Code Motion for Explicitly Parallel Programs*

Jens Knoop
Universität Dortmund †
knoop@ls5.cs.uni-dortmund.de

Bernhard Steffen
Universität Dortmund †
steffen@ls5.cs.uni-dortmund.de

Abstract

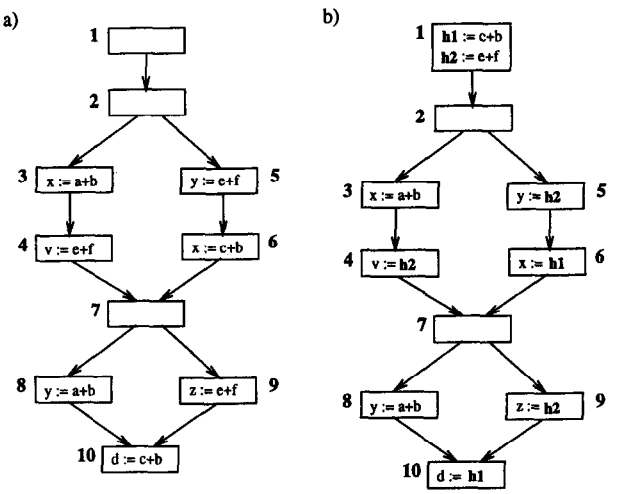
In comparison to automatic parallelization, which is thoroughly studied in the literature [31, 33], classical analyses and optimizations of explicitly parallel programs were more or less neglected. This may be due to the fact that naive adaptations of the sequential techniques fail [24], and their straightforward correct ones have unacceptable costs caused by the interleavings, which manifest the possible executions of a parallel program. Recently, however, we showed that unidirectional *bitvector analyses* can be performed for parallel programs as easily and as efficiently as for sequential ones [17], a necessary condition for the successful transfer of the classical optimizations to the parallel setting.

In this article we focus on possible subsequent code motion transformations, which turn out to require much more care than originally conjectured [17]. Essentially, this is due to the fact that interleaving semantics, although being adequate for correctness considerations, fails when it comes to reasoning about *efficiency* of parallel programs. This deficiency, however, can be overcome by strengthening the specific treatment of synchronization points.

Keywords: Code optimization, shared memory, interleaving semantics, synchronization, data-flow analysis, bitvector problems, code motion (partial redundancy elimination).

1 Motivation

Background. *Code motion* (*CM*) has proved to be a powerful technique for the optimization of sequential programs. Intuitively, it improves the efficiency of a program by avoiding unnecessary recomputations of values at runtime. This is achieved by replacing the original computations of a program by temporaries which are initialized at suitable program points while maintaining the program semantics. It is well-known that for sequential programs even *computationally optimal* results can be obtained, i.e., the number of computations on each program path cannot be reduced anymore by semantics preserving code motion.



The Sequential Argument Program — A Computationally Optimal Program

Figure 1: CM in the sequential setting.

Placing the computations at their "earliest down-safe" points leads to computationally optimal results [12, 14]. Intuitively, earliest down-safe program points are points where the computation under consideration is not available, but required on every program continuation leaving it. Following [14], the as-early-as-possible placement of computations can be achieved by computing the set of *up-safe* and *down-safe* program points of a computation, i.e., where it has been computed on every program path reaching the program point under

†Fachbereich Informatik, Universität Dortmund, Baroper Straße 301, D-44227 Dortmund, Germany.

*A poster presentation was given at CC'96, Linköping, Sweden.

PPoPP '99 5/99 Atlanta, GA, USA

consideration, and where it will be computed on every program continuation reaching the program's end node.[1] Thus, down-safe start nodes are earliest, as well as down-safe nodes with an unsafe predecessor or a predecessor modifying an operand of the computation under consideration. The as-early-as-possible placement strategy is illustrated in Figure 1. Note that the partially redundant computation of $a+b$ at node **8** cannot *safely* be eliminated, i.e., without affecting the semantics or impairing some program executions.

As both up-safety and down-safety are *unidirectional bitvector* problems, it is straightforward — using the framework of [17] — to transfer these analyses to parallel programs at almost no cost on the implementation and the computation side. This suggests to make the computationally optimal code motion of the as-early-as-possible placement strategy available for parallel programs, too, in order to avoid the unnecessary recomputation of values. An extension based on this idea has been sketched in [17] together with the conjecture that the resulting transformation leads to computationally optimal parallel programs. The following sections illustrate that this conjecture was too optimistic.

Pitfalls of CM in the Parallel Setting

Optimality. Computational optimality as considered for sequential programs is based on the relation "computationally better" between programs: a program G is computationally better than a program G' if every execution of G requires at most as many computations as the corresponding execution of G'. This relation is perfectly well-suited for comparing sequential programs. A sequential program G which is computationally better than a program G' is also "executionally better," i.e., any program execution of G will be faster than the corresponding execution of G'. Unfortunately, this does not carry over to the parallel setting as illustrated by the example of Figure 2. For clarity, the components of parallel statements are separated by parallels throughout this article. Note that both programs of Figure 2(b) and (c) lie in the kernel of the relation computationally better, i.e., on every program path from node **1** to node **10** they perform the same number of computations. Actually, they are even both computationally optimal. However, the runtime behaviour of the program of Figure 2(b), which results from the as-early-as-possible placement strategy, will be worse than that of the program of Figure 2(c). The point here is that "computationally better" is based on a simple count of the occurrences of computations on (sequentialized) program paths, i.e., on a pure interleaving view. It does not distinguish between computations in sequential and parallel program parts. The execution time of a parallel program, however, is extremely sensitive to this distinction: e.g., computations in a parallel component may be for "free." The execution time of the considered execution is determined by the computations of the bottleneck component (the component requiring most activity). Thus, the as-early-as-possible placement, which leads to computationally (and executionally) optimal results in the sequential setting, is inappropriate for the parallel setting as here computational optimality it is aiming at does not induce the desired *executional optimality.*

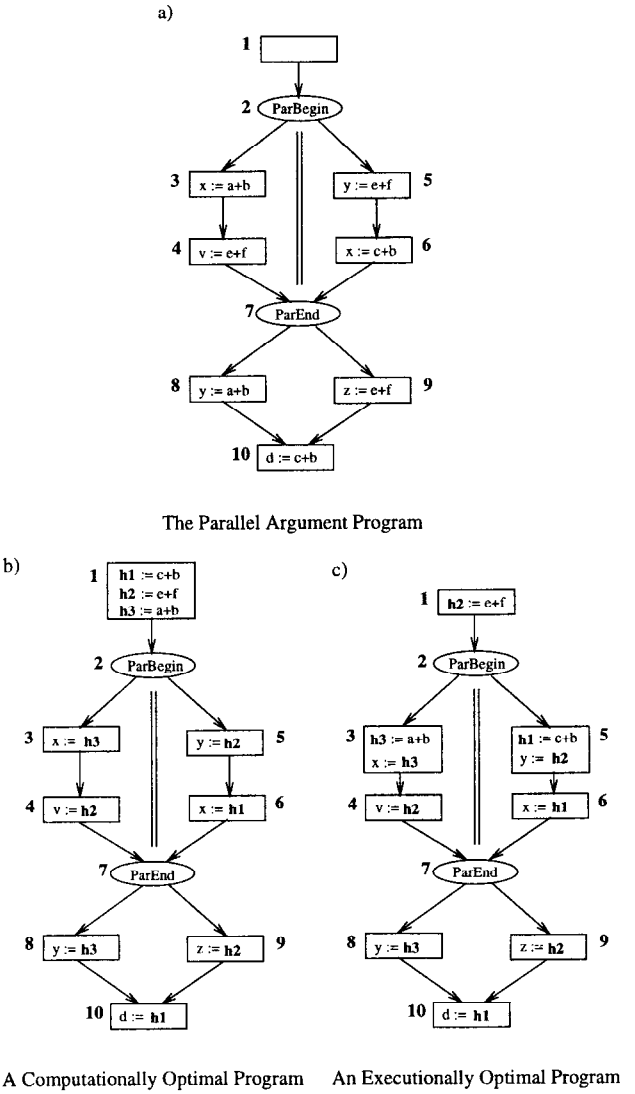


Figure 2: Computational vs. executional optimality.

