# 쉘로우 백트래킹을 사용한 Prolog 컴파일러의 최적화

## (Optimizing Prolog Compiler using Shallow Backtracking)

오 승 환 † 창 병 모 †† 신 동 하 ††† 최 광 무 ††††

(SeungHwan O) (Byung-Mo Chang) (Dongha Shin) (Kwang-Moo Choe)

**요 약** Prolog은 전통적인 컴퓨터 구조 상에서 수행되기 쉽지 않으므로 수행을 위한 가상 기계로써 WAM (Warren Abstract Machine) 이 제안되었다. 대부분의 Prolog 컴파일러는 WAM 코드를 기계어 코드로 변환하거나 에뮬레이션하여 수행하는데 이러한 방법은 각각 이식성과 효율성에 문제가 있다. 이를 해결하기 위하여 WAM 코드를 C로 변환하는 컴파일러가 제시되었는데 대표적인 컴파일러로 프랑스 INRIA에서 개발된wamcc 컴파일러를 들 수 있다. 그러나 이 컴파일러의 성능은 Prolog 최적화보다 생성된 C 코드의 최적화에 의존하고 있으므로 성능 개선의 여지가 있다. 본 논문에서는 Prolog 최적화 방법의 하나로서 쉘로우 백트래킹 최적화를 적용하여 이 컴파일러의 성능을 향상시킨다. 특히 지금까지 [1,2]에서 제시된 두 가지 최적화 방법을 구현하여 그 성능을 비교한다.

***Abstract*** It is difficult to compile Prolog programs on traditional computer architectures, so WAM (Warren Abstract Machine) was proposed as a virtual machine for Prolog execution. Most Prolog compiler executes WAM code by either emulating WAM code or translating WAM code to machine code. However, the two approaches have drawbacks in portability and efficiency, respectively. To overcome the drawbacks, wamcc compiler was proposed at INRIA, which compiles Prolog programs to WAM code in C. The performance of this compiler depends on optimization effect of C compiler rather than Prolog compiler. In this paper, the compiler is improved by applying shallow backtracking optimization technique which is one of optimizing techniques for Prolog language. In particular, two optimization techniques in [1,2] are compared.

## 1. 서 론

Prolog 컴파일러(compiler)는 전통적인 컴퓨터 구조에서 수행되기 쉽지 않다. 이러한 이유로 초창기의 Prolog는 인터프리터(interpreter)를 이용하여 수행되었다. 그러나 이러한 Prolog 수행의 문제점은 느린 수행 속도라 할 수 있으며 이를 해결하기 위해서 효과적인 컴파일러가 필요하다. 최근에 개발되고 있는 대부분의 컴파일러는 Warren에 의해서 제안된 추상 기계(abstract machine)인 Warren Abstract Machine (WAM) 에 기반을 두고 설계되고 있다 [3]. WAM을 기반으로 한 Prolog 컴파일러는 일단 Prolog 프로그램을 WAM 코드로 변환하고 이 코드를 에뮬레이션(emulation) 하거나 목적 기계어 코드(target machine language code)로 변환하여 수행한다 [4]. 그러나 이러한 두 가지 방법은 전자는 수행 효율에 있어서, 후자는 이식성(portability)에 있어서 결점을 가지고 있다. 특히 이식성은 최근에 관심을 모으고 있는 Internet 프로그래밍을 위한 논리 언어 사용에 있어서 중요한 필요조건이라고 할 수 있다 [5,6]. 이 두 조건을 모두 만족하는 WAM 의 구현 방법으로서 WAM 코드를 C로 컴파일하여 수행하는 방법이 제시되었는데 대표적인 컴파일러로 프랑스 INRIA에서 개발된 wamcc 컴파일러가 있다 [7]. 그러나 이 컴파일러는 표준 WAM에서 제시

· 이 연구는 한국 전자통신 연구원 학·연 공동연구 과제 "논리언어 컴파일러 연구"의 지원에 의한 것임

† 비 회 원 : 한국과학기술원 전산학과
†† 종신회원 : 숙명여자대학교 전산학과 교수
††† 정 회 원 : 상명대학교 정보과학과 교수
†††† 종신회원 : 한국과학기술원 전산학과 교수

논문접수 1997년 3월 12일
심사완료 1997년 10월 13일

한 최적화(optimization) 방법 이외의 최적화는 거의 사용하지 않고 있으나 C 컴파일러 단계에서의 최적화로 인한 매우 가능성 높은 성능을 보이고 있다 [8]. 따라서 wamcc 컴파일러에 최적화 기법을 적용하여 성능을 향상시키면 이식성을 갖추면서도 수행 효율이 뛰어난 Prolog 컴파일러를 개발할 수 있을 것으로 기대된다.

한편 Prolog 컴파일러 최적화를 위해서 다양한 방법이 제시되었는데, 단일화(*unification*) 최적화, 결정성(determinism) 이용, 백트래킹(backtracking) 최적화 등을 들 수 있다. 이러한 Prolog 컴파일러 최적화 방법 중의 하나로 쉘로우 백트래킹(shallow backtracking)을 최적화 하는 방법이 제시되었다. 이 개념은 Warren 에 의해서 처음 도입되었는데 같은 프로시저에 대해 더 시도할 클로우즈(clause)가 있는 상태에서 머리 단일화(head *unification*)가 실패하는 간단한 경우를 쉘로우 백트래킹이라 부르고 일반적인 경우를 딥 백트래킹(deep backtracking) 이라 불렀다 [9]. 만일 프로시저 호출(procedure call)과 쉘로우 백트래킹을 일으키는 머리 단일화 실패 사이의 기계 상태(machine state) 변화를 최소화하면 쉘로우 백트래킹은 딥 백트래킹보다 훨씬 효율적으로 구현될 수 있다. 그러나 쉘로우 백트래킹 최적화를 위해서는 WAM 의 구조와 WAM 컴파일러를 확장해야 한다. 이를 구현하는 방법은 [10] 과 [1] 에서 제시되었다. 이 방법에서는 만일 쉘로우 백트래킹이 일어나지 않는 경우에는 쉘로우 백트래킹 최적화를 하지 않는 경우에 비해서 약간의 비효율성이 생길 수 있다. 이러한 경우를 최적화 하기 위해 두개의 WAM 코드열을 생성하여 비효율성을 줄이는 방법이 제시되었다 [2].

본 논문에서는 wamcc 컴파일러의 수행 성능 향상을 위한 최적화 기법의 하나로서 쉘로우 백트래킹을 구현하여 그 수행 효율을 향상시켰다. 이를 위해 WAM을 확장하여 새로운 명령어와 레지스터을 추가하였으며, 이러한 확장된 WAM 구조에 맞는 WAM 코드 생성을 위하여 wamcc 컴파일러를 확장하였다. 특히 본 논문에서는 [1] 과 [2] 에서 제시된 쉘로우 백트래킹 최적화 기법을 구현하여 그 효과를 비교 분석하였다.

2절에서는 본 논문에서 구현한 최적화 방법인 쉘로우 백트래킹 최적화를 설명한다. 3절에서는 쉘로우 백트래킹 최적화를 구현하기 위한 방법과 wamcc 상에서 구현에 대해서 설명하고 실험 결과를 분석한다. 4절에서는 결론을 맺는다.

