Automatic Optimization of Dynamic Scheduling in Logic Programs

Germán Puebla and *Manuel Hermenegildo*

{german,herme}@fi.upm.es
Department of Artificial Intelligence
Technical University of Madrid (UPM)

(Abstract)

1 Dynamic Scheduling

Many modern logic programming languages provide more flexible scheduling than the Prolog traditional left-to-right computation rule. Computation generally also proceeds following some fixed scheduling rule but certain goals are dynamically "delayed" until their arguments are sufficiently instantiated to allow the call to run efficiently. This general form of scheduling is referred to as *dynamic scheduling*. Languages with dynamic scheduling also include constraint programming languages in which constraints which are "too hard" are delayed. In addition, most implementations of concurrent (constraint) programming languages essentially also follow a fixed left to right scheduling rule with suspension, where such suspension is controlled by the conditions in the *ask* guards. In fact, it has been shown that many such languages can be directly translated into (constraint) logic programs with dynamic scheduling with competitive efficiency. As a result, languages with dynamic scheduling are being seen more and more as very useful targets for prototyping or even implementing concurrent languages.

2 Optimization of Dynamic Scheduling

Dynamic scheduling increases the expressive power of (constraint) logic programs, but in most implementations it also introduces significant run-time overhead. The objective of our optimization is to reduce as much as possible this additional overhead by means of global analysis and program transformation, while preserving the semantics of the original programs. Previous work on optimization of dynamic scheduling has concentrated on detecting non-suspension (e.g., [4]) or on eliminating dynamic scheduling when it is not needed and/or producing reorderings in which dynamic scheduling is not needed any more (e.g., [1]). However, finding an order of literals in which no dynamic scheduling is needed does not guarantee that efficiency is improved. By this we do not mean to say that reordering should not be performed, but rather that it should only be performed when some conditions are met. In [7] we present optimization techniques which treat all the usual forms of delay declarations and are both correct and efficient in the sense of [5, 3], i.e., the observables are preserved and computation time is never increased (the *no slow-down* property is met). These optimizations include simplification of delay conditions in `when` meta-calls, and elimination of such meta-calls when not needed. Regarding `block` declarations, since they affect all the literals that call the corresponding predicate, they can in principle be simplified only if the simplification is allowed in *all* the such literals. This is overcome by means of *multiple specialization* which involves the generation of several versions of a predicate for different uses.

3 Implementation and Experimental Results

The optimization techniques presented in [7] have been implemented in the CIAO compiler [6] using newly available analysis techniques [2]. This is, to the best of our knowledge, the first implementation and integration in a compiler of an optimizing technique for dynamic scheduling. A series of benchmark programs have been implemented in a reversible way, so that they can be used in two modes of operation, forwards and backwards, through the use of suspension declarations. Note that though the declarative meaning of these programs explains both modes of operation, the fixed left-to-right scheduling rule does not allow running them backwards. The results show that optimization times are comparable to the time that the SICStus compiler takes to compile the same benchmarks into compact-code. For forward execution, all delay declarations are eliminated, obtaining a program which is as efficient as the original program designed to work forwards. This means that it is possible to write programs that are reversible (either by using the delay declarations directly or by using a higher level language for which the compiler generates delay declarations automatically) without incurring any run-time overhead when executing forwards. For backward execution, many of the delay declarations are needed and thus are not always completely eliminated by the optimizer. However, even in this case some speed-up is obtained due to the optimizer simplifying the suspension conditions.

References

1. J. Boye. Avoiding dynamic delays in functional logic programs. In *Programming Language Implementation and Logic Programming*, number 714 in LNCS, pages 12–27, Estonia, August 1993. Springer-Verlag.
2. M. García de la Banda, K. Marriott, and P. Stuckey. Efficient Analysis of Constraint Logic Programs with Dynamic Scheduling. In *1995 International Logic Programming Symposium*, Portland, Oregon, December 1995. MIT Press.
3. María José García de la Banda García. *Independence, Global Analysis, and Parallelism in Dynamically Scheduled Constraint Logic Programming*. PhD thesis, Universidad Politécnica de Madrid (UPM), July 1994.
4. M. Hanus. Analysis of Nonlinear Constraints in CLP(R). In *Tenth International Conference on Logic Programming*, pages 83–99. MIT Press, June 1993.
5. M. Hermenegildo and F. Rossi. Strict and Non-Strict Independent And-Parallelism in Logic Programs: Correctness, Efficiency, and Compile-Time Conditions. *Journal of Logic Programming*, 22(1):1–45, 1995.
6. M. Hermenegildo, F. Bueno, M. García de la Banda, and G. Puebla. The CIAO Multi-Dialect Compiler and System: An Experimentation Workbench for Future (C)LP Systems. In *Proccedings of the ILPS'95 Workshop on Visions for the Future of Logic Programming*, Portland, Oregon, USA, December 1995.
7. G. Puebla and M. Hermenegildo. Automatic optimization of dynamic scheduling in logic programs. Technical report, Technical University of Madrid, 1996. Available from `http://www.clip.dia.fi.upm.es/`.