특집

임베디드 소프트웨어의 Post-Pass 최적화 기법

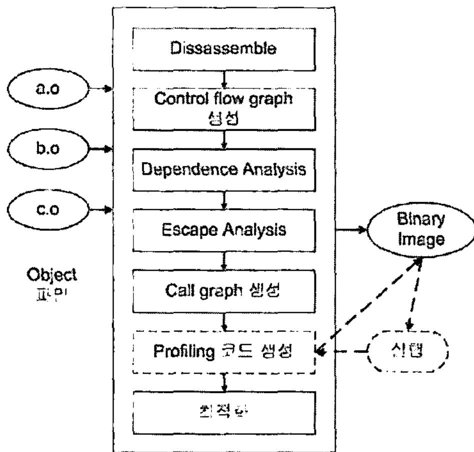
이재진 (서울대학교 컴퓨터공학부)

I. 서 론

임베디드 시스템은 보통 특정한 목적을 위해 설계되고 그 목적 달성을 위한 특정한 응용 프로그램을 실행시킨다. 이러한 임베디드 시스템을 위한 소프트웨어의 최적화는 기존 데스크 탑에서 사용하는 소프트웨어의 최적화에 비교해 다른 점이 있는데, 소프트웨어가 실행되는 하드웨어 플랫폼에 대한 정보를 미리 자세하게 알 수 있고, 특정한 소프트웨어만 실행하기 때문에 profiling 기법을 통한 소프트웨어 최적화가 가능하다는 것이다. 하지만, 임베디드 시스템은 일반적으로 생산 비용에 매우 민감하고, 실시간성으로 인하여 성능과 전력소모에 관한 여러 가지 제약조건이 많다. 즉, 좀 더 많은 최적화의 기회가 있는 반면에 최적화하기 위해 충족하여야할 조건이 기존의 데스크 탑 시스템을 위한 소프트웨어 보다 많다고 할 수 있다.

임베디드 소프트웨어의 최적화는 성능 향상, 코드 크기 감소, 및 저전력 소모의 세 가지를 주된 목표로 한다. 코드 크기의 감소는 주로 임베디드 시스템에 사용되는 메모리의 크기를 줄이게 되어 생산 비용을 감소시킨다. 이 방법은, 이전에 상대적으로 메모리의 크기가 작았던 데스크 탑 시스템에서 실행되는 소프트웨어의 최적화를 위해 많이 연구가 되었지만 지금은 임베디드 시스템 분야에서만 연구되고 있다. 성능 향상과 저전력 소모는 현재의 고성능 컴퓨터 시스템 및 데스크 탑 시스템에서 많이 연구가 되고 있는 분야이고, 임베디드 소프트웨어에 부여된 제약조건을 만족하는 범위 안에서 이전에 개발된 기법을 그대로 임베디드 소프트웨어의 최적화에 적용할 수도 있다. 슈퍼컴퓨터 시스템을 위해서 개발된 최적화 기법이, 적용 비용이 시간이 지남에 따라 감소하면서 10년 정도 후에 데스크 탑 시스템의 최적화에 적용되어 온 예를 생각하면, 데스크 탑 시스템과 임베디드 시스템의 관계는 이전의 슈퍼컴퓨터 시스템과 데스크 탑 시스템의 관계와 비슷하다고 할 수 있다.

본 고는 이전에 데스크 탑 시스템을 위한 소프트웨어의 최적화를 위해 연구가 되었고 많이 쓰이지는 않았지만[1] 최근에 임베디드 소프트웨어 최적화에서 널리 연구되고 있는 post-pass 최적화 기법을 소개하고, 이를 이용하여 임베디드 소프트웨어의 코드가 사용하는 메모리의 크기를 최적화하는 기법을 소개한다. 또, 최근에 연구가



〈그림 1〉 SNACK - pop의 구조

되기 시작한 scratch-pad 메모리를 이용하여 링크 시, 또는 실행 이미지 단계의 임베디드 소프트웨어를 최적화하는 기법을 소개한다.

