To be published in the
Proceedings of the 12th GI Workshop on "Alternative Konzepte für Sprachen und Rechner"
Physikzentrum Bad Honnef, May 2–4 1995

Optimal Code Motion for Parallel Programs

– Extended Abstract *–

Jens Knoop † Bernhard Steffen† Jürgen Vollmer ‡

Keywords: Parallelism, interleaving semantics, synchronization, compiler, program optimization, data flow analysis, bitvector problems, code motion, partial redundancy elimination.

Motivation

Parallel languages are of growing interest, as they are more and more supported by modern hardware environments. However, despite their importance [SHW, SW, WS], there is currently very little work on classical *analyses* and *optimizations* for parallel programs. Probably, the reason for this deficiency is that a naive adaptation fails [MP] and the straightforward correct adaptation needs an unacceptable effort which is caused by considering all interleavings that manifest the possible executions of a parallel program.

Thus, either heuristics are proposed to avoid the consideration of all the interleavings [McD], or restricted situations are considered, which do not require to consider the interleavings at all [GS].[1] Completely different is the approach of abstract interpretation-based state space reduction proposed in [CH1, CH2]. This, however, requires the construction of an appropriately reduced version of the global state space, which is often still unmanageable.

In [KSV1] we have recently demonstrated that for the large class of *bitvector analyses*, which are most relevant in practice, there is an elegant way to avoid the state explosion problem. In fact, we have shown how to construct arbitrary bitvector analysis algorithms for parallel programs with shared memory that

1. optimally cover the phenomenon of *interference*
2. are as *efficient* as their sequential counterparts and
3. *easy* to implement.

The key for obtaining this result was the observation that during bitvector analyses the different interleavings of the executions of parallel components need not be considered, although they are semantically different. As a consequence, all the well-known bitvector algorithms for liveness, availability, very business, reaching definitions, definition-use chains (cf. [He]) can easily be adapted for parallel programs at almost no cost on the runtime and the implementation side.

This is practically important as there is a broad variety of powerful classical program optimizations including *code motion* (cf. [KRS1, KRS2]), *strength reduction* (cf. [KRS3]), *partial dead code elimination* (cf. [KRS4]), and *assignment motion* (cf. [KRS5]) which are solely based on bitvector analyses. All these optimizations are now available for parallel programs.

*For the full version of this paper see [KSV2].

†Fakultät für Mathematik und Informatik, Universität Passau, Innstrasse 33, D-94032 Passau, Germany. E-mail: {knoop,steffen}@fmi.uni-passau.de

‡Institut für Programm- und Datenstrukturen, Universität Karlsruhe, Vincenz-Prießnitz-Straße 3, D-76128 Karlsruhe, Germany. E-mail: vollmer@ipd.info.uni-karlsruhe.de

[1] In [GS] this is achieved by requiring data independence of the parallel components.

Here, we demonstrate this by sketching a code motion algorithm, which is unique in placing the computations in a parallel program *computationally optimal*. Intuitively, this means that in the program resulting from our algorithm there is no program path, on which the number of computations can be reduced any more by means of a semantics preserving code motion transformation. Moreover, this algorithm is as efficient as its underlying sequential counterpart of [KRS1, KRS2].

Fundamental for the proof of optimality is the Parallel Bitvector Coincidence Theorem of [KSV1], which provides a sufficient condition for the coincidence of the specifying parallel *meet over all paths* solution of a data flow analysis problem and the parallel *maximal fixed point* solution that is computed by our algorithm. Fundamental for proving the efficiency result is the restriction to bitvector problems, which due to their structural simplicity do not require the consideration of the different interleavings (see [KSV1] for details).

The power of the new algorithm is illustrated by means of the example of Figure 1, where the components of parallel statements are visualized by means of parallel vertical lines. Here, our algorithm is unique to obtain the optimization result displayed in Figure 2. It eliminates the partially redundant computations of $a + b$ at the nodes **3**, **10**, **12**, **14**, **21**, **22**, **30** by moving them to the nodes **2**, **11** and **19**, but it does not touch the partially redundant computations of $a + b$ at the nodes **7** and **9**, which cannot safely be eliminated.

