Reducing Scheduling Overheads for Concurrent Logic Programs

Andy King and Paul Soper
Department of Electronics and Computer Science,
University of Southampton, Southampton, S09 5NH, UK.

Abstract

Strictness analysis is crucial for the efficient implementation of the lazy functional languages. A related technique for the concurrent logic languages (CLLs) called schedule analysis is presented which divides at compile-time a CLL program into threads of totally ordered atoms, whose relative ordering is determined at run-time. The technique enables the enqueuing and dequeuing of processes to be reduced, synchronisation tests to be partially removed, introduces the possibility of using unboxed arguments, and permits variables to be migrated from a heap to a stack to affect a form of compile-time garbage collection. The implementation is outlined and some preliminary results are given.

1 Introduction

Traub [1] has proposed dependence analysis as a technique for reducing the run-time overheads of the lenient functional languages. The analysis presented in this paper arose because the lenient functional languages and the concurrent logic languages (CLLs), as described in [2], share similar synchronisation mechanisms. Based on this observation a reinterpretation and reformulation of dependence analysis, called schedule analysis, has been developed for the (CLLs).

Schedule analysis is concerned with deducing at compile-time a partial schedule of processes, or equivalently the guard and body atoms of a clause, which is consistent with the program behaviour. Program termination characteristics are affected if an atom which instantiates a shared variable is ordered after an atom that matches on that variable. In order to avoid this an ordering of the atoms has to be determined which does not contradict any data dependence. In general the processes cannot be totally ordered and thus 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 threads.

An additional motivation for schedule analysis is that it allows a number of important optimisations. These are surveyed in section 2. The role of schedule analysis in uniprocessor and multiprocessor implementations of the CLLs is also discussed. Section 3 explains how dependencies between atoms can identify pairs of atoms which must be allocated to different threads. Finally theorem 1, a safety result, states the conditions under which atoms can be partitioned into threads and ordered within a thread whilst preserving the behaviour of the program. The final procedure can be used with existing compile-time analysis techniques. In section 4 we outline our implementation are give some preliminary results. Section 5 presents the concluding discussion.

2 Motivation

In addition to reducing enqueuing and dequeuing of processes by a scheduler, schedule analysis permits several useful optimisations to be applied within a thread. The optimisations all depend on the existence of a total ordering of atoms within a thread.

Gregory [3] uses the sequential and parallel conjuncts of kernel Parlog to express ordered guard and body atoms to enable matching and unification to be partially replaced with assignment and assignment to be partially removed. Synchronisation instructions (which correspond to `DATA/1` atoms in kernel Parlog), if repeated within a sequential conjunct, can also be removed. Crammond [4] explains how variables which are shared between ordered atoms can be allocated to the environments of a stack rather than a heap. Dividing the atoms of a clause into threads of totally ordered atoms extends the scope of these optimisations. Furthermore synchronisation instructions can be removed if producer atoms are ordered before the consumer atoms within the same thread.

Boxing analysis plays a role in realising the speedup of strictness analysis, and also appears to be useful in schedule analysis. Boxing analysis determines whether an argument of a predicate has to be boxed (tagged and referenced indirectly by a pointer) or can be unboxed (is of known type and can be placed in a machine register to be referenced directly without a pointer). Unboxed arguments can often be used if a producer atom is ordered before the consumer atoms within the same thread. Moreover if each clause of a predicate definition synchronises on an argument then it is possible to move the synchronisation instruction to immediately before the invoking atom in the parent clause. In many cases the synchronisation instruction can then be shown to be redundant.

In a multiprocessor implementation there is a tradeoff between scheduling at compile-time and scheduling at run-time. Schedule analysis permits useful optimisations to be applied within a thread but also limits parallelism. Thus schedule analysis should be applied only when parallelism is inappropriate. Parallelism is always inappropriate for a uniprocessor, and can often be inappropriate for a multiprocessor. To give an efficient and balanced untilisation of a multiprocessor a CLL program may be divided into grains, the constituent processes of a grain being executed on a single processor. The division of a CLL program into grains can be performed either manually by the programmer annotating code, or automatically by the compiler applying granularity analysis [5]. Since overheads still occur within a grain, because parallelism has to be emulated, schedule analysis can then be applied to a grain to reduce these overheads.

3 Outline of schedule analysis

In this section we briefly outline the main points of schedule analysis without formal definitions or proofs. A detailed account of the method can be found in [6]. Schedule analysis is based on overestimating the relevances associated with sharing, variable producers and variable consumers. These are assumed to be already derived, for instance by the abstract interpretation techniques reported by King and Soper [7] and Codish, Dams and Yardeni [8].

