are to be traded against nodes of T_{DS}. A node $n \in N$, which belongs to S_{DS} is connected to all nodes in T_{DS}, which at least have to be added to a down-safety region whenever a predecessor of n is added to the region, too. Starting with the

[4] We shall identify vertices of the bipartite graph with their corresponding nodes in the flow graph whenever their membership in S_{DS} or T_{DS} is unambiguous from the context. Note, however, that flow graph nodes have different representations in S_{DS} and T_{DS}, as S_{DS} and T_{DS} are required to be disjoint.

relevant original computation points, one can successively construct any down-safety region by trading points of the current earliestness frontier against earlier ones. Note that the earliest nodes, i.e., those in the frontier of the whole range of down-safe program points, are not part of the lower layer as they cannot be traded in this way.

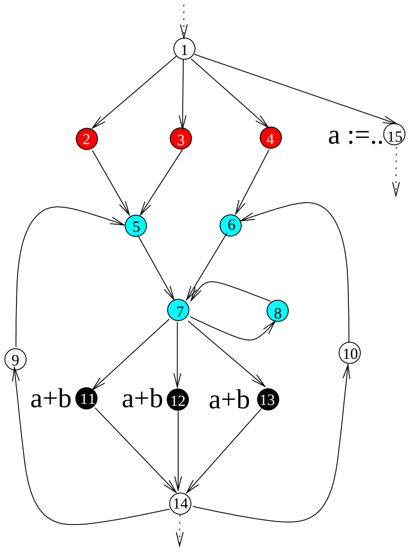


Fig. 3. Running example.

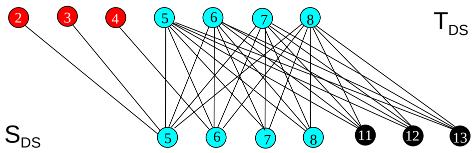


Fig. 4. The bipartite graph for Fig. 3.

Having constructed the bipartite graph, we compute a maximum matching on it, and thereafter the smallest or largest tight set. Next, we show how to exploit this information for the transformation, and the properties it enjoys.

4.2 Main Results

In this section we summarize the main results on our approach for code-size sensitive code motion. Theorem 3 says that a tight set TS induces a down-safety region with body TS. According to Theorem 2, the corresponding earliestness frontier of TS induces a semantics and performance preserving code motion transformation. Given a tight set, the first part of Theorem 4 allows us to determine the earliestness frontier of the down-safety region induced by TS only by inspecting TS and the bipartite graph underlying it. The second part of Theorem 4 gives us the code-size gain resulting from the code motion transformation induced by TS assuming that all original computations are replaced. It is given by the deficiency of TS. As this is maximal because TS is tight, this implies that the space gain is maximal, too. Moreover, it is positive as the deficiency of the empty set, which is 0, is a lower bound for the deficiency of a tight set. Together this means, that there is no code replication, and that the code size of the resulting program is indeed minimal. This is summarized in Theorem 5.

Theorem 3 (Tight Sets). *Let* $\mathsf{TS} \subseteq \mathsf{S}_{\mathsf{DS}}$ *be a tight set. Then* $\mathcal{R}_{\mathsf{TS}} =_{df} \Gamma(\mathsf{TS}) \cup (Comp \setminus UpSafe)$ *is a down-safety region with* $Body(\mathcal{R}_{\mathsf{TS}}) = \mathsf{TS}$.

Theorem 4 (Down-Safety Regions). *Let $\mathcal{R}$ be a down-safety region. Then*

1. $\mathit{EarliestFrontier}_{\mathcal{R}} = ((\mathit{Comp} \setminus \mathit{UpSafe}) \cup \Gamma(\mathit{Body}_{\mathcal{R}})) \setminus \mathit{Body}_{\mathcal{R}}$
2. $|\mathit{Comp} \setminus \mathit{UpSafe}| - |\mathit{EarliestFrontier}_{\mathcal{R}}| = \mathit{defic}(\mathit{Body}_{\mathcal{R}})$