본 고의 구성은 제 II 절에서 post-pass 최적화의 개요에 대하여 설명하고, 제 III 절에서 post-pass 최적화를 이용한 demand paging 기법을 소개하며, 제 IV 절에서 현재 많은 연구가 되고 있고 실제 임베디드 프로세서에서 많이 채용하고 있는 scratch-pad 메모리를 위하여 이들 post-pass 최적화 및 demand paging 기법을 응용하는 방법을 소개한다. 제 V 절에서 우리나라의 연구 동향 및 향후 전망을 알아본다.

II. Post-Pass 최적화 기법

Post-pass 최적화는 주로 어셈블리나 오브젝트 코드에 부가적인 최적화 기법을 적용하는 것을 일컫는다. 주로 컴파일러가 기존의 최적화기법을 적용한 다음 생성되는 어셈블리 코드나 오브젝트 코드, 또는 실행 이미지에 적용되기 때문에, 한번에 하나의 파일 또는 모듈을 처리하는 컴파일러보다 프로그램에 대해 더 많은 정보를 가지고 최적화를 수행할 수 있다. 즉, 프로그램을 구성하는 일부분이 아니라 라이브러리를 포함한 전체 프로그램을 보면서 최적화를 행하므로 최적화의 기회가 상대적으로 많다. 또, 컴파일러 최적화가 프로그램의 중간 표현단계에서 주로 이루어지는 반면 post-pass 최적화는 기계어 단계에서 이루어지므로 더욱 세밀한 최적화를 수행할 수 있고 타겟이 되는 프로세서 또는 시스템에 특정한 최적화를 할 수 있는 장점이 있다. 더욱이, 소스 코드를 판매하지 않고 소프트웨어 컴포넌트 단위로 오브젝트 코드를 거래하는 임베디드 소프트웨어 시장의 특성을 볼 때, 타겟 플랫폼에 특정한 최적화를 수행할 수 있는 기회는 이들 컴포넌트를 조합하여 소프트웨어의 실행 이미지를 생성할 때뿐이다. 따라서 post-pass 최적화는 중요하고 효율적인 임베디드 소프트웨어 최적화 기법이라고 할 수 있다.

그림 1은 서울대학교 컴파일러 연구실에서 개발 중인 소프트웨어 개발 도구의 일부분인 SNACK-pop (Seoul National university Advanced Compiler tool Kit - Postpass Optimizer)의 개략적인 전체 구조를 보여 준다[2]. 이 post-pass 최적화기는 오브젝트 파일이나 실행 이미지 파일을 입력으로 받는다. 이들 파일은 현재 많이 사용되고 있는 실행 및 오브젝트 파일 형식인 ELF (Executable Linkable Format) 형식으로 구성되어 있다. 이들 입력 파일은 역어셈블되어 어셈블리 코드로 변환된다. 이렇게 변환한 코드는 다시 제어 흐름 그래프 (control flow graph)로 변환된다. 실행 이미지나 오브젝트 파일 같이 binary 코드를 바탕으로 제어 흐름 그래프를 구성하는 것은 몇 가지 제약이 있는데, 코드를 basic block으로

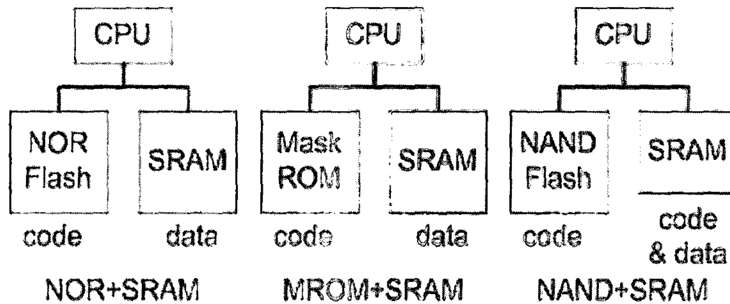
나눔에 있어 타겟 프로세서의 branch delay slot을 채우는 명령어의 처리가 그 중 한 예이다. 하지만 이런 제약은 기존의 제어 흐름 그래프의 구조를 약간 수정함으로써 없앨 수 있다. 제어 흐름 그래프를 생성한 후 post-pass 최적화기는 여러 가지 최적화에 필요한 프로그램 분석을 하게 되는데, 그 첫 번째가 dependence analysis이며, 이 단계에서 명령어와 명령어 간의 종속성을 판단하게 된다. 명령어간의 종속성은 최적화 단계에서 한 개의 명령어를 전체 코드 내에서 이동시킬 수 있는 범위를 알려 주게 된다. 명령어가 사용하는 레지스터 간에 존재하는 종속성은 판별하기가 쉬우나, 여러 가지 주소 지정 방법(addressing mode)으로 표현되는, 명령어가 참조하는 메모리 위치 간의 종속성은 정확하게 판별하기에 힘이 많이 들기 때문에 보수적인 접근 방법을 써서 실제로 종속성이 존재하지 않더라도 존재 할만 한 곳은 존재한다고 판별하는 분석 방법을 쓰게 된다. 어셈블리 차원이나 실행 이미지 차원의 프로그램 분석 방법은 많이 연구가 되지 않은 분야이며 현재 존재하는 실행 이미지 분석 방법보다 더 정확한 분석방법을 개발하는 것이 향후 연구 과제가 될 것이다.