"Recursive" Assignments. For sequential programs it is important that all computation patterns can be independently processed without affecting correctness or optimality of the complete transformation. In the paral-

[1] Up-safety and down-safety are also known as *availability* and *anticapibility* (*very busyness*). The coining of these traditional terms, however, relies on a mostly technical motivation. They neither reflect the duality of the two properties, nor their semantical essence, which is to express a safety property.

lel setting, however, correctness can be violated in the presence of "recursive" assignments, i.e., assignments whose left-hand side variables occur in their right-hand side terms. This is illustrated by the two examples of Figure 3. Note that the transformation of Figure 3(b) preserves *sequential consistency* (cf. [20, 28]) of the program of Figure 3(a), i.e., every observable behaviour for an interleaving of the program of (b) can also be observed for some (in general different) interleaving of the program of (a). Sequential consistency, however, gets lost as soon as the left-hand side variable z of the statement at node **3** is replaced by c making this statement like the one of node **5** a recursive one, too (cf. Figure 3(c)). As a consequence, in any interleaving sequence, in which node **3** is executed prior to node **5**, the use of c in node **6** refers to the new value of c and vice versa. This is illustrated by the program transformation displayed in Figure 3(d): the interleaving **5 – 6 – 3 – 4** of the program of (d) assigns in the parallel statement the value 5 to the variables a and x, and the value 8 to the variables c and y. This is impossible for any interleaving of the program of (c), regardless of considering assignments atomic or not.

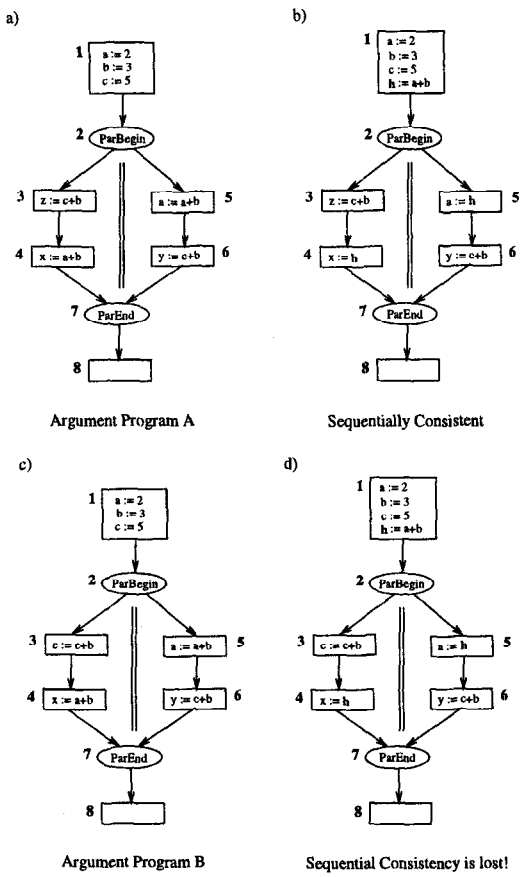


Figure 3: Loss of sequential consistency I.

Considering assignments atomic, however, even the independent treatment of different occurrences of the same computation pattern can cause the loss of sequential consistency in the presence of recursive assignments. This is illustrated by the example of Figure 4. Whereas the transformations of Figure 4(b) and (c) preserve *sequential consistency* of the argument program of Figure 4(a), sequential consistency gets lost if both individually correct transformations of (b) and (c) are combined as shown in (d): each interleaving of the program of (d) assigns the value 5 to the occurrences of variable a at node **4** and node **5**. This is impossible for any interleaving of the program of (a).

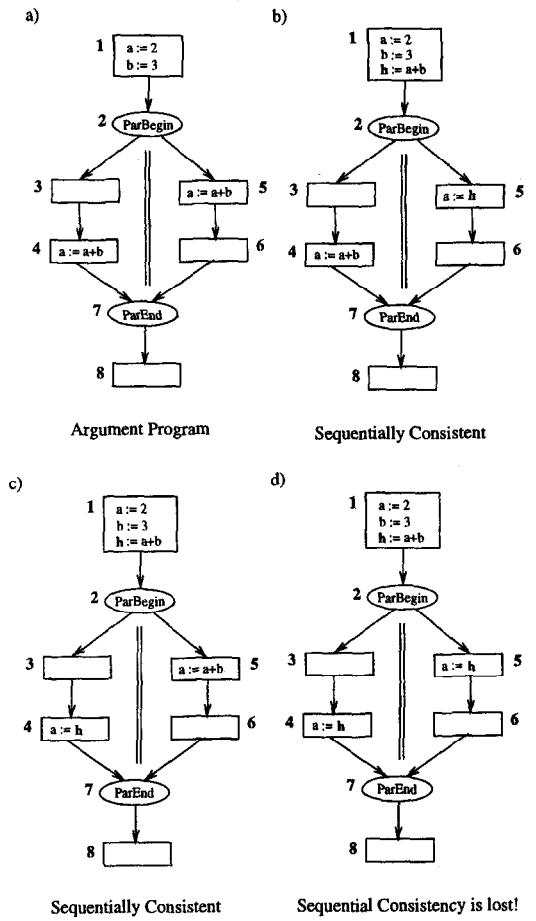


Figure 4: Loss of sequential consistency II.

In other words, the examples above demonstrate that an (existential!) trace-based requirement like sequential consistency is in general not adequate as correctness constraint for composite optimizations: the sequential composition of sequentially consistent optimizations is not guaranteed to be sequentially consistent, too.

Up-Safety and Down-Safety. A program point n is *up-safe* for a computation t, if every program path reaching n contains a computation of t, which is not followed by a modification of any operand of t. Dually, n is

down-safe for ***t,*** if every program path **starting** in *n* and reaching the end node contains a computation oft, which is not preceded by a modification of an operand of ***t.*** In the sequential setting up-safety of a program point n for ***t*** guarantees that there is a set it *M* of program points computing ***t,*** and commonly dominating ***n,*** where every program point on a path leaving a node of *M* and reaching n is up-safe as well. Conversely, down-safety of ***n*** for ***t*** guarantees that there is a set *M* of program points, which compute *t,* and commonly post-dominate *n,* and where each program point on a path leaving n and reaching a node of **M** is down-safe, too. These facts are illustrated in Figure 5. In the sequential setting they are sufficient to establish the correctness of the as-early-as-possible placement strategy: they guarantee (1) that a temporary which is initialized at a down-safe program point can be used on every program continuation reaching the end node without an intermediate redefinition of the temporary itself or of an operand of the computation, and (2) that at an up-safe program point the computation under consideration can be made available in a temporary which is initialized in a set of down-safe program points commonly dominating it.

