논문 2008-45CI-1-4

DSP용 코드 생성에서 주소 포인터 할당 성능 향상 기법

(Improvement of Address Pointer Assignment in DSP Code Generation)

이 희 진*, 이 종 열*

(Heejin Lee and Jong-Yeol Lee)

요 약

DSP에서 제공되는 주소 생성 유닛은 데이터 패스와 병렬적으로 주소 연산을 수행할 수 있게 해 줌으로써, DSP 코드 생성에 중요한 역할을 한다. 프로그램 변수들의 메모리 레이아웃을 결정하는 문제는 주소 생성 유닛의 기능을 이용하여 주소 연산용 명령어를 줄이는 최적화이다. 메모리 레이아웃 생성 단계와 주소 포인터 할당 단계로 구분 되는 이 최적화에서 본 논문은 주소 연산 코드의 수가 최소가 되도록 DSP용 코드 생성의 효과적인 주소 포인터 할당 문제를 다룬다. 제안하는 알고리즘은 고정된 메모리 레이아웃을 가질 때 주소 포인터 할당을 수행하는 기존의 알고리즘의 시간 복잡도를 줄이는 기법이다. 메모리 크기와 수행 시간을 줄이기 위해 알고리즘을 수행할 때 핵심적인 요소들만을 고려하도록 강한 가지치기 방법을 사용하였다. 또한 주소 포인터 할당 문제는 메모리 레이아웃에 영향을 크게 받는 문제이기 때문에 본 논문은 주어진 메모리 레이아웃을 갱신하여 반복적으로 성능을 개선하는 방법을 제안한다. 약 3,000여개의 실제 프로그램으로부터 얻은 변수 접근 시퀀스를 제공하는 OffsetStone 벤치마크를 이용한 실험결과를 통해 본 논문에서 제안한 기법과 알고리즘을 테스트 했다. 제안한 방법은 전통적인 방법보다 평균 25.9%의 적은 주소 코드를 생성해 냄을 보인다.

Abstract

Exploitation of address generation units which are typically provided in DSPs plays an important role in DSP code generation since that perform fast address computation in parallel to the central data path. Offset assignment is optimization of memory layout for program variables by taking advantage of the capabilities of address generation units, consists of memory layout generation and address pointer assignment steps. In this paper, we propose an effective address pointer assignment method to minimize the number of address calculation instructions in DSP code generation. The proposed approach reduces the time complexity of a conventional address pointer assignment algorithm with fixed memory layouts by using minimum cost-nodes breaking. In order to contract memory size and processing time, we employ a powerful pruning technique. Moreover our proposed approach improves the initial solution iteratively by changing the memory layout for each iteration because the memory layout affects the result of the address pointer assignment algorithm. We applied the proposed approach to about 3,000 sequences of the OffsetStone benchmarks to demonstrate the effectiveness of the our approach. Experimental results with benchmarks show an average improvement of 25.9% in the address codes over previous works.

Keywords : 주소 생성 (code generation), 디지털 시그널 프로세서 (digital signal processors), 주소 할당 (offset assignment), 주소 포인터 할당 (address pointer assignment)

I. 서 론

많은 내장형 시스템 아키텍처들에서는 더욱 강력한 처리 능력과 여러 가지 기능들을 가지도록 하기 위해서 마이크로프로세서뿐 아니라 DSP도 함께 탑재하고 있다. 그러나 기존의 컴파일러들은 DSP에서 제공하는 전용의 하드웨어들의 적극적인 활용이 어렵기 때문에 이러한 아키텍처에서 비효율적이다. 또한 DSP는 종종 제한된 프로그램 메모리의 크기와 실시간 수행에 따른 제약들을 요구하여 내장형 시스템 디자인 분야에서 코드 최적화가 중요한 문제이다.

프로그램 변수의 주소를 할당하는 문제 (offset assignment problem), 즉 변수들의 메모리 레이아웃에

* 학생회원, ** 평생회원, 전북대학교 전자정보공학부
(Division of Electronics and Information Engineering, Chonbuk National University)
접수일자: 2007년11월13일, 수정완료일: 2008년1월11일

대한 최적화는 코드 생성의 중요한 부분이다. 이는 주소 코드가 전체 프로그램 비트의 50%가 넘는 양을 차지할 수 있고, 전통적인 마이크로프로세서에서 6개의 명령어마다 1개의 주소 코드가 나오기 때문이다[1]. 따라서 주소 할당 최적화는 코드 크기와 프로그램 실행 시간을 줄이는 데 중요한 영역이다.

TI의 TMS320C2x/5x/6x 시리즈와 같은 많은 DSP 아키텍처들은 주소 생성을 위한 전용 하드웨어인 주소 생성 유닛 (AGU) 을 제공한다. AGU는 다음 주소를 연산을 수행하는데 중앙의 데이터 연산과 병렬적으로 빠르게 연산할 수 있도록 독립된 덧셈/뺄셈기, 레지스터 파일들을 가진다. 내장형 시스템은 자동 증감 및 자동 수정 주소 연산 방식을 갖는 간접 주소 방식을 제공한다. 자동 증감 방식은 주소 레지스터를 사용하여, 자동 수정 방식은 수정 레지스터를 사용하는 간접 주소 방식을 말한다. 데이터 패스 자원을 사용하지 않는 AGU에서의 주소 생성으로 인하여 변수들의 접근이 많은 자동 증감, 자동 수정 주소 방식을 사용하도록 할 때 높은 명령어 수준의 병렬 수행이 이루어질 수 있다. 그러나 이러한 주소 방식들의 높은 활용은 메모리의 변수의 위치들을 매우 주의 깊게 배치해야 함을 요구한다. 따라서 컴파일러는 간결한 주소 코드를 생산해 내기 위해 자동 증감, 자동 수정 주소 방식의 사용을 최대화 시키도록 메모리에서 관련 있는 변수들의 위치를 주의 깊게 결정해야만 한다.