Theorem 5 (Optimality). *Let $\mathsf{TS} \subseteq \mathsf{S_{DS}}$ be a tight set, let $\mathcal{R}_{\mathsf{TS}}$ be the down-safety region induced by TS, and let $\mathit{Insert}_{SpCM} =_{df} \mathit{EarliestFrontier}_{\mathcal{R}_{\mathsf{TS}}} = \mathcal{R}_{\mathsf{TS}} \setminus \mathsf{TS}$ be the set of program points of the earliestness frontier of $\mathcal{R}_{\mathsf{TS}}$. Then the program resulting from the induced code motion transformation (i.e., inserting at $\mathit{EarliestFrontier}_{\mathcal{R}}$ and replacing all original computations) is* code-size *minimal, and the code-size gain over the original program equals $\mathit{defic}(\mathsf{TS}) =_{df} |\mathsf{TS}| - |\Gamma(\mathsf{TS})| \geq 0$ being maximal.*

Note that the code-size gain does not depend on whether the transformation is based on the largest or smallest tight set. In fact, this choice only affects the ranking of computational and lifetime quality. Intuitively, the larger the tight set is, the larger its induced down-safety region is. The larger the down-safety region is, the "earlier" its earliestness frontier is, and the "earlier" the insertions of computations take place. Hence, the use of largest tight sets ranks computational quality higher than lifetime quality according to the earliestness principle, while this is vice versa for smallest tight sets according to the latestness principle. Figure 4.2 illustrates this for the running example. The program of part a) shows the down-safety region and earliestness frontier induced by the largest tight set, while part b) shows this for the smallest tight set. Note that there is in fact no difference in code-size between the two programs. However, the transformation of Figure 4.2(a) optimizes computational quality as second order goal, while the one of Figure 4.2(b) does this for lifetime quality.

4.3 Sparse Code Motion: The SpCM-Algorithm at a Glance

In this section we present the general pattern of our code-size sensitive CM-algorithm called SpCM for a computation t and a program G. It is as follows:

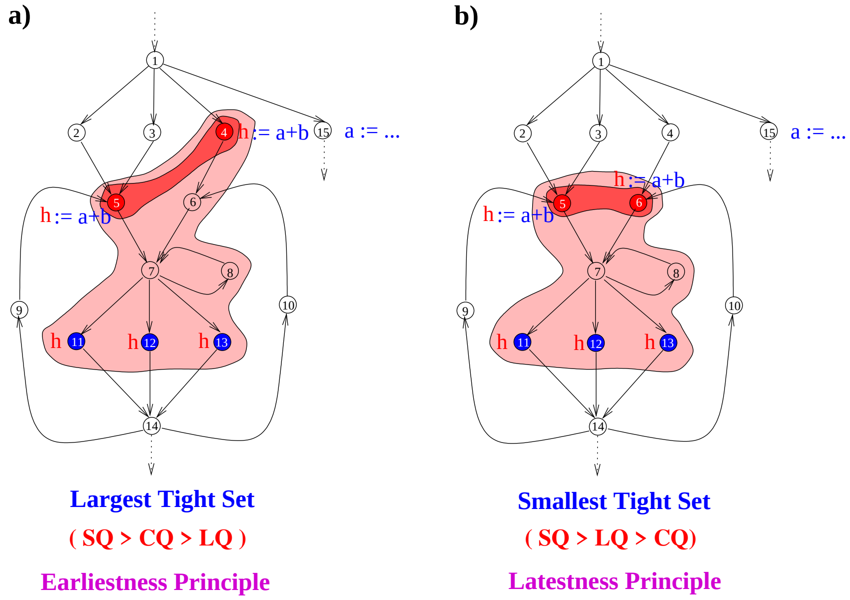
- **Preprocess**
  - *Optionally:* Perform lazy code motion (**LCM**) on G obtaining $G_{\mathbf{LCM}}$.
  - Compute the predicates required by busy code motion (**BCM**) (i.e., up-safety and down-safety) either for the original program G or, if the optional step has been executed, for $G_{\mathbf{LCM}}$.
- **Main Process**
  - **Reduction Phase**
    - Construct the bipartite graph $\mathcal{B}_{\mathsf{DS}}$.
    - Compute a maximum matching on $\mathcal{B}_{\mathsf{DS}}$.
  - **Optimization Phase**
    - Compute the largest/smallest tight set of $\mathcal{B}_{\mathsf{DS}}$.
    - Determine the insertion points according to Theorem 5(a), and replace all original computations by the temporary associated to t.

