Faster Lifetime-optimal Speculative Partial Redundancy Elimination for Goto-free Programs

Xuran Cai, Amir Kafshdar Goharshady

Faster Lifetime-optimal Speculative Partial Redundancy Elimination for Goto-free Programs

Xuran Cai and Amir Goharshady

Hong Kong University of Science and Technology
Clear Water Bay, Hong Kong, China
xcaiay@connect.ust.hk, goharshady@cse.ust.hk

Abstract. Lifetime-optimal Speculative Partial Redundancy Elimination (LOSPRE) is one of the most classical, ubiquitous and effective techniques used by compilers for redundancy elimination, i.e. avoiding unnecessary recomputations of the same expression. State-of-the-art methods for LOSPRE over structured programs are based on treewidth, i.e. they first compute a tree decomposition of the control-flow graph of the program and then perform dynamic programming on this decomposition. In this work, we consider a different decomposition approach which is called series-parallel-loop (SPL) and was recently introduced in [8]. We present an efficient linear-time LOSPRE algorithm that builds upon SPL decompositions. We then provide extensive experimental results over the Small Device C Compiler (SDCC) benchmarks, demonstrating that our algorithm outperforms the highly-optimized treewidth-based approach of SDCC.

Keywords: LOSPRE · Structured Programs · Compiler Optimization · Control-flow Graphs · Graph Decompositions.

1 Introduction

RE and its Extensions. Redundancy elimination (RE), i.e. avoiding repeated and unnecessary computations of the same expression, has been a goal of optimizing compilers since their early days. Put simply, if the same expression e is used in several different locations in a program, it might be beneficial to compute e once, store it in a temporary variable, and then use it whenever the program reaches any of the locations that need e. One of the first formalizations of this problem was provided in 1970 as Global Common Subexpression Elimination (GCSE) [17]. Later approaches considered removing redundancies that appear only in a subset of paths of the control-flow graph, leading to Partial Redundancy Elimination (PRE) [37]. An enhancement to PRE, introduced by Lazy Code-Motion (LCM) [33], focuses on achieving lifetime optimality by minimizing the lifetimes of the temporary variables it introduces. This is also helpful for reducing register pressure. Another classical improvement is that of Speculative PRE

(SPRE) [7,28], which selects the path for adding computations based on profiling information with the goal of maximizing the benefits of PRE. Putting the ideas of LCM and SPRE together leads to Lifetime-Optimal SPRE (LOSPRE), which is currently the most expressive approach to redundancy elimination and subsumes all other methods mentioned above.

Example. Consider the C function in Figure 1 (left). If the compiler directly compiles the code as written by the programmer without any optimization, it will lead to the intermediate representation shown in Figure 1 (center). Note that the gray nodes in this IR all compute the same expression `a+b`. Applying LOSPRE to this expression will lead to the IR shown in Figure 1 (right). In this case, `a+b` is computed only once and saved in a temporary variable `temp`. Then, all future uses of `a+b` are replaced by `temp`. Naturally, this optimization is sound only if there are no changes to the values of `a` and `b` between successive uses of `temp`. Moreover, it comes at a cost. Adding extra computations might change the code size and keeping temporary variables may increase register pressure, affecting performance. Such costs can be formalized and minimized by LOSPRE. See Section 2 for a more formal treatment.

```
int f(int a, int b){
 if(a+b>3){
  return a+b-3;
 }
 else {
  return a+b;
 }
}
```

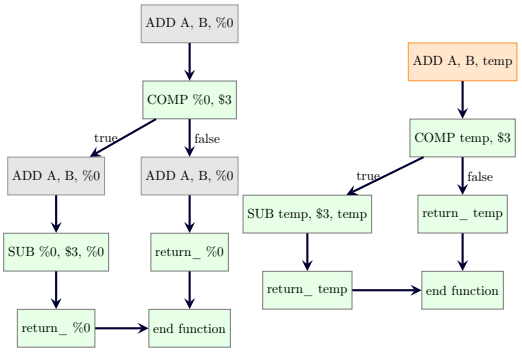

Fig. 1: A simple C function and its intermediate representation (IR) before and after optimization.

Sparsity of CFGs. A control-flow graph (CFG) is simply a graph that has one vertex corresponding to each statement of the program and a directed edge between two vertices whenever their corresponding statements may be executed successively. For example, in Figure 1, we have shown the IRs as CFGs. Some applications use a coarser definition of CFG, in which every vertex corresponds to a basic block of program statements. In both cases, it is well-known that control-flow graphs of real-world programs are sparse, resemble trees and can be decomposed into sets of vertices of size at most 8 that are connected to each other in a tree-like manner. More formally, CFGs of goto-free structured programs have a treewidth of at most 7 [41]. This celebrated result has been

used in a wide variety of program analysis, compiler optimization and model-checking problems [24], which rely on the small treewidth property to obtain faster algorithms for μ-calculus model-checking [38], data-flow analysis [12,27], MDPs [16,3,1], reachability analysis [40,10], algebraic program analysis [19,15,11], register allocation [35,5,41], optimizing cryptocurrency miners' revenues [36,4], cache management [2,13], reliability [26], chemical descriptors [18] and equality saturation [25]. The original bound in [41] applied to Pascal and C programs, but further works have extended it to other languages, such as Ada [6], Java [29,14] and Solidity [9], as well as to path decompositions [20]. There are also negative results, showing that bounded treewidth does not always help in verification [23]. Finally, the recent work [8] provides a new notion of decomposition, called series-parallel-loop (SPL), which exactly captures the set of control-flow graphs of structured goto-free programs and formalizes their sparsity.

