분산메모리 멀티프로세서 시스템을 위한 바인딩 환경(QCE)

이 용 두[†] · 김 희 철[††] · 채 수 환[†††]

요 약

바인딩환경은 로직프로그램의 OR병렬수행 성능에 중대한 영향을 준다. 특히 PE에 대한 원격 교차 접근은 시스템성능 저하를 초래하기 때문에 비단일주소공간을 갖는 병렬컴퓨터구조상의 분산실행에서는 이문제가 중요시된다. 비단일주소공간을 갖는 시스템에 관한 몇가지 바인딩 방법이 제안되어 있지만, 그들은 환경폐쇄및 역단일화와 같은 부가적동작이 요구된다. 본 논문에서는 비단일주소공간구조에서 높은 준폐쇄성의 새로운 바인딩 환경을 제안한다. 이방법은 단일주소공간에서와 비단일주소 공간 양쪽에 결합된 복합모델이다. 제안된 바인딩 방법은 단일화나 역단일화도 필요없는 명시적 폐쇄 동작이 아닐때에 대단히 효율적이고, 원격접근이 없이 한정접근을 유지한다.

The QCE: A Binding Environment for Distributed Memory Multiprocessors

Yong-Doo Lee[†] · Hie-Cheol Kim[††] · Soo-Hoan Chae[†††]

ABSTRACT

In the OR-parallel execution of logic programs, binding environments have a critical impact on the performance. Particularly, this is true for distributed execution on parallel systems with a non-single address space. The reason is that in such systems, the remote accesses across processing elements deteriorate the performance. To solve this problem, some binding methods were previously proposed specifically for a non-single address space. However, compared with the binding methods for a single address space, they are far less efficient due to the overhead of newly introduced operations such as environment closing and back-unification. In this paper, we propose a new binding method geared particularly toward achitectures with a non-single address space. The proposed binding environment is a hybrid that combines both the binding methods for a single address space and those for a non-single address space. It acomplishes high efficiency by making closing operations unnecessary both at unification and at back-unification, while maintining the restricted accesses.

1. Introduction

Logic programming based on universally quantified Horn clauses has become a prominent programming paradigm for symbolic computing. Indeed, PROLOG is one of the most popular logic programming languages because of its many advantages in terms of ease of programming and declarative semantics. As aplications often demand high computing resources due to their huge computation, many researches[1, 2, 3, 4,

† 정 회 원:대구대학교 정보통신공학부 교수
†† 정 회 원:삼성데이터시스템 선임연구원
††† 정 회 원:한국항공대학교 컴퓨터공학과 부교수
논문접수:1996년 7월 4일, 심사완료:1996년 8월 30일

10] have been conducted to develop parallel execution techniques.

Distributed implementation of logic programs suffers from severe inefficiency. Moreover, the inefficiency becomes higher when the system memory is organized in a non-single address space. According to previous researches, the inference mechanism of logic languages, particularly the environment stacking and the runtime traversal of a parallel search tree is the main source of the efficiency.

In the OR-parallel execution of logic programs, all descendant nodes being executed concurrently share the parent's environment. That means each descendant must have a virtual copy of the parent's environment to avoid the conflict with bindings made by the other descendants. In this respect, binding methods have a critical impact on the performance of OR-parallel execution of logic languages. Particularly, this is applied more seriously to distributed execution, because the accesses across processing elements (PEs) would usually lower system utilization.

A number of binding methods[1, 2, 4, 9], which we will call shared binding environments, have been developed for the parallel execution on shared memory machines. Specific for distributed implementations of logic programs, some other binding methods have been designed[3, 6, 7]. These will be referred to as closed binding environments because they restrict (close) variable accesses within each processing element. These closed binding methods achieve the restricted accesses through some new operations such as environment closing and back unification. However, the operations usually incur intolerable overhead particularly when application programs have large amount of complex terms. In consequence, under the closed binding methods, the performance of a thread of tasks, which will be scheduled within a processing element (PE), is quite lower than under shared binding methods.