## 2. 쉘로우 백트래킹

이 절에서는 쉘로우 백트래킹 최적화에 대해서 설명하고 그 구현을 위한 WAM 의 확장과, 컴파일러의 확장에 대해서 설명한다. 또한 쉘로우 백트래킹 최적화가 가져올 수 있는 비효율성에 대해 설명하고 이러한 비효율성을 줄이기 위해서 Carlsson에 의해 제시된 두개의 코드 열을 생성하는 방법에 대해서 설명한다.

### 2.1 WAM의 소개

WAM 은 전통적인 컴퓨터 구조상에서 Prolog를 빠르게 수행하기 위해 Warren 이 제시한 메모리 구조(memory structure)와 명령어 집합(instruction set)으로 이루어진 추상 기계이다 [11,3]. 이후 WAM 은 Prolog 구현에 있어서 하나의 표준으로 정착 되었다.

WAM 은 그림 1과 같이 메모리와 레지스터들로 구성된다. 메모리는 사용 용도에 따라 코드(code), 힙(heap), 스택(stack), 트레일(trail), 그리고 PDL(Push-Down List) 로 나누어지며, 이들은 연속된 하나의 주소 공간에 순차적으로 위치한다. 코드는 수행될 WAM 명령어 프로그램이 저장되는 곳이며, 두 레지스터 P 및 CP가 코드 메모리의 한 곳을 가리킨다. 레지스터 P는 프로그램 카운터(program counter)로서 다음 수행될 명령어가 있는 곳의 위치를 가지고 있으며, 레지스터 CP는 절차 수행 후 돌아올 곳의 위치를 가지고 있다. 힙 메모리는 프로그램 수행 중 생성 혹은 소멸되는 텀(term)들을 저장하는 곳이다. 힙의 실제 사용 공간은 프로그램이 수행하는 도중 증가 혹은 감소한다. 레지스터 S, HB, H는 힙 메모리를 가리킨다. 레지스터 S는 단일화의 읽기 모드(read mode)에서 다음에 매치될 텀이 있는 곳을 가리키고, 레지스터 H는 다음 사용 가능한 힙의 위치를 가리키며, 레지스터 HB는 가장 최근의 선택점(choice point)이 형성되었을 때의 H 값을 가진다. 스택은 프로시저의 수행 도중 생성되는 선택점 프레임(choice point frame) 및 영구 변수(permanent variable)의 내용을 가지는 환경 프레임(environment frame)을 저장하는 곳이다. 이 스택은 기존의 명령형 언어의 스택과 비슷한 개념이나 그 구조 및 용도는 조금 다르다. 레지스터 B는 스택 내에 저장된 최근의 선택점 프레임을 가리키고, 레지스터 E는 최근의 환경 프레임을 가리킨다. 그리고 레지스터 B0는 cut을 구현하기 위한 레지스터이다. 트레일은 백트랙시 변수들에 대한 바인딩(binding)을 이전 선택점 상태로 돌려놓기 위하여 힙 메모리 중 되돌려 놓아야 하는 부분을 기록하는 메모리이다. PDL은 단일화 관련 명령어 수행시 일시적으로 사용되는 메모리이다. 또한 WAM은 프로시저 호출시 인수의 전달을 위하여 A0,...,An 인수 레지스터를 가지고 있다. 이 인수 레지스터는 힙의 한 메모리

셀과 같은 크기를 가진다.

WAM에서 선택점은 Prolog의 비결정적인 연산을 위해 한 프로시저 호출시 그때의 기계 상태를 스택에 저장하여 백트랙시 프로시저 호출 이전의 상태로 돌아가게 하기 위해 필요하다. 선택점을 저장하는 선택점 프레임에는 그 선택점이 생성될 때의 레지스터들이 저장된다. 선택점 프레임의 내용은 그림 1에 나타나 있다. WAM의 명령어는 크게 4가지로 분류할 수 있다. 먼저 텀을 힙 메모리에 구성하는 put/set 명령어, 힙 메모리에 이미 구성된 텀과 새로운 텀을 비교하여 단일화를 수행하는 단일화 명령어, 스택에 환경 프레임을 생성/제거하거나 프로시저의 호출 및 되돌아옴(return)을 처리하는 제어 명령어, 스택에 선택점 프레임을 생성 혹은 제거하는 선택 명령어가 있다.

put/set 명령어에는 put_variable, put_value, put_structure, set_variable, set_value, set_structure 명령어가 있다. 단일화 명령어에는 get_structure, get_variable, get_value, unify_variable, unify_value 명령어가, 제어 명령어에는 call, execute, allocate, deallocate 명령어가 있다. 선택 명령어에는 try_me_else, retry_me_else, trust_me 명령어가 있다.

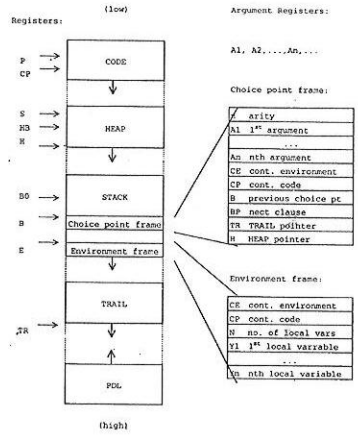


그림 1 WAM 구조

이중 선택 명령어는 쉘로우 백트래킹 개념과 연관이 크므로 선택 명령어에 대해 설명한다. try_me_else 는 새로운 선택점 프레임을 생성하고 선택점 프레임을 초기화하며 선택점 프레임에 인자 레지스터들을 저장한다. retry_me_else 는 선택점으로 백트랙한 뒤 수행되는 명령어로 인자 레지스터를 선택점 프레임에서 복구하고, 트레일을 이전 상태로 복구하며 선택점 속의 다음 선택점 항목을 갱신한다. trust_me 는 한 프로시저에 대해 마지막 클로우즈를 호출했을 때 수행되며 인자 레지스터를 선택점 프레임에서 복구하고, 트레일을 이전 상태로 복구하며 다음 선택점을 선택점 프레임 안의 이전 선택점 값으로 (즉 이전 프로시저를 나타내는 선택점으로) 정한다.

### 2.2 쉘로우 백트래킹의 개요

쉘로우 백트래킹의 개념은 [9] 에서 제시되었는데 쉘로우 백트래킹은 프로시저 호출에 대한 클로우즈 머리와의 단일화가 실패할 때 그 프로시저에 대해 아직 시도해 보지 않은 다른 클로우즈가 남아 있을 경우에 일어나는 백트래킹이다. Prolog 프로그램 수행 중에 행해지는 AND/OR 트리 탐색으로 설명하면 쉘로우 백트래킹은 한 OR 노드 바로 뒤에 AND 노드를 시도할 때 일어난다. 또 하나의 백트래킹 유형은 딥 백트래킹 이라고 하며 이것은 프로시저 호출이 클로우즈와 단일화에 실패하고 더 이상 시도할 클로우즈가 남아 있지 않은 경우에 일어난다. 이 경우 시스템은 이전 골로 되돌아가 다른 골을 시도해야 한다. 그러나 쉘로우 백트래킹은 같은 프로시저에 대한 다른 클로우저를 시도해야 함으로 현재의 레지스터 상태 중 많은 부분이 바뀌지 않아도 된다. 따라서 프로시저 호출과 쉘로우 백트래킹 사이에서의 레지스터 상태(register state)의 변화를 최소화 하면, 쉘로우 백트래킹은 딥 백트래킹보다 훨씬 효율적으로 구현될 수 있다.