Algorithms for LOSPRE. LOSPRE was initially solved by the min-cut-PRE (MC-PRE) algorithm of [42], which has a runtime of $O(n^3)$, where n is the number of nodes in the program's control-flow graph. The same work also shows that MC-PRE is equivalent to solving a weighted minimum cut problem on a directed graph. Thus, applying Karger's classical algorithm [32] leads to an improved runtime bound of $O(n^2 \log^3(n))$. An almost-quadratic runtime is considered too slow for many practical scenarios, especially just-in-time compilation [34,30,39]. This led to the development of suboptimal approximation approaches [30,39] or methods such as [31] that is empirically shown to work faster in practice while having the same asymptotic worst-case complexity. A breakthrough in LOSPRE was achieved by [34] in 2021, which provided a novel approach based on tree decompositions and treewidth. The algorithm of [34] considers the control-flow graph G of the program, computes a tree decomposition of G and then performs dynamic programming on this tree decomposition, leading to a LOSPRE solution that takes $O(\mathtt{tw}(G) \cdot 2^{\mathtt{tw}(G)} \cdot n)$ time, where n is the number of vertices in the CFG and $\mathtt{tw}(G)$ is its treewidth.

Our Contribution. In this work, we consider the problem of LOSPRE over structured goto-free programs. Our approach builds upon and extends the ideas of both [34] and [8] to obtain a faster linear-time algorithm for LOSPRE. More specifically, we exploit the sparsity of CFGs in order to design an algorithm with $O(n)$ runtime. However, unlike [34], we do not use tree decompositions. Instead, we use SPL decompositions, which were defined in [8]. This leads to a much simpler algorithm since SPL decompositions exactly capture the set of CFGs, i.e. we are not solving the problem on a larger set of graphs than necessary. Additionally, SPL decompositions do not ignore the directions of edges in the CFG. This simplicity pays off in practice. We provide extensive experimental results over the Small Device C Compiler (SDCC), which is a highly-optimized compiler using [34], and show that our approach obtains significant performance improvements over [34].

Organization. Section 2 provides a formal definition of the LOSPRE problem, following [34]. Section 3 is an overview of the SPL decomposition method of [8].

This is followed by our new LOSPRE algorithm in Section 4. Finally, Section 5 provides an experimental comparison of our approach and [34] over the SDCC benchmarks.

2 LOSPRE

In this section, we present LOSPRE. We use the terminology and notation of [34]. LOSPRE is an intraprocedural analysis that considers a single function of the program, modeled as a control-flow graph (CFG) $G = (V, E)$. For example, the two graphs of Figure 1 are CFGs. A CFG always has a single entry node and a single exit node. The entry node corresponds to the beginning of the function and the exit node to its termination. The entry node has no incoming edge. Conversely, the exit node has no outgoing edge.

Use Sets. Consider an expression e. We define the *use set* U of e as the set of all nodes of the CFG in which the expression e is computed.

Life Sets. Our goal is to precompute the expression e at a few points, save the result in a temporary variable `temp`, and then use `temp` in place of e in every node of U. We denote the lifetime of the variable `temp` by L and call it our *life set*.

Invalidating Set. We say a node v of the CFG invalidates e if the statement at v changes the value of e. For example, if $e =$ `a+b`, then the statement `a = 0` invalidates e. We denote the set of all invalidating nodes by I. These nodes play a crucial role in LOSPRE since they force us to update the value saved in `temp` by recomputing e. We assume that the entry and exit nodes are invalidating since LOSPRE is an intraprocedural analysis that has no information about the program's execution before or after the current function.

Calculation Set. Given the sets U, L and I above, we have to make sure the value of our temporary variable `temp` is correct at every node in $U \cup L$. Thus, for every edge $(x, y) \in E$ of the CFG where $x \notin L$ and $y \in U \cup L$, we have to insert a computation `temp` $= e$ between x and y. Similarly, if $x \in I$, then the value stored at `temp` becomes invalid after the execution of x, requiring us to inject the same computation between x and y. Formally, the computation `temp` $= e$ has to be injected into the following set of edges of the CFG:

$$C(U, L, I) = \{(x, y) \in E \mid x \notin L \setminus I \ \wedge \ y \in U \cup L\}.$$

We call this the *calculation set*.

Example. Figure 2 shows an example of LOSPRE. The top part of the figure is a CFG in which the use set of an expression e is shown in gray, i.e. we need the value of e at vertices $U = \{2, 4, 5, 7\}$. The invalidating set is shown in orange, i.e. the vertices in $I = \{1, 6, 8\}$ invalidate e. The middle and bottom parts each show one possible optimization. We show the lifetime of our temporary variable in green.

In the middle part, the temporary variable is alive at $\{2, 3\}$. Thus, the computation $\texttt{temp} = e$ has to be injected into the edge $(1, 2)$. We can then use `temp` instead of e in locations $2, 4$ and 5. However, we need to recompute e in the edge $(6, 7)$. In this case our computation set is $\{(1, 2), (6, 7)\}$. The edges in the computation set are shown in blue.

In the bottom part, the temporary variable is alive only at position 3. Thus, we first compute e when passing through $(1, 2)$ so that we have its value at 2. We then recompute e when going through $(2, 3)$ and save it at a temporary variable `temp`. This temporary variable is then used in place of e in 4 and 5. This example shows a tradeoff in which fewer repetitions of the computation lead to a longer lifetime for the temporary variable, which increases register pressure and is undesirable for register allocation.

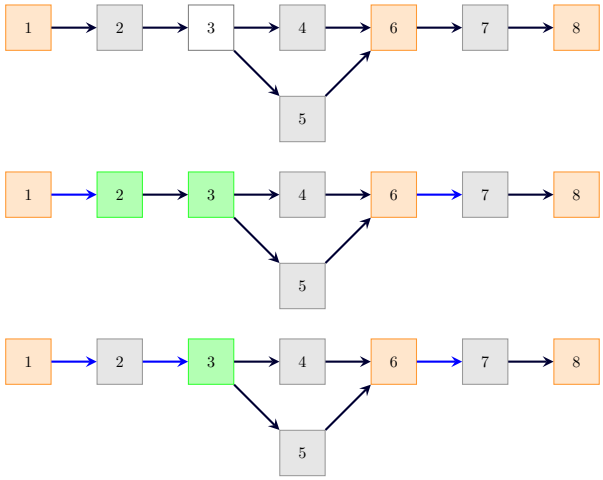

Fig. 2: An example of LOSPRE. The use set is shown in gray, the invalidating set in orange, and the life set in green. The edges of the calculation set are shown in blue.