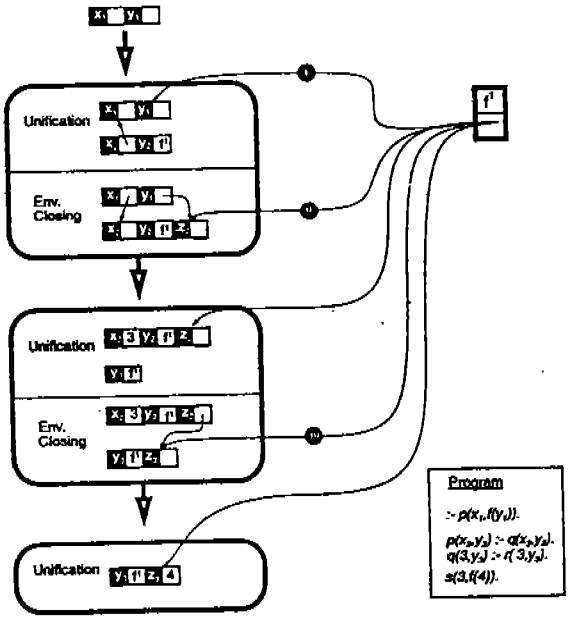
The central thesis of this paper is that with respect to a thread of tasks scheduled within a PE, even the binding methods for distributed implementations should still retain the efficiency of shared binding methods. To validate the thesis, we analyze the main sources of the inefficiency of the existent closed binding methods. Then, we propose a new binding method, which we will call the quasi-closed environment (QCE). As a hybrid of shared and closed binding methods, the QCE is aimed at both maintaining the efficiency of the shared binding method and archiving the restricted assess of closed binding methods.

The rest of this paper is organized as follows. In section 2, we will present the analysis of the closed binding methods in terms of the overhead. In section 3, we will discuss the proposed binding method with focus on the principles and the implementation techniques. In section 4, a qualitative performance comparison will be made between the proposed binding method and other relevant closed binding methods. Finally, a concluding remark is offered in section 5.

2. Analysis of Closed Binding Methods

Stated earlier, binding methods for distributed implementation are organized to restrict variable accesses always within local environments on a PE, because the remote accesses in shared memory models are too expensive. The variable importation method [7] employs back-unification along with head unification, using import vectors. The closed environment [3], a successor to the variable importation, brings out efficient memory usage by using closing operations without import vectors. In [6], the closed environment is optimized to avoid copying of grounded terms. In these binding environments, for every OR-task, its environment is closed against some other levels of the tree. This environment closing is achieved through the *closing operation* performed both at unification and at back-unification. Its precise algorithm is found in [3].

From the implementation perspective, we examine the closing algorithm [3] and identify the following three main sources of overheads: 1) For environment



(Fig. 1) An example of the closed environment

closing, the current environment should be scanned to check those references that reach outside the environment. That means all the complex terms reachable from the environment must be scanned. 2) When a variable in a working frame has a reference binding to a variable in a reference frame, the variable must be imported into the working frame. The importation entails the adjustment of the variable for making it refer to the newly created slot and also requires an extension of the working frame at runtime. 3) Provided a variable is included in a complex term, the term must be copied because of the single assignment property of PROLOG variables.

Fig. 1 depicts the results of unification and environment closing for a sequence of OR-tasks, where a reference is expressed simply with an arrow. It also shows the contents of environment frames. In the figure, a variable is contained in complex term f and is represented by the first slot. As a result of environment closing, the first slot is updated three times. In each update, it refers to a slot in three different frames. Because the offset values in those frames are not the same, *i.e.*, 2, 3 and 2, complex term f is copied three times.

The structure copying problem in the above case is similar to the case of functional programs. However, the structure copying problem in logic programs is quite severe. In the forward processing, once a non-ground structure is passed to a body goal, the structure should be copied for all the OR-nodes spawned by the body goal. For the search path of each OR-node, the structure must be repeatedly copied along the search path until the structure becomes grounded. As an example, suppose that the average number of OR-nodes for each body goal in the OR-task tree is n and a structure created in an OR-node k becomes ground after being passed to an OR-node separated from OR-node k by m levels. In this case, the total number of structure copying will be $O(n^m)$.