쉘로우 백트래킹의 예를 살펴보자. 다음 Prolog 프로시저 memberchk는 리스트로 표현되는 집합에 원소를 하나 더하는 프로시저이다.

```
memberchk(Item, [Item|_]) :- !.
memberchk(Item, [_|Rest]) :- memberchk(Item,Rest).
```

이 예는 명백히 어떤 호출에 대해서 두 클로우즈 중의 하나만 만족할 수 있다. 그러나 컴파일 시에 어떤 클로우즈가 호출에 대해 만족할지를 결정하는 것은 불가능하며, 인덱싱(indexing)도 불가능하다. 이 프로시저 수행은 다음과 같이 이루어 진다.

1) 선택점이 스택에 생성된다. 2개의 인자 레지스터와 WAM 레지스터들이 선택점에 저장된다.
2) 두번째 인자 레지스터가 [Item|_] 과 단일화된다. Item은 첫번째 인자이다.
3) 단일화가 성공하면 선택점이 스택에서 제거되고 일부 레지스터가 복귀된 뒤에 프로시저 수행이 끝난다.
4) 단일화가 실패하면 인자 레지스터와 WAM 레지

스터들이 복귀되고 나머지 클로우즈로 수행이 계속된다.

5) 두번째 인자 레지스터는 [_|Rest] 와 단일화되며 프로시저는 새로운 두번째 인수 Rest로 자신을 호출한다.

이 경우 Prolog 기계가 하는 대부분의 일은 불필요하다 인자 레지스터는 첫번째 클로우즈와 단일화하는 동안 변화하지 않으며 대부분의 WAM 레지스터도 변화하지 않는다. 따라서 선택점을 바로 레지스터에 넣고 바로 선택점을 제거하는 것은 성능의 손실을 가져온다. 이러한 성능의 손실의 원인은 Prolog에 if-then-else 문을 수행할 수 있는 방법이 선택점 생성 밖에 없기 때문이다.

딥 백트래킹이 일어나면 선택점에 저장된 정보는 모두 유용하다. 반면 쉘로우 백트래킹이 일어날 때는 상당한 양의 일을 하지 않을 수 있다 선택점에 모든 정보를 저장하는 대신 선택점에 정보를 저장하는 것을 몸체(body)의 서브골(subgoal)을 호출할 때로 연기하면 보다 효율적인 코드를 얻을 수 있다.

쉘로우 백트래킹의 최적화를 WAM 상에서 구현하기 위한 방법은 Meier에 의해서 제시되었다 [1]. 기본적인 개념은 *try_me_else* 명령에서 선택점에 기계 상태를 모두 저장하는 것이 아니라 머리 단일화 동안에 바뀔 가능성이 있는 일부의 기계 상태만을 저장하는 것이다. 여기에는 상당수의 WAM 레지스터와 인자 레지스터(argument register)가 포함된다. 선택점의 완성은 새로 추가된 명령어 neck 이 도달될 때까지 지연된다. neck 명령은 각각의 클로우즈에 대한 컴파일된 WAM 코드에 삽입되어서 적절한 지점에서 선택점을 완성하거나 갱신한다. neck 명령어는 그 클로우즈에서 머리 단일화의 성공을 알 수 있는 가장 빠른 지점에 삽입된다. 따라서 neck 이전에서 머리 단일화가 실패하게 되면 선택점을 만드는데 필요한 시간이 neck 명령어에서 선택점을 완성하는데 드는 만큼 절약된다. 또한 이 경우 다음 시도할 클로우즈에서 선택점에서 기계 상태를 회복할 필요도 없으므로, 이에 필요한 시간도 절약된다. 그러나 이를 위해서는 선택점에 저장되어야 하나 neck 명령어가 나타날 때까지 연기되는 기계 상태가 머리 단일화하는 동안 보존되어야 한다. 기본적인 코드 생성 방법에 따라 WAM 코드를 생성하면 인자 레지스터는 머리 단일화 동안 변화하지 않으나, wamcc 컴파일러는 레지스터 할당 최적화를 통하여 불필요한 레지스터 이동을 최소화함으로 머리 단일화하는 동안 인자 레지스터의 내용이 바뀔 수 있다. 예를 들어 다음과 같은 재귀적 클로우즈 append/3 은

```
append ([X|L1],L2,[X|L3]) :-
        append (L1,L2,L3).
```

레지스터 할당 최적화에 의해 다음과 같은 WAM 코드로 컴파일된다.

```
get_list A1
unify_variable X4
unify_variable A1
get_list A3
unify_value X4
unify_variable A3
execute append/3
```

이 예에서 머리 단일화를 수행하는 동안 unify_variable A1 명령에 의해서 인자 레지스터 A1의 내용이 바뀔 수 있다. 따라서 쉘로우 백트래킹 최적화를 구현하기 위해서는 이 코드에 곧바로 neck 을 삽입하는 것만으로는 부족하며, 머리 단일화 동안 바뀔 가능성이 있는 레지스터를 다른 레지스터에 옮겨서 머리 단일화를 수행하고, 머리 단일화가 끝난 뒤, 즉 neck 명령 뒤에서 put_value 명령어를 사용하여 원래의 인자 레지스터로 그 값을 옮겨 주어야 한다. 위의 예에 대하여 올바르게 neck 명령을 삽입한 코드는 다음과 같다.

```
get_list A1
unify_variable X4
unify_variable X5
get_list A3
unify_value X4
unify_variable X6
neck
put_value X5,A1
put_value X6,A3
execute append/3
```

따라서 쉘로우 백트래킹이 일어나지 않을 경우, neck 명령이 삽입된 코드는 그렇지 않은 코드에 비해 더 많은 put_value 명령을 포함하게 된다 그러나 일반적으로 쉘로우 백트래킹이 매우 자주 일어나기 때문에, 이 늘어난 put_value 명령에 의해서 생기는 비효율성보다는 쉘로우 백트래킹을 최적화함으로써 얻어지는 성능의 향상이 크다고 알려져 있다.[1, 2]

위에서 서술했듯이 쉘로우 백트래킹 최적화를 위해서는 레지스터 할당 최적화를 부분적으로 포기해야 할 필요가 있다. 또한 할당 최적화를 포기해야 하는 명령어가 코드 중에 없더라도 쉘로우 백트래킹이 실제로 일어나지 않는다면 아무 일도 하지 않는 neck 명령이 비효율