A relevance relation is constructed by overestimating the atoms which produce a variable and overestimating the atoms which consume the variable. A relevance is included for each such producer to consumer dependence. In the following we use the notation $(p \in)\ P_W$ for the set of predicate symbol occurrences in the program W, with a typical element p, and $(v \in)\ V$ for the set of program variables, with typical element v. For brevity we refer to the atom with predicate symbol p and also the clause defining p by the same symbol p. To describe the procedure for construction the relevance relation a producer map $\mathcal{P} : P_W \rightarrow 2^V$ and a consumer map $\mathcal{C} : P_W \rightarrow 2^V$ are introduced such that: $v \in \mathcal{P}(p)$ if p can affect v; and $v \in \mathcal{C}(p)$ if v can affect p. Specifically $v \notin \mathcal{P}(p)$ if v can be shown to be completely matched or ignored by p and $v \notin \mathcal{C}(p)$ if v can be shown to be completely instantiated or ignored by p. Sharing is encapsulated by the mappings $\mathcal{S} : P_W \rightarrow 2^{\mathbb{N}} \times \mathbb{N}$ and $\mathcal{V} : P_W \times \mathbb{N} \rightarrow 2^V$ which respectively indicate which arguments of an atom can share, and identify the variables in an argument of an atom. More exactly $\langle m, n\rangle \in \mathcal{S}(p)$ if the terms of the mth and nth arguments of the atom p can share, and $v \in \mathcal{V}(p, n)$ if the variable v is part of the nth argument of p. A relevance relation on the set of body atoms Q_p for the clause p can be constructed in terms of $\mathcal{P}$, $\mathcal{C}$, $\mathcal{S}$ and $\mathcal{V}$.

Definition 1 *The relevance relation δ_p is defined by: $\langle q, q'\rangle \in \delta_p$ if and only if*

1. $\langle m, m'\rangle \in \mathcal{S}(p')$ *and* $v \in \mathcal{V}(p, m)$ *and* $v' \in \mathcal{V}(p, m')$ *and* $v \in \mathcal{P}(q)$ *and* $v' \in \mathcal{C}(q')$ *and* $q \neq q'$ *or*

2. $v \in \mathcal{P}(q)$ *and* $v \in \mathcal{C}(q')$ *and* $q \neq q'$.

Note that since δ_p is defined edge-wise it is not necessarily transitive. Although producers and consumers are intuitively connected with relevance, the connection for sharing is indirect are arises through the potential for feedback which can introduce additional relevances into the relevance relation. This is explained in [6].

The δ_p relation summarises the behaviour of clause p independently of the initial query and it can be used to partition the atoms of Q_p into threads of totally ordered atoms. Threads are formed by identifying pairs of atoms which must be allocated to different threads. Pairs of atoms are related in just four ways according to the categories of figure 1 (where δ_p^+ denotes the transitive closure of δ_p). For category one, either q always precedes q' or q sometimes precedes q', so that for both cases q can be ordered before q' within the same thread. Category two is the symmetric variant of category one. For category three the atoms q and q' can be arbitrarily ordered because neither

Category	*Characteristic*	*Order*
1	$\langle q, q' \rangle \in \delta_p^+$ and $\langle q', q \rangle \notin \delta_p^+$	q precedes q'.
2	$\langle q', q \rangle \in \delta_p^+$ and $\langle q, q' \rangle \notin \delta_p^+$	q' precedes q.
3	$\langle q, q' \rangle \notin \delta_p^+$ and $\langle q', q \rangle \notin \delta_p^+$	neither q precedes q' nor q' precedes q.
4	$\langle q, q' \rangle \in \delta_p^+$ and $\langle q', q \rangle \in \delta_p^+$	either q precedes q' or q' precedes q, or q and q' coroutine.

Figure 1: Categorising atom pairs.

q precedes q' nor q' precedes q. Category four either identifies coroutining activity, or different sequences for which q precedes q' in one sequence and q' precedes q in another sequence. In either case the atoms q and q' must be assigned to different threads and the ordering resolved at run-time. Of these four categories only category four corresponds to pairs of atoms that must be allocated to different threads. This is encapsulated as the relation σ_p on Q_p called the separation relation.

Definition 2 *σ_p on Q_p is defined by: $\langle q, q' \rangle \in \sigma_p$ if and only if $\langle q, q' \rangle \in \delta_p^+$ and $\langle q', q \rangle \in \delta_p^+$.*

Atoms which are related by σ_p must be allocated to different threads.

Definition 3 *$\{Q_p^1, \ldots, Q_p^t\}$ is a partition of Q_p such that $q \in Q_p^i$ and $q' \in Q_p^j$ with $i \neq j$ if $\langle q, q' \rangle \in \sigma_p$. o_p^i is a total ordering on Q_p^i such that if $\langle q, q' \rangle \in o_p^i$ then $\langle q', q \rangle \notin \delta_p^+$.*

Q_p^i expresses the constituent atoms of a thread, o_p^i expresses the ordering of atoms within a thread, and t expresses the number of threads. Each o_p^i is chosen not to contradict δ_p.