Dependence analysis가 끝이 나면 escape analysis를 수행하는데, escape analysis는 한 함수의 코드 영역 안에 든, 읽기만 허용이 된 (constant) 데이터(예를 들어, 프로그램 내 어떤 변환 테이블의 베이스 주소)가 다른 함수에 의해 참조되는지 판별하는 단계이다. 원래의 escape analysis는 한 함수의 local 변수가 다른 함수에 의해 참조 되는지를 (예를 들어, local 변수의 주소가 인자로 다른 함수로 전달되는 경우) 판별하여 그런 경우만 그 변수에 스택영역을 할당하고 그렇지 않으면 레지스터를 쓸 수 있도록 하는 최적화 기법에 이용된 분석방법이다. post-pass 최적화의 escape analysis는 한 함수의 코드 영역 안에 든 읽기만 허용이 된 데이터가 다른 함수에 의해 참조 되면, 그 데이터 영역을 참조하는 함수의 끝에 복사하여 각 함수가 자신이 참조하는 읽기만 허용이 된 데이터를 코드 영역에 가지고 있도록 하여 전체 프로그램을 함수 단위로 나눌 수 있도록 하는 정보를 제공한다.

Escape analysis가 끝난 다음에 정적 호출 그래프(static call graph)를 생성하게 되는데, 정적 호출 그래프는 프로그램 안에 들어 있는 함수 간의 호출 관계를 나타내는 방법 중의 하나이다. 각 함수는 그래프의 node가 되고 호출하는 함수에서 호출되는 함수로 방향성이 있는 edge가 존재한다. 정적 호출 그래프의 단점은 함수 포인터가 있을 경우 함수들의 호출 관계를 완벽하게 표현하지 못한다는 것이다. 함수 포인터가 존재하는 프로그램의 완벽한 정적 호출 그래프 생성도 앞으로 풀어야 할 연구 과제이지만 함수 포인터의 경우는 예외로 하여 조금 부정확한 정적 호출 그래프를 사용할 수도 있다.

Profile 코드 생성 단계는 앞서 수행한 분석을 바탕으로 원래의 프로그램에 profiling에 필요한 측정 코드를 삽입하는 단계이다. 예를 들어 함수의 시작 부분에 counter 코드를 삽입하여 프로그램이 실행 될 때 그 함수가 호출 되는 횟수를 기록할 수 있다. 측정 코드의 삽입이 완료되면 post-pass 최적화기가 실행 이미지를 생성한 다음 이 코드를 실행하여 profile 정보를 얻는다. 이 profile 정보는 다시 post-pass 최적화기로 입력이 되어 프로그램의 최적화에 필요한 정보를 (예를 들어, 각 edge에 호출 횟수가 부기된 정적 호출 그래프) 제공할 수 있다.

