전역 레지스터 할당 알고리즘 분석

박 종 득(朴 鐘 得)
공주대학교 전자계산학과
전화 : (0416) 850-8519 / 팩스 : (0416) 858-1536

The Analysis of Global Register Allocation Algorithms

Jong Deuk Park
Dept. of Computer Science Kongju National University
E-mail : pjd@knu.kongju.ac.kr

Abstract

In this paper, an compiler system is ported and modified for register allocation experiments. This compiler system will enable various global register allocation. Lcc is introduced and Chaitin's graph coloring algorithm is executed with cmcc on DEC ALPHA 255/300. Several functions of SPEC92INT is used as inputs of the compiler system.

I. 서론

그래프 컬러링은 레지스터 할당을 모델링하는데 널리 사용되는 알고리즘이다. 컴파일러의 레지스터 할당부(register allocator)는 동시에 라이브(live)한 라이브 범위(live range)에 대하여 서로 다른 레지스터를 할당한다. 라이브란 어떤 변수가 정의된 후 마지막으로 참조되기까지의 프로그램 구간을 말한다. 하나의 라이브 범위를 단위 노드로 하고 동시에 라이브한 노드들을 서로 에지로 연결하면 간섭그래프(interference graph)가 만들어진다. 레지스터 할당부는 간섭그래프에 대해 그래프 컬러링 알고리즘을 적용하여 N-컬러링을 구한다. 여기서 N은 실제 레지스터(physical register)의 수이다. 어떤 간섭그래프가 N-컬러링이 된다면, 그래프에 있는 모든 변수를 레지스터에 할당할 수 있다는 의미가 있으므로 프로그램의 실행 속도가 그만큼 빨라진다.

대표적인 그래프 컬러링 알고리즘으로는 Chaitin의 알고리즘이 많이 쓰이고 있다[1]. Chaitin의 알고리즘은 다음과 같은 두 단계로 구성된다.

· 간단화(simplification)

간섭그래프의 어떤 노드 n가 N보다 작은 차수(degree; 노드가 가지는 에지의 수)를 가지면, 레지스터 배정 단계에서 n의 이웃 노드에 관계없이 적어도 하나의 컬러가 배정될 수 있다. n을 제거하고 나서 남아있는 그래프와 n에 인접한 모든 에지가 N으로 컬러될 수 있다면, 이 그래프는 N-colorable하다고 한다. 간섭그래프에서 구속되지 않는 라이브 범위(unconstrained live range)를 제거하는 과정을 간단화라고 한다. 구속되지 않는 라이브 범위란 N 보다 적은 에지수를 가지는 노드를 말한다. 간단화는 그래프에 하나의 노드도 남아 있지 않을 때까지 계속된다. 그런데, 간단화 과정이 도중에 중단되는 경우도 있다. 그래프에 남아 있는 모든 라이브 범위가 N이상의 차수를 가질 때이다. 이 경우에는, 한 라이브 범위를 선

택하여 대피시키게 되며, 해당 노드와 연결된 에지는 그래프에서 제거된다. 이렇게 하면 노드들의 차수가 줄어들 수 있으므로 다시 간단화가 가능해진다. 이 때 대피할 노드의 선택은 휴리스틱하게 이루어진다. 즉 해당 라이브 범위를 메모리로 이동하는 비용을 측정하여 비용이 적은 노드를 대피시키는 것이다. 간단화는 그래프가 비어 있게 될 때까지 계속된다.

· 레지스터 배정(register assignment)

레지스터 배정은 간단화 단계 다음에 실행되며 라이브 범위에 대해 실제 레지스터(physical register)를 선정한다. 레지스터 배정은 간단화 단계에서 노드가 그래프로부터 제거된 역순으로 컬러를 할당한다.

Chaitin의 방법은 간단화 단계에서 바로 대피할 노드를 선택하도록 되어 있다. 그런데 실제로는 이 노드가 컬러를 배정받을 수도 있다. Briggs[2]는 이러한 점을 보완하여 *Chaitin*의 알고리즘을 개선시켰다. Briggs는 대피 노드가 선택되면 무조건 대피를 시키지 않고 일단 레지스터 배정 가능성을 검토하기 위해 대피를 일정 기간 지연시킨다. 그래도 대피할 수 밖에 없다면 비로소 대피 결정이 이루어진다. Chow[3]는 라이브 범위에 우선 순위 함수를 적용시키는 관계로 레지스터에 배정되는 라이브 범위의 순위를 매길 수 있다. 우선 순위 함수를 적용시키게 되면 Chain의 알고리즘처럼 레지스터를 배정받을 수 있는 라이브범위를 대피하는 것과 같은 문제점을 사전에 방지할 수 있다.