AGU의 하나의 주소 레지스터를 이용하는 주소 할당 문제 (simple offset assignment problem, SOA) 는 Bartley에 의해 처음 연구되었다[2]. 그는 NP-complete인 최대 가중 해밀턴 경로 커버 (maximum-weighted Hamiltonian path cover, MWPC)으로 문제를 모델링했고 문제 해결을 위해 휴리스틱 알고리즘을 제안했다. 또한 Liao et al.은 Kruskal의 최대 신장 트리 알고리즘에 기반한 알고리즘을 제안하여 SOA 해결하는 휴리스틱을 제안하였고[3], AGU에서 제공하는 여러 개의 주소 레지스터를 이용하는 방법 (general offset assignment problem, GOA) 으로 문제를 확장하였다. Leupers와 Marwedel은SOA/GOA 해법의 질을 향상시키도록 tie-breaking 휴리스틱과 변수 배분 방법을 제안하여 Bartley와 Liao et al.의 연구를 확장하였다[4]. Leupers와 David는 임의적인 레지스터 파일 크기와 자동 증가 영역을 가정하여 GOA 문제를 해결하였는데, 그들의 방식은 유전자 알고리즘에 기반한 방식이었다[5]. Sudarsanam et al.은 $-l$부터 $+l$까지의 가변적인 범위의 자동증감을 고려하였고[6], Gebotys는 고정된 메모리 레이아웃이 주어진 코드에서 모든 변수 접근들을 주소 레지스터에 할당하는 문제를 네트워크 플로우 문제로 모델링하여 최적화를 시도했다[7]. Atri et al.은 초기 SOA 결과와 이로부터 다른 휴리스틱 알고리즘을 적용한 결과를 비교하며 점진적으로 성능을 향상 시키는 방법을 제안했으며[8], Ottoni et al.은 변수들의 메모리 배치를 합병하여 메모리 사용 영역을 줄이고 그래프를 단순화 시켜 최적화를 시도하는 SOA 알고리즘을 제안했다[9]. Rao와 Pande는 교환 법칙이 성립되는 입력에 대하여 변수 접근의 재배치로 시퀀스의 최적화를 연구했다[10]. Lim et al.은 하이브리드 알고리즘을 이용하여 코드 스케쥴링과 SOA 문제를 결합하는 시도를 하였고[11], Choi와 Kim은 코드 스케쥴링을 포함하는 Schedule-driven SOA/GOA 알고리즘을 제안했다[12]. Xue et al.은 여러 기능 유닛 (functional units)이 제공되는 구조에서 주소 할당과 스케쥴링 알고리즘을 제안했다[13]. 또한 Wess와 Zeitlhofer는 최적화 된 메모리 레이아웃이 존재하는 것을 가정하고 주소 포인터 할당 문제로 GOA 문제를 모델링하여 해법을 찾도록 새로운 접근방법을 제안하였다[14].

이전의 대부분의 변수의 주소를 계산하는 용도로만 쓰이는 명령어의 수를 줄이도록 메모리 레이아웃을 결정하는 방식이나, Wess와 Zeitlhofer가 제안한 방식은 주소 할당에서 서로 독립적으로 수행될 수 있는 단계들을 메모리 레이아웃 생성과 주소 포인터 할당 단계로 나누어 최적화를 시도했다. 그러나 그들이 제안한 알고리즘에는 주소 포인터 할당 알고리즘 수행 전 미리 최적화된 메모리 레이아웃을 가졌다는 단점이 존재한다.

본 논문은 주소 연산 명령어의 수를 줄임으로 전체 코드 크기를 줄여서 생성된 코드의 실행 성능이 향상되도록 주소 포인터 할당 문제를 효과적으로 수행하는 방법을 다룬다. 이를 위해 앞에서 언급한 Wess와 Zeitlhofer가 제안한 알고리즘을 기반으로 효과적인 2단계 알고리즘을 제안한다. 첫 번째 단계에서는 적절한 메모리 레이아웃을 결정하고 수행 속도를 향상 시키는 기법 사용한 주소 포인터 할당 알고리즘을 통하여 적절한 해결책을 구한다. 두 번째 단계에서는 첫 번째 단계에서 구해진 처음의 해결책을 반복적으로 개선시킨다.

본 논문의 Ⅱ장에서는 주소 포인터 할당 알고리즘을 사용한 주소 할당 문제에 대해 알아보며, Ⅲ장에서는 기존의 주소 포인터 할당 알고리즘의 수행 속도를 빠르게 하는 기법과 초기의 성능을 더욱 개선하는 반복 기

법을 제안한다. Ⅳ장에서는 제안한 알고리즘의 효과를 보여주기 위해 벤치마크 프로그램의 실험 결과를 보여주며, 마지막으로 Ⅴ장에서는 본 논문의 결론을 맺는다.

Ⅱ. 주소 할당 문제

1장에서 언급한 여러 개의 주소 레지스터를 이용하는 주소 할당 문제는 프로그램 변수들의 메모리상의 위치를 결정하는 메모리 레이아웃 생성 단계와 생성된 메모리 레이아웃을 사용하여 각 주소 레지스터의 포인터들을 할당하는 주소 포인터 할당 단계로 나눌 수 있다.

1. 메모리 레이아웃 생성

메모리 레이아웃은 주소 할당 문제에서 매우 중요한 요소이다. 앞에서 언급한 것처럼 메모리 레이아웃에 따라서 주소 코드가 많아질 수도 있고 적어질 수도 있기 때문이다. 전통적인 컴파일러들은 변수가 사용된 순서대로 변수들의 메모리 레이아웃을 결정한다. (order of first used, OFU) 이는 메모리 레이아웃을 결정하는데 추가적인 시간을 전혀 소모하지 않는다.

SOA 문제의 많은 연구에서는 그래프 이론에 기반하여 메모리 레이아웃을 결정한다. 그림 1은 순차적으로 결정한 메모리 레이아웃과 그래프 이론에 기반한 메모리 레이아웃을 비교하여 보여준다.

그림 1의 (a)와 같은 변수들의 접근이 있다고 할 때 (b)는 OFU 방식으로 결정된 메모라 레이아웃을 나타내며, (c)는 변수 접근을 무방향 그래프로 나타낸 그림이고 (d)는 Leupers와 Marwedel이 제안한 tie-breaking 휴리스틱[4]을 적용하여 결정한 메모리 레이아웃을 표시한다.

또한 SOA의 확장으로 GOA 문제를 접근하는 이전의 연구 방법들[3~5]은 각 변수들을 사용 가능한 주소 레지스터의 개수만큼 나누어서 각 주소 레지스터에 대하여

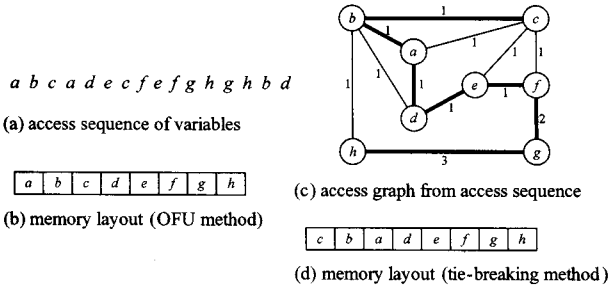


그림 1. 메모리 레이아웃 생성의 예제
Fig. 1. An example of memory layout generation.