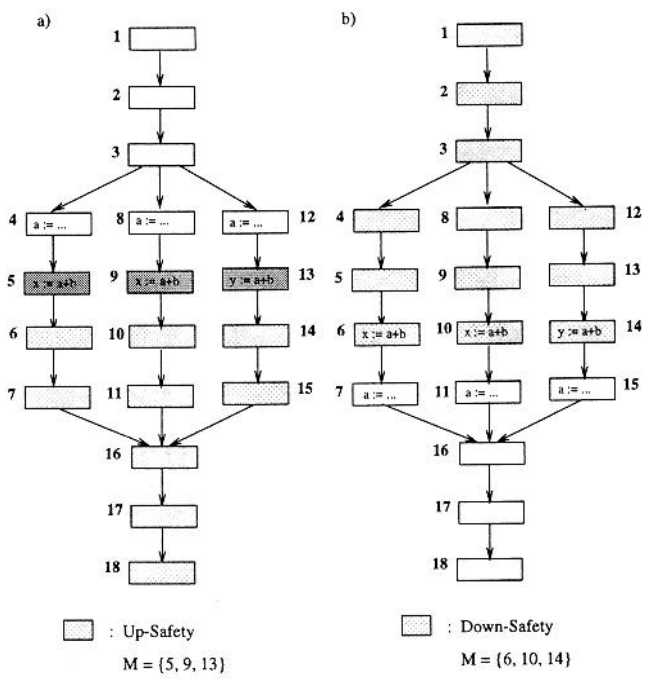


Figure 5: Sequential setting.

Unfortunately, these facts generally do not carry over to parallel programs. This is illustrated in Figure 6. Note that for every program interleaving node **3** and node **16** are down-safe and up-safe on entering and leaving the parallel statement, respectively. However, none of the internal nodes of the parallel statements enjoys these properties. The point here is that up-safety and down-safety at node **16** and node **3,** respectively, hold for individual interleavings. This becomes obvious by considering the corresponding nondeterministic sequential product program, the "unfolded" version of ***a*** parallel program (cf. Figure 6). The product program makes all interleavings of a parallel program explicit. In fact, up-safety and down-safety then hold in the usual path-based sense, i.e, the particular occurrence guaranteeing the property can be precisely pin-pointed. However, in the compact representation as a parallel program (cf. Figure 6), which can be considered an abbreviation of the in the worst case exponentially larger unfolded (product) program, this is impossible. In fact, depending on the chosen interleaving the properties are guaranteed by different occurrences of *a + b.*

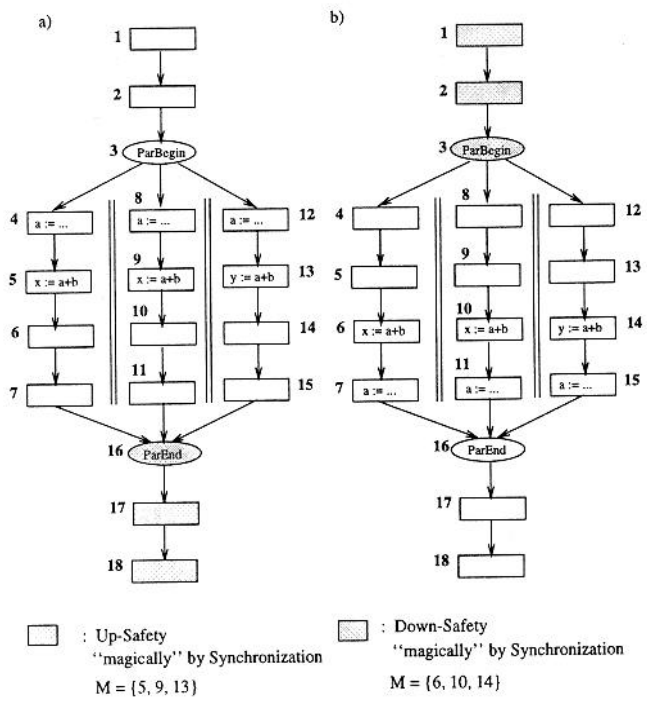


Figure 6: Parallel setting.

As ɑ consequence, a "naive" adaption of the computationally optimal placing strategies known for sequential programs like the earliest down-safe placement can impair the efficiency and even corrupt the semantics. This is illustrated in Figure 7. Obviously, node **1** is an earliest down-safe point. However, the initialization made here cannot be guaranteed to be used. Hence, the runtime efficiency may be impaired. On the other hand, the initialization at node **12** is suppressed as the value under consideration is up-safe there. In the sequential setting the suppression would be correct as up-safety implies that the value is already stored in a temporary. However, in the parallel setting this cannot be guaranteed.

In this article, we show how to overcome these pitfalls of CM for parallel programs within the framework of

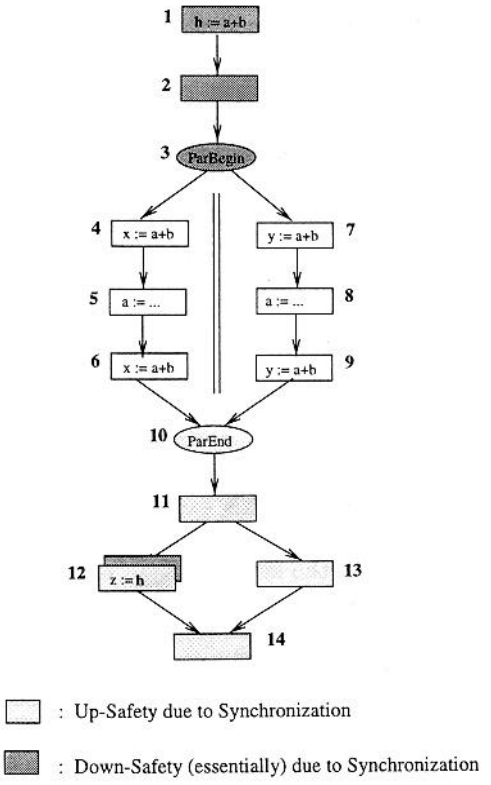


Figure 7: Safety "anomaly" due to synchronization.

[17]. It turns out that all modifications required to compute the program properties required by (proper) parallel code motion are limited to the generic computation algorithm of the framework, and concern the computation of data-flow information at synchronization points only. The modifications leave the general structure of the sequential algorithm for the as-early-as-possible placement strategy invariant. In fact, the new algorithm inherits the simple structure of its underlying sequential counterpart, the *busy code motion (BCM)* transformation of [12, 14], and it is similarly efficient. Like the *BCM-transformation,* it is composed of only two unidirectional bitvector data-flow analyses. The power of the new algorithm is illustrated in the example of Figure 10(a), where it is unique to obtain the optimization of Figure 10(b).

Related Work. One of the pioneering approaches for analyzing parallel programs has been proposed by Shasha and Snir [28]. It is concerned with analysing the side-conditions guaranteeing sequential consistency. Intuitively, sequential consistency means that a parallel execution of a program must behave like some interleaving of the program's parallel components. Later approaches like the one of Krishnamurthy and Yelick [18] improved on the complexity of Shasha and Snir's approach for a specific class of programs, or were concerned with analysing different variants of parallel programs differing in the language primitives concerning for example synchronization, the kind of parallelism or the restrictions applying to shared variables. Typical examples are the approaches of Chow and Harrison [l], Dwyer and Clarke [5], Grunwald and Srinivasan [6], Long and Clarke [22], or Srinivasan and Wolfe [30]. The majority of these approaches is designed for problems which are specific for parallel programs like mutual **exclusion or** the detection of dead-locks and data races. In fact, only a minority like Grunwald and Srinivasan's approach [6] is dealing with a problem known from sequential optimization: reaching-definitions analysis. Though they indicate that reaching-definitions information is important in sequential optimization for instance for common subexpression elimination, dead code elimination or constant propagation, they do not present any optimization. Similarly this holds for the approaches of Srinivasan, Hook, and Wolfe [29], Srinivasan and Wolfe [30], and of Wolfe and Srinivasan [32]. They focus on data structures and algorithms for transforming parallel programs into SSA-like program representations, but do not consider (classical) optimizations. An optimization is proposed by Krishnamurthy and Yelick [19]. They present an approach for optimizing remote accesses on distributed memory machines. Note that this is a problem specific for parallel architectures.