Choice of Priority	Apply	To	Using	Yields	Auxiliary Information Required
$\mathcal{LQ}$	**Not meaningful: The identity, i.e., G itself is optimal!**				
$\mathcal{SQ}$	**Subsumed by $\mathcal{SQ} > \mathcal{CQ}$ and $\mathcal{SQ} > \mathcal{LQ}$!**				
$\mathcal{CQ}$	**BCM**	G			UpSafe(G), DnSafe(G)
$\mathcal{CQ} > \mathcal{LQ}$	**LCM**	G		**LCM**(G)	UpSafe(G), DnSafe(G), Delay(G)
$\mathcal{SQ} > \mathcal{CQ}$	**SpCM**	G	**Largest tight set**	$\mathbf{SpCM}_{\mathsf{LTS}}$(G)	UpSafe(G), DnSafe(G)
$\mathcal{SQ} > \mathcal{LQ}$	**SpCM**	G	**Smallest tight set**		UpSafe(G), DnSafe(G)
$\mathcal{CQ} > \mathcal{SQ}$	**SpCM**	**LCM**(G)	**Largest tight set**		UpSafe(G), DnSafe(G), Delay(G) UpSafe(**LCM**(G)), DnSafe(**LCM**(G))
$\mathcal{CQ} > \mathcal{SQ} > \mathcal{LQ}$	**SpCM**	**LCM**(G)	**Smallest tight set**		UpSafe(G), DnSafe(G), Delay(G) UpSafe(**LCM**(G)), DnSafe(**LCM**(G))
$\mathcal{SQ} > \mathcal{CQ} > \mathcal{LQ}$	**SpCM**	**DL**($\mathbf{SpCM}_{\mathsf{LTS}}$(G))	**Smallest tight set**		UpSafe(G), DnSafe(G), Delay($\mathbf{SpCM}_{\mathsf{LTS}}$(G)), UpSafe(**DL**($\mathbf{SpCM}_{\mathsf{LTS}}$(G))), DnSafe(**DL**($\mathbf{SpCM}_{\mathsf{LTS}}$(G)))

The complexity of the complete algorithm — given by $\mathcal{O}(|N|^{\frac{5}{2}})$ — is determined by the complexity of computing the maximum matching (cf. [10]).

The table above, where $\mathcal{CQ}$, $\mathcal{LQ}$, $\mathcal{SQ}$ stand for computational, lifetime and code size quality, respectively, summarizes the SpCM-variants that can be derived from this pattern, and that differ in the prioritization of goals they support. Note that some prioritizations do not show up in this table. This is because some of them are not meaningful. Essentially, this is because the requirement of lifetime optimality determines the result and hence the transformation uniquely. Considering e.g. lifetime quality as the first order goal causes that the identity transformation is the unique optimal transformation. Obviously, this is not intended when dealing with optimization transformations.

The third and fourth row of this table summarize the prioritizations handled by classical PRE. The prioritizations of the following rows require the application of our code-size sensitive approach to PRE. In each case the last column lists the unidirectional bitvector analyses (also known as GEN/KILL-analyses) required as auxiliary information. Note that except for the prioritization shown in the last row, the other ones result simply from switching between largest and smallest tight sets, and applying sparse code motion to the original program, or the one resulting from lazy code motion. In the first case, code size is ranked higher than computational quality, while this is vice versa in the second case. Intuitively, this is because sparse code motion is performance preserving. Hence, starting with the computationally optimal program resulting from lazy code motion, it comes up with a computationally optimal program of minimal code size.