Costs. There are two types of costs associated with the process above: (i) injecting calculations into the edges in $C(U, L, I)$ and (ii) keeping an extra variable `temp` at every node in L. These costs are dependent on the goals pursued by the compiler. For example, a compiler aiming to minimize code size will focus on (i). On the other hand, if our goal is to ease register pressure, we would want to minimize (ii). LOSPRE is an expressive framework in which these costs are modeled by two functions

$$c : E \to K$$

and

$$l : V \to K.$$

where K is a totally-ordered set with an addition operator, c is a function that maps each edge to the cost of adding a computation of e in that edge and l is similarly a function that maps each vertex of the CFG to the cost of keeping the temporary variable `temp` alive at that vertex.

Based on the discussion above, we are now ready to define our main problem.

Definition 1 (LOSPRE). *Given a CFG $G = (V, E)$, a use set U, an invalidating set I and two cost functions $c : E \to K$ and $l : V \to K$, the LOSPRE problem is to find a life set L that minimizes the total cost*

$$\textsc{Cost}(G, U, I, L, c, l) = \sum_{e \in C(U,L,I)} c(e) + \sum_{v \in L} l(v).$$

Examples of Cost Functions. Suppose our goal is to optimize for execution time. We use profiling to find frequencies of execution for each edge. We then set $K = \mathbb{R}$ and $c(x, y)$ to the frequency with which the edge (x, y) was executed. Finally, we set $l(x) = 0$ for all vertices x. As another example, suppose that we optimize for code size but also want to achieve lifetime-optimality, i.e. not keeping the temporary variable alive when it is not necessary. In this case, we let $K = \mathbb{R}^2$ (using lexicographic ordering) and assign $c(x, y) = (1, 0)$ to every edge (x, y) and $l(x) = (0, 1)$ to every vertex x.

3 SPL Decompositions

In this work, we build our algorithm on top of a decomposition method introduced in [8]. This decomposition method is called SPL (series-parallel-loop) and is an extension of series-parallel graphs with an extra loop operation. It is shown in [8] that a graph is a CFG of a structured program if and only if it has an SPL decomposition.

Structured Programs [41]. We say a program is structured if it can be generated using the following grammar:

$$\begin{aligned} P := {} & \epsilon \mid \texttt{break} \mid \texttt{continue} \mid P; P \\ & \mid \texttt{if } \varphi \texttt{ then } P \texttt{ else } P \texttt{ fi} \mid \texttt{while } \varphi \texttt{ do } P \texttt{ od}. \end{aligned} \tag{1}$$

Here, ϵ is any atomic operation that has no effect on control flow, such as an assignment to a variable. It is easy to define other structures such as `for` and `switch` as syntactic sugar. See [41] for details. We say a program generated by the grammar above is *closed* if every `break` and `continue` statement appears inside a `while` loop's body.

SPL Graphs [8]. An SPL graph $G = (V, E, S, T, B, C)$ is a directed graph (V, E) with four distinct special nodes $S, T, B, C \in V$, which are respectively called the *start*, *terminate*, *break* and *continue* nodes, generated by the grammar below:

$$G := A_\epsilon \mid A_{\texttt{break}} \mid A_{\texttt{continue}} \mid G \otimes G \mid G \oplus G \mid G^{\circledast} \tag{2}$$

We now explain the operations in this grammar.

Atomic SPL graphs. There are three different atomic SPL graphs: A_ϵ, $A_{\texttt{break}}$, and $A_{\texttt{continue}}$. All of them contain only the four special nodes and only one edge as shown in Figure 3.

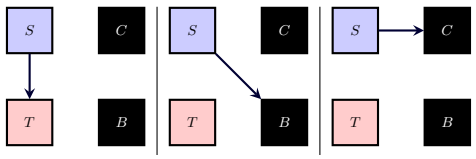

Fig. 3: Atomic SPL graphs: A_ϵ (left), $A_{\texttt{break}}$ (middle), and $A_{\texttt{continue}}$ (right) [8].

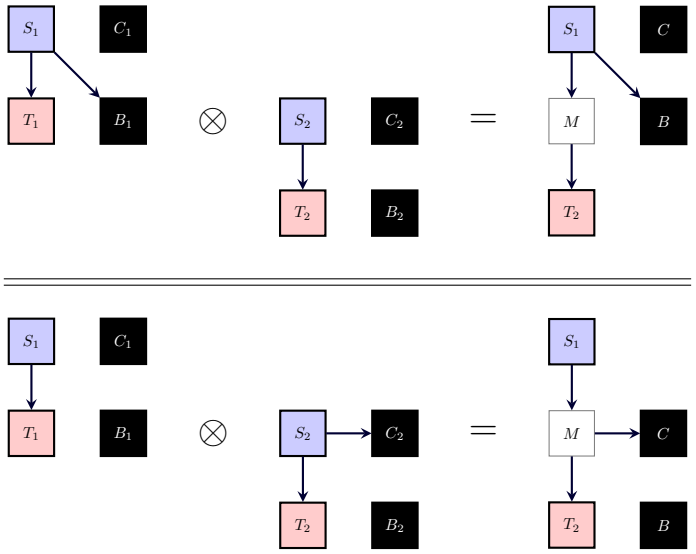

Fig. 4: Two examples of the series operation $\otimes$ [8].

SPL Operations. SPL defines three operations. Let $G_1 = (V_1, E_1, S_1, T_1, B_1, C_1)$ and $G_2 = (V_2, E_2, S_2, T_2, B_2, C_2)$ be two disjoint SPL graphs. Then, the graphs obtained by the following operations are also SPL graphs.