There are only very few approaches dealing with classical optimizations of explicitly parallel programs as those of Lee, Midkiff and Padua [21] and Knoop [11] dealing with constant propagation or of Knoop [10] dealing with partial dead-code elimination. As demonstrated by Midkiff and Padua [24] straightforward transfers of sequential techniques fail, in particular for code motion. This article contributes to revealing the inherent reasons of these failures for code motion. As they do not rely on a specific structural richness of the parallel setting, we consider a setting as simple and as concise as possible, which allows us to extract the problem's essence. However, our approach itself is not limited to this setting.

Structure of the Article. In Section 2 we recall the essential parts of the framework of [17], which are necessary for developing our CM-algorithm for parallel programs. Subsequently, we show how to overcome the pitfalls of CM in a parallel setting, and present our algorithm, which is unique to eliminate partially redundant computations in a parallel program in Section 3. Section 4, finally, contains our conclusions.

2 The Parallel Setting

This section sketches our setup, which has been presented in detail in [17]. We consider a parallel imperative programming language with interleaving semantics.

Parallelism is syntactically expressed by means of a **par** statement whose components are executed in parallel on a shared memory. As usual, we assume that there are neither jumps entering a component of a parallel statement from outside nor vice versa.

Similar to [29] and [6], we represent a parallel program by a nondeterministic *parallel flow graph* $G^* = (N^*, E^*, \mathbf{s}^*, \mathbf{e}^*)$ with node set N^* and edge set E^* as illustrated in Figure 2. As in a sequential flow graph, nodes $n \in N^*$ represent the statements, and edges $(m, n) \in E^*$ the nondeterministic branching structure of the progam under consideration, while $\mathbf{s}^*$ and $\mathbf{e}^*$ denote the distinct *start node* and *end node* of the graph. They are assumed to represent the empty statement **skip**, and to be free of incoming and outgoing edges, respectively. Parallel statements are represented by subgraphs, which are encapsulated by a **ParBegin** node and a **ParEnd** node representing both **skip**. For clarity, **ParBegin** nodes and **ParEnd** nodes are represented by ellipses in the figures. Additionally, the component graphs of a parallel statement are separated by two parallel lines as shown in Figure 2.

Important in order to capture the interference of parallel components is the notion of an interleaving predecessor of a node. In contrast to a sequential flow graph G, where the set of nodes which might precede a node n at runtime is precisely given by the set $Pred_G(n)$ of its predecessors in the graph, in a parallel flow graph the interleaving of parallel components must be taken into account, too. A node n occurring in a component of some parallel statement can at runtime also be preceded by any node of another component of this parallel statement. As in [17] we denote these "potentially parallel" nodes of a node n as its *interleaving predecessors*, denoted by $Pred^{Itlvg}_{G^*}(n)$. In the example of Figure 2 node **3** is the "ordinary" predecessor of node **4**, while node **5** and node **6** are its interleaving predecessors.

The key for defining the operational semantics of a parallel program is the notion of a parallel program path. To this end we recall that the interleaving semantics of parallel imperative programs can be defined via a translation that reduces them to (much larger) nondeterministic programs, which represent all the possible interleavings explicitly. These "product" programs directly induce the notion of a (finite) feasible path of a parallel program, or for short, of a *parallel path*: a node sequence of a parallel program is a parallel path if and only if it is a path in the corresponding product program. We denote the set of all parallel paths from m to n or to a predecessor of n by $\mathbf{PP}_{G^*}[m, n]$ and $\mathbf{PP}_{G^*}[m, n[$, respectively. In the remainder of this section we recall how to perform unidirectional *bitvector* data-flow analyses for parallel programs as easily and as efficiently as for sequential ones.

Data-flow Analysis. In essence, *data-flow analysis* (*DFA*) provides information about the program states which may occur at some given program points during execution (cf. [7, 26, 25]). Theoretically well-founded are DFAs based on *abstract interpretation* (cf. [2, 3, 23]). Usually the abstract semantics is tailored to deal with a specific problem, and is specified by a *local semantic functional* $[\![\ ]\!] : N^* \to (\mathcal{C} \to \mathcal{C})$. It gives abstract meaning to every program statement (here: every node of a parallel flow graph G^* with node set N^*) in terms of a transformation function on a complete lattice $(\mathcal{C}, \sqcap, \sqsubseteq, \bot, \top)$, whose elements express the DFA-information of interest. In our framework this carries over to the parallel setting.

A local semantic functional can easily be extended to cover parallel paths. This extension is the key for defining the parallel version of the *meet-over-all-paths* (*MOP*) approach in the sense of Kam and Ullman [8]. Its solution specifies the intuitively desired solution of a DFA-problem. The *MOP*-approach (in the parallel setting the *PMOP*-approach) directly mimics possible program executions in that it "meets" (intersects) all informations belonging to a program path starting in $\mathbf{s}^*$ and reaching the program point $n \in N^*$ under consideration.

The *PMOP*-Solution: $\forall\, c_0 \in \mathcal{C}\ \forall\, n \in N^*.$
$PMOP(n)(c_0) = \sqcap \{ [\![p]\!](c_0) \mid p \in \mathbf{PP}_{G^*}[\mathbf{s}^*, n[\}$

This directly reflects our desires, but is in general not effective. For unidirectional bitvector problems, however, there exists an elegant and efficient way for computing the *PMOP*-solution by means of a fixpoint computation.

Remark 2.1 The local semantic functional as introduced above gives meaning to assignments rather than to their right-hand side terms. Assignments are thus implicitly considered atomic. However, interleavings between the evaluation of right-hand side terms and their subsequent assignments to the left-hand side variable can easily be modelled by (conceptually) splitting assignments of the form $x := t$ into the sequence $x_t := t$; $x := x_t$, where x_t is a fresh variable (cf. [10]).

Bitvector Analyses. Unidirectional bitvector problems are characterized by the simplicity of their local semantic functional $[\![\ ]\!] : N^* \to (\mathcal{B} \to \mathcal{B})$. It specifies the effect of a node n on a particular component of the bitvector, where $\mathcal{B}$ is the lattice $(\{ff, tt\}, \sqcap, \sqsubseteq)$ of Boolean truth values with $ff \sqsubseteq tt$ and the logical "and" as meet operation $\sqcap$ (or its dual counterpart). Important for our efficient fixpoint approach are the following obvious facts on the semantic domain $\mathcal{F}_\mathcal{B}$ of the monotonic Boolean functions $\mathcal{B} \to \mathcal{B}$ of bitvector analyses: (1) $\mathcal{F}_\mathcal{B}$ consists of the constant functions $Const_{tt}$ and $Const_{ff}$, together with the identity $Id_\mathcal{B}$ on $\mathcal{B}$ only. (2) All functions of $\mathcal{F}_\mathcal{B}$ are distributive. (3) $\mathcal{F}_\mathcal{B}$, together with the

pointwise ordering between functions, forms a complete lattice with least element $Const_{ff}$ and greatest element $Const_{tt}$, which is closed under function composition. Based on these facts, the following lemma, which follows by a simple induction on q, will be the key to the efficient computation of the "interleaving effect." It pin-points the specific nature of a domain of functions which only consists of constant functions and the identity function on a set M.

Lemma 2.2 (Main Lemma)
Let $f_i : \mathcal{F}_{\mathcal{B}} \to \mathcal{F}_{\mathcal{B}}$, $1 \leq i \leq q$, $q \in I\!N$, be functions on $\mathcal{F}_{\mathcal{B}}$. Then: $\exists\, k \in \{1,\ldots,q\}.\ f_q \circ \ldots \circ f_2 \circ f_1 = f_k \wedge \forall\, j \in \{k+1,\ldots,q\}.\ f_j = Id_{\mathcal{B}}$.