앞서 행한 프로그램 분석 단계에서 얻은 정보



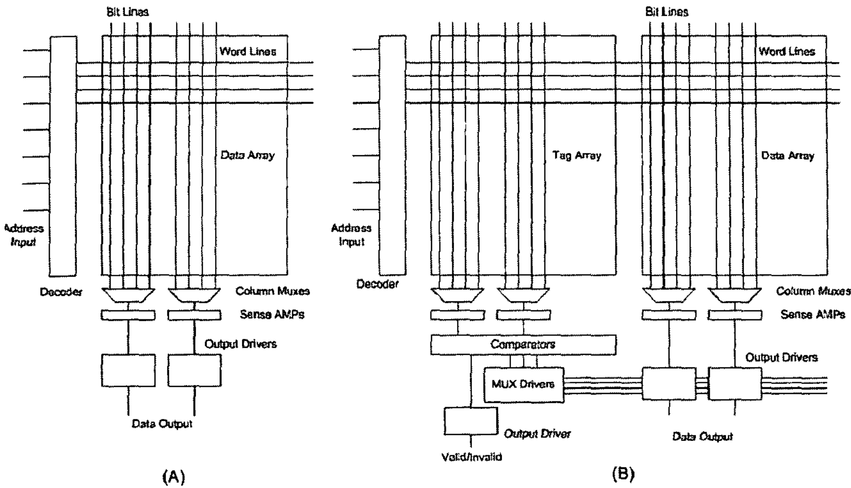
〈그림 2〉 저급 임베디드 시스템의 메모리 시스템

를 가지고 post-pass 최적화기는 프로그램 최적화를 수행하게 되는데, 이 단계에서 기존의 컴파일러가 행하는 대부분의 최적화 기법이 적용 될 수 있으며, 이를 이용하여 코드 크기 감소, 성능 향상, 저전력 소모를 달성할 수 있다. 이 단계에서 특히 유용한 기존 컴파일러 최적화 기법은 instruction scheduling, register allocation, function inlining 등이 있다. 또, 기존의 방법과 다르고 post-pass 최적화의 장점을 살린 최적화 기법을 적용할 수 있는데, 다음에 나오는 두개의 절에서 이러한 post-pass 최적화 기법을 소개하겠다.

III. Post-Pass 최적화를 이용한 Demand Paging 기법

그림 2는 저급(low-end)의 휴대전화와 메모리 카드 제어기에 사용되는 임베디드 시스템에 대한 세 가지 종류의 메모리 구조를 보여 준다.[2] 이들은 코드 저장소로 사용되는 NOR와 NAND 타입의 플래시 메모리와 데이터나 작업메모리로 사용되는 SRAM으로 구성되어 있다. 이런 종류의 임베디드 시스템은 기존의 운영체제에 존재하는 가상 메모리가 존재하지 않으며 프로세서가 MMU(Memory Management Unit)도 가지고 있지 않다. 이러한 종류의 저급 임베디드 시스템에 사용되는 메모리 구조는, SRAM에 소모되는 비용을 절감하고, 펌웨어에 대한 기능의 추가나 디버깅을 위해 코드를 쉽게 수정할 수 있어야 한다는 설계상의 제약이 있다. 최근의 경향을 보면 펌웨어의 크기가 보안 기능 같은 새로운 기능의 추가로 인해 수 KB에서 수백 KB로 증가하고 있기 때문에, 코드가 차지하는 메모리를 효율적으로 관리하는 것이 시스템의 설계할 때 중요한 고려 사항이 되고 있다.

그림 2의 NOR+SRAM은 NOR 타입의 플래시 메모리를 코드 저장소로 사용하고 직접 코드를 플래시 메모리 상에서 실행하며 SRAM을 데이터를 위한 작업 메모리로 사용하는 경우를 이야기한다. 이 경우 펌웨어가 플래시 메모리에 저장되기 때문에 펌웨어의 기능 향상과 디버깅에 따른 수정이 용이하다. MROM+SRAM은 NOR 타입의 플래시 메모리보다 가격이 싼 Mask ROM(EEPROM)으로 NOR 플래시 메모리를 대치한 경우이며 펌웨어를 수정할 때 마다 Mask ROM을 새로 만들어야 하기 때문에 여기에 들



〈그림 3〉 On-chip scratch-pad 메모리와 on-chip 캐시의 구조

어가는 비용의 부담이 크다. NAND+SRAM의 경우는, NAND 플래시 메모리의 특성 때문에 저장된 코드가 실제로 실행될 때 SRAM에 복사된 후 실행되게 된다. 코드가 SRAM 상에서 실행되므로 수행 속도가 빠르다는 장점이 있으나 SRAM을 작업 메모리의 역할도 하므로 같은 크기의 코드를 실행하려면 SRAM의 크기가 NOR+SRAM이나 MROM+ SRAM보다 커져야 하는 단점이 있다.

