Ordering Optimisations for Concurrent Logic Programs

Andy King and Paul Soper

Department of Electronics and Computer Science,
University of Southampton, Southampton, S09 5NH, UK.

Abstract

Ordering optimisations are optimisations that can be applied to a concurrent logic program when the atoms of a clause are known to be ordered. In this paper ordering optimisations are reviewed, reformulated and refined. The paper explains how ordering optimisations can be realised in terms of abstract interpretation and shows that by, building on schedule analysis, simple, efficient and accurate forms of abstract interpretation can achieved. The paper outlines how to: identify instances of unification which can be simplified or removed; distinguish repeated synchronisation instructions; indicate which redundant checks can be removed when producers are ordered before consumers in the same thread; identify which variables can be accessed without dereferencing; indicate where variable initialisation and unification can be simplified; and show which variables can be allocated to an environment. Some safety checks can also be removed by using mode information.

1 Introduction

Schedule analysis[1] can be used to introduce sequential threads into a concurrent logic program. The analysis is an adaptation of the dependence analysis[2] proposed by Traub for lenient functional languages. A partial schedule of processes is determined at compile-time which does not contradict any data-dependence of the program. The data-dependencies are inferred from producer and consumer analysis[3]. In general all processes cannot be totally ordered and so the analysis leads to a division into threads of totally ordered processes. In this way the work required of the run-time scheduler is reduced to ordering sequential threads rather than ordering individual processes. This is likely to be a useful optimisation since the overhead of invoking a scheduler is significant.

Parlog supports sequential conjunction and thus provides a way to order to order the evaluation of atoms within a clause. Typically its use as a programming device is limited, and Gregory[4] argues for its inclusion in Parlog primarily as a mechanism for controlling granularity. Schedule analysis builds on this work by providing a mechanism which is both automatic and formally proven (the behaviour of the program is preserved) for removing the overheads of fine-grained parallelism.

Gregory[4] documents several optimisations that can be applied in connection with sequential conjunction. These optimisations, which we collectively call ordering optimisations, generalise to threads. Gregory advocates that the programmer should indicate where sequential conjunction can be introduced into the program. Schedule analysis, instead of relying on the programmer, systematically generates threads. Schedule analysis is likely, therefore, to identify circumstances for which atoms can be ordered, which the programmer does not. A consequence of this is likely to be more scope for optimisation. Indeed, in connection with schedule analysis, ordering optimisations first proposed for sequential conjunction might assume a new importance. A notable special case for which ordering optimisations are of particular value is when a concurrent logic program is compiled to a uniprocessor.

In this paper we review, reformulate and refine ordering optimisations. We explain how ordering optimisations which relate to sequential conjunction can be reformulated in terms of abstract interpretation. We argue that by building on threads, simple and effective forms of abstract interpretation can be achieved. This approach not only extends the scope of the ordering optimisations, but additionally, the information inferred by abstract interpretation can be used to identify where other optimisations can be applied. We show how optimisations reported in the Prolog literature can be adapted to threads by combining mode analysis and type analysis, described in Section 2, with reference analysis, described in Section 3. (Section 3.2 presents an exception to this since we detail how to perform the safety check at compile-time. This optimisation is just as applicable to Prolog as it is to a concurrent logic language, and yet it does not seem to have been mentioned in the Prolog literature.) Section 4 presents our concluding discussion.

2 Mode analysis and type analysis

Mode analysis is a well-established analysis technique which infers how the arguments of a predicate (or an atom) can be expected to be bound when the predicate (or the atom) is invoked. Knowing that an argument is bound, or unbound, for instance, enables unification to be specialised. Type analysis can be regarded as an extension of mode analysis. Instead of inferring that arguments are expected to be unbound, bound or ground, the classification is refined to additionally deduce type information. Type analysis typically might infer that a variable is unbound or bound to a number, string, list or tuple. A knowledge of which types can be anticipated can, for example, be used to further simplify unification.

Mode analysis and type analysis can be regarded as forms of top-down abstract interpretation. A consequence of this is that the analyses critically depend on control. Debray[5] explains how modes can be inferred when a total (left-to-right) ordering of the atoms of a clause is replaced with a partial ordering. This approach can be refined to deduce type information too. However, the generality of substituting a total ordering with a partial ordering reduces the accuracy, and therefore the usefulness, of the mode or type information. Furthermore, the and-parallelism of a concurrent logic program means that the only ordering that exists between the atoms of a clause is imposed by the data-dependencies between the atoms. Therefore, without considering the ordering imposed by data-dependencies, the mode or type information inferred by the technique of Debray[5] is likely to be of little use.