SOA 방식처럼 메모리 레이아웃을 결정한다. 그래프 이론에 입각한 SOA의 메모리 레이아웃 결정 방식은 많은 시간을 필요로 하지 않지만, 휴리스틱하게 접근하는 GOA의 메모리 레이아웃 결정 방식은 사용 가능한 주소 레지스터의 개수와 변수의 개수가 많아질수록 많은 시간을 요구한다.

2. 주소 포인터 할당

메모리 레이아웃이 결정되어 있다면 각 주소 레지스터의 포인터들을 어떤 변수 접근에 할당 해 줄 것인지만 결정하면 되는데, 이것을 주소 포인터 할당 (address pointer assignment, APA) 이라 부르며 Wess와 Zeitlhofer에 의해 연구되었다. 그들은 이전 단계에서 최적화된 메모리 레이아웃을 결정하였다고 가정하고 주소 포인터의 갱신 비용 함수를 최소화하는 APA 알고리즘을 제안하였다. 비용 함수는 다음과 같다.

$$C = I_K + (K+1)C_K \tag{1}$$

식 (1)에서 I_K는 주소 포인터 초기화의 수, K는 사용 가능한 주소 포인터의 수, C_K는 주소 포인터가 갱신된 횟수를 나타낸다. 식 (1)을 가지고 수행하는 APA 알고리즘은 변수 접근 시퀀스와 메모리 레이아웃을 입력으로 받으며, 할당 트리를 만들며 APA를 수행한다.

그림 2는 메모리 레이아웃과 변수 접근 시퀀스가 각각 그림 1(a), 1(d)와 같으며 사용 가능한 주소 레지스터의 개수가 세 개일 때 APA 알고리즘의 할당 트리 구축 과정의 예를 보여준다. 0-비용 오프셋 영역은 ±1로 가정되었다.

할당 트리의 같은 레벨에서의 모든 노드들은 접근 시퀀스의 같은 위치에서 각각 다른 주소 포인터 세팅과 매칭된다. 즉, 그림 2의 (i) 노드에서 다음 접근 시퀀스 c를 가리키는데 주소 포인터 AP1, AP2, AP3를 사용할

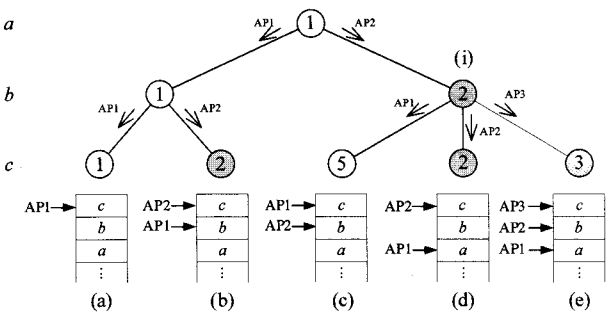


그림 2. 할당 트리 구축 과정 예제
Fig. 2. An example of the pruning of assignment tree construction.

수 있다. 또한 트리의 루트 노드에서 단말 노드까지의 각 패스는 주소 포인터들의 순차적인 메모리 접근과 매칭된다. 포인터의 초기화 비용은 그림 2의 루트 노드에서 보이는 것 같이 1이며, 포인터 할당은 각 노드의 색과 노드 안에 있는 숫자 (식 (1)에 의해 얻은 비용 *C*)로 표시된다. 그림 2의 루트 노드는 첫 번째 주소 포인터 (AP1) 가 할당되어 흰색, *C=1+(3+1)0=1*의 비용을 갖고, (i) 노드는 두 번째 주소 포인터 (AP2) 가 할당되어 회색을 나타내고 *C=2+(3+1)0=2*의 비용을 갖는다. 그리고 변수 접근 시퀀스의 시작 원소 (여기서는 *a*) 는 임의로 포인터 할당을 한다. 균질 주소 레지스터 파일 (homogeneous address register file) 의 조건에서는 첫 번째 주소 포인터를 할당한다.

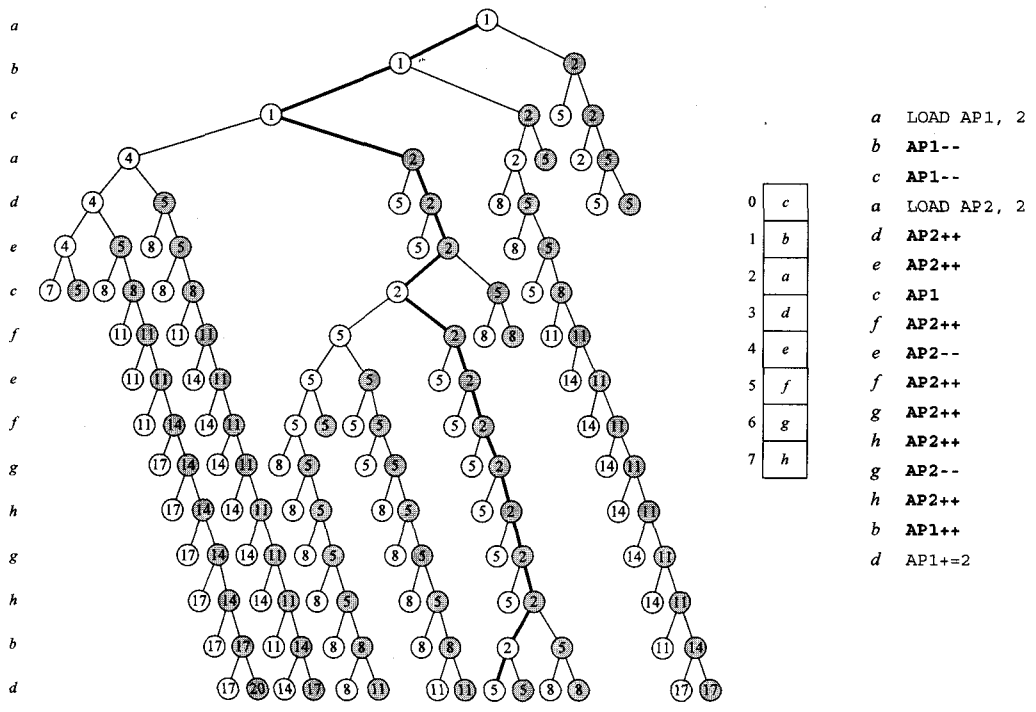
Wess와 Zeitlhofer는 균질 주소 레지스터 파일에서는 그림 2의 (b)와 (c)가 동일한 포인터 세팅을 갖는 것을 발견하였다. 연산 복잡도를 줄이기 위해 동일한 포인터 세팅을 갖는 노드들의 그룹에서는 할당 트리 구축을 진행해 나갈 때 단 하나의 적은 비용을 갖는 노드만을 유지하도록 하며, 동일한 비용을 같은 노드들은 임의적으로 선택하게 하였다 (arbitrary equivalent tie-nodes breaking). 그들은 할당 트리 구축을 진행하는 APA 알고리즘에 arbitrary equivalent tie-nodes breaking을 적용하여 수행 시간을 줄였다.