1. *Series Operation.* $G_1 \otimes G_2$ is generated by taking the union of G_1 and G_2 and merging the pairs of vertices $M = (T_1, S_2)$, $B = (B_1, B_2)$, and $C = (C_1, C_2)$. The distinguished vertices of $G_1 \otimes G_2$ are (S_1, T_2, B, C). It is easy to verify

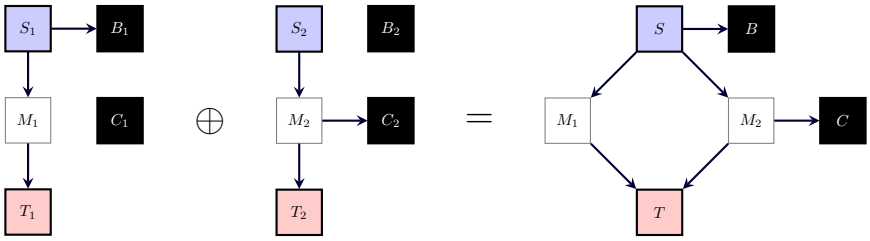

Fig. 5: An example of the parallel operation $\oplus$ [8].

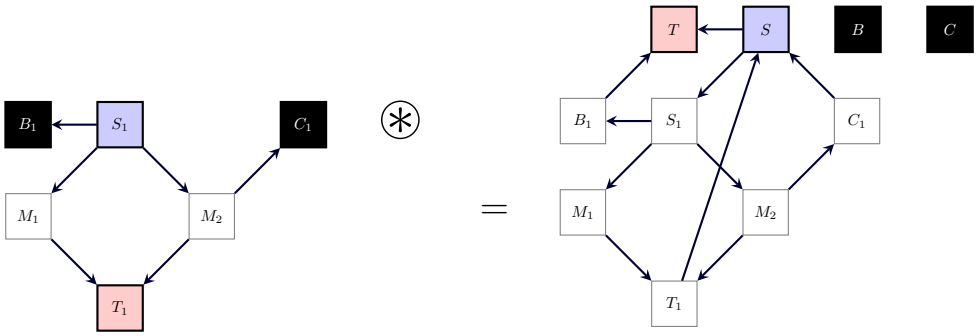

Fig. 6: An example of the loop operation [8].

that the series operation is associative. Figure 4 shows two examples of the series operation.

2. *Parallel Operation.* $G_1 \oplus G_2$ is generated by taking the union of G_1 and G_2 and merging the pairs of vertices $S = (S_1, S_2)$, $T = (T_1, T_2)$, $B = (B_1, B_2)$, and $C = (C_1, C_2)$. The special vertex tuple of $G_1 \oplus G_2$ is (S, T, B, C). Figure 5 shows an example of this operation.
3. *Loop Operation.* $G_1^{\circledast}$ is generated by adding four new vertices S, T, B, C to G_1 and then adding the following edges: $(S, S_1), (S, T), (T_1, S), (C_1, S)$, and (B_1, T). The special vertex tuple of $G_1^{\circledast}$ is (S, T, B, C). Figure 6 shows an example of the loop operation.

We say an SPL graph $G = (V, E, S, T, B, C)$ is *closed* if there are no incoming edges to the vertices B and C.

SPLs as CFGs. Given the above definitions of structured programs and SPL graphs, we have the following homomorphism which maps every structured program to its control-flow graph. Moreover, this homomorphism preserves closedness, i.e. closed programs are mapped to closed graphs. A graph is an SPL graph if and only if it is the control-flow graph of a program [8].

$$\mathsf{cfg}(\epsilon) = A_\epsilon \qquad \mathsf{cfg}(\texttt{break}) = A_{\texttt{break}} \qquad \mathsf{cfg}(\texttt{continue}) = A_{\texttt{continue}}$$

$$\mathsf{cfg}(P_1; P_2) = \mathsf{cfg}(P_1) \otimes \mathsf{cfg}(P_2)$$

$$\mathsf{cfg}(\texttt{if } \varphi \texttt{ then } P_1 \texttt{ else } P_2 \texttt{ fi}) = \mathsf{cfg}(P_1) \oplus \mathsf{cfg}(P_2)$$

$$\mathsf{cfg}(\texttt{while } \varphi \texttt{ do } P_1 \texttt{ od}) = \mathsf{cfg}(P_1)^{\circledast}$$

SPL Decomposition. Given a closed program P, we can first parse it based on the grammar in (1) to generate a parse tree. Subsequently, by applying our homomorphism above to this parse tree, we can derive a parse tree according to (2) for its control-flow graph. We use the term *SPL decomposition* to refer to the parse tree of the CFG according to (2). It is easy to verify that this process takes linear time. See Figure 7 as an example.

4 Our LOSPRE Algorithm

In this section, we present a linear-time algorithm for LOSPRE using SPL decompositions. As in Definition 1, the input to our algorithm consists of a closed program P, its control-flow graph $G = (V, E)$, a use set $U \subseteq V$, an invalidating set $I \subseteq V$ and two cost functions $c : E \to K$ and $l : V \to K$. Our goal is to find a life set $L \subseteq V$ that minimizes

$$\textsc{Cost}(G, U, I, L, c, l) = \sum_{e \in C(U,L,I)} c(e) + \sum_{v \in L} l(v).$$

Step 1 (Initialization). Our algorithm computes an SPL decomposition of $G = \mathsf{cfg}(P)$ by first parsing P and then applying the homomorphism of the previous section.