Codognet *et al.*[6] and Codish *et al.*[7] detect possible deadlocks in concurrent logic programs by reasoning about groundness and sharing and including special machinery in the analysis to model the possible suspension of processes. Such analyses consider the data-dependencies between atoms of the clause and therefore provide a framework to infer accurate mode and type information. However, a simpler and more efficient alternative is to apply the analysis of Debray to threads. The method is particularly attractive since the partial order defined by the threads often carries more information than the partial order defined by the data-dependencies. Thus accurate modes and types are likely to be inferred. The fundamental difference between this technique and the approach described by Codognet *et al.*[6] and Codish *et al.*[7] is that data-dependencies are considered at an earlier stage of compilation, in the generation of threads.

2.1 Write-first occurrence optimisation

Gregory[4] first described how the unification of a variable with a term can be replaced with a form of assignment if the variable is a write-first occurrence of a variable. Gregory[4] lists conditions which define a write-first occurrence of a variable. These conditions guarantee that a write-first occurrence of a variable is unbound prior to unification. The definition of a write-first occurrence of a variable, however, concerns only local properties of a clause, and the scope of the optimisation can be improved by considering global properties by mode analysis. Specifically the unification of a variable with a term can be specialised to a form of assignment if mode analysis infers that the variable is unbound prior to unification. Write-first occurrence is a useful optimisation because the unification algorithm is usually implemented as a complex piece of code and thus is usually accessed out-of-line. In-lining can be used, however, to implement the write-first occurrence of a variable.

Type information can also be used to simplify unification. The unification of two variables can, for example, be reduced after type analysis to just a comparison for equality if one variable is known to be a constant and the other is known to be bound. The gains from in-lining can be considerable. Not only is branching to and returning from an out-of-line routine made unnecessary but also the code is exposed to subsequent improving optimisations such as common sub-expression elimination and register allocation. Even if in-line code cannot be produced type information can still be useful. If both variables involved in the unification are found to be bound, for instance, then a more suitable entry-point to the unification code can be selected.

2.2 Repeated synchronisation instruction optimisation

A synchronisation instruction checks that a variable is bound. In the kernel Parlog language described by Gregory[4], for instance, a synchronisation instruction corresponds to a `DATA/1` atom. If the variable is bound the synchronisation instruction succeeds; otherwise suspension occurs until the variable is bound. Gregory[4] describes how repeated `DATA/1` atoms which occur across a sequential conjunction can be removed. More generally, synchronisation instructions which are repeated within a thread are not required and can be removed. This redundancy can be detected by mode analysis by ensuring that on exit from a synchronisation instruction the mode of the variable is updated to bound. If the mode analysis infers that the synchronisation instruction tests a bound variable then the instruction can be removed. Again reformulating the optimisations as mode analysis broadens the scope for the optimisation.

2.3 Producers before consumers optimisation

Within the single framework of mode analysis a further optimisation can be incorporated. Threads are generated so as not to contradict any data-dependence between producers and consumers. Consequently producers are frequently ordered before consumers. Some of the synchronisation instructions in the consumers are thus made redundant. Mode analysis can detect this redundancy so that superfluous synchronisation instructions can be removed. Additionally type information can identify which type tests and which type checks are extraneous and therefore can be removed. The producers before consumers optimisation is likely to be useful for code which evaluates arithmetic expressions because most of the code used to perform arithmetic is actually spent checking that the arguments are of the right type.

3 Reference analysis

The logical variable enables variables to be unified together without being bound. A unification which binds a variable must also bind all the variables to which it is aliased. The conventional variable representation achieves this by representing a variable as a pointer. On the unification of two unbound variables one variable pointer to set to reference the other. To determine the binding of a variable, therefore, the chain of pointers emanating from a variable has to be followed. This is called dereferencing. Dereferencing is a significant factor in performance because it is incorporated into many abstract machine operations and is usually implemented as a non-trivial cycle of memory fetch, check and loop instructions. Furthermore, chains of pointers introduce additional complexity to garbage collection. These overheads can be reduced by applying reference analysis to threads.

Taylor[8] and Mariën *et al.*[9] proposed reference analysis as a way of keeping track of the length of pointer chains. Types derived by type analysis are paired with length information to indicate whether a term can be accessed without a pointer, with just a single pointer, or requires a chain of pointers of unknown length to be followed. Reference analysis, like mode and type analyses, depends on control and therefore threads, for its accuracy and usefulness. However, unlike mode and type analysis, reference analysis cannot be directly applied to threads. This is because reference analysis is concerned with sequential behaviour, that is, the order in which variables are created and references are established. Suppose, for instance, that a variable is shared across two threads and in both threads the variable participates in a unification with a non-ground term. Then an alias can be formed straddling the two threads with one of the aliased variables in one thread and the other aliased variable in the other. The order in which the threads are scheduled and evaluated determines which of the aliased variables is set to reference the other. Thus reasoning about the length of pointer chains for variables which are aliased across threads is problematic. Notice, however, that unaliased variables can always be accessed without a chain of pointers. The adaptation of reference analysis to threads thus corresponds to simply determining whether a variable is possibly aliased or is definitely unaliased. This form of alias analysis can be formulated in terms of the mode analysis described by Debray[5] to give an efficient and effective reference analysis for threads.

