Static granularity optimization of a committed-choice language Fleng

Takuya Araki and Hidehiko Tanaka

School of Engineering, the University of Tokyo,
7-3-1 Hongo, Bunkyo-ku, Tokyo 113, Japan.
{araki,tanaka}@mtl.t.u-tokyo.ac.jp

Abstract. The committed-choice language Fleng can extract much parallelism easily even from irregular programs using dataflow synchronization. However, there is a large overhead because the granularity of execution is very fine. If granularity of a program is coarsened, such an overhead can be reduced. This can be attained by fusing several goals into one goal, but this may cause deadlock. In this paper, we propose a safe goal fusion algorithm that statically optimizes granularity of a Fleng program. We implemented the algorithm and evaluated it on a parallel computer PIE64. The evaluation shows that enough speedup can be attained by this method.

1 Introduction

Committed-choice languages can extract much parallelism easily even from programs with irregular parallelism such as symbolic computation. This is achieved by single assignment variable and dataflow synchronization.

However, the granularity of execution is so fine that overheads of parallel execution, such as context switching, synchronization, and communication, is large. Therefore, to reduce these overheads is the key of efficient implementation of these languages.

Because the main source of the overhead is fine grain execution, the overhead can be reduced by making granularity coarse. However, it is not easy to make granularity coarse, because that may cause deadlock.

In this paper, we present a safe goal fusion algorithm. The basic idea of the algorithm is derived from separation constraint partitioning [4]. We implemented the algorithm for a committed-choice language Fleng [3] and evaluated its effectiveness on a parallel computer PIE64.

2 Committed-choice language Fleng

Fleng is a kind of parallel symbol processing language. Its ancestor is Prolog and the syntax is similar to it. However, the semantics is very different from Prolog, because a Fleng program doesn't backtrack. GHC and KL1 [8] are other examples of committed-choice languages.

The characteristics of Fleng are:

– execute all goals in parallel
– synchronize using single assignment variable

These make it possible to extract much parallelism easily.

Take a short program for example:

$$\underbrace{\texttt{foo(A,R)}}_{\textit{head}}\texttt{:-}\underbrace{\texttt{add(A,1,B),mul(B,2,R)}}_{\textit{body}}$$

This is a definition of a *predicate* `foo` that executes `R = (A + 1) * 2`. A definition of a predicate consists of *clauses*. In this case, it consists of only one clause. A Clause is separated by "`:-`"; the left side of a clause is called a "*head*", and right side is called a "*body*".

The unit of computation is called a *goal*. If an initial goal `foo(A,R)` is given, this goal is rewritten into `add(A,1,B)` and `mul(B,2,R)`. These two goals are executed in parallel. But in this case, `mul` cannot be executed before the value of `B` is calculated. Thus, the execution of `mul` is *suspended* if `B` is not bound. When `B` is bound by `add`, `mul` is *activated*. To realize this mechanism, variables are *single assignment variables* and the value of the variable cannot be changed.

Branch can be expressed as follows:

```
foo(true,R):- R = 1.
foo(false,R):- R = 0.
```

This program consists of two clauses. This program means, if the first argument is `true` then executes `R = 1`, if it is `false` then executes `R = 0`. Like the previous example, if the first argument is not bound, the goal is suspended. When it is bound, the goal is activated and branches according to the value.

Arithmetic operations are defined such as:

```
add(#A,#B,R):- compute(+,A,B,R).
```

Here, "`#`" means that the predicate waits for the variable with "`#`" to be bound. `compute` is executed without suspension and it can be compiled into a few assembly codes. The input arguments of `compute` should be bound, which is guaranteed by "`#`"s of the head in this case. If there is a sequence of `compute`s, they are executed sequentially.

3 Goalfusion

3.1 Problem of goalfusion

We propose *goalfusion* as a coarsening method of Fleng programs. Goalfusion is a method that fuses goals into one goal, but the method cannot always be applied.