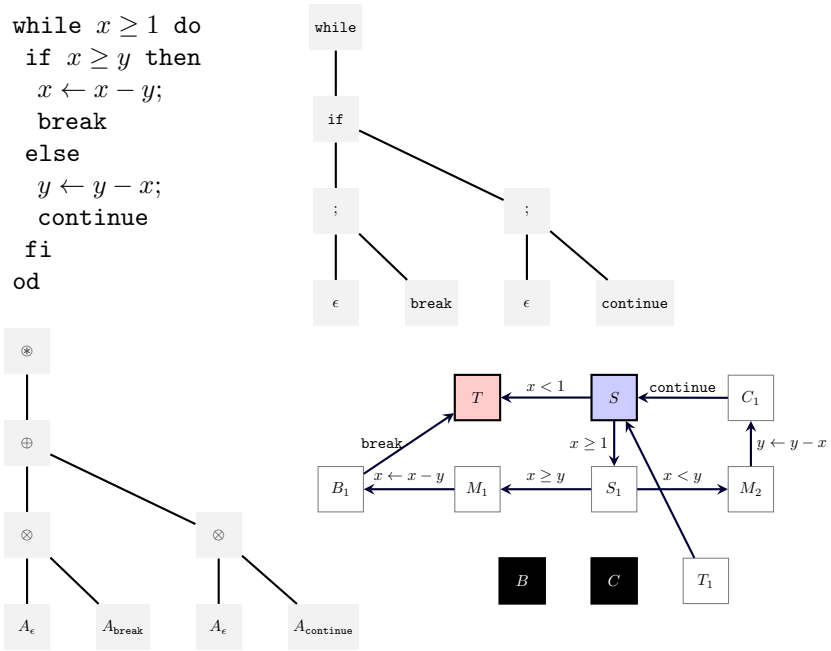

Fig. 7: A program P (top left), its parse tree (top right), the corresponding parse tree of $G = \mathsf{cfg}(P)$ (bottom left) and the graph $G = \mathsf{cfg}(P)$ (bottom right) [8]. The edges of the graph are labeled according to the statements in the program.

Step 2 (Dynamic Programming). Our algorithm proceeds with a bottom-up dynamic programming on the SPL decomposition. Note that each node u of the SPL decomposition corresponds to an SPL subgraph $G_u = (V_u, E_u, S_u, T_u, B_u, C_u)$ of G which is either an atomic SPL graph (when u is a leaf) or obtained by applying one of the SPL operations to the graphs corresponding to the children of u. See Figure 7. Let $\Gamma_u = \{S_u, T_u, B_u, C_u\}$ be the set of special vertices of G_u. For every $X \subseteq \Gamma_u$, we define a dynamic programming variable $\texttt{dp}[u, X]$. Our goal is to compute this dynamic programming value such that

$$\texttt{dp}[u, X] = \min_{L \subseteq V_u \ \wedge \ L \cap \Gamma_u = X} \textsc{Cost}(G_u, U, I, L, c, l).$$

Intuitively, we are considering a subproblem of the original LOSPRE in which the graph is limited to G_u. Moreover, we only consider those solutions (life sets) L for which $L \cap \Gamma_u = X$. The value in $\texttt{dp}[u, X]$ should then give us the minimum cost among all such solutions. Below, we present how our algorithm computes $\texttt{dp}[u, X]$ for every vertex u of the decomposition based on the $\texttt{dp}[\cdot, \cdot]$ values at its children:

(2.1) *Atomic Nodes*: If G_u is an atomic SPL graph, then the only vertices in G_u are the four special vertices. Therefore, we must have $L = X$. Our algorithm computes each $\texttt{dp}[u, X]$ as:

$$\texttt{dp}[u, X] = \textsc{Cost}(G_u, U, I, X, c, l) = \sum_{e \in C(U,X,I) \cap G_u} c(e) + \sum_{v \in X} l(v).$$

(2.2) *Series Nodes*: Suppose $G_u = G_v \otimes G_w$ where v and w are the children of u in the SPL decomposition. Let $X \subseteq \Gamma_u$ and $X_v \subseteq \Gamma_v$ be subsets of special vertices of G_u and G_v, respectively. We say that X and X_v are *compatible* and write $X \leftrightharpoons X_v$ if the following conditions are satisfied:
- $S_v \in X_v \Leftrightarrow S_u \in X$;
- $B_v \in X_v \Leftrightarrow B_u \in X$;
- $C_v \in X_v \Leftrightarrow C_u \in X$.

Intuitively, compatibility means that the subsets X and X_v make the same decisions about including vertices in the life set L. Since $S_u = S_v$, they should either both include it or both exclude it. Similarly, B_u is obtained by merging B_v and B_w. Therefore, the decisions made for B_u and B_v must match. The same applies to C_u which is a merger of C_v and C_w.
Now consider $X_w \subseteq \Gamma_w$. We say that X_w and X are compatible and write $X \leftrightharpoons X_w$ if the following conditions are satisfied:
- $T_w \in X_w \Leftrightarrow T_u \in X$;
- $B_w \in X_w \Leftrightarrow B_u \in X$;
- $C_w \in X_w \Leftrightarrow C_u \in X$.

The intuition is the same as the previous case, except that we now have $T_u = T_w$. Finally, we say that X_v and X_w are compatible and write $X_v \leftrightharpoons X_w$ if
- $T_v \in X_v \Leftrightarrow S_w \in X_w$.

This is because T_v and S_w are the same vertex of the CFG.
In this step, our algorithm sets

$$\mathtt{dp}[u, X] = \min_{\substack{X \leftharpoondown X_v \\ X \leftharpoondown X_w \\ X_v \leftharpoondown X_w}} \mathtt{dp}[v, X_v] + \mathtt{dp}[w, X_w] - [T_v \in X_v] \cdot l(T_v) - [B_v \in X_v] \cdot l(B_v) - [C_v \in X_v] \cdot l(C_v).$$

This is because every edge in G_u appears in either G_v or G_w but not both. Thus, the cost of the edges would simply be the sum of their costs in the two subgraphs. However, when it comes to vertices, T_v and S_w are merged, as are B_v and B_w, and C_v and C_w. Hence, we have to make sure we do not double count the cost of liveness for these vertices. Since this cost is counted in both `dp` values at the children, we should subtract it.