이러한 NAND+SRAM 구조의 단점을 해결하기 위해, post-pass 최적화를 통하여 기존의 가상 메모리에서 사용되던 demand paging 기법을 사용할 수 있다. SRAM 상의 작은 영역이 코드 실행을 위한 영역으로 할당되고 프로그램은 세그먼트(segment)라 불리는 여러 개의 작은 부분으로 나뉜다. 가각의 세그먼트는 적어도 하나의 함수를 담고 있으며, 프로그램이 실행되면서 이 함수가 호출되면, 이 함수를 담고 있는 세그먼트가 SRAM 상의 코드 실행 영역에 올라가 실행 되게 된다. post-pass 최적화기는 이를 관리하는 페이지 관리자 (page manager)를 자동으로 생성하고 이를 생성된 프로그램의 실행 이미지에 붙여서 하나의 실행 이미지로 만든다. 원래 프로그램에 있던 모든 함수 호출 명령과 함수로부터 복귀 명령은 페이지 관리자를 호출하는 명령으로 post-pass 최적화기에 의해 대치된다. 페이지 관리자는 항상 SRAM 상에 상주하면서 함수 호출 명령어의 타겟 주소가 SRAM 상의 코드 실행 영역에 존재하는가를 확인하고, 존재하면 제어를 그 타겟 주소로 넘기고, 그렇지 않으면 타겟 주소가 들어 있는 세그먼트를 플래시 메모리에서 SRAM에 얹은 다음 SRAM 상의 타겟 주소로 넘긴다. 이러한 방법으로 약 20%의 성능이 감소되는 반면 SRAM의 크기는 약 30%정도 줄어든다 [2]. 더 개선된 프로그램 분석 방법을 쓰면 페이지 관리자의 호출에 따른 성능 감소폭을 줄일 수 있으며, 임베디드 시스템 설계자는 demand paging을 함으로써 야기되는 성능 감소와 SRAM 크기 감소간의 trade-off 관계를 잘 고려하여 시스템을 설계 하여야 한다.

IV. Scratch-Pad 메모리를 위한 Post-Pass 최적화 기법

전원을 전적으로 배터리에 의존하는 모바일 임베디드 시스템은 배터리 용량의 한계 때문에 효율적인 에너지의 사용이 매우 중요하다. 특히 메모리 시스템은 전체 임베디드 시스템에서 소모되는 에너지에서 큰 비중을 차지한다. 일반적인 임베디드 프로세서에서 SRAM을 바탕으로 만들어진 on-chip 캐시(cache)는 전체 chip 전력의 25%~45%를 소모한다[3]. 이에 반하여, 이러한 on-chip SRAM을 주소 지정이 가능한 scratch-pad 메모리로 사용하면 같은 용량의 캐시에 비해 1/3 정도의 크기를 가지고 40%정도 더 적은 전력을 소모한다. 그림 3은 scratch-pad 메모리와 캐시의 구조를 비교한 것이다[3][4]. On-chip 캐시는 tag array 참조와 comparator와 같은 부가적인 회로로 인하여 같은 용량의 scratch-pad 메모리에 비하여 더 많은 전력을 소모한다.

하지만, 비용의 문제로 인하여 scratch-pad 메모리의 크기는 외부 SRAM 또는 DRAM 보다 훨씬 작은 수 Byte에서 수 KByte 정도이다. 따라서 작은 크기의 scratch-pad 메모리를 효율적으로 사용하여 프로그램의 성능을 높이면서 전력 소모를 줄이도록 하는 것이 매우 중요하다. 이를 위하여 프로그램에서 자주 호출되는 함수들을 scratch-pad 메모리에 올려 실행하거나 자주 접근되는 변환 테이블 같은 데이터를 scratch-pad 메모리에 올려서 실행하는 많은 연구가 있었지만 대부분의 경우 scratch-pad 메모리에 할당 가능한 크기의 데이터만 고려하거나, 배열과 같이 큰 데이터는 일부를 scratch-pad 메모리에 할당하여 제한된 방법으로 이를 참조하는 정도였다[5]. 이렇게 크기만 고려한 정적인 scratch-pad 메모리 할당 방법은 사용가능한 scratch-pad 메모리의 일부 영역이 남아 있더라도 할당 하려는 데이터의 크기 때문에 이 영역을 잘 활용할 수 없고 이미 할당된 영역은 할당된 데이터가 더 이상 사용되지 않더라도 새로운 데이터의 할당을 위해 다시 사용될 수 없다. 또, 자주 사용되는 데이터에 대한 scratch-pad 메모리의 동적인 할당에 대한 연구가 있었으나 scratch-pad 메모리와 off-chip DRAM 간의 복사 비용만 고려하고 참조되는 각각의 변수에 대한 가중치를 고려하지 않았기 때문에 효율적인 scratch-pad 메모리의 사용이 될 수 없었다.