성의 원인이 될 수 있다. 따라서 한 프로시저에 대해 결정적으로 어떤 클로우즈가 실행될지 결정된 경우에는 neck 명령이 삽입되지 않은 코드를 사용하는 것이 더 효율적이다. 이러한 경우를 위해 두개의 코드 열을 생성하는 방법이 Carlsson에 의해 제시되었다. [2] 이 방법에서는 try_me_else 명령과 retry_me_else 명령을 통해 클로우즈가 실행될 때는 neck 이 삽입된 코드를 사용하고, 그 외의 모든 경로로 클로우즈가 실행될 때는 neck 이 삽입되지 않은 코드를 사용한다. 이 방법의 단점은 코드 크기가 커진다는 것이며 이것은 특히 단위 클로우즈 데이타베이스같은 경우에 있어서 불리하다. 이 방법을 사용하여 생성된 WAM 코드의 예는 다음과 같다.

```
switch_on_term(L1,L5,fail,L6,fail)
% Level 2 인덱싱에 의한 분기, 첫번째 인자의 타입에 따라 L1, L5, L6 으로
%분기한다. L5, L6 는 Carlsson 방법에 의해 새로 생성된 neck 을 포함하지 않는
%코드이다. 종래의 경우 L5,L6 는 각각 L2, L4 로 분기하게 된다.
L1
    try_me_else L3
L2      % neck 을 사용한 코드
    get_nil A0
    get_nil A1
    get_nil A2
    neck
    proceed
L3
    trust_me
L4      % neck 을 사용한 코드
    get_list A0
    ....
    unify_variable X4
    neck
    put_value X5,A2
    execute pair/3
L5      % neck 을 사용하지 않은 코드
        (L2 클로우즈의)
    get_nil A0
    get_nil A1
    get_nil A2
    proceed
L6      % neck 을 사용하지 않은 코드
        (L4 클로우즈의)
    get_list A0
    ....
    unify_variable X4
    execute pair/3
```

## 3. 구 현

이 절에서는 쉘로우 백트래킹 최적화를 구현하기 위한 WAM 확장에 대해서 설명하고 확장된 WAM 코드 생성을 위한 wamcc 컴파일러의 확장에 대해서 설명한다.

### 3.1 쉘로우 백트래킹 최적화를 위한 WAM 의 확장

쉘로우 백트래킹 최적화를 WAM 상에서 구현하기 위한 방법은 [1] 에서 제시되었다. 쉘로우 백트래킹의 최적화를 WAM 상에서 구현하기 위한 기본적인 아이디어는 try_me_else 명령어에서 선택점에 대한 공간을 할당하고 선택점의 모든 내용을 저장하는 것이 아니라, 머리 단일화 동안에 변화할 가능성이 있는 기계의 상태만을 선택점에 저장하고 머리 단일화가 성공한 다음에 나머지 기계 상태들로 선택점을 완성하는 것이다. 코드 상에서 머리 단일화가 성공했는지를 판별할 수 있는 가장 빠른 지점에 새로운 명령 neck을 추가한다. neck 명령은 각각의 클로우즈에 대한 컴파일된 WAM 코드 안에 삽입되어 머리 단일화가 성공한 경우 그 프로시저에 대해 다른 클로우즈를 시도할 수 있을 때 선택점을 완성하거나 갱신하는 일을 한다.

쉘로우 백트래킹 최적화를 구현하기 위해서는 먼저 두개의 레지스터를 새로 정의해야 한다. 하나는 컴파일러가 단일화 실패가 일어날 때 딥 백트래킹이 일어나야 하는지 쉘로우 백트래킹이 일어나야 하는지 알 수 없으므로 이를 구별하는 레지스터가 필요하다. 이 새로운 레지스터를 D 라고 한다. 또한 현재 선택점이 완성되어 있는지 아닌지를 나타내는 레지스터가 필요하다. 이 레지스터는 최초로 머리 단일화가 성공한 뒤의 neck 명령에서 한번 선택점이 완성된 뒤 다른 클로우즈 안의 neck에서 다시 불필요하게 선택점을 완성시키는 것을 피하기 위해 필요하다. 이 레지스터를 ChP 라고 부른다. 이 두 레지스터는 각각 불리안(boolean) 값을 가진다. 따라서 try_me_else, retry_me_else, trust_me 명령어들은 이 새로운 레지스터들에 따라 그 의미가 바뀌어야 한다. 또한 부분 선택점(partial choice point) 만이 만들어진 상태에서 머리 단일화가 성공할 경우 선택점을 완성하는 neck 명령이 필요하다. 이 명령어들의 의미는 다음과 같다.

1) try_me_else: 이 명령어는 스택에 선택점 크기 만큼

의 공간을 확보하고 여기에 선택점 항목들을 저장한다. WAM 레지스터들 중 머리 단일화 동안에 변화할 수 있는 레지스터는 E, B 와 TR 이다. 따라서 이 레지스터들은 try_me_else 명령에 의해서 선택점에 미리 저장되어야 한다. 레지스터 CP,H 는 머리 단일화 동안에 변화할 수 없으므로 이 레지스터들은 neck 명령어를 만날 때 선택점에 저장된다. 그리고 인자 레지스터의 경우 레지스터 할당 최적화 때문에 머리 단일화 동안 내용이 바뀌도록 WAM 코드가 생성될 수 있지만, 여기서는 그러한 경우 인자 레지스터가 아닌 임시 변수 레지스터를 사용하도록 WAM 코드가 생성되었다고 가정하므로 인자 레지스터도 선택점에 저장할 필요가 없다. 레지스터 할당 최적화때문에 인자 레지스터의 내용이 바뀌지 않도록 레지스터 할당을 고치는 알고리즘은 다음 절에서 서술된다. 또한 try_me_else 명령에서는 새로운 레지스터 D 와 ChP 의 값이 정해져야 한다. D 는 쉘로우 백트래킹을, ChP는 부분적 선택점을 나타내는 false로 각각 정해진다. try_me_else 명령어의 의미는 다음과 같이 정의된다.

```
try_me_else L
    newB = new_choice_point_frame
    Stack[newB] = num_of_arguments
    % 스택에 인자 갯수 저장
    n = Stack[newB]
    BP(newB) = L
    % 스택안의 선택점 프레임에 레지스터 값들을 저장
    TR(newB) = TR   % BP(newB) 는 newB 어드레스로 시작하는 선택점
    CB(newB) = E    % 프레임 안에서 BP 레지스터를 저장하는 어드레스를
    B(newB) = B     % 나타낸다.
    B = newB
    D = false
    ChP = false
```

2) retry_me_else: 이 명령어는 선택점의 다음 선택점 항목을 갱신하고 선택점에서 기계 상태들을 복구한다. 그러나 만일 현재의 백트래킹이 쉘로우 백트래킹이라면 기계 상태에 대한 정보가 현재 레지스터 안에 남아있으므로 불필요하다. 따라서 이 경우에는 retry_me_else 명령은 트레일 만을 복구한다. 만일 백트래킹이 딥 백트래킹이라면 선택점에서 기계 상태들을 복구하고 D 와 ChP의 값을 다시 정해주어야 한다. D 는 쉘로우 백트래킹을 나타내는 false로, ChP는 선택점이 존재함을 나타내는 true로 정해진다. retry_me_else 명령어는 다음과 같이 정의된다.

```
retry_me_else L
    n = Stack(B)
    unwind_trail(TR(B))  % 트레일 복구
    BP(B) = L
    if Deep = 0          % 딥 백트래킹이면
       CP = CP(B)        % 레지스터 값들을 선택점에서 복구
       E = CE(B)
       H = H(B)
       for i to n        % 인자 레지스터 값을 선택점에서 복구
           Ai = Ai(B)
       rof
    fi
    D = false            % D, ChP 레지스터 값 재조정
    Chp = true
```