In addition to the structure copying, another drawback of closed binding methods is that the closing operation is applied to unification and back-unification uniformly in every OR-task. As a matter of fact, the closing operations are not necessary for the intra-PE task threads because the environments for those tasks will always stay local to the PE; therefore, the overhead caused from the closing operations cannot be justified on an intra-PE thread.

The concept of the closed binding environment is valuable for distributed implementation of logic programs. However, the above analysis indicates that the closed binding environment is not a suitable option for the distributed implementation of logic programs due to its unrealistic overhead.

3. The Quasi-Closed Environment(QCE)

This section presents the idea behind the QCE method and also addresses a variety of implementation issues.

3.1 Principles

The problem of complex term copying originates from the variable naming convention. In non-closed binding environments, a variable has a global name. For example, parallel models established on top of the Warren Abstract Machine WAM[8] utilize memory addresses for variable names. Under this global naming convention, it is not necessary to rename a variable when a structure containing the variable is passed to an OR-node; that is, it is not necessary to copy the structure. In the closed binding environment, every variable has a name scoped locally within the environment of a task. In this case, a complex term containing a variable must be copied, because a variable must be renamed whenever the complex term is passed between tasks. The goal of the Quasi-Closed Environment (QCE) is to achieve restricted accesses of the closed environment, while maintaining the efficiency of the non-closed binding methods. The main idea behind the QCE is to employ both the global and the local naming convention. For this reason, variables are divided into two classes; if a variable appears in arguments of complex terms, it is defined as an *instance* variable; otherwise, as a *frame* variable. Given this classification, frame variables are named according to the local naming convention, whereas instance variables are named according to the global naming convention.

Because all the variables contained in structures have global names, the QCE method avoids structure copying in an intra-PE task thread. Moreover, it does not need explicit environment closing at unification with the help of a tagging scheme to be discussed shortly.

However, two new issues arise in the QCE method: 1) the management of instance variables: especially the detection of instance variables. 2) the management of auxiliary structures for storing the bindings of instance variables.

3.2 Management of Frame Variables

For frame variables, the QCE maintains closed environments without explicit closing. To do that, unification is performed such that the closing effect is obtained for frame variables. This is achieved by a reference binding made always from variables in a reference environment to those in a working environment. To support such reference bindings, variables are marked with tags indicating their environments. When a variable is bound with a reference to another variable, the tag is compared to decide the direction. For this, an explicit scan of the whole environment including complex terms is required at every goal activation. In the QCE, it is sufficient to change the tags of variables only in the working frame since the variables in complex terms are managed separately. The management of frame variables are thus summarized as follows. (i) Each variables in a working environment is created with a tag indicating the frame. (ii) At unification, any reference binding between variables uses the tags to decide the direction of the reference. (iii) When variables in the working environment frame of a task is passed to its child task, their tags are adjusted during argument generation.

3.3 Management of Instance Variables

Being globally named, instance variables have unique names both inside and outside of a task. A binding made for an instance variable in a task is stored in an auxiliary data structure as it is done in general non-closed binding methods. However, two issues, the detection of instance variables and the management of auxiliary data structures, must be clearly addressed.

Naming instance Variables

In PROLOG, complex terms are created on the heap either 1) when arguments are generated for the activation of a goal that has a complex terms at its arguments, or 2) when a complex term appearing at a clause head is bound to a variable during unification. Both cases use a common algorithm to detect and name the instance variables.

```
BUILD-STRUCTURE (HP, E, H, A, name, arity, Flag)
/* instance variable detection and naming during construction of a structure */
1:  begin
2:      ptr =: HP;
3:      H[ptr] =: name;
4:      ptr =: ptr + 1;
5:      H[ptr] =: arity;
6:      ptr =: ptr + 1;
7:      for all argument A[i]
8:        begin
9:          Dereference A[i] to a term T.
10:         if Flag = true
11:           begin
12:             if tag(T) = unbound local variable
13:               begin
14:                 Get name $Ai for the T
15:                 E[T] =: $Ai;
16:                 H[ptr] =: $Ai;
17:               end
18:             else
19:                 H[ptr] =: T;
20:         end
21:         else
22:             H[ptr] =: T;
23:         ptr =: ptr + 1;
24:         HP =: ptr;
25:       end
26: end.
```