그림 3은 그림 1(a)의 접근 시퀀스와 그림 1(d)의 메모리 레이아웃, 두 개의 주소 레지스터를 가지고 Wess와 Zeitlhofer의 APA 알고리즘을 통해 완성된 APA 트리와 APA 알고리즘을 통해 생성된 주소 코드를 보인다. 그림 3(a)의 굵은 선이 APA 트리에서 최적의 결과를 갖는 패스이다. 최적의 결과를 토대로 생성된 주소 연산용 코드를 그림 3(b)에서 보인다. 그림에서 보이듯이 APA 알고리즘을 수행하면 변수들의 열 일곱 번의 메모리 접근을 처리하기 위해 세 개의 주소 연산용 코드 (주소 포인터 초기화 코드 두 개와 주소 포인터 갱신 코드 한 개) 만이 필요함을 알 수 있다.

III. 제안하는 알고리즘

1. 최소 비용 노드 분할

2.2절에서 살펴본 것처럼 Wess와 Zeitlhofer는 균질 주소 레지스터 파일 상에서 arbitrary equivalent tie-nodes breaking을 사용하여 할당 트리를 구축하는



(a) complete assignment tree by equivalent tie -nodes breaking (b) address assignment

그림 3. equivalent tie-nodes breaking을 사용하여 완성된 APA 트리와 생성된 주소 코드

Fig. 3. The complete assignment tree using equivalent tie-nodes breaking and the generated address codes.

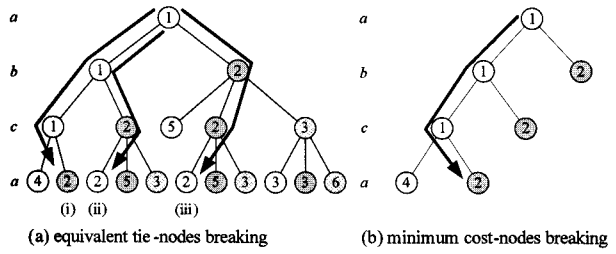


그림 4. 기존 Wess와 Zeitlhofer의 breaking과 제안하는 breaking의 차이 비교

Fig. 4. The comparison of the equivalent tie-nodes breaking of Wess and Zeitlhofer and minimum cost-nodes breaking method of our approach.

데 소요되는 연산의 복잡도를 줄여서 APA 알고리즘을 수행하였다. 그러나 우리는 그들의 연산 복잡도를 더 줄이는 최소 비용 노드 분열 (minimum cost-nodes breaking) 을 제안한다.

그림 4는 Wess와 Zeitlhofer의 equivalent tie-nodes breaking과 우리가 제안하는 minimum cost-nodes breaking의 비교를 위해 그림 2에서 보인 APA 트리 구축의 확장을 보인다. 사용 가능한 주소 레지스터의 개수는 세 개이며, 메모리 레이아웃과 변수 접근 또한 그림 2에서 사용한 것과 같다. 그림 4은 그림 2보다 한 단계 더 트리의 구축을 진행한 결과를 보인다. 그림 4(a)의 (i)과 (ii), (iii)을 보면 각 노드에서의 주소 포인터들의 세팅은 다를 수 있지만 같은 비용 함수 값을 갖는다. 루트 노드에서부터 단말 노드까지의 패스가 달라서 주소 포인터의 초기화와 갱신이 일정하게 결정되지 않더라도 할당 트리 구축의 최종 목적은 비용 함수가 최소가 되도록 하는 것이기 때문에 본 논문에서는 트리의 레벨에서 항상 최소의 비용을 갖는 노드만이 트리 구축을 계속할 수 있도록 하는 minimum cost-nodes breaking을 제안한다. 그림 4(b)는 그림 4(a)와 같은 조건에서 항상 최소의 비용을 갖는 노드만이 트리 구축 진행을 계속하도록 본 논문에서 제안하는 breaking을 사용했을 때 생성된 APA 트리의 노드를 보여준다.

그림 5는 minimum cost-nodes breaking을 사용하여 구축된 전체 APA 트리를 보여준다. 사용 가능한 주소 레지스터의 개수는 두 개로 설정하였으며, 메모리 레이아웃과 변수 접근 시퀀스는 그림 2와 동일한 조건으로 알고리즘을 수행할 때 APA 트리의 전체 모습을 보여준다. 그림 3(a)와 그림 5의 APA 트리의 생성된 노드 개수 차이를 보아 알 수 있듯이 제안하는 minimum cost-nodes breaking은 equivalent tie-nodes breaking에 비해 APA 알고리즘의 연산 복잡도를 줄인다.

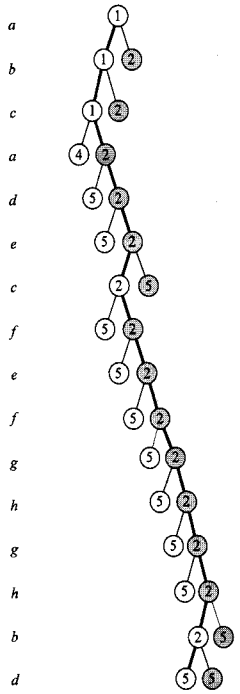


그림 5. minimum cost-nodes breaking을 사용하여 완성된 APA 트리

Fig. 5. The complete assignment tree using minimum cost-nodes breaking.

우리는 실험을 통해서 equivalent tie-nodes breaking을 사용한 APA 알고리즘과 우리의 minimum cost-nodes breaking을 사용한 APA 알고리즘이 같은 결과 즉, 같은 주소 코드를 생성하며 적은 시간이 필요함을 확인하였다.

본 논문에서 제안하는 minimum cost-nodes breaking을 사용한 APA 알고리즘의 시간 복잡도 계산을 위해 변수 접근 시퀀스의 길이를 n이라고 하고 사용 가능한 주소 레지스터의 개수를 K라고 하자. 먼저 아무런 최적화가 이루어지지 않는 simple exhaustive tree의 경우 APA 트리의 각 레벨에서 생성 가능한 노드의 개수는 K^{n-1}이며 따라서 할당 트리 전체 노드의 개수는 K^n-1이다. 따라서 simple exhaustive tree의 알고리즘은 지수적으로 증가하는 복잡도를 갖는다. equivalent tie-nodes breaking을 사용하는 APA 알고리즘은 트리의 각 레벨에서 K^{n-1}의 노드를 생성하며 각 노드마다 K개의 가능성을 테스트 하므로 $O(nK^n)$의 복잡도를 갖는다. 그러나 우리가 제안하는 minimum cost-nodes breaking을 사용한 APA 알고리즘은 항상 최소의 비용을 갖는 노드만 남기고 그렇지 않은 노드들은 삭제하기 때문에 worst case의 경우에 트리의 한 레벨에서 n^K의 노드 개수를 갖는다. 따라서 우리가 제안하는 minimum

cost-nodes breaking을 사용한 APA 알고리즘은 $O(n \cdot n^K)$로 $O(n^K)$의 시간 복잡도를 갖는다.