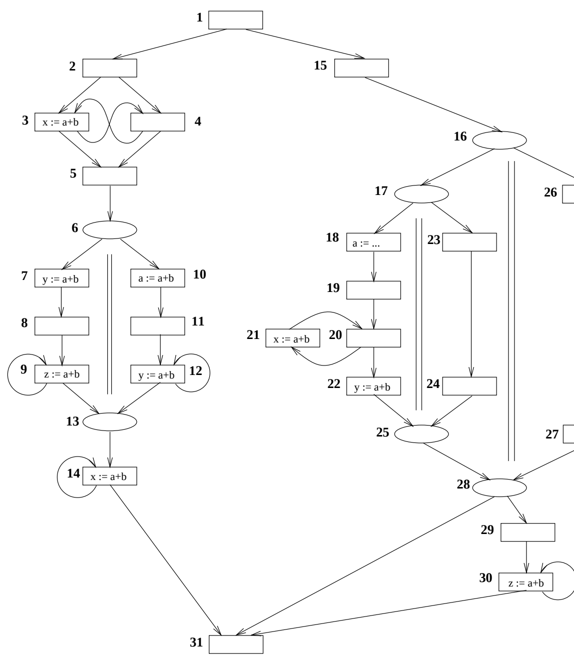

Figure 1: The Motivating Example

In the following we give a sketch of this algorithm, which works for programs of a parallel imperative programming language with an interleaving semantics, where parallelism is syntactically expressed by means of a parallel statement whose components are assumed to be executed independently and in parallel on a shared memory.

Sketch of the Algorithm

Intuitively, 'code motion' 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 that are initialized at suitable program points (cf. [MR]).

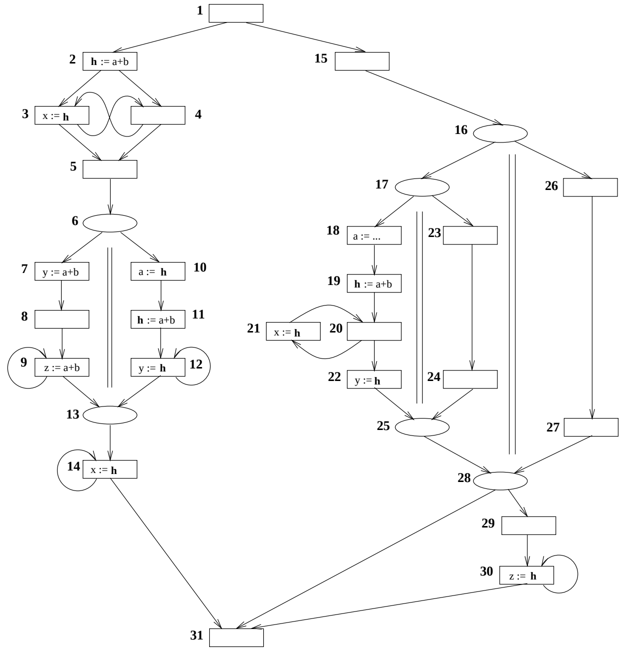

Figure 2: The Computationally Optimal Transformed Program

As in the sequential setting, the central idea to achieve *computationally optimal* results is to place the initializations of the temporaries *as early as possible* in a program, while maintaining *safety* and *correctness* (cf. [KRS1, KRS2]). Intuitively, 'safety' means that there is no program path, on which the computation of a new value is introduced; 'correctness' means that the temporaries are properly initialized, i.e., they always represent the same value as the computation they replace.

In order to determine the program points that are earliest in this sense, it suffices to compute the set of *down-safe* and *up-safe* program points (cf. [KRS2]). Intuitively, a program point is 'down-safe' ('up-safe'), if there is no program path starting at this point reaching the end node (if there is no program path from the start node of the program reaching it), on which the computation of a new value is introduced. In fact, a program point is safe if and only if it is down-safe or up-safe. Moreover, it is *earliest* or *as early as possible* in order to be more precise, if and only if it is safe and if one of its immediate predecessors is not safe or modifies an operand of the computation under consideration.