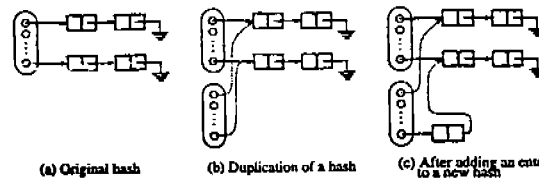
(Fig. 2) The extended algorithm to create complex

From the definition in the QCE, the variables inside complex terms are instance variables. We detect instance variables when complex terms are created. With this approach, we avoid scanning of complex terms at every task creation. The algorithm to create complex terms is shown in (Fig. 2).

Given the execution of a logic program based on the procedural interpretation, by induction on the depth n of the search tree, we can verify the completeness of the detection algorithm. Initially, at the root of the search tree for a program P, every instance variable is detected during argument generation; therefore, every variable included in arguments to the node of the 1^{th} depth is adequately named. Suppose that all the instance variables in the node of the $(n-1)^{th}$ depth were completely detected and thus appropriately named. On a node of the n^{th} depth, we can detect during argument generation all the unbound local variables belonging to arguments that are complex terms. By doing it, we do not need to scan complex terms to name the instance variables here since by induction hypothesis, the variables are already named. Therefore, all variables in complex terms passed to $(n+1)^{th}$ are appropriately named.

Management of Auxiliary Structures

At forward execution, the pointer to the parent's auxiliary structure is passed as a part of an input environment. Hence, every reference to a binding of an instance variable is performed locally in the task. At backward execution, the pointer to a child auxiliary structure carrying bindings of instance variables is passed back to the parent task as a part of the output environment. In the QCE, a hash table, shown in (Fig. 3(a)), is employed as the auxiliary data structure. As the duplication is always performed in a single address space, we optimize the duplication by just providing a copy of a head as shown in (Fig. 3 (b)). Enqueuing an element to the hash is shown in (Fig. 3(c)).



(Fig. 3) Operation on the auxiliary store

3.4 Back-unification

Back-unification done upon the successful completion of a task consists of simple retrievals of bindings, since no explicit closing is performed in our method. That means the binding of a variable in the parent frame is retrieved from those of local variables if exist, whereas tags of the variable is changed from *reference to variable* if the binding does not exist.

3.5 An Example of the QCE

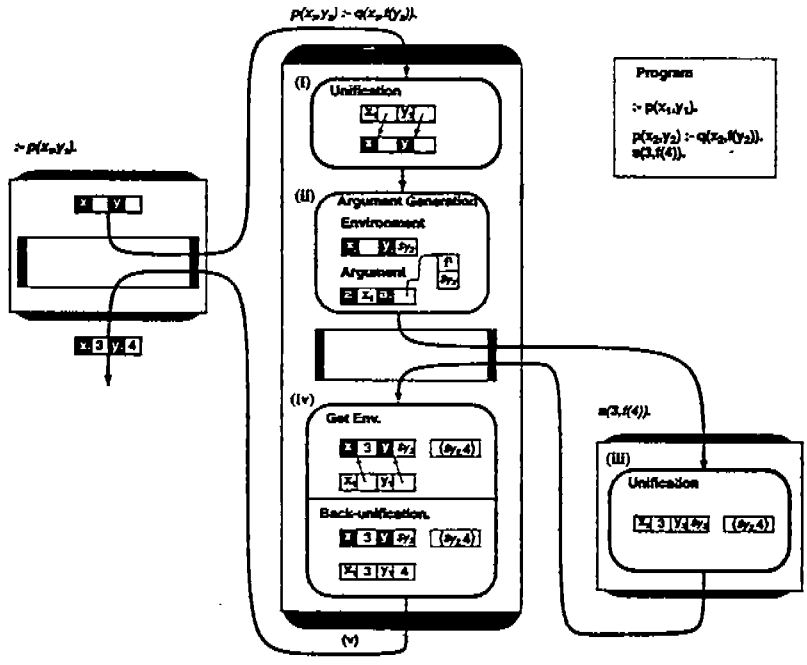
Using an example, we illustrate the QCE with focus on the structure sharing and the general principles of the QCE which include head unification, generation of instance variables, and back-unification.