2. 반복 기법을 통한 성능 향상

2.1절에서 언급했듯이 APA를 수행하기 전 단계에서 결정한 메모리 레이아웃은 APA에 큰 영향을 미치는 중요한 요소이다. 메모리 레이아웃에 매우 의존적인 APA의 결과를 메모리 레이아웃의 개선을 통해 더욱 향상시키도록 반복하는 기법을 제안한다.

반복 기법에 대한 아이디어는 다음과 같다. 그림 1의 (c)와 (d)처럼 변수 접근 시퀀스로 그린 접근 그래프로부터 얻은 메모리 레이아웃은 SOA 문제에서 사용하는 방식으로 주소 레지스터가 한 개인 경우에 해당한다. 그렇기 때문에 초기 APA 알고리즘을 통해서 각 주소 레지스터에 할당된 서브 시퀀스들로 다시 접근 그래프를 그리는 방법은 여러 주소 레지스터가 존재하는 문제에 대하여 최적화의 가능성을 갖는다. 각 주소 레지스터들에 할당된 서브 시퀀스로부터 접근 그래프를 그리게 되면 초기의 접근 그래프에서 상대적으로 높은 무게(weight) 를 갖는 에지는 남고, 낮은 무게를 갖는 에지는 사라지는 경향이 나타난다. 따라서 메모리에서 인접하도록 위치해야 하는 변수들이 선택될 확률이 높아진다. 따라서 반복 기법은 더 좋은 성능을 가져온다.

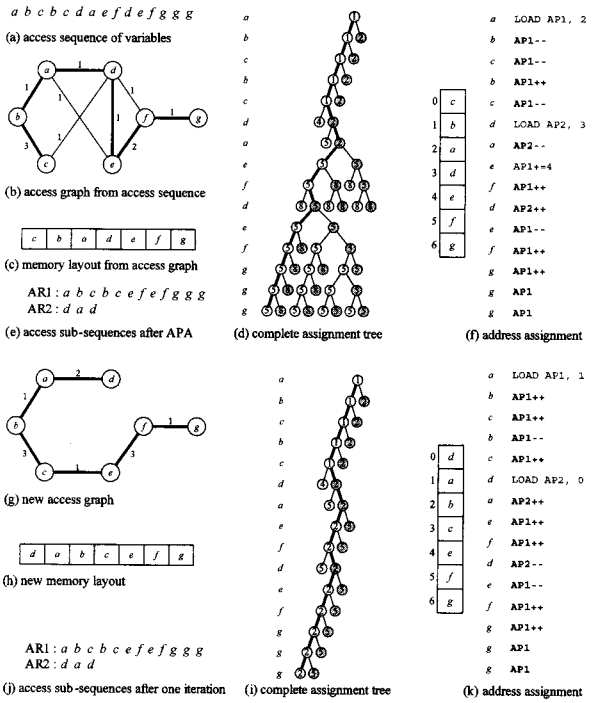


그림 6. 반복을 통한 성능 향상의 예제

Fig. 6. An example of the improving of code performance for address pointer assignment.

그림 6은 두 개의 주소 레지스터를 가지는 AGU를 사용한 반복 기법의 예제를 보여준다. 그림 6(a)는 초기 변수 접근 시퀀스이며, 그림 6(b)는 변수 접근 시퀀스로부터 그린 접근 그래프, 그림 6(c)는 이 접근 그래프에 tie-breaking 휴리스틱을 적용하여 도출된 메모리 레이아웃을 보여준다. 그림 6(c)의 메모리 레이아웃을 기반으로 본 논문에서 제안한 minimum cost-nodes breaking을 사용한 주소 포인터 할당 알고리즘의 결과인 APA 트리, 접근 서브 시퀀스, 주소 할당을 각각 그림 6(d), (e), (f)에서 보여준다. 초기 솔루션으로부터 두 번의 주소 포인터 초기화 코드와 한 번의 주소 포인터 갱신 코드가 필요함을 알 수 있다.

반복을 위해 그림 6(e)의 접근 서브 시퀀스로 새롭게 그린 접근 그래프를 그림 6(g)에서 보여준다. 위에서 언급했듯이 그림 6(e)의 접근 서브 시퀀스로 그린 접근 그래프는 그림 6(b)의 초기 접근 그래프보다 에지의 개수가 줄어들었다. 그림 6(e)의 접근 그래프로부터 얻은 새로운 메모리 레이아웃을 그림 6(h)에서 보여주고, 그림 6(i), (j), (k)에서 그림 6(h)의 메모리 레이아웃으로 반복 수행되는 APA 알고리즘의 결과를 보여준다. 그림 6(f)와 (k)를 비교하면 알 수 있듯이, 반복 전엔 1번의 주소 포인터 갱신 코드가 필요하지만, 반복으로 바뀐 메모리 레이아웃을 이용하면 추가적인 주소 포인터 갱신 코드가 필요하지 않음을 알 수 있고, 프로그램의 수행 속도와 자원이 절약될 것을 예상할 수 있다.

반복 기법에서는 종료 조건이 중요한 역할을 한다. 제안하는 반복 기법은 접근 그래프에서 높은 무게의 에지들만 남기 때문에 메모리 레이아웃을 결정하기 위해 그리는 반복할 때마다 접근 그래프가 단순해져 간다. 따라서 반복할 때마다 메모리 레이아웃이 수렴하는 현상이 나타나기 때문에 제안하는 반복 기법은 무한 반복이 일어나지 않는다.

IV. 실험 결과

우리가 제안하는 mininum cost-nodes breaking을 적용한 APA 알고리즘을 평가하기 위하여, 본 논문에서 제안한 알고리즘은 C 언어를 이용하여 구현하였고, Intel Pentium 4 프로세서 3.0GHz, 1GB RAM의 Linux 머신에서 실행되었다. 또한 OffsetStone 벤치마크에서 제공하는 3,406개의 시퀀스로 테스트하였다[15]. 벤치마크에서 많이 사용되는 응용 프로그램들 중 적은 시퀀스 길이를 갖는 프로그램과 긴 시퀀스 길이를 갖는 프로그

표 1. 적은 시퀀스 길이의 APA 알고리즘 비교
Table 1. The comparison of the breaking techniques with small access sequence length.