3) trust_me: 이 명령어는 한 프로시저의 가장 마지막 클로우즈를 가리킨다. retry_me_else 명령과 유사하게 D 레지스터의 값에 따라서 의미가 변화한다. 만일 딥 백트래킹이라면 선택점에서 기계 상태를 복구해야 하지만 쉘로우 백트래킹이라면 선택점에서 기계 상태를 복구할 필요는 없다. 두 경우 모두 트레일을 제거하고 선택점을 가리키는 레지스터 B 를 이전 선택점으로 복구한다. trust_me 명령어는 다음과 같이 정의된다.

```
trust_me
    n = Stack(B)
    unwind_trail(TR(B))          % 트레일 복구
    if Deep = 0                  % 딥 백트래킹이면
       CP = CP(B)                % 레지스터 값들을 선택점에서 복구
       E = CE(B)
       H = H(B)
       for i to n
          Ai = Ai(B)
       rof
    fi
    B = B(B)                     % 선택점을 이전 선택점으로 복구
```

4) neck: 이 명령어는 클로우즈에서 머리 단일화가 성공했는지 확인할 수 있는 가장 앞에 위치해서 부분 선택점만이 존재할 경우 선택점을 완성하게 된다. 그

의미는 다음과 같이 정의된다.

```
neck
  if D = false
    if ChP = false  % D 와 ChP 가 모두 false 이면 부분 선택점이 존재
      CP(B) = CP % 선택점에 저장하지 않은 레지스터들을
      H(B) = H   % 저장하여 선택점을 완성
        for i = 1 to n
          Ai(B) = Ai
        rof
      D = true
    fi
  fi
```

D가 false 이고 ChP도 false 인 경우는 이 프로시저에서 최초로 머리 단일화가 성공한 경우로서 선택점을 완성해야 하며, D 가 false 이고 ChP가 true 인 경우는 현재 선택점으로 한번 돌아온 뒤에 머리 단일화가 성공한 경우로 D 만을 딥 백트래킹을 나타내는 true 로 정해주면 된다. 만일 D 가 true 이면 neck 명령은 아무일도 하지 않는다.

### 3.2 wamcc 컴파일러의 확장

wamcc 컴파일러 상에서 쉘로우 백트래킹 최적화를 위한 WAM 코드 생성을 위해서 각 클로우즈 별로 생성된 WAM 코드에 대해서 레지스터 할당을 쉘로우 백트래킹과 모순되지 않게 수정하는 알고리즘과 각 WAM 코드 열 안에 neck 명령을 삽입하는 알고리즘을 구현하였다. wamcc 컴파일러가 Prolog 언어로 구현되어 있으므로 Prolog로 구현된 이 알고리즘들은 wamcc에서 인덱싱 최적화를 수행하는 단계와 C 코드를 생성하는 단계 사이에 삽입되어 neck 명령을 포함한 WAM 코드 및 C 코드를 생성한다. wamcc에서 생성하는 C 코드 상에는 WAM 명령들이 C 마크로의 형태로 표현되고, WAM 명령어들을 나타내는 마크로는 라이브러리 상에 정의되어 있다. 따라서 수정된 WAM 명령들과 레지스터, 그리고 새롭게 추가된 neck 명령어가 C 언어로 구현되어 기존 wamcc 라이브러리에 추가되었다.

확장된 wamcc 컴파일러의 전체 구조는 그림 2에 나타나 있다.

쉘로우 백트래킹 최적화를 위해서는 전술했듯이 머리 단일화 도중에 인자 레지스터의 내용이 바뀌어서는 안된다. 그러나 레지스터 할당 최적화에 의해서 인자 레지스터의 내용이 머리 단일화 도중에 바뀌도록 WAM 코드가 생성될 수 있다. 따라서 쉘로우 백트래킹 최적화를 위해서는 컴파일러가 이렇게 인자 레지스터의 내용이 바뀔 수 있는 레지스터 할당을 찾아 인자 레지스터가 머리 단일화 도중 바뀌지 않도록 레지스터 할당을 고쳐야 한다. 여기서는 이렇게 레지스터 할당을 수정하는 알고리즘을 기술한다.

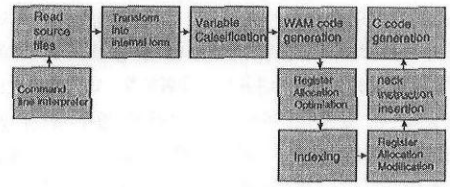


그림 2 확장된 wamcc 컴파일러의 구조

기본적으로 해야 할 일은 머리 단일화 도중에 바뀔 가능성이 있는 인자 레지스터를 찾아, 이 레지스터의 내용이 머리 단일화 도중에 바뀌지 않도록 임시 변수 레지스터(temporary variable register)로 바꾸어 주는 일이다. 따라서 임시 변수 레지스터 (인자 레지스터와 같은 기억장소를 사용하는)중에 함수의 인자 갯수를 넘어서는 레지스터에 대해서는 고려할 필요가 없다. 또한 인자 레지스터가 처음 몇 명령에서는 오직 인자 레지스터의 내용을 읽어오기 위해 사용될 수 있으므로, 이 경우에 인자 레지스터를 읽어들이는 명령의 인자는 변화될 필요가 없다. 따라서 여기서는 한 클로우즈에 해당하는 WAM 명령어의 리스트 중에서 neck 명령 이전에 인자 레지스터가 최초로 쓰여지기 위해 사용되는 명령을 찾아, 그 명령에서 쓰여지는 인자 레지스터를 임시 변수 레지스터로 바꾸어 주고, 이후에 neck 명령까지 모든 그 인자 레지스터가 사용된 명령어를 찾아 인자 레지스터를 임시 변수 레지스터로 바꾸어준 뒤, neck 명령 바로 뒤에, 임시 변수 레지스터를 인자 레지스터로 복귀시키는 put_value 명령을 추가해주어야 한다. 이러한 WAM 코드의 변화에 의해서 생기는 비효율성은 추가되는 put_value 명령 만큼 만이다. 단 아래에 제시된 Algorithm 1 에서는 삽입된 명령이 put_x_value 명령인데, 이것은 put_value 명령 중에서 X 레지스터를 사용하는 명령을 wamcc 에서 구현의 편의를 위해 put_x_vlaue 명령으로 구분하기 때문이다. X 레지스터는 임시 변수를 저장하기 위해 사용되며 Y 레지스터는 영구 변수를 저장하기 위해 사용된다. 이러한 명령어의 구분은 unify 명령에 대해서도 마찬가지이다.

neck 명령 이전에는 put 명령은 사용될 수가 없으므

로, 표준 WAM 명령에서 인자 레지스터의 내용을 바꿀 가능성이 있는 명령은 unify_variable 과 unify_value 명령 뿐이다. 그러나 wamcc 에서는 unify_variable 과 unify_value 명령 외에도 인자 레지스터의 내용을 바꿀 수 있는 명령이 Prolog의 내장 함수를 구현하기 위해 추가되어 있다. 이러한 명령에는 builtin_2, builtin_3, functon_1, funcion_2, math_load_x_value 명령들이 있다

쉘로우 백트래킹 최적화를 위해서 수정된 레지스터 할당 알고리즘은 다음과 같다.