(2.3) *Parallel Nodes:* We can handle parallel nodes in the same manner as series nodes, i.e. finding compatible masks at both children and ensuring that there is no double-counting of the costs of vertices. To be more precise, let $G_u = G_v \oplus G_w$. The compatibility conditions we have to check are as follows:

$$\begin{gathered} X \leftharpoondown X_v \Leftrightarrow \\ (S_u \in X \Leftrightarrow S_v \in X_v \ \wedge \ T_u \in X \Leftrightarrow T_v \in X_v \ \wedge \ B_u \in X \Leftrightarrow B_v \in X_v \ \wedge \ C_u \in X \Leftrightarrow C_v \in X_v)\,; \\ X \leftharpoondown X_w \Leftrightarrow \\ (S_u \in X \Leftrightarrow S_w \in X_w \ \wedge \ T_u \in X \Leftrightarrow T_w \in X_w \ \wedge \ B_u \in X \Leftrightarrow B_w \in X_w \ \wedge \ C_u \in X \Leftrightarrow C_w \in X_w)\,; \\ X_v \leftharpoondown X_w \Leftrightarrow \\ (S_v \in X_v \Leftrightarrow S_w \in X_w \ \wedge \ T_v \in X_v \Leftrightarrow T_w \in X_w \ \wedge \ B_v \in X_v \Leftrightarrow B_w \in X_w \ \wedge \ C_v \in X_v \Leftrightarrow C_w \in X_w)\,. \end{gathered}$$

With the same argument as in the previous case, our algorithm sets

$$\mathtt{dp}[u, X] = \min_{\substack{X \leftharpoondown X_v \\ X \leftharpoondown X_w \\ X_v \leftharpoondown X_w}} \mathtt{dp}[v, X_v] + \mathtt{dp}[w, X_w] - [S_v \in X_v] \cdot l(S_v) - [T_v \in X_v] \cdot l(T_v) - [B_v \in X_v] \cdot l(B_v) - [C_v \in X_v] \cdot l(C_v).$$

(2.4) *Loop Nodes:* Finally, we should handle the case where $G_u = G_v^{\circledast}$. This case is quite simple. By construction, in comparison to G_v, the graph G_u has four new vertices

$$V_{\text{new}} = \{S_u, T_u, B_u, C_u\}$$

and five new edges

$$E_{\text{new}} = \{(S_u, S_v), (S_u, T_u), (T_v, S_u), (C_v, S_u), (B_v, T_u)\}.$$

The two graphs G_u and G_v do not share any special vertices, i.e. $\Gamma_u \cap \Gamma_v = \emptyset$. Moreover, for every edge $(x, y) \in E_{\text{new}}$ we can decide whether (x, y) is in the calculation set solely based on X and X_v. This is because $x, y \in X \cup X_v$. More specifically, (x, y) is in the calculation set if and only if

$$\varphi(X, X_v, x, y) := [x \notin X \cup X_v \setminus I \ \wedge \ y \in U \cup X \cup X_v]$$

Thus, our algorithm sets:

$$\mathtt{dp}[u, X] = \sum_{x \in V_{\text{new}} \cap X} l(x) + \min_{X_v \subseteq \Gamma_v} \mathtt{dp}[v, X_v] + \sum_{(x,y) \in E_{\text{new}}} \varphi(X, X_v, x, y) \cdot c(x, y).$$

Step 3 (Computing the Final Answer). Let r be the root of the SPL decomposition. By definition, we have $G_r = G$. The algorithm outputs $\min_{X \subseteq \Gamma_r} \texttt{dp}[r, X]$ as the minimum possible cost for the given LOSPRE input. This is because G_r is the entire CFG G and any solution L will conform to exactly one of the different possible values of X at r. As is standard in dynamic programming approaches, one can reconstruct the optimal life set L that leads to this minimal cost by retracing the steps of the algorithm and remembering which choices led to the optimal value at each step.

Theorem 1. *Given a LOSPRE instance consisting of a closed structured program P, its control-flow graph G with n vertices, a use set U, an invalidating set I and two cost functions $c : E \to K$ and $l : V \to K$, the algorithm above solves the LOSPRE problem of Definition 1 in $O(n)$ and outputs*

$$\min_L \textsc{Cost}(G, U, I, L, c, l) \qquad \textit{and} \qquad \arg\min_L \textsc{Cost}(G, U, I, L, c, l).$$

Proof. Correctness is already argued above. Thus, we focus on the runtime analysis. The SPL decomposition has $O(n)$ vertices and can be computed in $O(n)$ as mentioned at the end of Section 3. At each vertex u of the decomposition, we have $2^4 = 16 = O(1)$ different possible values for X. The computations in Step (2.1) are over graphs with only four vertices and thus take $O(1)$ time. In Step (2.2) we have at most two compatible X_v's for each X. This is because inclusion or exclusion of the vertices S_v, B_v and C_v in X_v is uniquely determined by X and only T_v remains to be chosen. Similarly, for every fixed X, X_v, there is a unique X_w. Thus, computing each $\texttt{dp}[u, X]$ in this step takes $O(1)$ time. In Step (2.3), every X induces a unique X_v and a unique X_w. Hence, this step takes $O(1)$ time to compute each $\texttt{dp}[u, x]$ value. In Step (2.4), we try $2^4 = O(1)$ different X_v's for each X. Thus, the total runtime of Step 2 is $O(n)$. Finally, Step 3 takes the maximum of $2^4 = O(1)$ values. □

5 Experimental Results

Implementation. We implemented our LOSPRE algorithm in `C++` and integrated it with the Small Device C Compiler (SDCC) [21,22].

Baseline. We compared our algorithm's runtime with the treewidth-based approach of [34], which is the current state-of-the-art in LOSPRE. This approach has an asymptotic runtime of $O(\texttt{tw}(G) \cdot 2^{\texttt{tw}(G)} \cdot n)$ where $\texttt{tw}(G)$ is the treewidth of G. We did not consider other previous methods since they are significantly slower with runtime bounds of $O(n^3)$ or $O(n^2 \cdot \log^3 n)$. Additionally, SDCC already includes a highly-optimized variant of algorithms for finding tree decompositions. Furthermore, [34] is also implemented as part of SDCC.

Machine. The results were obtained on an Ubuntu 24.04 machine, equipped with a 1.6 GHz dual-core Intel Core i5 processor and 4 GB of RAM.

Benchmarks. We exactly followed the setup of [34], utilizing the SDCC regression test suite as our benchmark set. This suite comprises a total of $20,244$ instances for LOSPRE. These benchmarks are embedded programs expected to operate in resource-constrained environments. Therefore, the focus is on code size optimization. More specifically, the goal is to minimize the total number of computations in the resulting 3-address code. Thus, we use $K = \mathbb{Z}^2$ with lexicographic ordering. The cost assigned to each edge (x, y) is $c(x, y) = (1, 0)$. We also enforce lifetime-optimality by assigning the cost $l(x) = (0, 1)$ to every vertex x. We only compare the runtimes. There is no output comparison since both our approach and [34] find an optimal solution for LOSPRE, creating 3-address codes of the same size. We enforced a time limit of 10 minutes and a memory limit of 4 GB for each benchmark.