Bench. (|V|/|S|)	K	eq-APA		eq-APA (W=100)		min-APA	
		# a/i[1]	time[2]	# a/i	time	# a/i	time
complex (6/12)	2	2	< 1	2	< 1	2	< 1
	3	2	< 1	2	10	2	< 1
	4	2	60	3	30	2	< 1
	6	2	1220	3	70	2	< 1
	8	2	4760	3	100	2	< 1
fir (8/22)	2	4	< 1	4	< 1	4	< 1
	3	3	60	4	40	3	< 1
	4	3	1720	6	120	3	< 1
	6	3	140220	6	240	3	< 1
	8	3	48 m[3]	6	370	3	< 1
biquad (12/25)	2	6	< 1	6	< 1	6	< 1
	3	4	660	9	60	4	< 1
	4	4	38220	10	140	4	< 1
	6	-[4]	-	10	230	4	< 1
	8	-	-	10	350	4	< 1

표 2. 큰 시퀀스 길이의 APA 알고리즘 비교
Table 2. The comparison of the breaking techniques with large access sequence length.

Bench. (|V|/|S|)	K	eq-APA (W=100)		min-APA		min-APA (W=100)	
		# a/i	time	# a/i	time	# a/i	time
jpeg (99/277)	2	115	380	67	< 1	67	< 1
	3	119	1140	50	10	50	10
	4	121	2340	41	440	42	220
	6	122	3940	-	-	42	1440
	8	122	5860	-	-	45	2720
motion (280/734)	2	377	1110	222	< 1	222	< 1
	3	382	2980	156	20	156	20
	4	383	5920	114	220	114	210
	6	384	9530	-	-	91	2560
	8	384	14330	-	-	96	5770
cpp (598/1377)	2	527	2040	219	< 1	219	< 1
	3	531	5560	109	80	109	70
	4	532	11360	94	64520	96	1610
	6	533	18880	-	-	100	7070
	8	533	28570	-	-	101	9150

램 3개씩을 선별하여 결과를 보인다. 예제로 보이는 벤치마크들은 complex (complex multiplier), fir, biquad (biquad_one_section), jpeg, motion (motion estimation), cpp이다.

표 1과 표 2는 각각 시퀀스의 길이가 작은 경우와 큰 경우에 제안하는 minimum cost-nodes breaking을 사용한 APA 알고리즘의 효용성을 보인다.

표 1과 2는 각각 적은 변수의 개수와 적은 접근 시퀀스의 길이를 갖는 벤치마크와 많은 변수의 개수와 긴 시퀀스의 길이를 갖는 벤치마크에서 Wess와 Zeitlhofer에 의해 제안된 equivalent tie-nodes breaking을 적용한 APA 알고리즘과 우리가 제안한 minimum cost-nodes breaking을 적용한 APA 알고리즘의 결과를 사용 가능한 주소 레지스터의 개수의 변화에 따라 비교하여 보여준다. 비교 대상은 각 알고리즘을 사용하여 생성된 주소 인스트럭션의 수와 그에 걸린 시간이다. 표에서 |V|는 벤치마크 프로그램에서 사용된 변수의 개수, |S|는 변수 시퀀스의 길이, K는 사용 가능한 주소 레지스터의 수를 나타낸다. 또한 eq-APA는 equivalent tie-nodes breaking을 적용한 APA 알고리즘, min-APA는 minimum cost-nodes breaking을 적용한 APA 알고리즘을 의미한다.

같은 메모리 레이아웃을 사용한다고 가정할 때 (여기서는 tie-breaking 방식으로 결정한 메모리 레이아웃을 사용하였음) K의 수가 커질수록 같은 벤치마크에서 주소 인스트럭션을 계산하는 시간이 많이 필요하게 된다. eq-APA의 경우 K가 증가함에 따라 연산에 많은 시간을 필요로 하게 된다.

표 1에서 fir의 경우 K=8일 때 48분의 시간이 소요되었으며, biquad의 경우 K가 6 이상인 경우에 정해놓은 시간 내에 솔루션을 얻을 수가 없었다. 그래서 할당 트리의 각 레벨에서 최대로 생성 가능한 노드의 개수 (W) 를 제한하여 시간을 줄였다. 3번째, 5번째 열을 비교하여 알 수 있듯이 W를 제한하면 그만큼 결과는 좋지 않을 수 있다.

표 2에서 테스트한 벤치마크에서 W를 제한하지 않은 eq-APA 알고리즘의 수행 결과는 이렇다. K=2일 때 jpeg 벤치마크는 1.4분이 걸려 min-APA와 같은 결과를 얻었지만, 그 이외의 모든 경우에는 정한 시간 이내에 솔루션을 얻을 수가 없었다. 때문에 우리는 긴 시퀀스 길이를 갖는 벤치마크에 대하여 eq-APA는 W를 제한한 실험값을 사용한다. W를 100으로 제한한 eq-APA의 실험은 너무 많은 주소 코드를 생성했지만 min-APA와 같은 조건으로 비교하기 위하여 우리는 100으로 설정한 W를 사용한다. 알고리즘 min-APA의 경우에도 eq-APA보다는 많은 실험에서 솔루션을 얻었지만 K가 증가함에 따라 제한된 시간 내에 솔루션을 얻을 수 없는 경우가 있었다. 그리하여 min-APA에도 W를 제한하여 결과를 얻었고, 이를 비교하였다. 그렇지만 eq-APA와는 다르게 min-APA에서 W를 100으로 제한하였을 때 적은 K에서는 결과값이 같으며, K가

1 주소 코드의 수 (The number of address instruction)
2 CPU 시간의 단위 : msec
3 분 (minute)
4 최대 시간 (1 hour) 내에 완료되지 못한 실험

4 이상일 때 W를 제한한 결과가 조금 좋지 않을수 있지만, 최적의 값과 거의 비슷한 것을 실험을 통해 알 수 있다. 우리가 제안한 breaking을 사용한 min-APA는 eq-APA와 같은 결과를 얻으면서 시간을 크게 절감한다.

많은 메모리와 많은 시간을 요구하는 APA 알고리즘에서 제안한 minimum cost-nodes breaking은 모든 경우에 equivalent tie-nodes breaking과 같은 최적의 주소 인스트럭션을 얻을 수 있었고, 수행 시간은 equivalent tie-nodes breaking보다 훨씬 적은 시간을 요구하는 것을 확인할 수 있다.