Interference. The relevance of Main Lemma 2.2 for bitvector problems is that it restricts the means of possible interference within a parallel program: each possible interference in a parallel program is due to a single statement in a parallel component, whose execution can be interleaved with the statement at the program point n under consideration, i.e., one of n's *interleaving predecessors* of $Pred^{Itlvg}_{G^*}(n)$. This is a consequence of Main Lemma 2.2 and the fact that for each node $m \in Pred^{Itlvg}_{G^*}(n)$, there exists a parallel path leading to n, whose last step requires the execution of m. Together with the obvious existence of a path to n that does not require the execution of any statement of $Pred^{Itlvg}_{G^*}(n)$, this implies that the only effect of interference is "destruction." This motivates the introduction of the predicate *NonDest* defined for each node $n \in N^*$ by

$$NonDest(n) =_{df} \forall\, m \in Pred^{Itlvg}_{G^*}(n).\ [\![\, m \,]\!] \in \{Const_{tt}, Id_{\mathcal{B}}\}$$

Only the constant function given by the precomputed value of this predicate is used below to model interference (cf. Definition 2.3), and in fact, the Parallel Bitvector Coincidence Theorem 2.4 guarantees that this modelling is sufficient.

Synchronization. In order to leave a parallel statement, all parallel components are required to terminate. The information, which is necessary to model this effect, can be computed by a hierarchical algorithm which only considers purely sequential programs. The central idea coincides with that of interprocedural DFA (cf. [9, 27]): we need to compute the effect of complete subgraphs, in this case of complete parallel components. This information is computed in an "innermost" fashion and then propagated to the next surrounding parallel statement. In essence, the complete three-step procedure $\mathcal{A}$ is a straightforward hierarchical adaptation of the functional version of the *maximal-fixed-point* (*MFP*) approach in the sense of Kam and Ullman [8] to the parallel setting. Here we only consider the second step realizing the synchronization at end nodes of parallel statements in more detail. In essence, this step can be reduced to the case of parallel statements $\bar{G}$ with purely sequential components $G_1, \ldots, G_k$. Thus, the global semantics $[\![\, G_i \,]\!]^*$ of the component graphs G_i of $\bar{G}$, $1 \leq i \leq k$, can be computed as in the sequential case. Afterwards, the global semantics $[\![\, \bar{G} \,]\!]^*$ of $\bar{G}$ is given by:

$$[\![\, \bar{G} \,]\!]^* = \begin{cases} Const_{ff} & \text{if } \exists\, G' \in \mathcal{G}_C(\bar{G}).\ [\![\, end(G') \,]\!] = Const_{ff} \\ Id_{\mathcal{B}} & \text{if } \forall\, G' \in \mathcal{G}_C(\bar{G}).\ [\![\, end(G') \,]\!] = Id_{\mathcal{B}} \\ Const_{tt} & \text{otherwise} \end{cases}$$

where $\mathcal{G}_C(\bar{G})$ denotes the set $\{G_1, \ldots, G_k\}$. Again, Main Lemma 2.2 is the key for proving the correctness of this step, i.e., $[\![\, G \,]\!]^*$ coincides with the desired *PMOP*-solution. As before the point is that a single statement is responsible for the entire effect of a path through a parallel statement. Thus, its effect is already given by the projection of this path onto the parallel component containing the vital statement. This is exploited in the synchronization step above. Adapting this step will be the key for overcoming the pitfalls of parallel CM in Section 3. After the hierarchical preprocess, whose correctness is a consequence of the Hierarchical Coincidence Theorem of [17], the following equation system is the key for characterizing the *PMOP*-solution of a unidirectional bitvector problem algorithmically:

Definition 2.3 *The functional $[\![\ ]\!] : N^* \to \mathcal{F}_{\mathcal{B}}$ is defined as the greatest solution of the equation system given by: $[\![\, n \,]\!] =$*

$$\begin{cases} Id_{\mathcal{B}} & \text{if } n = \mathbf{s}^* \\ [\![\, pfg(n) \,]\!]^* \circ [\![\, start(pfg(n)) \,]\!] \sqcap Const_{NonDest(n)} & \text{if } n \in N^*_X \\ \sqcap \{ [\![\, m \,]\!] \circ [\![\, m \,]\!] \mid m \in Pred_{G^*}(n) \} \sqcap Const_{NonDest(n)} & \text{otherwise} \end{cases}$$

where pfg denotes a function, which maps a node of a parallel subgraph to the smallest parallel subgraph containing it, and where start is a function, which maps a graph to its start node, and where N^*_X denotes the set of end nodes of parallel subgraphs.

In analogy to the *MFP*-solution of [8] for the sequential case, we can now define the $PMFP_{BV}$-solution of unidirectional bitvector problems for the parallel setting:

The $PMFP_{BV}$-Solution:

$$\forall\, b \in \mathcal{B}\ \forall\, n \in N^*.\ PMFP_{BV}(n)(b) = [\![\, n \,]\!](b)$$

As in the sequential case, the $PMFP_{BV}$-solution is practically relevant because it can efficiently be computed. Moreover, it coincides with the desired *PMOP*-solution (cf. [17]):

Theorem 2.4 (Parallel BV-Coincidence Th.) *Given a parallel flow graph G^*, and a local semantic functional $[\![\ ]\!] : N^* \to \mathcal{F}_{\mathcal{B}}$, the PMOP-solution and the $PMFP_{BV}$-solution coincide.*

3 Code Motion for Parallel Programs

In this section we develop our CM-algorithm for parallel programs, and demonstrate how to overcome the pitfalls illustrated in Section 1. In order to allow a simple and unparameterized notation we develop our algorithm with respect to an arbitrary, but specific program G^* and an arbitrary, but specific computation t. Without loss of generality we assume that the right-hand side terms of assignments contain at most one operator, i.e., we consider 3-address code. Additionally, we assume that all edges leading to a node outside the set of end nodes of parallel statements with more than one predecessor have been split by inserting a synthetic node. This is typical for CM-transformations (cf. [4, 12, 14]) in order to avoid the blocking of the code motion process by edges leading from a node with more than one successor to a node with more than one predecessor, which in sequential optimization are called *critical edges*.

3.1 Admissible Code Motion

As mentioned in Section 1 a CM-transformation *CM* must preserve the semantics. Intuitively, this requires that *CM* is *admissible*, i.e., *safe* and *correct*: "safe" means that on no program path the computation of a new value is introduced by inserting a computation of t; "correct" means that at program sites where the temporary $\mathbf{h}$ replaces an original computation of t, it always represents the same value as t. A sufficient condition, which is usually considered for code motion, is that two computations of t represent the *same value* on a path if no operand of t is modified between them. Thus, *CM* is admissible if $Insert_{CM}$ implies safety, and $Replace_{CM}$ implies correctness, where $Insert_{CM}$ and $Replace_{CM}$ are two predicates for nodes specifying the insertion and replacement points of *CM*. Note that the admissibility constraint holds analogously for the sequential and the parallel setting. In the following section we define the as-early-as-possible placement valid for the *sequential* setting. Subsequently, we show how to modify the DFA-analyses in order to overcome the pitfalls of code motion for *parallel* programs, when transferring the placement strategy to the parallel setting.

3.2 Earliest Down-Safe Placements

For sequential programs computational (and simultaneously executional) optimality can be obtained by placing computations at their "earliest (down-safe)" computation points, and by subsequently replacing all original computations by the temporary introduced for the computation under consideration. A node n is earliest (for t), if it is

- down-safe, i.e., if the value of t is required on every continuation of a program execution leaving n and reaching the end node,
- not up-safe, i.e., if the value of t is not already available at n, and if it is
- either the start node, or if the placement of t in some of n's predecessors would not be safe (it would introduce a new value on some path) or would not be transparent due to a modification of one of t's operands (a computation there would not yield the same value as in n).