Touati and Despain[10] present statistical evidence which suggests that the unification of two unbound variables accounts for only a small proportion of unifications and therefore small, usually zero length, chains of pointers can be expected. Taylor[8] gives figures for reference analysis which coincide with this because the majority of lengths inferred were actually zero. Indeed, in later revisions of the compiler, Taylor[11] restricts reference analysis to infer that either a chain is zero length or unknown length. Thus the adaptation of reference analysis to threads is likely to accurately determine the length of a high proportion of the pointer chains.

3.1 Dereference optimisation

The adaptation of reference analysis to threads identifies variables which are guaranteed to be unaliased. Unaliased variables are not associated with a chain of pointers and therefore do not need to be dereferenced. This is the dereference optimisation. The dereference optimisation is likely to be more effective in simplifying variable access for a shared-memory implementation than a distributed-memory implementation. For a shared-memory implementation an unaliased variable guarantees immediate access to the variable whereas for a distributed-memory implementation communication might still be required with another processor.

3.2 Environment optimisations

Crammond[12] explains how variables which are shared between ordered atoms can be allocated to an environment. Variables which are not allocated to an environment are placed in the heap. Environments are useful because they are stored in a stack and thus reduce the overhead of garbage collecting the heap. The environments employed by Crammond differ from the those conventionally used in Prolog[13]. Instead of allocating unbound variables to an environment, the variables are stored in the heap and a reference to the variable placed in the environment. Crammond explains that this is necessary for avoiding dangling references and also simplifies garbage collection. Specifically if two environment variables in the stack are bound together then the younger variable (nearer the top of a stack) must be set to point to the older variable (nearer the bottom of the stack), otherwise when the environment containing the younger variable is deallocated, a dangling reference is created. For Prolog, since there is a single stack, the decision of which variable is the younger can be made by address comparison. For a concurrent logic language, the multiple argument stacks make the relative age of a variable more difficult to determine. Therefore, instead introducing extra machinery to deal with age comparison, Crammond advocates migrating all variables from the argument stacks to the heap. Placing all variables in the heap also simplifies the garbage collection of the argument stacks since all pointers in the argument stacks reference the heap. (In general, garbage collection of the argument stacks is required because, although the argument stacks act as stacks for allocation, items are not necessarily deallocated in reverse order and thus large holes can potentially develop.)

Threads increase the potential for environments in two ways. First, schedule analysis is likely to identify circumstances, undetected by the programmer, for which atoms can be ordered. Consequently there is likely to be a movement of data from the heap to the argument stacks. This is suggested by statistical evidence presented by Crammond[12] which shows that (without schedule analysis) very few Parlog clauses create environments. Threads change this. Second, the adapted reference analysis can be used to introduce variables into an environment. A variable which is guaranteed to be unaliased can be safely allocated to an environment since a dangling reference will never be created when the environment is deallocated. Furthermore, apart from the self-reference of an unbound variable, all pointers in the argument stacks reference the heap so that the overhead of compacting an argument stack is still low.

The handling of environments is not completely solved, however, and dangling references can still occur. Although all the pointers in the argument stacks reference the heap, pointers in the argument registers can possibly reference the argument stacks. Specifically an argument register can point to an unbound variable contained in an environment so that on deallocation a dangling reference can be formed. Because of the way variables are copied, a dangling reference cannot occur if the last occurrence of a variable is known to be bound. For Prolog a safety check is employed to test if the variable is bound[13]. If so the variable can be safely deallocated. If not a new variable is created in the heap and the environment variable copied to the heap variable. The safety check, instead, can be performed by mode analysis by inferring the mode of the last occurrence of each environment variable. If the last occurrence of an environment variable is bound then the safety check can be dispensed with. This enables environment handling to be refined further.

3.3 Initialising variable optimisation

Within a thread, atoms frequently instantiate unbound and unaliased variables. Conventionally such variables are initialised when the arguments are constructed, and thus typically the variable is initialised, examined in the process of unification and then bound. Van Roy[14] and Taylor[11] suggest how mode analysis and aliasing analysis can be applied to Prolog to reveal which arguments of an atom correspond to unbound and unaliased variables. This information, in turn, enables the code which initialises and unifies such variables to be simplified. The mode analysis and reference analysis of threads deduce just this information so that the principle behind the initialising variable optimisation can be applied to threads.