표 3은 기존의 방법과 초기 메모리 레이아웃이 다르게 결정된 경우의 APA 알고리즘의 결과를 비교하여 보인다.

표 3에서 보이는 GOA는 Leupers와 Marwedel에 의해 제안된 알고리즘이며, min-APA_OFU와 min-APA_TB는 각각 OFU 방식과 tie-breaking 방식에 따라 결정된 메모리 레이아웃으로 minimum cost- nodes breaking을 적용한 APA 알고리즘을 나타낸다. APA 알고리즘은 각각 W=100으로 제한한 상태의 결과값을 보여준다. 각 알고리즘의 수행 시간은 메모리 레이아웃을 결정하는 데 걸린 시간도 포함한다.

우리가 테스트한 벤치마크에서는 OFU 방식의 메모리 레이아웃을 사용한 APA 알고리즘이 GOA에 비해 평균 23.8%, tie-breaking 방식의 메모리 레이아웃을 사용한 APA 알고리즘에 비해 평균 22.9% 적은 주소 인스트럭션을 얻을 수 있었다.

표 4는 반복 기법에서 중요한 요건인 반복의 종료 조건에 대해 실험한 결과를 보인다. OffsetStone의 전체 3,406개 시퀀스들 중에서 주소 레지스터 개수의 변화에 따라 반복 기법의 APA-MVA 알고리즘을 사용했을 때 최대 반복 횟수를 보인다. 최대 반복은 100으로 설정했다. 3.2절에서 언급한 것처럼 반복의 종료는 새로운 서브 시퀀스로 접근 그래프를 그려서 메모리 레이아웃을

표 3. GOA와 메모리 레이아웃이 다른 APA 알고리즘의 결과 비교 (W=100)
Table 3. The comparison of the results of GOA and APA algorithm with different memory layouts. (W=100)

Bench.	K	GOA		min-APA_OFU		min-APA_TB		Gain (%)	
		# addr_instr	time (ms)	# addr_instr	time (ms)	# addr_instr	time (ms)	OFU/GOA	OFU/TB
complex	2	2	< 1	2	< 1	2	< 1	0.0	0.0
	3	3	< 1	2	< 1	2	< 1	33.3	0.0
	4	3	< 1	2	< 1	2	< 1	33.3	0.0
	6	3	< 1	2	< 1	2	< 1	33.3	0.0
	8	3	< 1	2	< 1	2	< 1	33.3	0.0
fir	2	4	< 1	3	< 1	4	< 1	25.0	25.0
	3	4	< 1	3	< 1	3	< 1	25.0	0.0
	4	4	< 1	3	< 1	3	< 1	25.0	0.0
	6	4	< 1	3	< 1	3	< 1	25.0	0.0
	8	4	< 1	3	< 1	3	< 1	25.0	0.0
biquad	2	5	< 1	4	< 1	6	< 1	20.0	33.3
	3	5	< 1	3	< 1	4	< 1	40.0	25.0
	4	4	< 1	3	< 1	4	< 1	25.0	25.0
	6	6	< 1	3	< 1	4	< 1	50.0	25.0
	8	6	< 1	3	< 1	4	< 1	50.0	25.0
jpeg	2	75	20	57	< 1	67	< 1	24.0	14.9
	3	58	20	35	10	50	< 1	39.7	30.0
	4	42	10	27	30	42	220	35.7	35.7
	6	12	10	21	570	42	1440	-75.0	50.0
	8	12	10	21	1280	45	2710	-75.0	53.3
motion	2	260	370	188	< 1	222	< 1	27.7	15.3
	3	180	290	119	10	156	30	33.9	23.7
	4	121	200	82	60	114	210	32.2	28.1
	6	54	150	53	1330	91	2570	1.9	41.8
	8	47	140	44	3750	96	5770	6.4	54.2
cpp	2	164	2740	104	< 1	219	10	36.6	52.5
	3	147	2840	74	60	109	70	49.7	32.1
	4	127	2330	67	1010	96	1590	47.2	30.2
	6	119	1940	63	3810	100	7180	47.1	37.0
	8	112	1910	70	7780	101	9280	37.5	30.7
avg.								23.8	22.9

표 4. GOA와 초기 APA 알고리즘, 반복 APA 알고리즘의 결과 비교
Table 4. The comparison of the results by GOA and initial APA and iterative APA algorithms.

Bench.	K	GOA	min-APA_OFU	Iterative min-APA_OFU			Gain (%)	
		# of addr_instrs	# of addr_instrs	# of addr_instrs	# of iterations	time (ms)	Iter/GOA	Iter/APA
jpeg	2	75	57	40	4	< 1	46.7	29.8
	3	58	35	28	3	10	51.7	20.0
	4	42	27	24	5	120	42.9	11.1
	6	12	21	19	3	1590	-58.3	9.5
	8	12	21	21	2	3810	-75.0	0.0
motion	2	260	188	170	3	10	34.6	9.6
	3	180	119	104	4	40	42.2	12.6
	4	121	82	68	2	180	43.8	17.1
	6	54	53	42	3	5350	22.2	20.8
	8	47	44	44	2	11110	6.4	0.0
cpp	2	164	104	99	5	20	39.6	4.8
	3	147	74	68	2	230	53.7	8.1
	4	127	67	64	2	3080	49.6	4.5
	6	119	63	61	2	11300	48.7	3.2
	8	112	70	67	3	30450	40.2	4.3
average							25.9	10.4

얻었을 때 이전 단계와 같은 메모리 레이아웃을 얻게 되면 반복이 멈추도록 하였다. 실험을 통하여 대부분의 벤치마크가 10회가 넘지 않는 반복 동안 종료되었음을 확인할 수 있었다. K=3인 경우 anthr 벤치마크에서 유일하게 예외가 일어났는데 실험 결과를 분석한 결과, 해당 벤치마크는 3가지의 메모리 레이아웃이 반복됨을 알 수 있었으며, 이를 제외한 모든 벤치마크는 최대 반복 횟수에 다다르기 전에 자동 종료되었다. 특히 K가 4 이상인 경우는 OffsetStone 전체 중 2,000개 이상의 시퀀스에서 최대 2회만 반복했음을 확인할 수 있었다. 본 논문에서 제안하는 반복 기법은 현실적으로 적용이 가능함을 실험 결과를 통해 알 수 있다.