For example:

```
foo(U,V,R,S):- add(U,U,R), mul(V,V,S).
add(#A,#B,R):- compute(+,A,B,R).
mul(#A,#B,R):- compute(*,A,B,R).
```

This program executes `R = U + U` and `S = V * V`. `add` and `mul` are fused into one goal:

```
foo(U,V,R,S):- add_mul(U,V,R,S).
add_mul(#U,#V,R,S):- compute(+,U,U,R), compute(*,V,V,S).
```

In this case, `add` and `mul` are fused into `add_mul`, and the overhead of goal invocation and synchronization is reduced. However, this transformation is *wrong*.

In the original program, if `foo(1,Tmp,Tmp,S)` is called, it is rewritten into `add(1,1,Tmp)` and `mul(Tmp,Tmp,S)`. Then `Tmp` is bound to 2 by `add`, `S` is bound to 4 by `mul`.

But in the transformed program, `foo(1,Tmp,Tmp,S)` is rewritten into `add_mul (1,Tmp,Tmp,S)`; `add_mul(1,Tmp,Tmp,S)` waits for `Tmp` to be bound forever. Thus, transformed program causes deadlock. That means this transformation changed the semantics of the program, so such a transformation should be avoided.

Why the fusion of `add` and `mul` causes deadlock? Figure 1 shows the dataflow graph of the original program and the transformed program.

In the dataflow graph of the original program, `mul` depends on `add` through `Tmp`, which is an argument of `foo`. This dependency does not always exist. This kind of dependency is called a *potential dependency* [4]. This dependency makes a *cyclic dependency* after goalfusion. A Cyclic dependency means that the goal depends on its own output, which causes deadlock. The original program does not cause deadlock because it does not have any cyclic dependencies.

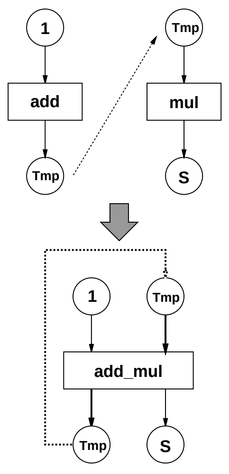

Fig. 1. Dataflow graph

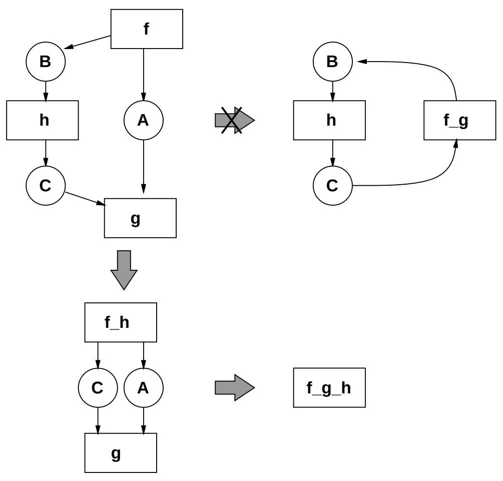

Fig. 2. Algorithm of goalfusion

3.2 Algorithm of goalfusion

To avoid deadlock, cyclic dependencies should be avoided. So the basic idea of the algorithm is: *two goals can be fused only when there is no indirect dependency between these goals.* This is because indirect dependencies always make cyclic dependencies when these two goals are fused.

Consider figure 2 for example; it is a dataflow graph of a program. `A`, `B` and `C` are variable and `f`, `g` and `h` are goals. In this program, `f` and `g` cannot be fused because `g` indirectly depends on `f` through `h`. If `f` and `g` are fused, the indirect dependency makes a cyclic dependency.

On the other hand, `f` and `h` are safely fused into `f_h`. Though `h` depends on `f`, the dependency is direct. The direct dependency disappears after fusion because the dependency exists inside of the fused goal.

Then `f_h` and `g` can be fused into one goal likewise.