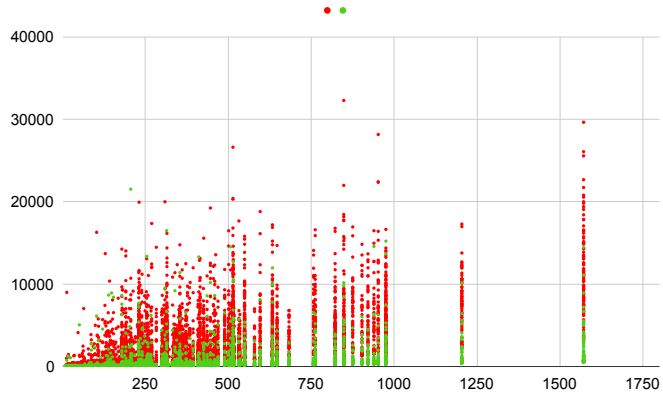

Fig. 8: Runtime comparison of the treewidth-based algorithm in [34] (red) and our approach (green). The x axis is the size of the CFG and the y axis is time in μs.

Runtimes. Figures 8–9 provide runtime comparisons between [34] and our approach. On average, our algorithm takes 222.38 μs, while the treewidth-based approach of [34] has an average runtime of 1349.14 μs. The maximum runtime was 21,524 μs for our algorithm compared to 32,284 μs for [34]. Our algorithm significantly outperforms [34] in the vast majority of benchmarks. We identified only 19 instances where our runtime exceeded 10,000 μs, whereas [34] takes more than 10,000 μs in 277 instances.

Discussion. In summary, our approach is approximately six times faster than the previous state-of-the-art for LOSPRE. This represents a significant improvement, particularly given that the treewidth-based approach of [34] is already highly optimized and included in the well-established SDCC compiler. We believe there are two main reasons for this speedup: (i) computing SPL decompositions is

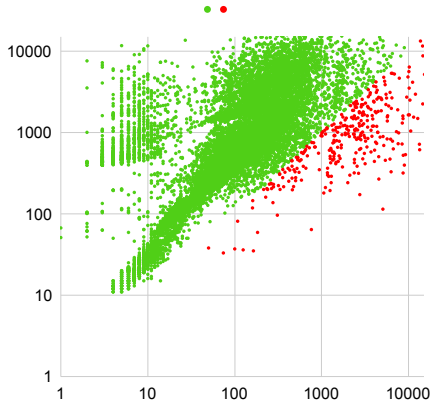

Fig. 9: Runtime comparison of the treewidth-based algorithm in [34] and our approach. The x axis is our runtime and the y axis is [34]'s runtime. Each dot represents one LOSPRE instance. Instances for which our approach was faster are shown in green and those for which [34] was faster are shown in red. The axes are in logarithmic scale.

much faster than tree decompositions, and (ii) our algorithm's runtime does not depend on parameters such as treewidth and the constant factor hidden in our $O(n)$ asymptotic runtime analysis is small in practice.

6 Conclusion

We provided a novel algorithm for lifetime-optimal speculative partial redundancy elimination over structured goto-free programs. Our algorithm exploits the sparsity of the program's control-flow graph. However, unlike the previous state-of-the-art, we do not consider tree decompositions and treewidth. Instead, we build our algorithm based on the different notion of series-parallel-loop (SPL) decompositions. SPL decompositions can be computed much faster than tree decompositions and our algorithm's runtime is also independent of parameters such as treewidth. As a result, we observed a significant 6x runtime improvement in LOSPRE over benchmarks from the Small Device C Compiler. Our algorithm works directly on the CFG of a source program, whereas many compilers, such as LLVM, have an intermediate representation in the SSA form, which has been exploited for various compiler optimization tasks. An interesting direction of future research is to explore whether our speedups can be combined with those obtained from SSA representations. On the other hand, a significant limitation of our approach is that it can only be applied to goto-free programs. Extending the SPL decomposition and our algorithm to support goto statements is another interesting direction.

Acknowledgments

We are grateful to the anonymous reviewers for raising points that significantly improved this work. The research was partially supported by the Hong Kong Research Grants Council ECS Project Number 26208122.