It is possible for $\{o_p^1, \ldots, o_p^t\}$ to describe a division into threads which affects the behaviour of the clause p. The problem stems from the sequential nature of threads. Collectively $\{o_p^1, \ldots, o_p^t\}$ can introduce extra non-trivial cycles into δ_p. This is because the totally ordered threads induce extra dependencies between atoms. It is as if these extra dependencies are included in another relevance relation which is a superset of the original relevance relation. The superset relevance relation can require a different division into threads. In this case the original partition is inappropriate and can potentially affect program behaviour. The observation that the partition can affect termination if the threads collectively introduce extra non-trivial cycles into the relevance relation motivates the following safety result.

Definition 4 *τ_p on Q_p is defined by $\tau_p = \cup_{i=1,\ldots,t} o_p^i$.*

Definition 5 *An interleave ι_p of $\{o_p^1, \ldots, o_p^t\}$ is a total relation on Q_p such that $\tau_p \subseteq \iota_p$ and if $\langle q, q' \rangle \in \iota_p$ then $\langle q', q \rangle \notin \tau_p$.*

	Data	Get_Const and Get_List	Bind	Unify	Minus and Plus	Less
nfib/2	**1/441**	89/89	**0/264**	177/177	352/353	
nrev/2	**31/496**	91/91		466/466		
sieve/2	473/473	258/258		**247/275**	28/28	51/51

Figure 2: Preliminary schedule analysis results.

Theorem 1 *If $\tau_p \cup \delta_p^+$ has no more non-trivial cycles than δ_p^+ then there exists an interleave ι_p of $\{o_p^1, \ldots, o_p^t\}$ such that for all initial queries ι_p does not contradict any data dependence on Q_p.*

An interleave expresses how the body atoms of a clause can be ordered by scheduling threads. In other words definition 5 states that the ordering of atoms in an interleave must not contradict the ordering of atoms in a thread. Theorem 1 is a safety result in the sense that if τ_p adds no extra non-trivial cycles to δ_p^+ then for all initial queries the threads can always be scheduled so as to resolve all data dependencies. Specifically theorem 1 describes a procedure for safely partitioning the atoms of a clause into threads of totally ordered atoms in such a way that termination characteristics are preserved.

4 Implementation and Preliminary Results

Schedule analysis has been implemented and integrated into an existing FParlog86 compiler. δ_p^+ is calculated as the fixed-point of the Boolean adjacency matrix for δ_p [9]. The problem of finding an optimal partition of Q_p, one which minimises the number of threads t, is NP-complete [10]. Therefore, instead, a good partition is found in polynomial-time by a sequential colouring algorithm [11]. Each o_p^i is formed by topologically sorting the relation induced by δ_p^+ on Q_p^i. The number of non-trivial cycles in δ_p^+ and $\tau_p \cup \delta_p^+$ is counted by a backtracking algorithm [12]. The prototype schedule analysis module has been coded in 350 lines of FParlog86, and typically equates to 10% of execution time of the compiler (excluding the generation of mode information by abstract interpretation). Some preliminary results obtained with the prototype implementation are given in figure 2.

Figure 2 lists the instruction count for three benchmark programs: `nfib/2` which counts the number of reductions required to calculate the tenth number in the Fibonacci sequence; `nrev/2` which computes the naive reverse of a thirty element list; and `sieve/2` which finds the first ten prime numbers by a sieve-based method. The instruction counts are presented in the form c/c^* where c and c^* are the instruction counts obtained with/without applying schedule analysis. Note how the synchronisation, binding and unifying instructions can often be removed. Observe too that because `sieve/2` uses significant amounts of corouting few instructions can be removed from the program.

5 Discussion

A compilation technique called schedule analysis has been presented which divides a program into threads, whose relative ordering is determined at run-time. The analysis has been developed in a formal framework within which safety conditions are established. A practical procedure for constructing threads, which satisfies the safety conditions, is also presented. Schedule analysis plays a more central role than just another intermediate stage of compilation, since it enables the enqueuing and dequeuing of processes to be reduced, binding checks and variable tagging to be partially removed, and variables migrated from a heap to a stack to effect a form of compile-time garbage collection. Since the lenient functional languages are similar in a number of ways to the CLLs the benefits ensuing from dependence analysis suggest that schedule analysis is likely to be worthwhile even for microprocessors equipped with microcoded scheduling support.