Brief sketch of the algorithm is as follows:

1. Select one clause.
2. Analyze dependencies in the clause (described in section 3.3).
3. Fuse two goals if there are no indirect dependencies between them.
4. Repeat 2-3 until any two goals cannot be fused.

Here, potential dependencies and dependencies which go through one or more other goals are treated as indirect dependencies.

3.3 Dependency analysis

In order to fuse two goals, it is necessary to guarantee that there is no indirect dependency between these goals. In this section, we will describe how to analyze dependencies in a program.

Mode inference In a Fleng program, it is not known from syntax whether an argument is used as input or output. To analyze dependencies precisely, input/output mode information is needed. [9] describes a mode inference method including structured data, but we did simple mode inference that does not include structured data. The algorithm is borrowed from [9].

The algorithm is based on the following ideas:

1. If a variable is output of a goal, the variable is input of other goals that share the variable.
2. If a variable is input of all goals except one goal, the variable is output of the goal.

Variables whose mode cannot be inferred are treated as both input and output to analyze dependencies safely.

Detection of dependency In this section, we will describe how to detect dependencies from each program element.

Body goals: If a body goal cannot be reduced until an argument variable is bound, the goal directly depends on the variable (and the variable is input of the goal); and if an argument variable is guaranteed to be bound after a goal is reduced (by `compute` or `=`), the variable directly depends on the goal (and the variable is output of the goal).

In addition, a body goal may make dependencies from any input variables to any output variables, because sub goals of the goal may make dependencies.

Of course, sub goals may not make dependencies; but in order to detect potential dependencies safely, dependencies from all input variables to all output variables should be assumed.

If the definition of a body goal is known, the information is propagated using inter-clause analysis described below.

Unification: Unifying variable with atomic value does not make a dependency. However, unifying variable with variable should be treated to make dependencies of both directions. For example, `A = B` is treated to make dependencies from `A` to `B` and that from `B` to `A`. Actually `A` does not depend on `B` and vice versa; but `A = B` may be used as a path of dependencies.

Unifying variable with structured data is more complex. For example, `A = {B,C}` (`{B,C}` means a vector which consists of two elements, `B` and `C`) should be treated to make dependencies from `A` to `B` and `C`, and those from `B` and `C` to `A`, because this unification may be also used as paths of dependencies.

For example:

```
foo(...):- A = {B,C}, bar(A,A1), A1 = {X,Y}, ...
bar({B,C},A1):- add(B,B,X), mul(C,C,Y), A1 = {X,Y}.
```

In the definition of `foo`, `X` depends on `B` and `Y` depends on `C`. These dependencies cannot be detected unless dependencies from `B` and `C` to `A` and those from `A1` to `X` and `Y` are assumed.

Head: The caller site (i.e. head) may make dependencies between arguments. The first example whose body goals cannot be fused is of this type.

While analyzing dependencies, a head and body goals are alike in the sense that both may make dependencies between arguments. So a head can be treated in the same way as a body goal during mode inference and dependency analysis. However, the inferred mode of a head is reversed to what is seen from the outside.

For example:

```
foo(U,V,R,S):- add(U,U,R), mul(V,V,S).
```

This program is same as the first example whose body goals cannot be fused.

Figure 3 shows the dataflow graph of this program including the head, which is represented in italic. Inferred mode of the head `foo` is: `R` and `S` are input and `U` and `V` are output (The mode of `foo` seen from outside is: `U` and `V` are input and `R` and `S` are output).

This dataflow graph shows that there are indirect dependencies from `R` to `V` and from `S` to `U`, which indicates that `add` and `mul` cannot be fused.