In the illustration, we take the following pictorial notation. The execution is depicted in an abridged form. An OR-task is represented in a rectangular embellished with shades both on the top and the bottom. Inside the OR-task, the results of primitive functions such as head-unification, argument generation and back-unification are depicted by rectangulars with rounded edges. The representation of a variable in an environment frame consists of a name and a content. The reference binding is represented simply by an arrow. A structure reachable from the environment is represented by a rectangular with a set of vertically stacked slots. The first slot at the top has a structure name with a superscript indicating the number of the copies resulting from structure copying. An instance variable is represented by a variable name preceded with '$' sign, and 'x' and 'y' are used as variable names. Subscripts are used in the variable names for notational convenience.

In (Fig. 4), goal $p(x_1, y_1)$ is tried for clause $p(x_2, y_2) :- q(x_2, f(y_1))$. Together with the result of unification in (i), we can see the environment and arguments before the goal activation in (ii). An instance variable $\$y_2$ is produced at the stage of argument generation. The arguments are unified with goal $q(x_2, f(y_2))$. As one of the result, the instance variable $\$y_2$ is instantiated with 4 and stored in an auxiliary store indicated by a rectangular box with a shared boundary in (iii). Finally, the result is propagated back to the upper OR-task and the back-unification takes place as does in (iv). The resulting environment has an instantiation on every slot and is passed to the task in the upper level.

4. Comparison with Other Researches

Both the closed environment (CE) and the quasi-closed environment (QCE) pursue restricted accesses.



(Fig. 4) An example : the operation of the QCE

However, the QCE method is designed with emphasis on an optimization of structure handling to avoid structure copying. Indeed, when a program does not contain structure data, there will be no differences between the two method. In this section, we compare the QCE with other relevant methods particularly with regard to the structure handling.

In the ROPM method[5], the binding method is again organized using the closed environment on top of tuples. Each time after unification, the input tuple is closed with respect to a reference tuple, and back-unification engaging another closing is performed when returning a result. This method has also an optimization in structure handling. Based on the classification of structures with three types (*general, ground,* and *closed structures*), the method avoids copying the closed structures, using an indirect pointer called a *moecule*. As a matter of fact, the optimization is effective only when a large portion of structures in a program is in a closed form at runtime. Therefore, it is hard to get substantial benefit form the optimization because in general, the portion of closed structures is small in real world applications.

Three binding methods are compared in terms of principal operations made with respect to the structures in ⟨Table 1⟩. The first three columns concern with the requirement of the structure copying for the three classes of structures, and the forth one is the requirement of structure scanning at environment closing. The last one is whether frames are extended or not at environment closing. In pure closed environments, a working environment frame is to be extended to import all the reachable variables in the reference environment frame. It is noticed that the QCE is devoid of any type of structure copying, scanning of structures, and frame extension.

5. Concluding Remarks

In this paper, we presented a new binding method, called quasi-closed environment (QCE). It has been designed for maintaining the efficiency of the shared binding methods at intra-PE task threads, while for avoiding remote accesses as doing in the closed binding method. Indeed, the main feature of the QCE method is that no explicit closing operations are required both at unification and at back-unification. As a result, the QCE method does not involve with scanning and copying of structures, as opposed the closed binding environment. Featured with the restricted accesses and high single thread performance, the QCE method is a viable solution to the binding environment for distributed implementation.