Some of the benefits of schedule analysis are linked with replacing bounded-depth scheduling with depth-first scheduling. The scheduling of guard and body atoms is said to be and-fair [2] if any atom capable of being evaluated will eventually be evaluated. And-fairness is only guaranteed by depth-first scheduling if the branch of the SLD-tree emanating from each atom is bounded and can be extended without indefinite suspension. Although the compile-time detection of bounded SLD-tree branches is in general undecidable. Francez [13], Ullman and Van Gelder [14], Walther [15], Apt *et al.* [16], Bezem [17], Van Gelder [18], Plümer [19] and Wang [20] have shown that the termination of logic programs can be usefully detected at compile-time. It has been assumed in this work that an important class of clauses can be identified for which the constituent atoms can be depth-first scheduled without compromising and-fairness or for which depth-first is preferred on the grounds of efficiency [21].

Acknowledgments

We would like to thank Ken Traub whose dependence analysis motivated much of this work and Hugh Glaser and Pieter Hartel for helpful discussions.

References

[1] Traub, K.R. (1989). "Compiling as Partitioning: A New Approach to Compiling Non-strict Functional Languages", *in Proceedings of the Fourth International Conference on Functional Programming*, pp. 75–88. ACM Press.

[2] Shapiro, E.Y. (1989). "The Family of Concurrent Logic Programming Languages", *Journal of ACM Computing Surveys*, 21 (3): 413–510.

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

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

[5] King, A. and P. Soper (1990). "Granularity Analysis of Concurrent Logic Programs", *in The Fifth International Symposium on Computer and Information Sciences*, Nevsehir, Cappadocia, Turkey.

[6] King, A. and P. Soper (1990). "Schedule Analysis of Concurrent Logic Programs", Technical Report 90-22, Department of Electronics and Computer Science, Southampton University, Southampton, S09 5NH.

[7] King, A. and P. Soper (1991). "A Semantic Approach to Producer and Consumer Analysis", *International Conference on Logic Programming Workshop on Concurrent Logic Programming*, Paris, France.

[8] Codish, M., D. Dams, and E. Yardeni (1990). "Derivation and Safety of an Abstract Unification Algorithm for Groundness and Aliasing Analysis", Technical Report CS90-28, Department of Computer Science, Weizmann Institute of Science, Rehovot 76100, Isreal.

[9] Carré, B. (1979). *Graphs and Networks.* Clarendon Press, Oxford.

[10] Karp, R.M. (1972). *Complexity of Computer Computations*, chapter Reducibility among Combinatorial Problems, pp. 85–103. Plenum Press.

[11] D.W. Matula, G. Marble and J.D. Isaacson (1972). *Graph theory and computing*, chapter Graph colouring algorithms, pp. 109–122. Academic Press, London. Edited by R.C. Read.

[12] Tiernan, J.C. (1970). "An efficient search algorithm to find the elementary ciccuits of a graph", *Communications of the ACM*, 13: 722–726.

[13] Francez, N., O. Grumberg, S. Katz, and A. Pnueli (1985). "Proving Termination of Logic Programs", *in Proceedings of Logics of Programs Conference*, pp. 89–105, Brooklyn, NY. Springer-Verlag.

[14] Ullman, J.D. and A. Van Gelder (1988). "Top-down Termination of Logical Rules", *Journal of the ACM*, 35 (2): 345–373.

[15] Walther, C. (1988). *Automated Termination Proofs.* PhD thesis, University of Karlsruhe.

[16] Apt, K.R., R.N. Bol, and J.W. Klop (1989). "On the safe termination of Prolog programs", *in Proceedings of the Sixth International Conference on Logic Programming*, pp. 353–368, Lisboa, Portugal. MIT Press.

[17] Bezem, M. (1989). "Characterizing Termination of Logic Programs with Level Mappings", *in Proceedings of the North American Conference on Logic Programming*, pp. 69–80, Case Western Reserve University, Cleveland, Ohio.

[18] Gelder, A. Van (1990). "Deriving Constraints Among Argument Sizes in Logic Programs", *in Proceedings of the Ninth ACM Symposium on Principles of Database Systems*, Nashville, Tennessee.

[19] Plumer, L. (1990). "Termination Proofs for Logic Programs based on Predicate Inequalities", *in Proceedings of the Seventh International Conference on Logic Programming*, Jerusalem, Isreal.

[20] Wang, B. and R. K. Shyamasundar (1990). "Towards a Characterisation of the Termination of Logic Programs", *in The Second International Workshop on Programming Language Implementation and Logic Programming*, pp. 204–221, Linkoping, Sweden. Springer-Verlag.

[21] Sato, M., H. Shimizu, A. Matsumoto, K. Rokusawa, and A. Goto (1987). "KL1 Execution Model for PIM Cluster with Shared Memory", *in Proceedings of the Fourth International Conference*, pp. 338–355.