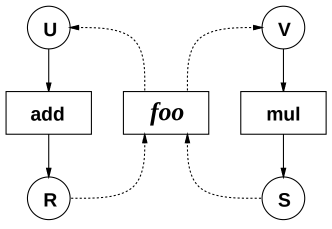

Fig. 3. Dependencies through a head

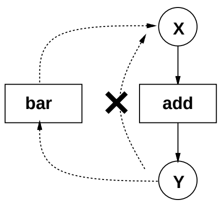

Fig. 4. Removal of a potential dependency that is contradicted by a certain dependency

Removal of contradicted dependencies Analyzed dependency may contain dependencies that cannot exist.

For example:

```
foo(...):- bar(Y,X), add(X,X,Y),...
```

In the dependency analysis, a dependency from `Y` to `X` is detected from a body goal `bar`; but there cannot be such a dependency. Because `add` makes a dependency from `X` to `Y` certainly, if `Y` depends on `X`, the program is in deadlock. If the given program is right and runs without deadlock, there cannot be such a dependency (Fig. 4).

In this study, we used the method described in [4]: *remove potential dependencies that are contradicted by certain dependencies.* Here, dependencies that consist of only direct dependencies (and dependencies that are propagated as certain dependencies by inter-clause analysis) are treated as certain dependencies. Other dependencies are treated as potential dependencies.

3.4 Inter-clause analysis

We described analysis within a clause so far. We will describe inter-clause analysis in this section.

There are two directions of passing information in inter-clause analysis: from a callee to a caller (bottom-up) and from a caller to a callee (top-down). But in the current implementation, top-down information propagation is not implemented to avoid complexity, though it can be implemented almost in the same way as bottom-up method.

And there are two kinds of information to pass: input/output modes and dependencies between variables. Dependencies are classified into certain dependencies and potential dependencies. This classification is used to remove potential dependencies that is contradicted by certain dependencies.

If the information varies according to the conditional branch, the information should be merged to be safe.

Dependencies are analyzed from the top goal and the call tree is traversed in the depth-first order. The goal already analyzed is memorized and analysis is stopped when the goal was already analyzed; it avoids infinite loop while analyzing (mutually) recursive goals.

For example:

```
foo(U,V,R,S):- add(U,U,R), mul(V,V,S).
bar(...):- foo(U,V,R,S),...
```

The information analyzed in `foo` is propagated to `bar` as follows.

The input/output mode of `foo` can be inferred using the method described above. However, because the analyzed mode is what is seen from body goals, the mode needs to be reversed. In this case, `U` and `V` are input, and `R` and `S` are output.

The dependencies to pass is what is analyzed without head. This is the dependencies made inside of `foo`. In this case, certain dependencies from `U` to `R` and `V` to `S` are analyzed. There are no other potential dependencies.

It is notable that the propagated dependency information indicates that dependencies from `U` to `S` and from `V` to `R` do not exist. It cannot be known only from the mode information.

4 Implementation and evaluation

4.1 Implementation

We implemented this method as a preprocessor of a compiler. It inputs a Fleng program and outputs a Fleng program whose granularity is coarsened. The preprocessor is about 6,000 lines of a Fleng program.

And the Fleng compiler is extended to manage conditional branch within a goal in order to fuse goals that branches at heads. This branch is described as `(Cond --> Then; Else)`. This branch assumes that all variables needed to evaluate `Cond` are bound, and it is compiled into a simple branch operation like procedural languages.

In addition, if a goal can be executed without suspension, it is inlined.

An example of transformation is as follows:

```
abs(A,R):-
  greater(A,0,IsGt), abs1(IsGt,A,R).
abs1(true,A,R):- R = A.
abs1(false,A,R):- sub(0,A,R).
```

This program is transformed into:

```
abs(A,B):- C = 0, greater_abs1(A,C,B).
greater_abs1(#A,#B,C) :-
  compute(>,A,B,D),
  ((D == true)--> C = A; E = 0, compute(-,E,A,C)).
```