The prioritization shown in the last row requires two applications of sparse code motion. First, SpCM is applied to the original program G with respect to the largest tight set. This yields the program $SpCM_{\mathsf{LTS}}(G)$. According to the earliestness principle it is computationally best among the programs of minimal size. Among those, however, it is the one being life-time worst. The uniquely determined life-time best one is now computed by first subjecting $SpCM_{\mathsf{LTS}}(G)$ to the delayability analysis known from lazy code motion (cf. [3, 4]). This yields the program $\mathbf{DL}(SpCM_{\mathsf{LTS}}(G))$ preserving the computational quality, but possibly enlarging the code size. The final step then reestablishes code-size optimality while preserving computational quality and keeping life-times under the preceding restrictions as small as possible. This means, the computationally best code-size optimal program with minimal register pressure results finally from applying sparse code motion to $\mathbf{DL}(SpCM_{\mathsf{LTS}}(G))$ with respect to the smallest tight set.

5 Conclusions

Traditionally, code motion algorithms focus on dynamic aspects of performance optimization, which are summarizable by the terms of computational quality and register pressure. Static criteria like code size are not taken into account. In fact, the state-of-the-art code motion algorithms do not provide any control on their impact on code size. They can cause code replication in a completely unpredictable manner, which limits their adequacy for size-critical application areas like smart cards and embedded systems.

We have therefore added a third dimension to partial redundancy elimination, by considering *code size* as a further optimization goal in addition to the more traditional goals of computation costs and register pressure. This has resulted in a family of *sparse* code motion algorithms, each optimally capturing a predefined choice of priority between these three optimization goals, e.g. code size can be minimized while (1) guaranteeing at least the performance of the argument program, or (2) even computational optimality. Minimal register pressure is bound to be of least priority, as it typically uniquely determines a code motion transformation under the given circumstances.

Currently, we are exploring the full spectrum of the three-dimensional space of optimization goals, in order to better understand the impact of specific choices of priority. As indicated in Section 4.3, not all of them make sense, but most of them will have a justification in specific application scenarios. Moreover, we are investigating extensions to further optimization techniques like *partial dead-code elimination* [5] and *assignment motion* [6].

References

1. A. V. Aho, J. E. Hopcroft, and J. D. Ullman. *Data Structures and Algorithms*. Addison-Wesley, Reading, Massachusetts, 1983.
2. J. E. Hopcroft and R. M. Karp. An $n^{\frac{5}{2}}$ algorithm for maximum matchings in bipartite graphs. *SIAM Journal of Computing*, 2(4):225 – 331, 1973.
3. J. Knoop, O. Rüthing, and B. Steffen. Lazy code motion. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation* (*PLDI'92*) (*San Francisco, California*), volume *27*,7 of *ACM SIGPLAN Notices*, pages 224 – 234, 1992.
4. J. Knoop, O. Rüthing, and B. Steffen. Optimal code motion: Theory and practice. *ACM Transactions on Programming Languages and Systems*, 16(4):1117–1155, 1994.
5. J. Knoop, O. Rüthing, and B. Steffen. Partial dead code elimination. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation* (*PLDI'94*) (*Orlando, Florida*), volume *29*,6 of *ACM SIGPLAN Notices*, pages 147 – 158, 1994.
6. J. Knoop, O. Rüthing, and B. Steffen. The power of assignment motion. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation* (*PLDI'95*) (*La Jolla, California*), volume *30*,6 of *ACM SIGPLAN Notices*, pages 233 – 245, 1995.
7. L. Lovász and M. D. Plummer. Matching theory. *Annals of Discrete Mathematics*, 29, 1986.
8. E. Morel and C. Renvoise. Global optimization by suppression of partial redundancies. *Communications of the ACM*, 22(2):96 – 103, 1979.
9. O. Rüthing. *Interacting Code Motion Transformations: Their Impact and Their Complexity*. PhD thesis, Institut für Informatik und Praktische Mathematik, Christian-Albrechts-Universität zu Kiel, Kiel, Germany, 1997. Lecture Notes in Computer Science, vol. 1539, Springer-Verlag, Heidelberg, 1998.
10. O. Rüthing, J. Knoop, and B. Steffen. Sparse code motion. In *Conference Record of the 27th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (*POPL 2000*) (*Boston, Massachusetts*), pages 170 – 183. ACM, New York, 2000.