Algorithm 1. 레지스터 할당의 수정

Modify_Register_Allocation (WamCode : list of WAM instructions, Arity : integer)

MaxReg = A(i) 또는 X(i) 가 WamCode 안에 나타나는 가장 큰 i

for i =1 to Arity

FirstW = WamCode 의 시작부터 최초의 put 명령 사이에 최초로 나타난 A(i) (iArity) 에 값을 쓰는 명령

FirstW 의 출력 인자만을 A(i) 에서 A(i+MaxReg) 로 바꾼다.

FirstW 뒤와 최초의 put 명령 사이의 모든 A(i) 와 X(i) 를 A(i+MaxReg) 와 X(i+MaxReg) 로 바꾼다.

WamCode 의 최초의 put 명령 바로 앞에 put_x_value(X(i+MaxReg),A(i)) 명령을 삽입한다.

rof

쉘로우 백트래킹 최적화를 위해서는 Prolog 프로그램의 각 클로우즈에 해당하는 WAM 코드 마다 머리 단일화가 성공했는지를 판별할 수 있는 최초의 위치에 neck 명령이 삽입되도록 컴파일러가 수정되어야 한다. 하나의 Prolog 클로우즈에 해당하는 WAM 코드에 neck 명령을 삽입하는 알고리즘은 [12] 에서 제시되어 있다. 여기서는 이 알고리즘을 기술하기로 한다.

Algorithm 2 : neck 명령의 삽입

Insert_neck_Instruction(L : 하나의 클로우즈를 나타내는 WAM .코드의 리스트,

Lout : neck 명령이 삽입된 WAM 코드의 리스트)

L 안의 최초의 put 명령 P 를 찾는다.

L = L1 @ [P|L2] 라 할 때, % @ 는 리스트의 연결 (concatenation) % 을 나타내는 연산자 (operator) 이다

Lout = L1 @ [neck,P|L2]

if L1 안에 cut 명령이 있으면

L1 = L3 @ [cut|L4] 이라 할 때,

Lout = L3 @ [neck,cut|L4] @ [P,L2]

fi

Lout = L5 @ [Pneck,neck] @ L6 이라 하면,

while Pneck 이 실패 가능성이 없는 명령어일 동안

Lout = L5 @ [neck,Pneck] @ L6

L5, Pneck, L6 을 다시 할당.

elihw

Lout = L7 @ [neck|L8] 이라 하자

for L7 .안의 각 명령어 H 에 대해서

if H 가 get_varible(A,B) 이고 B 가 변수인 머리 인자(head argument)이면, Lout 에서 H 를 neck 명령어 바로 뒤로 옮긴다

rof

위 알고리즘에서 실패할 수 없는 명령이란 unify_-variable 이나 deallocate처럼 명령어의 정의에 따라 명령어의 수행 도중 실패가 일어나 백트래킹의 원인이 될 가능성이 없는 명령을 말한다. 여기서 고려해야 할 실패 가능성이 없는 명령어는 다음과 같다.

unify_x_varialbe

unify_y_variable

proceed

allocate

deallocate

한편 변수인 머리 인자에 해당하는 get_variable (A,B) 명령이 neck 명령어 뒤로 이동될수 있음은 이 명령의 효과를 이용하면 보일 수 있다. A 는 neck 이전까지 다시 사용되지 않는다. 왜냐하면 put 명령들이 나타나기 전의 모든 A 의 사용은 이후 B로 대치되기 때문이다. 또한 A 는 인자 레지스터가 아니기 때문에 neck 자신에 의해서도 사용되지 않는다. 또한 neck 까지 B 레지스터가 정의될 수 없다. 이것은 neck 까지 모든 입력된 인자가 변화하지 않고 보존된다는 가정 때문이다. 따라서 get_variable 명령을 neck 뒤로 이동시킬 수 있다.

이와 같이 의미에 영향을 주지 않는 범위 내에서 명령어들을 neck 명령 뒤로 이동시키는 것은 머리 단일화 실패 시에 필요 없는 일을 가능한 하지 않게 하여 성능에 좋은 영향을 준다.

### 3.3 두개의 코드열을 사용한 최적화 방법의 구현

두개의 코드열을 사용하여 쉘로우 백트래킹 최적화의 성능을 향상시키는 방법은 [2] 에서 제시되었다. 본 논문에서는 wamcc 상에서 두개의 코드열을 사용하는 최적화 방법을 구현하기 위해서 컴파일러에서 인덱싱 명령을 생성하는 부분을 수정하였다. wamcc 에서는 2 레

벨 인덱싱(2 level indexing)을 사용하고 있다.[11] 즉, 먼저 주어지는 인자의 타입(type)에 따라 인덱싱을 하고, 만일 다시 인자의 값에 따라 인덱싱이 가능하면 그에 대해서 한번 더 인덱싱을 하게 된다. 이러한 인덱싱은 코드에 switch_on 명령어를 삽입하고 코드의 순서를 재배열함으로써 이루어진다. 따라서 만일 switch_on 명령들에 의하여 지시되는 레이블이 try_me_else 나 retry_me_else 를 가리키고 있는 부분이 아니라면 그 레이블에 해당하는 클로우즈는 switch_on 명령어에서 해당하는 인덱싱에 의해 분기했을 때 결정적으로 수행할 클로우즈가 정해져 다른 클로우즈를 시도할 필요가 없는 경우이거나, trust_me_else 명령어 이후에 수행되는 클로우즈임을 나타낸다. 따라서 이러한 경우에는 neck 을 포함한 코드를 수행할 필요가 없다. 이러한 경우에는 프로시저를 컴파일한 WAM 코드 이후에, 해당되는 클로우즈들에 대해 neck 을 포함하지 않은 WAM 코드를 생성하고 각각 새로운 레이블들을 붙인 다음, switch_on 명령어에서 새로 만들어진 레이블로 분기하도록 switch_on 명령들을 고치면 된다. 또한 만일 어떤 프로시저에 대해서 try_me_else 명령어가 사용되지 않은 경우에는 그 프로시저는 그 자체로 결정적인 경우이므로 역시 neck 명령을 사용한 코드를 수행할 필요가 없다. 따라서 이러한 경우에는 neck 명령을 사용하지 않는 코드만을 생성한다.

다음은 이러한 두 개의 코드열을 생성하는 알고리즘이다.

Algorithm 3 : 두 개의 코드열 생성
Two_Code_Generate(L : 한 프로시저에 해당하는 인덱싱 된 WAM 명령어의 리스트)
Split_Code(L,Ls) % L .안에서 레이블들이 가리키는 각 클로우즈에 해당하는 % WAM 명령어의 리스트들을 원소로 가지는 리스트 Ls를 % 반환하는 프로시저
m=number of lables in L
if Ls 의 헤드의 해드가 switch_on_terms 명령이면
Ls 의 헤드의 헤드를 switch_on_terms(L1,L2,L3,L4, L5) 라 하자.
for i=1 to 5
if Ls[Li]의 헤드가 try_me_else 또는 retry_me_else 가 아니면 코드 열 Ls[Li] 에 대해 neck 을 사용하지 않는 코드 Ls[Li] 을 생성,
label(m+1) 과 Ls[Li] 을 L 뒤에 추가.
L 안의 switch_on_terms 명령 안의 레이블 Li 를 m+1 로 대체
m = m + 1
fi
rof
else
if Ls 의 헤드의 해드가 try_me_else 명령이 아니면
L 전체를 neck 을 사용하지 않는 코드로 대체
% L 이 나타내는 프로시저는 클로우즈 하나만으로 구성된다.
fi
fi

## 4. 실험 및 결과 분석

본 논문에서 쉘로우 백트래킹 최적화의 성능을 실험하기 위해 사용한 프로그램은 다음과 같다. 순수하게 구현된 WAM과 컴파일러의 확장된 부분만이 성능에 미치는 영향을 측정하기 위해서 일반적인 Prolog 프로그램 외에 순수하게 쉘로우 백트래킹만을 하는 프로그램과 딥 백트래킹만을 하는 프로그램을 포함시켰다.