표 5는 우리가 제안한 반복 기법의 성능을 측정한다. 표 3의 실험값을 토대로 메모리 레이아웃 갱신 단계를 통해 APA 알고리즘을 반복했을 때의 결과를 GOA와 초기 APA 알고리즘의 결과와 비교하여 보인다. 앞의 실험들과 마찬가지로 우리의 minimum cost-nodes breaking이 적용된 APA 알고리즘을 사용하였고, 수행 속도의 단축을 위하여 APA 트리의 W를 100으로 제한하였다. 초기 메모리 레이아웃은 표 3에서 가장 좋은 값을 얻은 OFU 방식의 메모리 레이아웃을 사용하였다. 실험 값을 통해 알 수 있듯이 모든 경우에 GOA 보다 좋아지지 않을 수 있다. 왜냐하면 GOA와 초기 APA, 반복 APA의 메모리 레이아웃은 모두 다르기 때문이다. 그럼에도 불구하고 반복 APA는 우리가 실험한 벤치마크에 대하여 GOA보다 평균 25.9%, 초기 APA보다 평균 10.4%의 적은 주소 코드를 만들었다. 또한 실험의 모든 반복 횟수는 최대 5번을 넘지 않은 것으로 알 수 있듯이 반복 기법은 추가적인 많은 시간을 필요로 하지는 않는다.

표 5. 반복 종료 횟수에 대한 실험
Table 5. The comparison of the number of iterations.

K	min-APA_OFU		min-APA_TB	
	Bench. (|V|/|S|)	# of iterations	Bench. (|V|/|S|)	# of iterations
2	f2c (61/130)	14	gsm decode (437/874)	15
			klt (88/247)	9
3	anthr (27/66)	100	cavity (569/1603)	11
	gsm encode (388/776)	31	mpeg2 (80/235)	8
4-8	-	2	-	2

V. 결 론

본 논문은 내장형 시스템의 전력 소모와 메모리 공간, 실행 시간 등의 다루기 어려운 요구 조건들의 충족을 위하여 디지털 시그널 프로세서용 코드 생성에서 변수들의 고정된 메모리 레이아웃이 있는 상황에서의 주소 포인터 할당 문제에 대한 알고리즘을 개선하는 새로운 breaking 테크닉을 제안했다. 주어진 메모리 레이아웃에서의 최적의 값을 얻어 변수들의 주소 연산을 위해

존재하는 주소 인스트럭션을 최소화하도록 구성되어 있는 APA 알고리즘의 시간 복잡도를 한층 줄이는데 목적을 두었다.

실험에 OffsetStone 벤치마크를 이용하여 제안한 시간 복잡도를 줄이는 테크닉과 반복 알고리즘에 대하여 시뮬레이션 하여 기존의 GOA 방식과 비교하였을 때 많은 시간을 들이지 않고도 평균 25.9%의 이득을 얻을 수 있음을 보였다.

참 고 문 헌

[1] G. Araujo, "Code Generation Algorithm for DSP's," Ph.D. dissertation, Dept. Elect. Eng., Princeton Univ., Princeton, NJ, 1997.

[2] D. H. Bartley, "Optimizing stack frame accesses for processors with restricted addressing modes," *Software-Practice and Experience*, vol. 22, no. 2, pp. 101-110, 1992.

[3] S. Liao, S. Devadas, K. Keutzer, S. Tjiang, and A. Wang, "Storage assignment to decrease code size," in *ACM Trans. Program. Languages Syst.*, vol. 18, no. 3, pp. 235-253, 1996.

[4] R. Leupers and P. Marwedel, "Algorithm for address assignment in DSP code generation," in *Proc. Int. Conf. Computer-Aided Design*, pp. 109-112, 1996.

[5] R. Leupers and F. David, "A uniform optimization technique for offset assignment problem," in *Proc. Int. Symp. Syst. Synthesis*, pp. 3-8, 1998.

[6] A. Sudarsanam, S. Liao, and S. Devadas, "Analysis and evaluation of address arithmetic capabilities in custom DSP architectures," in *Proc. Design Automation Conf.*, pp. 287-292, 1997.

[7] C. Gebotys, "DSP address optimization using a minimum cost circulation technique," in *Proc. Int. Conf. Computer-Aided Design*, pp. 100-103, 1997.

[8] S. Atri, J. Ramanujam, and M. Kandemir, "Improving offset assignment for embedded processors," in *Proc. Int. Workshop on Languages and Compilers for Parallel Computing*, pp. 158-172, 2000.

[9] D. Ottoni, G. Ottoni, G. Arajuo, and R. Leupers, "Improving offset assignment through simultaneous variable coalescing," in *Proc. Int. Workshop Software Compilers Embedded Syst.*, 2003.

[10] A. Rao and S. Pande, "Storage assignment using expression tree transformation to generate compact and efficient DSP code," in *Proc. ACM SIGPLAN Conf. Program. Language Design and Implementation*, pp. 128-138, 1999.

[11] S. Lim, J. Kim, K. Choi, "Scheduling-based code size reduction in processors with indirect addressing mode," in *Proc. Int. Symp. Hardware/Software Codesign*, pp. 165-169, 2001.

[12] Y. Choi and T. Kim, "Address assignment combined with scheduling in DSP code generation", in *Proc. Design Automation Conference*, pp. 225-230, 2002.

[13] C. Xue, Z. Shao, Q. Zhuge, B. Xiao, M. Liu, and E. H.-M. Sha, "Optimizing address assignment and scheduling for DSPs with multiple functional units," in *IEEE Trans. Circuits and Systems II*, vol. 53, no. 9, pp. 976-980, September 2006.

[14] B. Wess and T. Zeitlhofer, "Optimum address pointer assignment for digital signal processors," in *Proc. Int. Conf. Acoustics, Speech, and Signal Processing*, vol. 5, pp. 121-124, 2004.

[15] R. Leupers, "Offset assignment showdown: Evaluation of DSP address code optimization algorithms," in *Proc. Int. Conf. Compiler Construction*, Lecture Nodes on Computer Science, LNCS 2622, Springer-Verlag, Poland, 2003, http://www.address-code-optimization.org.

저 자 소 개

이 희 진(학생회원)
2006년 전북대학교 컴퓨터공학과 학사 졸업.
2006년 3월~현재 전북대학교 전자정보공학부 석사과정.
<주관심분야 : DSP, 컴파일러>

이 종 열(평생회원)
1993년 한국과학기술원 전자전산학과 졸업 (B.S.).
1996년 한국과학기술원 전자전산학과 졸업 (M.S.).
2002년 한국과학기술원 전자전산학과 박사 (Ph.D.).
2002년 9월~2003년 9월 하이닉스 반도체 선임연구원.
2003년 10월~2004년 2월 한국과학기술원 BK21 초빙교수.
2004년 3월~현재 전북대학교 전자정보공학부 조교수.
<주관심분야: SoC 설계, 내장형 프로세서 설계, 내장형 소프트웨어 최적화>