The initialising variable optimisation can be implemented by designating as uninitialised any argument of an atom which coincides with an unbound and unaliased variable on entry to the associated predicate, and corresponds to a bound variable on exit from the predicate. The onus of initialising an uninitialised argument is moved from the atom (the caller) to the predicate (the callee) to enable initialisation and unification of the variable to be refined.

Taylor[11] reports that the gains from applying the initialising variable optimisation to Prolog are substantial. This is presumably because of the frequency with which the optimisation can be applied. The drawback of the optimisation is likely to be in the garbage collection of the heap. Spurious references temporarily introduced by uninitialised arguments will not significantly complicate compaction of the argument stacks but may well dictate that extra pointers need to be traversed in the compaction of the heap. The spurious references which are introduced by the optimisation can, however, limit the applicability of the technique. This is because the optimisation can be applied to variables which are used only within a single thread. Without this restriction it is possible, for instance, for one thread to temporarily generate a spurious reference, and concurrently, for another thread to synchronise on that variable waiting for it to be instantiated. Finding which variables are used in which threads will not significantly complicate the optimisation since the check is merely a syntactic issue.

4 Conclusions

Schedule analysis plays a more central rôle than just another intermediate stage of compilation, since as well as reducing the enqueuing and dequeuing of processes, ordering optimisations can additionally be applied within a thread. Many ordering optimisations can be reformulated in terms of abstract interpretation, which not only extends the scope of these optimisations, but additionally facilitates other optimisations. This improves the integration and economy of the compilation process. In broad terms the optimisations follow from combinations of mode analysis, type analysis and reference analysis. Building these forms of abstract interpretation on threads leads to simple, efficient and accurate analyses. Mode analysis and type analysis can be used to: identify instances of unification which can be replaced with a form of assignment; distinguish repeated synchronisation instructions which can be removed; and indicate which redundant checks can be removed when producers are ordered before consumers in the same thread. Reference analysis can be used to: identify which variables can be accessed without dereferencing; indicate where variable initialisation and unification can be simplified; and show which variables can be allocated to an environment. Some safety checks can also be removed by using mode information.

References

[1] A. King and P. Soper, "Reducing scheduling overheads for concurrent logic programs," in *International Workshop on Processing Declarative Knowledge*, (Kaiserslautern, Germany), (1991).

[2] K. R. Traub, *Implementation of Non-strict Functional Programming Languages.* Pitman, (1991).

[3] A. King and P. Soper, "A semantic approach to producer and consumer analysis," in *International Conference on Logic Programming Workshop on Concurrent Logic Programming*, (Paris, France), (1991).

[4] S. Gregory, *Parallel Logic Programming in Parlog, The Language and its Implementation.* Addison-Wesley, (1987).

[5] S. K. Debray, "Static analysis of parallel logic programs," in *Proceedings of the Fifth International Conference and Symposium on Logic Programming*, pp. 711–732, MIT Press, (1988).

[6] C. Codognet, P. Codognet, and M. Corsini, "Abstract interpretation for concurrent logic languages," in *Proceedings of the North American Conference on Logic Programming*, (Austin, Texas), MIT Press, October (1990).

[7] M. Codish, M. Falaschi, and K. Marriott, "Suspension analysis of concurrent logic programs," in *Proceedings of the Eighth International Conference on Logic Programming*, (Paris, France), pp. 331–345, MIT Press, (1991).

[8] A. Taylor, "Removal of dereferencing and trailing in prolog compilation," in *Proceedings of the Sixth International Conference on Logic Programming*, pp. 48–60, MIT Press, (1989).

[9] A. Mariën, G. Janssens, A. Mulkers, and M. Bruynooghe, "The impact of abstract interpretation: an experiment in code generation," in *Proceedings of the Sixth International Conference on Logic Programming*, pp. 33–47, MIT Press, (1989).

[10] H. Touati and A. Despain, "An empirical study of the warren abstract machine," in *Proceedings of the 1987 Symposium on Logic Programming*, (San Francisco, California), pp. 114–124, (1987).

[11] A. Taylor, *High Performance Prolog Implementation.* PhD thesis, Basser Department of Computer Science, July (1991).

[12] J. A. Crammond, *Implementation of Committed-Choice Logic Languages on Shared Memory Multiprocessors.* PhD thesis, Heriot-Watt University, Edinburgh, May (1988).

[13] D. H. D. Warren, "An abstract prolog instruction set," Tech. Rep. 309, Artificial Intelligence Center, SRI International, Menlo Park, California, August (1983).

[14] P. Van Roy and A. Despain, "The benefits of global dataflow analysis for an optimising prolog compiler," in *North American Conference on Logic Programming*, pp. 501–515, MIT Press, October (1990).