Table 1. Benchmark programs

Program	Code size	Input size	Binary size (byte)		Compilation time (sec)	
			Original	Optimized	Original	Optimizing
queen	55	9-Queens	8925	7918	3.58	7.04
primes	39	2000	7076	7367	2.87	5.00
qsort	31	10000	4365	5373	2.55	14.1
fme	1175	queen	194110	170054	135	539

4.2 Evaluation

We evaluated this method on the Parallel Inference Engine PIE64 [1]. PIE64 is a parallel computer that is designed to execute Fleng programs efficiently. PIE64 has 64 processor elements and is a distributed shared memory machine, that is, the address space is global. The processor of PIE64 supports multi-context processing with which latency of communication can be hidden. The interconnection network supports automatic load balancing, and the speed of the network is relatively high.

Execution time used to calculate relative speed is an average of 3 executions excluding garbage collection time.

The granularity of the transformed program is made to be as coarse as possible though the implementation allows to specify the granularity of the program. This is because even the coarsest granularity was finer than the optimal granularity.

Compilation time was evaluated on a workstation (Sun Ultra 1).

We selected "queen", "primes", "qsort" and "fme" as benchmark programs. Queen is a program that solves N-Queens problem. Primes is a prime number generator that uses Eratosthenes' sieve. Qsort is a quick sort program. Fme is a program that expands macros of Fleng program; it is a practical Fleng application. The input of fme is queen whose macros are not expanded. These programs except queen have relatively small parallelism.

Table 1 shows binary size and compilation time of benchmark programs. Binary size was expected to increase because new predicate definition was added. However, it did not increase very much and even decreased in two programs. This is because the optimizer removes predicates that became unused after goalfusion. Compilation time increased 2 to 6 times. We think this increase is reasonable considering the complex dataflow analysis.

Figure 5–8 show the speedup by granularity optimization. X-axis of the graph shows the number of processors, and Y-axis of the graph shows the relative speed normalized by the speed of the original program with one processor. The optimized programs except fme are 2 to 9 times faster than the original programs irrespective of number of processors. Optimized fme ran about 1.2 times faster than the original program. We think the limited speedup is due to the difficulty of global analysis of a large program.