Thus, the insertion points of the as-early-as-possible placement can be induced from the set of up-safe and down-safe program points. We are therefore simply left with specifying the local semantic functionals $[\![\ ]\!]_{us} : N^* \to (\mathcal{B} \to \mathcal{B})$ and $[\![\ ]\!]_{ds} : N^* \to (\mathcal{B} \to \mathcal{B})$ for up-safety and down-safety. They are defined for each node n of the argument program as shown below, where *Transp* and *Comp* denote as usual two local predicates being true for a node n, if n does not modify an operand of the computation t under consideration (i.e., the value of t is *transparent* for n ($Transp(n)$)), and if it contains a computation of t ($Comp(n)$), respectively:[2]

$$[\![n]\!]_{us} =_{df} \begin{cases} Const_{tt} & \text{if } Transp(n) \wedge Comp(n) \\ Id_{\mathcal{B}} & \text{if } Transp(n) \wedge \neg Comp(n) \\ Const_{ff} & \text{otherwise} \end{cases}$$

$$[\![n]\!]_{ds} =_{df} \begin{cases} Const_{tt} & \text{if } Comp(n) \\ Id_{\mathcal{B}} & \text{if } \neg Comp(n) \wedge Transp(n) \\ Const_{ff} & \text{otherwise} \end{cases}$$

These are just the semantic functionals known from the sequential case.[3] They can directly be fed into the generic algorithm of the framework of [17] (cf. Section 2), which then computes the set of up-safe and down-safe program points for sequential and parallel programs. Subsequently placing the computations at the "earliest down-safe" computation points and replacing all original computations by a reference to the corresponding temporary, leads to computationally (and simultaneously also executionally) optimal results in the sequential setting. However, for parallel programs neither executional optimality nor correctness of the transformation is guaranteed (see Section 1). In the following section we show how to elegantly overcome these problems.

[2] Note that up-safety requires a forward analysis of the argument program, whereas down-safety requires a backward analysis.

[3] In the literature these definitions are usually given in the following equivalent form:
$[\![n]\!]_{us}(b) = (b \vee Comp(n)) \wedge Transp(n)$ and $[\![n]\!]_{ds}(b) = Comp(n) \vee (Transp(n) \wedge b)$.

3.3 Overcoming the Pitfalls

3.3.1 Optimality

As illustrated in Section 1, the relation "computationally better" is inappropriate for comparing the efficiency of parallel programs. Hence, a strategy aiming at computational optimality like the as-earliest-as-possible one is inappropriate, too. As a new measure we introduce the relation "executionally better." As usual for code motion, assignments with a trivial right-hand side term (i.e., a variable or a constant) are considered to be for free, and assignments whose right-hand side term involves an operator are assumed to have unit costs.[4] Now the *execution time* of a parallel program path is given (structurally) as follows: for a parallel statement it is the maximum of the execution times of its components for the considered execution, and for a parallel program path, i.e., the sequential composition of elementary and parallel statements, it is the sum of the execution times of its components.

An admissible CM-transformation CM is *executionally better* than an admissible CM-transformation CM' if and only if for all paths p from the start node to the end node of the program the execution time of p in the program resulting from CM is less or equal to that of p in the program resulting from CM'.[5] Moreover, CM is *executionally optimal* if and only if it is executionally better than any other admissible code motion transformation. As executional optimality cannot be achieved in general, we will present an efficient algorithm which guarantees executional improvement only.

Note that in contrast to "computationally better," the relation "executionally better" separates the programs of Figure 2(b) and (c) as desired.

3.3.2 Recursive Assignments

The observation that an independent treatment of "recursive" assignments can lead to the loss of sequential consistency, is caused by the fact that recursive statements both *compute* and *modify* t. This property cannot be distinguished from a simple use in our abstract domain. While this distinction is unnecessary in the sequential setting, and unnecessary in the parallel one as long as one is only interested in the up-safety or down-safety property, this distinction is vital when using these properties for the placement transformation because of interference. This problem, however, can easily and elegantly be overcome in our framework by implicitly decomposing recursive assignments $x := t$ in parallel statements into sequences $x_t := t;\ x := x_t$, which are considered atomic, and where x_t is a fresh variable (cf. Remark 2.1): rather than changing the argument program, this implicit decomposition is realized by associating two semantic functions with recursive assignments (in parallel components). In fact, this is sufficient to completely decouple all computation patterns and their occurrences. Considering the examples in Figure 3(a) and (c) and the computation of down-safety for illustration, the effect of this implicit decomposition is that the predicate *NonDest* is set to false for nodes occurring in a parallel statement, whose right-hand side term contains an operand being assigned to in a parallel relative by a recursive or non-recursive assignment. In effect, this (together with the modification of Section 3.3.3) prevents the transformations displayed in Figures 3(b) and (d), and in Figures 4(b), (c), and (d). Note that this is a *semantic must* for the transformations of Figures 3(d) and 4(d) because they are semantically incorrect, and it is a *profitability must* for the transformations of Figures 4(b) and (c) as without additional (runtime!) information none of these transformations guarantees profitability, or is preferable to its counterpart.

3.3.3 Up-Safety and Down-Safety

The modification here must reestablish the two facts that (1) up-safety of a program point n guarantees that the computation under consideration can be made available in a temporary at n, and that (2) down-safety guarantees that the value of a temporary initialized at n can be used on *every* terminating program continuation. Denoting the new properties up-safe$_{par}$ and down-safe$_{par}$, this is illustrated for up-safe$_{par}$ in Figure 8.

To reestablish fact (1) on up-safety, it is sufficient to modify the synchronization step, when computing the semantics of parallel statements. In fact, the exit of a parallel statement is up-safe$_{par}$, if and only if the computation under consideration is available on entering the parallel statement and the parallel statement is transparent for it, or if it is made available by one of the parallel components, and none of the nodes of its same-level parallel relatives destroys the availability information. This is simply achieved by modifying the synchronization step of the three-step procedure $\mathcal{A}$ of Section 2 as follows:

$$[\![\bar{G}]\!]^* =$$

$$\begin{cases} Const_{tt} & \text{if } \exists G' \in \mathcal{G}_C(\bar{G}).\ [\![end(G')]\!] = Const_{tt} \wedge \\ & \quad \forall m \in Nodes(\mathcal{G}_C(\bar{G}) \setminus G').\ [\![m]\!] \neq Const_{ff} \\ Id_{\mathcal{B}} & \text{if } \forall G' \in \mathcal{G}_C(\bar{G}).\ [\![end(G')]\!] = Id_{\mathcal{B}} \\ Const_{ff} & \text{otherwise} \end{cases}$$

In order to guarantee that a temporary initialized on entering a parallel statement can be used at least once on every program continuation reaching the end node, the same modification as above proposed for up-safety

[4] Thus, we are implicitly assuming that all variables are shared. However, our results carry over to a refined model distinguishing between shared and local variables.

[5] Note that this relation is reflexive. *Executionally at least as good* would thus be the more precise, however, uglier term.

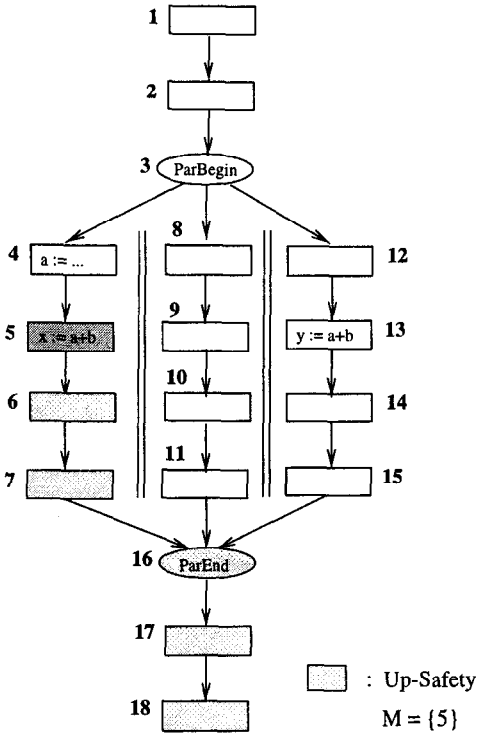


Figure 8: Up-safety refinement.