Technically, the set of down-safe and up-safe program points are characterized by the solutions of the *PMFP*-approach, the parallel version of the maximal fixed point approach in the sense of Kam and Ullman (cf. [KU]). In the parallel setting, however, the computation of the maximal fixed point solution is preceded by a preprocess which determines the semantics of parallel statements in terms of transformations of data flow informations. This preprocess is successively applied to all parallel statements of the argument program from inside to outside. It works by iteratively computing for every statement *st* of the currently investigated parallel statement a function, which transforms data flow information that is assumed to be valid at the entry of the parallel statement under consideration into the information that can be guaranteed before the execution of *st*. The point here is that for bitvector problems the interference caused by the interleaved execution of parallel components can completely be captured by only investigating the local effects of the elementary statements occurring in components that can be executed in parallel. The semantics of the parallel statement itself is then essentially given by the meet of the functions associated with the last statements of each of its parallel components. After this hierarchical preprocess (see [KSV1] for details), which is quite similar in spirit to the preprocess of computing the semantics of procedure calls in interprocedural data flow analysis (cf. [KS]), the maximal fixed point solution can be computed essentially as in the sequential case.

All partially redundant computations of a program are then eliminated by performing the following three step transformation for all program terms t:

- Declare a new temporary tmp_t.
- Insert initialization statements $tmp_t := t$ at the entry of all nodes being *earliest* for t.
- Replace all original computations of t occurring in nodes being *safe* for t by tmp_t.

Figure 3 shows the result of computing the set of down-safe and up-safe program points for the term $a + b$. Moreover, it shows the set of earliest program points and of the set of nodes, where an original computation of $a + b$ has to be replaced.

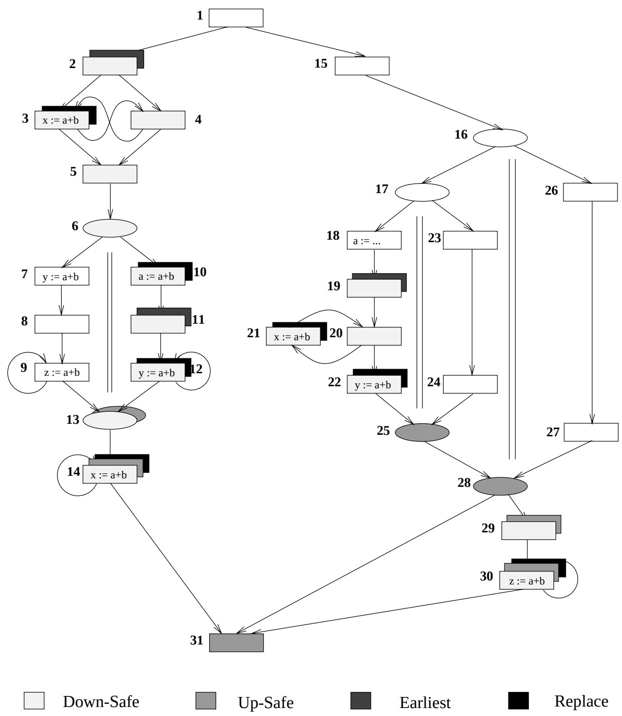

Figure 3: Down-Safe, Up-Safe, Earliest, and Replacement Program Points of $a + b$

Applying the procedure above to the program of Figure 1 results in the promised program of Figure 2. It is worth noting that the transformed program is indeed computationally optimal.

A variant of the code motion algorithm sketched here is implemented in the ESPRIT project COMPARE [Vo1, Vo2].

References

[CH1] Chow, J.-H., and Harrison, W. L. Compile time analysis of parallel programs that share memory. In *Conference Record of the 19^{th} International Symposium on Principles of Programming Languages (POPL'92)*, Albuquerque, New Mexico, 1992, 130 - 141.

[CH2] Chow, J.-H., and Harrison, W. L. State Space Reduction in Abstract Interpretation of Parallel Programs. In *Proceedings of the International Conference on Computer Languages, (ICCL'94)*, Toulouse, France, May 16-19, 1994, 277-288.

[GS] Grunwald, D., and Srinivasan, H. Data flow equations for explicitely parallel programs. In *Proceedings of the ACM SIGPLAN Symposium on Principles of Parallel Programming (PPOPP'93), SIGPLAN Notices 28*, 7 (1993).