References

[1] P. Bosco, C. Cecchi, C. Moiso, M. Port, and G. Soft, "Parallel PROLOG using stack Segments on Shared-Memory Multiprocessors," In *1984 Symposium on Logic Programming*, pp. 2-11, Feb. 1984.

[2] A. Ciepielewski and S. Andrzej, "*A Formal model for OR-Parallel Execution of Logic Programs,*" Information Processing 83, Elsevier-North Holland, 1983.

⟨Table 1⟩ Comparison of distributed binding methods

NR: not required

Binding Method	Ground Structures Copying	Closed Structures Copying	Unbound Structures Copying	Scanning of Structures	Extension of Frames
Closed Env. (CE)	NR	Required	Required	Required	Required
Closed Env. with molecules	NR	NR	Required	Required	Required
Quasi-Closed Env. (QCE)	NR	NR	NR	NR	NR

[3] J. Conery, "Binding Environments for Parallel Logic Programs in Non-Shared Memory Multiprocessors," In *1987 Symposium on Logic Programming*, pp. 159-170, IEEE Computer Society Press, Aug. 1984.

[4] B. Hausman, A. Ciepielewski, and A. Calderwood, "OR-parallel PROLOG make efficient on Shared Memory Multiprocessor," In *1987 Symposium on Logic Programming*, pp. 69-79, IEEE Computer Society Press, Aug. 1984.

[5] L. Kale, "The Reduced-OR Process Model for Parallel Execution of Logic Programs," *Journal of Logic Programming*, 11:55-84, 1991.

[6] L. Kale B. Ramkumar, and W. Shu. "A Memory Organization Independent Binding Environment for And and Or Parallel Execution of Logic Programs," In *Fifth International Conference and Symposium on Logic Programming*, pp. 1223-1240, MIT Press, Aug. 1988.

[7] G. Lindstorm, "OR-Parallelism on applicative Architectures," In *Second International Logic Programming Conference*, pp. 159-170, 1984.

[8] D. Warren. "Implementation of PROLOG Compiling Predicate Logic Programs," Technical Report Vols. 1 and 2, Reports Nos. 39 and 40, Dept. of Artificial Intelligence, University of Edinburgh, May 1977.

[9] D. Warren. "Efficient PROLOG Memory Management for Flexible Control Strategies," In *New Generation Computing 2*, volume 84, pp. 361-369, 1984.

[10] G. Gupta, E. Pontelli, and V. Costa, "Shared Paged Binding Array:AA Universial Data-Structure for Parallel Logic Programming," In *Proceedings of NSF/ICOT workshop on Parallel Logic Programming*, 1994.

이 용 두

1975년 한국 항공대학교 통신공학과 졸업(공학사)
1982년 영남대학교 대학원 전자공학과(공학석사)
1995년 한국항공대학교 대학원 전자공학과(공학박사)
1982년~현재 대구대학교 전자공학과 교수
1981년~1982년 (일)동경대학 전자공학과 객원교수
1991년~1993년 University of Southern California 교환교수
관심분야:컴퓨터구조(병렬처리, 분산처리), 컴퓨터통신.

김 희 철

1983년 연세대학교 전자공학과 졸업(학사)
1991년 Univ. of Southern California(EE&Sys, M.S.)
1996년 Univ. of Southern California(EE&Sys, Ph.D.)
1996년~현재 삼성데이터시스템 선임연구원
관심분야:컴파일러, 컴퓨터구조, 병렬처리시스템, 로직프로그래밍

채 수 환

1973년 한국항공대학교 전자공학과 졸업(학사)
1985년 Univ. of Alabama 전산공학과(M.S.)
1988년 Univ. of Alabama 전기전자공학과(Ph.D.)
1973년~1977년 공군교육사령부 통신학교 교관
1977년~1983년 금성통신(연구원)
1989년~현재 한국항공대학교 컴퓨터공학과 부교수
관심분야:컴퓨터구조, 병렬처리시스템.