1) queen8 & queen10 : 8queen 문제와 10queen 문제를 푸는 Prolog 프로그램
2) zebra : 제약 (constraints) 에 기반을 둔 논리적 퍼즐
3) crypt : 숫자 찾기 퍼즐을 푸는 프로그램
4) sendmore : SEND+MORE=MONEY 퍼즐 프로그램, 쉘로우 백트래킹을 전혀 하지 않는다. (모든 프로시저가 각각 단 하나의 클로우즈로 구성되어 있다.)
5) ham : 주어진 그래프의 해밀토니안 패스를 찾는 프로그램
6) membercheck : 인자가 리스트 안에 있는지를 검사하는 프로시저
7) shallow & deep : 각각 99번씩의 쉘로우 실패와 딥 실패를 행하는 프로그램

shallow 와 deep 프로그램들은 Pereira에 의해 작성된 ICOT 벤치마크에 주로 기반한 프로그램들이다 [13]. membercheck 프로그램은 3장에서 언급한 memberchk 프로시저이다. 나머지 프로그램들은 wamcc 컴파일러에 포함되어 있는 벤치마크 프로그램이며, Prolog의 벤치마크로서 흔히 사용되는 프로그램들이다 [8].

위 벤치마크 프로그램들은 각각 쉘로우 백트래킹 최적화를 포함하지 않은 wamcc 컴파일러와 Meier 에 의해 제안된 쉘로우 백트래킹 최적화 방법[1]을 확장한 wamcc 컴파일러, 그리고 앞에서 설명된 두개의 코드 열을 생성하는 쉘로우 백트래킹 최적화[2]를 확장한 wamcc 컴파일러에 의해 컴파일되어 수행되었다. 이 방법들을 각각 앞으로 방법 1, 방법 2, 방법 3 이라고 부른다.

먼저 프로그램들이 수행하는데 걸린 시간은 표 1과 같다. 수행 시간은 SUN SparcStation 2 상에서 gcc 2.5.8에 의해 컴파일된 실행 코드를 사용하여 msec 단위로 측정되었다. 단 shallow 프로그램과 back 프로그램 및 membercheck 프로그램은 컴파일시에 인덱싱이 일어나지 않아 방법 3이 무의미하므로 여기에 표시하지 않았다.

표 1

| | 방법 1 | 방법 2 | 방법 1에 대한 향상 | 방법 3 | 방법 1에 대한 향상 | 방법 2에 대한 향상 |
|---|---|---|---|---|---|---|
| que8 | 810 | 760 | 1.07 | 740 | 1.09 | 1.03 |
| que10 | 15640 | 14410 | 1.09 | 14070 | 1.11 | 1.02 |
| zebra | 2840 | 2810 | 1.01 | 2590 | 1.10 | 1.08 |
| crypt | 6290 | 6330 | 0.99 | 6000 | 1.05 | 1.06 |
| sendmore | 3280 | 3360 | 0.96 | 4040 | 0.98 | 1.01 |
| ham | 4520 | 4180 | 1.08 | 4040 | 1.12 | 1.03 |
| member | 4700 | 4230 | 1.11 | | | |
| shallow | 270 | 170 | 1.59 | | | |
| deep | 570 | 650 | 0.88 | | | |

표2 는 코드의 크기를 나타낸다. 단위는 Kbyte 이다.

표 2

| | | 방 법 1 | | 방 법 2 | | 방 법 3 | |
|---|---|---|---|---|---|---|---|
| | 원시코드 | C 코드 | 실행코드 | C 코드 | 실행코드 | C 코드 | 실행코드 |
| que8&que10 | 1.0 | 10.8 | 31.1 | 10.8 | 31.1 | 12.0 | 31.2 |
| zebra | 1.4 | 13.9 | 31.9 | 14.1 | 31.9 | 14.3 | 32.8 |
| crypt | 1.6 | 16.8 | 32.8 | 17.0 | 32.8 | 19.4 | 32.8 |
| ham | 1.1 | 17.3 | 32.8 | 17.5 | 32.8 | 23.9 | 32.8 |
| sendmore | 1.1 | 12.0 | 32.8 | 12.0 | 32.8 | 13.3 | 32.8 |
| member | 0.4 | 8.9 | 31.1 | 8.9 | 31.1 | | |
| shallow&deep | 1.5 | 13.6 | 32.0 | 15.0 | 32.0 | | |

쉘로우 백트래킹이 일어날 경우에는 구현된 최적화에 의해서 수행 속도 향상이 기대된다. 쉘로우 백트래킹만이 일어날 경우에 (shallow 프로그램) 수행 속도는 약 59% 의 향상을 보였다. 그러나 딥 백트래킹이 일어날 경우에는 neck 명령의 부담과 레지스터 할당 최적화의 파괴에 의한 부담 때문에 최적화를 하지 않았을 때에 비해서 오히려 성능에 부담이 된다. 이 부담의 정도는 딥 백트래킹만이 일어난 프로그램의 경우 (deep 프로그램) 약 12% 정도의 수행 속도의 저하로 나타났다. 그러나 일반적인 프로그램의 경우 쉘로우 백트래킹의 빈도가 높고, 딥 백트래킹에서 일어나는 성능의 부담보다는 쉘로우 백트래킹에서 일어나는 성능의 향상이 더 크기 때문에 전체적인 성능은 향상되는 것으로 나타난다. 한편 방법 3을 사용한 최적화에 의한 결과는 방법 1을 적용한 경우보다 향상된 성능을 나타내었다. 두개의 코드 열 생성 방법을 사용하면 많은 경우 딥 백트래킹이 일어날 때 쉘로우 백트래킹 최적화에 의해서 생기는 부담을 피할 수 있으므로 그만큼 더 쉘로우 백트래킹 시에 일어나는 성능의 향상이 전체 성능의 향상으로 나타나게 된다.

본 논문의 구현에 있어서 일반적인 프로그램들의 방법 2에 의한 쉘로우 백트래킹 최적화에 의한 성능 향상은 약 1% - 9% 정도이고 오히려 성능이 저하되는 예제도 있었다. 그러나 방법 3을 사용하여 최적화 하였을 때는 쉘로우 백트래킹이 전혀 일어나지 않는 예제를 제외하고는 모두 최적화 하지 않은 경우보다 성능 향상을 보였으며, 그 성능 향상은 약 5%-12% 에 이른다. 따라서 두개의 코드 열 생성 방법이 모든 클로우즈에 대해 쉘로우 백트래킹 최적화를 적용하는 것보다 더 좋은 성능을 나타내며, 보다 많은 범위에 있어서 성능의 향상을 가져온다. 방법 3의 방법 2에 대한 성능 향상 폭은 0% - 8% 정도이다.