[He] Hecht, M. S. Flow analysis of computer programs. Elsevier, North-Holland, 1977.

[KRS1] Knoop, J., Rüthing, O., and Steffen, B. Lazy code motion. In *Proceedings of the ACM SIGPLAN'92 Conference on Programming Language Design and Implementation (PLDI'92)*, San Francisco, California, *SIGPLAN Notices 27*, 7 (1992), 224 - 234.

[KRS2] Knoop, J., Rüthing, O., and Steffen, B. Optimal code motion: Theory and practice. *Transactions on Programming Languages and Systems 16*, 4 (1994), 1117 - 1155.

[KRS3] Knoop, J., Rüthing, O., and Steffen, B. Lazy strength reduction. *Journal of Programming Languages 1*, 1 (1993), 71 - 91.

[KRS4] Knoop, J., Rüthing, O., and Steffen, B. Partial dead code elimination. In *Proceedings of the ACM SIGPLAN'94 Conference on Programming Language Design and Implementation (PLDI'94)*, Orlando, Florida, *SIGPLAN Notices 29*, 6 (1994), 147 - 158.

[KRS5] Knoop, J., Rüthing, O., and Steffen, B. The power of assignment motion. To appear in *Proceedings of the ACM SIGPLAN'95 Conference on Programming Languag e Design and Implemantion (PLDI'95)*, La Jolla, California, June 18 - 21, 1995.

[KS] Knoop, J., and Steffen, B. The interprocedural coincidence theorem. In *Proceedings of the 4^{th} International Conference on Compiler Construction (CC'92)*, Paderborn, Germany, Springer-Verlag, LNCS 641 (1992), 125 - 140.

[KSV1] Knoop, J., Steffen, B., and Vollmer, J. Parallelism for free: Efficient and optimal bitvector analyses for parallel programs. In *Proceedings of the International Workshop on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'95)*, Aarhus, Denmark, BRICS Notes Series NS-95-2 (1995), 319 - 333.

[KSV2] Knoop, J., Steffen, B., and Vollmer, J. Optimal code motion for parallel programs. Fakultät für Mathematik und Informatik, Universität Passau, Germany, MIP-Bericht (1995).

[KU] Kam, J. B., and Ullman, J. D. Monotone data flow analysis frameworks. *Acta Informatica 7*, (1977), 309 - 317.

[McD] McDowell, C. E. A practical algorithm for static analysis of parallel programs. *Journal of Parallel and Distributed Computing 6*, 3 (1989), 513 - 536.

[MP] Midkiff, S. P., and Padua, D. A. Issues in the optimization of parallel programs. In *Proceedings of the International Conference on Parallel Processing, Volume II*, St. Charles, Illinois, (1990), 105 - 113.

[MR] Morel, E., and Renvoise, C. Global optimization by suppression of partial redundancies. *Communications of the ACM 22*, 2 (1979), 96 - 103.

[SHW] Srinivasan, H., Hook, J., and Wolfe, M. Static single assignment form for explicitly parallel programs. In *Conference Record of the 20^{th} ACM SIGPLAN Symposium on Principles of Programming Languages (POPL'93)*, Charleston, South Carolina, 1993, 260 - 272.

[SW] Srinivasan, H., and Wolfe, M. Analyzing programs with explicit parallelism. In *Proceedings of the 4^{th} International Conference on Languages and Compilers for Parallel Computing*, Santa Clara, California, Springer-Verlag, LNCS 589 (1991), 405 - 419.

[Vo1] Vollmer, J. Data flow equations for parallel programs that share memory. Tech. Rep. 2.11.1 of the ESPRIT Project COMPARE (1994), Fakultät für Informatik, Universität Karlsruhe, Germany.

[Vo2] Vollmer, J. Data flow analysis of parallel programs. To appear in *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT'95)*, Limassol, Cyprus, June 26 - 29, 1995.

[WS] Wolfe, M, and Srinivasan, H. Data structures for optimizing programs with explicit parallelism. In *Proceedings of the 1^{st} International Conference of the Austrian Center for Parallel Computation*, Salzburg, Austria, Springer-Verlag, LNCS 591 (1991), 139 - 156.