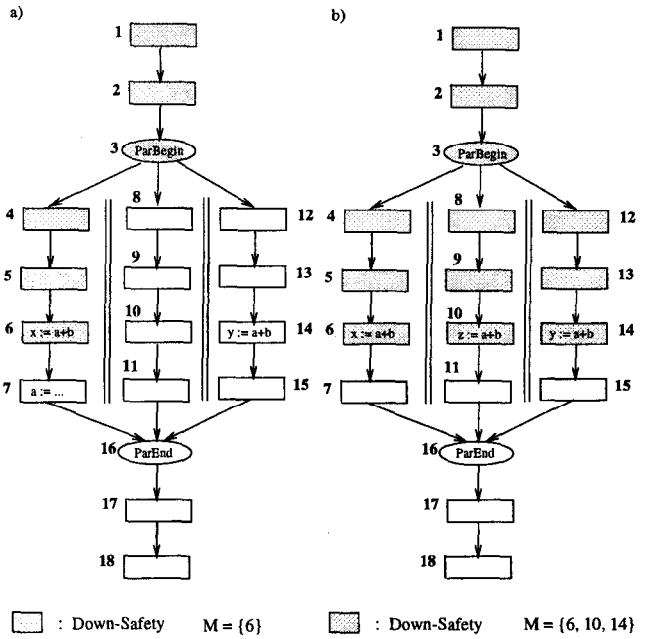


Figure 9: Down-safety refinement.

would suffice. This is illustrated in Figure 9(a). However, this would still allow moving a computation from a single component of a parallel program statement, where its execution is possibly for free, to a sequential program part, where it definitely counts. Thus, in order to never impair a program execution, we require that the entry of a parallel statement is down-safe$_{par}$ only if all its components satisfy this property and none of them contains an assignment modifying t as illustrated in Figure 9(b). In this situation moving a computation out of a parallel statement is safe as it is simultaneously moved out of all of its components, and hence, in particular, out of any corresponding bottleneck component.

As for up-safety$_{par}$ modifying the synchronization step of procedure $\mathcal{A}$ of Section 2 is sufficient:

$$[\![\bar{G}]\!]^* =$$

$$\begin{cases} Const_{tt} & \text{if } \forall G' \in \mathcal{G}_C(\bar{G}).\ [\![end(G')]\!] = Const_{tt} \wedge \\ & \quad \forall m \in Nodes(\mathcal{G}_C(\bar{G})).\ [\![m]\!] \neq Const_{ff} \\ Id_{\mathcal{B}} & \text{if } \forall G' \in \mathcal{G}_C(\bar{G}).\ [\![end(G')]\!] = Id_{\mathcal{B}} \\ Const_{ff} & \text{otherwise} \end{cases}$$

3.3.4 The Parallel CM-Transformation

After computing the set of up-safe$_{par}$ and down-safe$_{par}$ nodes the insertion points of our parallel CM-transformation are computed along the lines of Section 3.2 using up-safety$_{par}$ and down-safety$_{par}$. Subsequently, all original computations occurring in a $Safe_{par}$ node are replaced by the corresponding temporary, i.e., the insertion and replacement predicates are defined by $Insert(n) =_{df} Earliest_{par}(n)$, and $Replace(n) =_{df} Comp(n) \wedge Safe_{par}(n)$, where $Safe_{par}(n)$ denotes the disjunction of up-safe$_{par}(n)$ and down-safe$_{par}(n)$, and $Earliest_{par}(n)$ the conjunction of down-safe$_{par}(n)$ and the disjunction of n is equal to the start node and there is a predecessor of n failing the predicate $Safe_{par}$.

Intuitively, this transformation moves computations as far as possible in the opposite direction of the control flow while maintaining admissibility *and* the parallelism of the argument program. In contrast, a "pure" as-early-as-possible placement strategy maintains admissibility only. The new placement strategy realized by our algorithm for parallel code motion is an adequate natural adaption of the as-early-as-possible placement strategy for sequential programs to the parallel setting. The correctness and the profitability of our new algorithm are rather straightforward to prove. Moreover, we conjecture that it is impossible to improve this transformation without an explicit consideration of the bottleneck components of parallel statements. The example of Figure 10 illustrates the power of the complete transformation. It is particularly highlighted by the different treatment of the terms $a+b$, $c+d$, $e+f$, $g+h$, and $j+k$. The transformation removes the loop invariant computations of $g+h$ and $j+k$ by placing them inside the parallel statement in front of their respective loops. Similarly, this holds for the computation of $c+d$, which remains inside the parallel statement as its computation can be for free at this point, whereas it would definitely count at an earlier program point. In con-

trast, the computation of $a + b$ can safely be placed outside the parallel statement as it is computed in both parallel components. Combining this with the fact that $a + b$ is also computed in the left branch leaving node **6**, $a + b$ can safely be moved to node **1**.

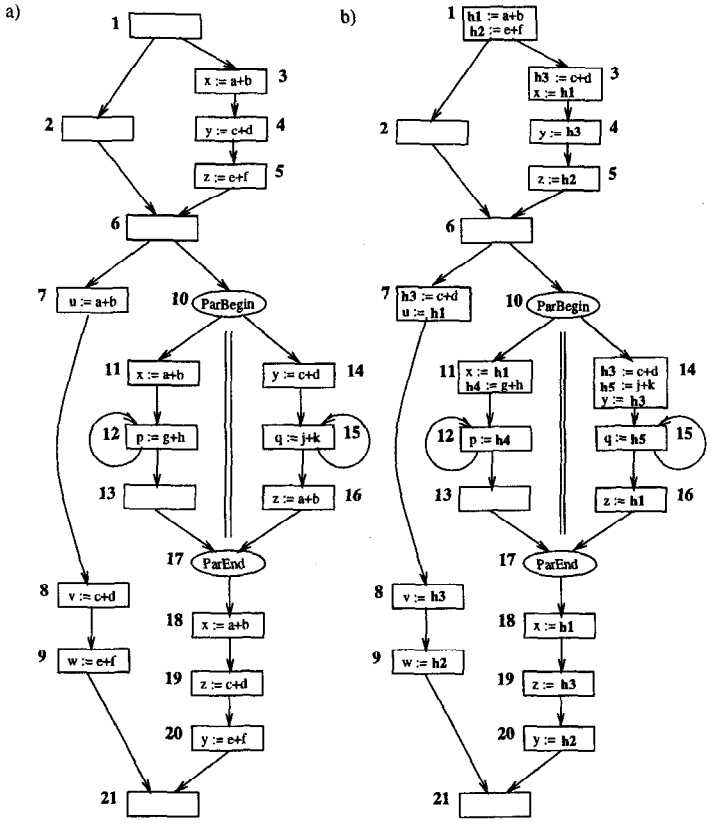


Figure 10: The power of the complete transformation.

4 Conclusions

Using the framework of [17] it is possible to transfer unidirectional bitvector analyses to parallel programs, and to solve them as efficiently as for sequential ones. This is highly relevant in practice because of the broad variety of powerful classical optimizations like *code motion* [14], *strength reduction* [13], *partial dead-code elimination* [15], and *assignment motion* [16], which only require bitvector analyses of this type. However, transferring transformations to the parallel setting is more problematic as we demonstrated here by means of code motion. The point is that thinking in terms of "interleaved" program paths is insufficient when considering performance. Our algorithm takes this observation into account. It is unique in eliminating partially redundant computations in a parallel program, while guaranteeing safety and executional improvement. It is worth noting that all the required modifications concern the generic algorithm of the framework only. Thus programmers applying the framework do not have to bother about them at all.