References

1. Ahmadi, A., Chatterjee, K., Goharshady, A.K., Meggendorfer, T., Safavi, R., Zikelic, Đ.: Algorithms and hardness results for computing cores of Markov chains. In: FSTTCS. pp. 29:1–29:20 (2022)
2. Ahmadi, A., Daliri, M., Goharshady, A.K., Pavlogiannis, A.: Efficient approximations for cache-conscious data placement. In: PLDI. pp. 857–871 (2022)
3. Asadi, A., Chatterjee, K., Goharshady, A.K., Mohammadi, K., Pavlogiannis, A.: Faster algorithms for quantitative analysis of MCs and MDPs with small treewidth. In: ATVA. vol. 12302, pp. 253–270 (2020)
4. Barakbayeva, T., Farokhnia, S., Goharshady, A.K., Gufler, M., Novozhilov, S.: Pixiu: Optimal block production revenues on Cardano. In: Blockchain (2024)
5. Bodlaender, H.L., Gustedt, J., Telle, J.A.: Linear-time register allocation for a fixed number of registers. In: SODA. pp. 574–583 (1998)
6. Burgstaller, B., Blieberger, J., Scholz, B.: On the tree width of Ada programs. In: Ada-Europe. pp. 78–90 (2004)
7. Cai, Q., Xue, J.: Optimal and efficient speculation-based partial redundancy elimination. p. 91–102. CGO (2003)
8. Cai, X., Goharshady, A.K., Hitarth, S., Lam, C.K.: Faster register allocation via grammatical decompositions of control-flow graphs (2024), `https://hal.science/hal-04672403`
9. Chatterjee, K., Goharshady, A.K., Goharshady, E.K.: The treewidth of smart contracts. In: SAC. pp. 400–408. ACM (2019)
10. Chatterjee, K., Goharshady, A.K., Goyal, P., Ibsen-Jensen, R., Pavlogiannis, A.: Faster algorithms for dynamic algebraic queries in basic RSMs with constant treewidth. ACM Trans. Program. Lang. Syst. **41**(4), 23:1–23:46 (2019)
11. Chatterjee, K., Goharshady, A.K., Ibsen-Jensen, R., Pavlogiannis, A.: Algorithms for algebraic path properties in concurrent systems of constant treewidth components. In: POPL. pp. 733–747 (2016)
12. Chatterjee, K., Goharshady, A.K., Ibsen-Jensen, R., Pavlogiannis, A.: Optimal and perfectly parallel algorithms for on-demand data-flow analysis. In: ESOP. pp. 112–140 (2020)
13. Chatterjee, K., Goharshady, A.K., Okati, N., Pavlogiannis, A.: Efficient parameterized algorithms for data packing. Proc. ACM Program. Lang. **3**(POPL), 53:1–53:28 (2019)
14. Chatterjee, K., Goharshady, A.K., Pavlogiannis, A.: JTDec: A tool for tree decompositions in soot. In: ATVA. vol. 10482, pp. 59–66 (2017)
15. Chatterjee, K., Ibsen-Jensen, R., Goharshady, A.K., Pavlogiannis, A.: Algorithms for algebraic path properties in concurrent systems of constant treewidth components. ACM Trans. Program. Lang. Syst. **40**(3), 9:1–9:43 (2018)
16. Chatterjee, K., Lacki, J.: Faster algorithms for Markov decision processes with low treewidth. In: CAV. pp. 543–558 (2013)

17. Cocke, J.: Global common subexpression elimination pp. 20–24 (1970), https://doi.org/10.1145/390013.808480
18. Conrado, G.K., Goharshady, A.K., Hudec, P., Li, P., Motwani, H.J.: Faster treewidth-based approximations for Wiener index. In: SEA. vol. 301, pp. 6:1–6:19 (2024)
19. Conrado, G.K., Goharshady, A.K., Kochekov, K., Tsai, Y.C., Zaher, A.K.: Exploiting the sparseness of control-flow and call graphs for efficient and on-demand algebraic program analysis. Proc. ACM Program. Lang. **7**(OOPSLA2), 1993–2022 (2023)
20. Conrado, G.K., Goharshady, A.K., Lam, C.K.: The bounded pathwidth of control-flow graphs. Proc. ACM Program. Lang. **7**(OOPSLA2), 292–317 (2023)
21. Dutta, S.: Anatomy of a compiler: A retargetable ANSI-C compiler. Circuit Cellar **121**(5) (2000)
22. Dutta, S., Drotos, D., Vigor, K., et al.: Small device C compiler (2003), http://sdcc.sourceforge.net/
23. Ferrara, A., Pan, G., Vardi, M.Y.: Treewidth in verification: Local vs. global. In: LPAR. vol. 3835, pp. 489–503 (2005)
24. Goharshady, A.K.: Parameterized and Algebro-geometric Advances in Static Program Analysis. Ph.D. thesis, Institute of Science and Technology Austria, Klosterneuburg, Austria (2020)
25. Goharshady, A.K., Lam, C.K., Parreaux, L.: Fast and optimal extraction for sparse equality graphs. In: OOPSLA (2024)
26. Goharshady, A.K., Mohammadi, F.: An efficient algorithm for computing network reliability in small treewidth. Reliab. Eng. Syst. Saf. **193**, 106665 (2020)
27. Goharshady, A.K., Zaher, A.K.: Efficient interprocedural data-flow analysis using treedepth and treewidth. In: VMCAI. vol. 13881, pp. 177–202 (2023)
28. Gupta, R., Berson, D., Fang, J.: Path profile guided partial redundancy elimination using speculation. In: Proceedings of the 1998 International Conference on Computer Languages (Cat. No.98CB36225). pp. 230–239 (1998)
29. Gustedt, J., Mæhle, O.A., Telle, J.A.: The treewidth of Java programs. In: ALENEX. pp. 86–97 (2002)
30. Horspool, R.N., Pereira, D.J., Scholz, B.: Fast profile-based partial redundancy elimination. In: JMLC. p. 362–376 (2006)
31. Jaiyen, B., Liu, J.: Implementing profile-guided speculative code motion in LLVM (2012)
32. Karger, D.R., Stein, C.: An $\tilde{O}$(n2) algorithm for minimum cuts. In: STOC. p. 757–765 (1993)
33. Knoop, J., Rüthing, O., Steffen, B.: Lazy code motion. In: PLDI. p. 224–234 (1992)
34. Krause, P.K.: lospre in linear time. In: SCOPES. p. 35–41 (2021)
35. Krause, P.K.: Optimal register allocation in polynomial time. In: CC. vol. 7791, pp. 1–20 (2013)
36. Meybodi, M.A., Goharshady, A.K., Hooshmandasl, M.R., Shakiba, A.: Optimal mining: Maximizing Bitcoin miners' revenues from transaction fees. In: Blockchain. pp. 266–273 (2022)
37. Morel, E., Renvoise, C.: Global optimization by suppression of partial redundancies. Commun. ACM **22**, 96–103 (1979)
38. Obdrzálek, J.: Fast mu-calculus model checking when tree-width is bounded. In: CAV. pp. 80–92 (2003)
39. Pereira, D.J.: Isothermality: making speculative optimizations affordable (2007), https://api.semanticscholar.org/CorpusID:64227807
40. Sankaranarayanan, S.: Reachability analysis using message passing over tree decompositions. In: CAV. pp. 604–628 (2020)

41. Thorup, M.: All structured programs have small tree width and good register allocation. Inf. Comput. **142**(2), 159–181 (1998)
42. Xue, J., Cai, Q.: A lifetime optimal algorithm for speculative PRE. ACM Trans. Archit. Code Optim. p. 115–155 (2006)