한편, 라이브 범위 분할(splitting)[4]이라고 하여 라이브 범위를 쪼개는 연구도 꾸준히 제기되어 왔다. 라이브 범위를 분할하게 되면 간섭그래프의 차수가 줄어드는 효과가 있다.

본 논문에서는 여러 가지의 레지스터 할당 알고리즘을 벤치마크프로그램에 적용하여 각 알고리즘의 레지스터 할당 결과를 측정하기 위한 시스템을 구축하였다. 또한, 라이브 범위를 분할(splitting)하여 그래프 컬러링 알고리즘에 미치는 영향을 아울러 측정하고자 준비하였다.

실험은 DEC ALPHA 255/300에서 카네기 멜론 대학의 cmcc[5]를 사용하여 SPEC92INT 벤치마크 프로그램으로 이루어졌다. 입력 프로그램을 컴파일하기 위해 Fraser[6]의 lcc 전단부를 활용하였다.

II. LCC

lcc는 C 언어 프로그램을 어셈블리 언어로 변환해주는 방식으로 구성되어 있다. lcc의 첫 단계는 C 프리프로세서(preprocessor)로서 헤더 파일과 매크로를 확장하고, 조건부 컴파일 코드를 선택한다. 다만 lcc는 파서와 코드 생성기 위주로 기술되어 있고, SPEC92int 벤치마크 프로그램을 DEC ALPHA 255/300에서 수행시키는 관계로 C 프리프로세서로는 Digital Unix에 탑재되어 있는 C 컴파일러를 사용하였다.

lcc의 파서는 추상구문트리(abstract syntax tree)를 출력한다. 트리의 각 노드는 단위 연산을 나타내고, 소스 코드에 묵시적으로 내재되어 있던 내용을 명시적으로 나타내도록 표현된다. 추상구문트리는 다시 DAG(Directed Acyclic Graph)로 변환되면서 공통 부표현(common subexpression)의 반복을 제거한다. DAG는 코드 생성기로 넘겨지고 코드 생성기는 특정 머신의 어셈블리 코드를 출력한다.

lcc는 기계독립적인 구조로 DAG를 출력하는 전단부와 코드생성을 하는 후단부로 나눌 수 있다. 따라서 특정 머신에 대한 코드를 생성하려면 코드 생성기의 기계어 정보 및 관련 프로시져들을 변환하면 된다.

lcc로 실행해보면 빠르다는 것을 느낄 수 있는데, lcc의 프로그램 크기는 다른 컴파일러에 비해 상당히 작은 편이다. 따라서 타겟 머신이 변경될 경우 lcc 후단부의 기계어 부분만을 개조하면 되므로 이식성이 높은 편이다.

한편, lcc는 최적화에 대한 고려는 많이 하고 있지는 않다. lcc에 적용되는 공통 부표현식 제거와 같은 최적화의 범위는 DAG로 한정되어 있다. 레지스터 할당의 경우 코드 생성시에 지엽적인 정보에 따라 처리되고 있다.

본 논문의 실험에서는 lcc에서 최적화 및 레지스터 할당으로 생성된 코드와 Chaitin의 그래프 컬러링 알고리즘을 적용하여 생성된 코드에 대한 레지스터 할당 결과를 비교하였다.

III. 그래프 컬러링

레지스터 할당 알고리즘 수행전에, 프로그램의 각 변수들은 가상의 레지스터(virtual register)로

표현된다. 가상의 레지스터 개수에는 제한을 두지 않는다. 그러나 레지스터 할당이 가상의 레지스터에 직접 적용되는 것은 아니며, 프로시져 단위로 라이브 범위에 적용된다. 그래서 하나의 가상 레지스터가 프로그램 전체에서 보면 여러 개의 값을 가질 수도 있으므로 분리된 라이브 범위로 나누어서 고려해야 한다.