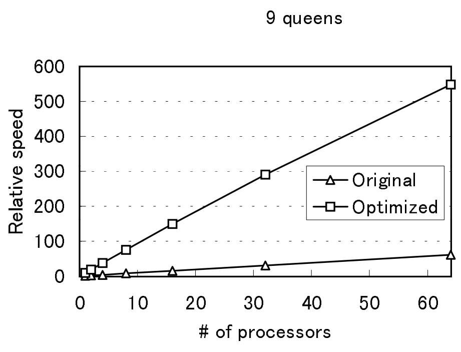

Fig. 5. Speedup of 9 queens

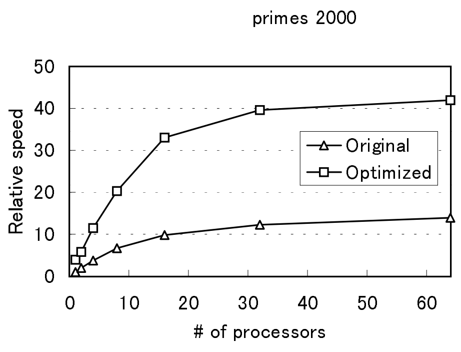

Fig. 6. Speedup of primes

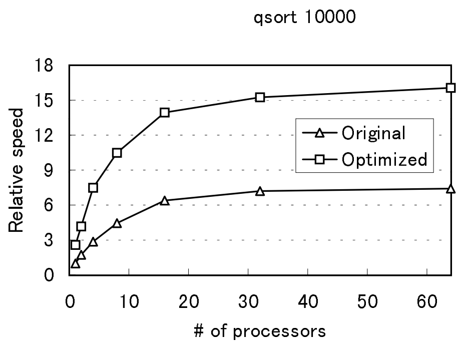

Fig. 7. Speedup of qsort

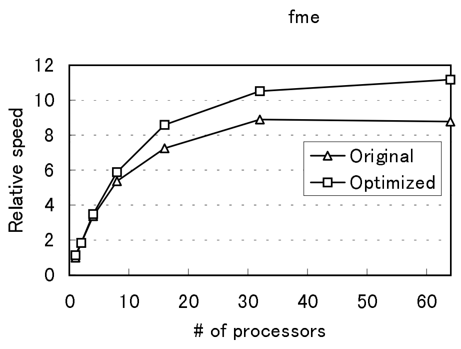

Fig. 8. Speedup of fme

5 Related work

Execution model of lenient functional languages such as Id is similar to that of committed-choice languages. Same kind of optimization as our work is also studied in the field of functional languages [5, 6, 7, 4]. Separation constraint partitioning (SCP) [4] is the latest algorithm of Id to make large threads from dataflow graphs; it strongly influenced our study.

The basic algorithm of our study is substantially same as SCP. SCP uses dataflow graphs as input, but our algorithm runs on the Fleng source code level. That makes it possible to process other source code level optimizations simultaneously. For example, if goalfusion enables any goals to be inlined, they are inlined. In addition, a transformed program is correct after every step of our algorithm, though it is not true in SCP.

On committed-choice languages, there is a study of sequentialization of programs [2]. In this study, they limit the programs to a class called "fully moded, feedback free". Feedback free means that there is no dependency through a head. It seems that their limitation is very strict; especially to limit programs to "feedback free" makes it impossible to write many kinds of programs.

6 Conclusion

In this paper, we proposed, implemented and evaluated a granularity optimization method of a committed-choice language Fleng. In this study, we used the goalfusion method to coarsen granularity of a program. An algorithm of goalfusion was proposed. We implemented and evaluated the algorithm. Several times speedup in small programs and about 1.2 times speedup in a realistic program were attained.

The reason of relatively small speedup in a realistic program seems to be due to difficulty of global analysis. Future work will focus on developing a granularity coarsening method for larger programs.

References

1. T. Araki, Y. Hidaka, H. Nakada, H. Koike, and H. Tanaka. System integration of the parallel inference engine PIE64. In *Workshop on Parallel Logic Programming attached to International Symposium on Fifth Generation Computer Systems 1994*, Dec. 1994.
2. B. C. Massey and E. Tick. Sequentialization of parallel logic programs with mode analysis. In *4th International Conference on Logic Programming and Automated Reasoning. LNCS 698*, pages 205–216. Springer-Verlag, 1993.
3. M. Nilsson and H. Tanaka. Fleng Prolog - the language which turns supercomputers into Prolog machines. In *Logic Programming '86, LNCS 264*, pages 170–179. Springer-Verlag, 1989.
4. K. E. Schauser, D. E. Culler, and S. C. Goldstein. Separation constraint partitioning – a new algorithm for partitioning non-strict programs into sequential threads. In *POPL '95*, pages 259–271. ACM, 1995.
5. K. E. Schauser, D. E. Culler, and T. von Eiken. Compiler-controlled multithreading for lenient parallel languages. In *FPCA '91, LNCS 523*, pages 50–72. Springer-Verlag, 1991.
6. K. R. Traub. Multi-thread code generation for dataflow architectures from non-strict programs. In *FPCA '91, LNCS 523*, pages 73–101. Springer-Verlag, 1991.
7. K. R. Traub, D. E. Culler, and K. E. Schauser. Global analysis for partitioning non-strict programs into sequential threads. In *LFP '92*, pages 324–334. ACM, 1992.
8. K. Ueda and T. Chikayama. Design of the kernel language for the parallel inference machine. *The Computer Journal*, 33(6):494–500, December 1990.
9. K. Ueda and M. Morita. A new implmentation technique for flat GHC. Technical report, ICOT, 1990. TR-560.

This article was processed using the LaTeX macro package with LLNCS style