We developed our algorithm for a minimum parallel setting in order to focus on the essence of the problems one encounters when transferring code motion to parallel programs. However, our technique can also be applied to extended settings, e.g. comprising explicit synchronization or extensions to task-parallel languages e.g. in the fashion of Java. This leads to extremely efficient however less precise analyses. We are currently empirically investigating the impact of language extensions to precision.

Acknowledgements. We thank Jürgen Vollmer for his contributions during the development of the analysis framework for parallel programs.

References

[1] J.-H. Chow and W. L. Harrison. Compile time analysis of parallel programs that share memory. In *Conf. Rec. 19th Symp. Principles Prog. Lang. (POPL'92)*, pages 130 – 142. ACM, NY, 1992.

[2] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Conf. Rec. 4th Symp. Principles of Prog. Lang. (POPL'77)*, pages 238 – 252. ACM, NY, 1977.

[3] P. Cousot and R. Cousot. Abstract interpretation frameworks. *J. Logic and Computation*, 2(4):511 – 547, 1992.

[4] D. M. Dhamdhere. Practical adaptation of the global optimization algorithm of Morel and Renvoise. *ACM Trans. Prog. Lang. Syst.*, 13(2):291 – 294, 1991. Tech. Corr.

[5] M. B. Dwyer and L. A. Clarke. Data flow analysis for verifying properties of concurrent programs. In *Proc. 2nd ACM SIGSOFT'94 Symp. on Foundations of Software Eng. (FSE'94)*, volume *19*,5 of *Software Eng. Notes*, pages 62 – 75, 1994.

[6] D. Grunwald and H. Srinivasan. Data flow equations for explicitly parallel programs. In *Proc. ACM SIGPLAN Symp. on Principles of Parallel Progr. (PPoPP'93)*, volume *28*,7 of *ACM SIGPLAN Not.*, pages 159–168, 1993.

[7] M. S. Hecht. *Flow Analysis of Computer Programs.* Elsevier, North-Holland, 1977.

[8] J. B. Kam and J. D. Ullman. Monotone data flow analysis frameworks. *Acta Informatica*, 7:305 – 317, 1977.

[9] J. Knoop. *Optimal Interprocedural Program Optimization: A new Framework and its Application.* PhD thesis, Univ. of Kiel, Germany, 1993. LNCS Tutorial 1428, Springer-V., 1998.

[10] J. Knoop. Eliminating partially dead code in explicitly parallel programs. *TCS*, 196(1-2):365 – 393, 1998. (Special issue devoted to *Euro-Par'96*).

[11] J. Knoop. Parallel constant propagation. In *Proc. 4th Europ. Conf. on Parallel Processing* (*Euro-Par'98*), LNCS 1470, pages 445 – 455. Springer-V., 1998.

[12] J. Knoop, O. Rüthing, and B. Steffen. Lazy code motion. In *Proc. ACM SIGPLAN Conf. on Prog. Lang. Design and Impl.* (*PLDI'92*), volume *27*,7 of *ACM SIGPLAN Not.*, pages 224 – 234, 1992.

[13] J. Knoop, O. Rüthing, and B. Steffen. Lazy strength reduction. *J. Prog. Lang.*, 1(1):71–91, 1993.

[14] J. Knoop, O. Rüthing, and B. Steffen. Optimal code motion: Theory and practice. *ACM Trans. Prog. Lang. Syst.*, 16(4):1117–1155, 1994.

[15] J. Knoop, O. Rüthing, and B. Steffen. Partial dead code elimination. In *Proc. ACM SIGPLAN Conf. on Prog. Lang. Design and Impl.* (*PLDI'94*), volume *29*,6 of *ACM SIGPLAN Not.*, pages 147 – 158, 1994.

[16] J. Knoop, O. Rüthing, and B. Steffen. The power of assignment motion. In *Proc. ACM SIGPLAN Conf. on Prog. Lang. Design and Impl.* (*PLDI'95*), volume *30*,6 of *ACM SIGPLAN Not.*, pages 233 – 245, 1995.

[17] J. Knoop, B. Steffen, and J. Vollmer. Parallelism for free: Efficient and optimal bitvector analyses for parallel programs. *ACM Trans. Prog. Lang. Syst.*, 18(3):268 – 299, 1996.

[18] A. Krishnamurthy and K. Yelick. Optimizing parallel SPMD programs. In *Proc. 7th Int. Conf. on Lang. and Compilers for Parallel Comp.* (*LCPC'94*), LNCS 892, pages 331 – 345. Springer-V., 1994.

[19] A. Krishnamurthy and K. Yelick. Optimizing parallel programs with explicit synchronization. In *Proc. ACM SIGPLAN Conf. on Prog. Lang. Design and Impl.* (*PLDI'95*), volume *30*,6 of *ACM SIGPLAN Not.*, pages 196 – 204, 1995.

[20] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Trans. Comput.*, 28(9):690 – 691, 1979.

[21] J. Lee, S. P. Midkiff, and D. A. Padua. Concurrent static single assignment form and constant propagation for explicitly parallel programs. In *Proc. 10th Int. Conf. on Lang. and Compilers for Parallel Comp.* (*LCPC'97*), pages 114 – 130, 1997.

[22] D. Long and L. Clarke. Data flow analysis of concurrent systems that use the rendezvous model of synchronization. In *Proc. ACM SIGSOFT Symp. on Testing, Analysis, and Verification* (*TAV'91*), volume *16* of *Software Eng. Notes*, pages 21 – 35, 1991.

[23] K. Marriot. Frameworks for abstract interpretation. *Acta Informatica*, 30:103 – 129, 1993.

[24] S. P. Midkiff and D. A. Padua. Issues in the optimization of parallel programs. In *Proc. Int. Conf. on Parallel Processing, Vol. II*, pages 105 – 113, 1990.

[25] S. S. Muchnick. *Advanced Compiler Design and Implementation.* Morgan Kaufmann, San Francisco, CA, 1997.

[26] S. S. Muchnick and N. D. Jones, editors. *Program Flow Analysis: Theory and Applications.* Prentice Hall, Englewood Cliffs, NJ, 1981.

[27] M. Sharir and A. Pnueli. Two approaches to interprocedural data flow analysis. In S. S. Muchnick and N. D. Jones, editors, *Program Flow Analysis: Theory and Applications*, chapter 7, pages 189 – 233. Prentice Hall, Englewood Cliffs, NJ, 1981.

[28] D. Shasha and M. Snir. Efficient and correct execution of parallel programs that share memory. *ACM Trans. Prog. Lang. Syst.*, 10(2):282 – 312, 1988.

[29] H. Srinivasan, J. Hook, and M. Wolfe. Static single assignment form for explicitly parallel programs. In *Conf. Rec. 20th Symp. on Principles of Prog. Lang.* (*POPL'93*), pages 260 – 272. ACM, NY, 1993.

[30] H. Srinivasan and M. Wolfe. Analyzing programs with explicit parallelism. In *Proc. 4th Int. Conf. on Lang. and Compilers for Parallel Comp.* (*LCPC'91*), LNCS 589, pages 405 – 419. Springer-V., 1991.

[31] M. Wolfe. *High performance Compilers for Parallel Computing.* Addison-Wesley, NY, 1996.

[32] M. Wolfe and H. Srinivasan. Data structures for optimizing programs with explicit parallelism. In *Proc. 1st Int. Conf. of the Austrian Center for Parallel Computation*, LNCS 591, pages 139 – 156. Springer-V., 1991.

[33] H. Zima and B. Chapman. *Supercompilers for Parallel and Vector Computers.* Addison-Wesley, NY, 1991.