레지스터 할당을 그래프 컬러링 문제로 모델하기 위해서는, 컴파일러가 처음에 간섭 그래프 IG를 구성해야 한다. IG에 있는 노드들은 라이브 범위에 대응되고 에지는 간섭을 나타낸다. 그러므로 라이브 범위 l_i가 l_j와 간섭 관계에 있으면, 노드 i에서 노드 j까지 에지가 연결된다. 에지가 연결된 두 노드는 동시에 라이브하므로 같은 레지스터에 할당될 수 없다. 간섭그래프에서 l_i와 간섭 관계에 있는 라이브 범위를 l_i의 이웃이라고 하며, 이웃의 수를 l_i의 차수(degree)라고 한다.

컴파일러의 레지스터 할당부는 간섭 그래프 IG의 N-컬러링을 구한다. 즉, 인접한 노드는 다른 컬러를 가지도록 IG의 노드에 N개의 컬러를 배정하자는 것이다. 그래프 컬러링은 NP-complete 하므로 컴파일러는 컬러링에 휴리스틱을 적용한다. N-컬러링이 발견되지 않을 경우, 그래프의 노드 중 하나 이상을 메모리에 대피시켜야 한다.

라이브 범위를 대피시키게 되면, 간섭 그래프에서 대피된 노드와 그 노드에 관련된 에지를 제거하여 새로운 간섭 그래프를 만든다. 이 과정은 N-컬러링이 구해질 때까지 반복 수행된다.

Chaitin의 그래프 컬러링은 그림 1과 같이 구성된다.

· Renumber: 라이브 범위 넘버링. 각 정의 위치에서 새로운 라이브 범위를 생성한다. 각 사용 위치에서 사용에 도달한 라이브 범위를 서로 합집합한다.

· Build: 간섭그래프 생성. 간섭그래프는 비트 매트릭스와 인접리스트의 집합으로 구성된다.

· Coalesce: 라이브 범위 결합. 두 라이브 범위 l_i와 l_j는 l_j의 초기 정의가 l_i와 같고 서로 간섭하지 않으면 결합이 가능하다. 두 라이브 범위의 결합은 복사(copy) 명령어를 제거하는 효과가 있다. 결합으로 인해 그래프가 변경되면 Build로 간다.

· Spill costs: 대피 비용 산출. 각 라이브 범위에 대해 대피시 추가 비용을 계산한다. 이 비용은 해당 라이브 범위를 대피하는데 요구되는 로드 및 스토어 수를 측정하고 가중치를 부여해서 얻을 수 있다.

· Simplify: 간단화. 컬러링할 노드의 순서를 정한다. 비어 있는 스택에 간섭그래프의 노드를 집어넣기위해 그래프에 하나의 노드도 남지않을 때까지 다음 스텝을 반복한다:

(a) 노드의 차수가 N보다 작은 노드가 있으면 이 노드 및 관련 에지를 그래프에서 제거하고 스택에 집어넣는다.

(b) 그렇지 않으면, 대피 노드를 선택한다. 선택된 대피노드 및 관련 에지를 그래프에서 제거하고 대피될 노드에 대피 표시를 한다.

· Spill code: 대피 코드 삽입. 대피된 라이브 범위에 대해 사용전에 로드 명령어와 정의 후에 스토어 명령어를 삽입한다.

· Select: Simplify에서 정한 순서대로 컬러를 부여하는 과정이다. 스택이 빌 때까지 다음 스텝을 반복한다:

(a) 스택의 현재 위치에 있는 라이브 범위를 꺼낸다.

(b) 이 라이브 범위를 간섭그래프에 삽입한다.

(c) 이웃과 구별되는 컬러를 부여한다.

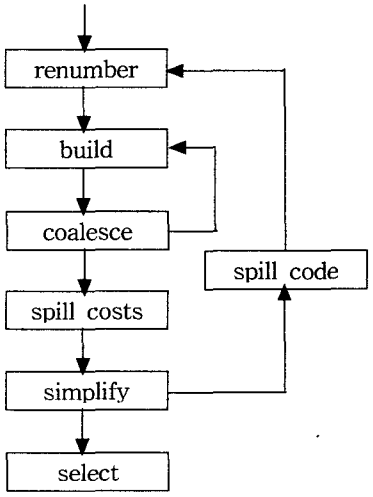

그림 1 Chaitin의 그래프 컬러링 과정
Fig. 1 Chaitin's Graph Coloring Phases

IV. 실험

그래프 컬러링 알고리즘의 할당 결과를 얻기 위해 3 가지 종류의 컴파일러를 사용하고, 벤치마크 프로그램으로는 SPEC92INT를 활용하였다. 벤치마크 프로그램의 프리프로세싱을 위해서