방법 3의 문제는 코드 크기가 커진다는 점이다. Carlsson 의 구현에서 방법 3을 적용 했을 때 코드 크기는 방법 2에 비해서 약 8% - 19% 정도 늘어나며 코드의 증가량은 약 평균 14% 이다. 그런데 본 논문의 구현은 WAM 의 구현을 위해서 C 언어를 사용하므로 컴파일러가 생성하는 코드는 C 코드이나 최종적으로 생성되는 실행 가능 코드는 C 컴파일러에 의해 생성된다. 따라서 코드의 크기를 비교하기 위해서는 C 코드뿐 아니라 최종적인 실행 가능 코드의 크기도 비교해야 한다. 본 논문에서 구현한 방법 의 방법 2에 대한 코드 증가율은 C 코드의 경우 1% - 36%이며 평균 약 14% 이다. 따라서 방법 3의 코드 크기 증가율은 Prolog 컴파일러가 생성한 C 코드의 경우 Carlsson 의 구현의 경우와 비슷하다고 할 수 있다. 그러나 최종적으로 생성된 실행 가능 코드의 경우 그 크기는 두개의 코드 열 방법과 모든 클로우즈에 쉘로우 백트래킹을 적용하는 방법 모두 거의 변함이 없다. 이것은 wamcc 컴파일러가 WAM 구현에 필요한 각종 모듈을 라이브러리로 구성하여 Prolog 코드에서 생성된 C 로 표현된 WAM 코드와 최종적으로 링크하는 구조로 되어 있기 때문이다.

## 5. 결 론

본 논문에서는 wamcc 컴파일러를 확장하여 Prolog

최적화 방법 중의 하나인 쉘로우 백트래킹 최적화를 구현하였다. 쉘로우 백트래킹 최적화를 위해서는 WAM 구조와 컴파일러를 확장해야 한다. 여기서는 Meier에 의해 제시된 방법에 따라 레지스터와 명령어가 확장된 WAM 코드를 wamcc 컴파일러의 WAM 라이브러리에 구현하였으며 확장된 WAM 코드를 생성하도록 wamcc 컴파일러를 확장하였다. 또한 쉘로우 백트래킹 최적화를 위해 삽입된 neck 인스트럭션과 파괴된 레지스터 할당 최적화 때문에 쉘로우 백트래킹이 일어나지 않을 경우에는 오히려 성능의 부담으로 작용할 수 있다. 이런 이유로 프로그램에 따라서는 오히려 쉘로우 백트래킹 최적화를 적용한 경우 성능이 저하되는 경우도 있을 수 있다. 여기서는 Carlsson에 의해 제시된 방법을 wamcc 컴파일러 상에 구현하였다.

쉘로우 백트래킹 최적화를 적용한 경우 전반적으로 성능이 향상되나 위에서 언급한 부담 때문에 성능이 저하되는 경우도 있었다. 이 경우 두개의 코드열을 사용한 방법으로는 일반적으로도 더 큰 성능 향상을 얻을 수 있었다. 두개의 코드 열을 사용한 방법의 단점은 코드의 크기가 커진다는 점인데, 본 논문에서 쉘로우 백트래킹 최적화를 구현한 wamcc 컴파일러는 C 코드의 형태로 표현된 WAM 코드를 C 컴파일러로 컴파일하여 라이브러리로 구성된 WAM 모듈과 링크시키는 구조로 되어 있기 때문에, 두개의 코드 열 방법에 의해 늘어난 정도의 코드 크기는 최종적인 수행 가능 코드의 크기에는 크게 영향을 주지 않았다. 따라서 이러한 구조를 가진 컴파일러의 경우에는 두개의 코드 열 방법을 사용하는 것이 유리하다.

앞으로는 단일화 최적화, 결정성 이용 등의 최적화 방법들을 이용하여 wamcc 컴파일러를 확장하여 이식성과 효율성을 동시에 만족시키는 보다 효율적인 Prolog 컴파일러를 개발할 예정이다.

## 참 고 문 헌


[1] M. Meier. Shallow Backtracking in Prolog Programs. Internal Report ECRC, 1987.

[2] Mats Carlsson. On the efficiency of optimizing shallow backtracking in compiled Prolog. In Logic Programming : Proceedings of the Sixth International Conference, 1989.

[3] D.H.D Warren. An Abstract Prolog Instruction Set, SRI International, Technical Note 309, 1983.

[4] S Research. SICSus Prolog Users manual. SICS Research Report R88007B, 1988.

[5] S. W. Loke and A. Davison. Logic progra- mming with the world-wide-web. In Proceed- ings of the 7th ACM Conference on Hyper- text, pp. 235-245. 1996.

[6] D. Cabeza, M. Hermenegildo, and S. Varma. The PiLLoW/CIAO Library for INTERNET/WWW Programming, Proceedings of the 1st Workshop on Logic Programming Tools for Internet Applications, pp. 43-62, 1996.

[7] D. Diaz. Wamcc 2.21 Users manual. INRIA-Rocquencourt, 1994.

[8] P. Codognet and D. Diaz. wamcc: Compiling Prolog to C. In Proceedings of the International Conference on Logic Programming, 1995.

[9] D.H.D Warren. IMPLEMENTING PROGLOG - compiling predicate logic programs, University of Edinburgh, D.A.I Research Report 39, 1977.

[10] E. Tick, Studies in Prolog Architectures, Stanford University, CSL-TR-87-329, 1987.

[11] A. Hassan Ait-Kaci, Warren's Abstract machine - A Tutorial Reconstruction, MIT Press, 1991.

[12] M. Carlsson, A Prolog Compiler and its Extension for OR-Parallelism, Swedish Institute of Computer Science, 1995.

[13] F Pereira, Prolog benchmark, Benchmark program in SB/Prolog program, 1986.



오 승 환

1995년 2월 KAIST 전산학과 학사. 1997년 2월 KAIST 전산학과 석사. 1997년 3월 ~ 현재 KAIST 전산학과 박사과정 재학



창 병 모

1988년 서울대학교 컴퓨터공학과 졸업(학사). 1990년 한국과학기술원 전산학과에서 공학석사 학위 취득. 1994년 한국과학기술원 전산학과에서 공학박사 학위 취득. 1994년 ~ 1995년 한국전자통신연구소 박사후 연수 연구원. 1995년 ~ 현재 숙명여자대학교 전산학과 조교수. 관심분야는 컴파일러 구성론(정적 분석, 코드 최적화), 논리 프로그래밍, 연역 데이타베이스

신 동 하

1980년 경북대학교 전자공학과 학사. 1982년 서울대학교 전자계산기공학과 석사. 1994년 Univ. of South Carolina(미국) 컴퓨터과학과 박사. 1982년 3월 1997년 2월 한국전자통신연구소 컴퓨터연구단, 책임연구원. 1997년 3월 ~ 현재 상명대학교 정보과학과 조교수. 관심분야는 프로그래밍 언어(논리), 컴파일러, 계산이론, 인공지능

최 광 무

1976년 서울대학교 전자공학과 졸업(학사). 1978년 한국과학기술원 전산학과 졸업(석사). 1984년 한국과학기술원 전산학과 졸업(박사). 현재 한국과학기술원 전산학과 부교수. 관심분야는 프로그래밍언어론, 논리 프로그램의 병렬 수행 및 컴파일러 구성임.