Digital Unix에 탑재된 C 컴파일러의 프리프로세서를 활용하였고, 파싱 프로그램으로는 레지스터 할당을 위해 개조된 lcc 컴파일러를 사용하였다. lcc의 전단부에서 출력된 DAG를 전문적인 최적화프로그램인 카네기 멜론 대학의 cmcc에 입력되도록 하였다. 플랫폼으로는 DEC ALPHA 255/300을 사용하였으며, 레지스터 할당 후의 출력은 mips 어셈블리 코드를 생성하도록 하였다. 표 1은 lcc의 레지스터 할당 결과와 본 논문에서 구성한 시스템에서 출력된 Chaitin의 레지스터 할당 결과를 나타낸 것이다.

표1의 *osrand* 함수 할당 결과에 '+'로 표시한 것은 정수 레지스터와 부동점 소수 레지스터를 구분하기 위한 것이다. 각 함수는 espresso, li, eqntott 중 일부 파일에 속해 있다. 즉 espresso의 cols.c, verify.c와 li의 unixstuff.c, xlsym.c, 또한 eqntott의 yystuff.c, bnode.c 등에 있는 함수들에 대한 레지스터 할당 결과를 나타낸 것이다. unix_stuff.c 중 일부 함수는 규모가 너무 작아서 생략하였으며, 몇몇 함수에 대한 결과는 지면 관계상 나타내지 않았다.

표1. 레지스터 할당 결과

Table 1. Result of the Register Allocation

함수명	lcc	N-컬러링	개선비율(%)
yylex	15	11	26.7
get_free	10	8	20
copy_bnode	13	10	23.1
xlsenter	7	5	28.6
xlmakesym	12	7	41.7
xlblind	10	8	20
xlxgetvalue	9	8	11.1
xlsetvalue	11	9	18.2
xlgetprop	11	7	36.4
xlremprop	8	7	12.5
findprop	9	6	33.3
osrand	5+3	5+2	12.5
osgetc	10	5	50
osputc	9	7	22.2
verify	16	14	12.5
PLA_permute	18	15	16.7
sm_col_alloc	6	5	16.7
sm_col_insert	12	9	25

또한, 개선 비율은 lcc에 대한 N-컬러링의 레지스터 절감 비율을 %로 나타낸 것이지만, N-컬러링이전에 lcc 보다 더 많은 최적화가 수행되었음을 감안할 필요가 있다.

V. 결론

본 논문에서는 다양한 레지스터 할당 알고리즘을 분석하기 위한 컴파일러 시스템을 구축하고자 하였다. 시스템 환경은 Unix 플랫폼으로 DEC ALPHA 255/300을 호스트로 하고, Pentium II PC로는 프로그램 분석을 할 수 있도록 하였다. C 컴파일러의 전단부로서 개조된 lcc를 활용하였고, 최적화부로는 카네기 멜론 대학의 cmcc를 사용하였다.

본 연구 결과로 다양한 레지스터 할당 실험이 가능하게 되었으며 아울러 컴파일러의 최적화 알고리즘과 코드 스케쥴링에도 활용이 가능할 것으로 기대된다.

참고문헌

[1] G. J. Chaitin, M. A. Auslander, A. K. Chandra, J. Cocke, M. E. Hopkins, and P. W. Markstein. "Register allocation via coloring", Computer Languages, pp. 47-57, 1981. 1

[2] P. Briggs, K. D. Cooper, K. Kennedy, and L. Torczon, "Coloring heuristics for register allocation", In Proc. ACM SIGPLAN '89 Conf. on Prog. Language Design and Implementation", pp. 275-284, 1989. 7

[3] F. C. Chow and J. L. Hennessy, "A priority-based coloring approach to register allocation", ACM Trans. on Prog. Lang. Syst., pp. 501-535, 1990. 10

[4] S. M. Kurlander and C. N. Fischer, "Zero-cost range splitting", In Proc. ACM SIGPLAN Symp. on Programming Language Design and Implementation, pp. 45-58. ACM, 1991. 6

[5] Guei-Yuan Lueh, " Issues in Register Allocation by Graph Coloring", CMU-CS-96-171, 1996. 11

[6] Christopher Fraser and David Hanson, "A Retargetable C Compiler: Design and Implementation", Benjamin/Cummings, 1995