이러한 기존의 방법대신에 post-pass 최적화를 통한 demand paging 기법이 scratch-pad 메모리의 동적인 할당에 사용될 수 있다. 프로그램의 코드만 고려했을 때, 각각의 함수를 SRAM 영역, SRAM의 paging 영역, 및 DRAM 영역의 세 가지 영역중 하나에 할당 할 수 있다. Post-pass 최적화에서 나오는 profiling 정보(함수 호출 횟수)와 함수를 각 영역에 할당했을 때 복사에 따른 전력 소모비용을 고려하면 이 할당의 문제는 근사적으로 Integer Linear Programming의 한 문제가 되며 이를 풀어 전력소모를 감소시킬 수 있다. SRAM의 paging 영역에 할당된 함수는, 앞서 언급한 post-pass 최적화를 이용한 paging 방법을 사용하여 호출 될 때마다 동적으로 SRAM에 복사되어 실행된다. 이 예는 scratch-pad 메모리와 post-pass 최적화 기법이 명령어 캐시를 대신할 수 있음을 보여주는 좋은 예이다.

V. 결 론

소프트웨어가 실행되는 하드웨어 플랫폼에 대한 정보를 미리 알 수 있고, 특정한 소프트웨어만 실행하기 때문에 profiling 같은 기법을 통한

최적화가 효과적이며, 오브젝트 파일이 전체 소프트웨어의 컴포넌트로서 거래되는 임베디드 소프트웨어의 특성은 post-pass 최적화의 좋은 타겟임을 알 수 있다. 더욱이, 라이브러리를 포함하는 전체 프로그램의 최적화와 타겟 플랫폼에 특성화된 최적화를 수행할 수 있다는 측면에서 post-pass 최적화는 임베디드 소프트웨어에 잘 들어맞는 최적화 기법이다. 현재 많이 사용되는 소프트웨어 개발 환경에 post-pass 최적화가 포함되어 있지 않지만, 그 유용성을 볼 때 앞으로 사용될 임베디드 소프트웨어 개발 환경은 이를 포함하고 있을 가능성이 크다.

임베디드 소프트웨어가 우리나라의 신성장 동력으로 지정되어 연구 및 개발이 많이 되고 있지만, 국가적인 지원은 임베디드 시스템을 위한 응용 소프트웨어 및 실시간 운영체제의 연구 개발에 국한되어 있다. 정작 임베디드 소프트웨어 개발의 기반이 되는 컴파일러 및 소프트웨어 개발 환경의 연구와 개발을 위한 국가적 차원의 지원은 거의 없는 상태이며, gcc와 같은 open source 개발 도구를 이식하여 새롭게 개발되는 임베디드 프로세서의 소프트웨어 개발 환경에 이용하는 정도의 수준이다. 어느 분야든 기반 기술의 연구 없이는 그 기술의 수명이 극히 짧다고 할 수 있을 것이며, 국가의 장래를 위해서 임베디드 소프트웨어 개발 환경의 연구를 위한 지원과 이를 위한 인력 양성이 절실하다 할 것이다.

참고문헌

[1] Robert Muth, Saumya Debray, Scott Watterson, "Alto: A Link-Time Optimizer for the Compaq Alpha", Software Practice and Experience, Vol. 31, pp. 67-101, January 2001

[2] Chanik Park, Junghee Lim, Kiwon Kwon, Jaejin Lee, Sang Lyul Min, "Compiler-Assisted Demand Paging for Embedded Systems with Flash Memory", Proceedings of the 4th International Conference on Embedded Software, September, 2004

[3] Rajeshwari Banaker, Stefen Steinke, Bo-sik Lee, M. Balakrishnan, Peter Marwedel, "Scratchpad Memory: A Design Alternative for Cache on-Chip Memory in Embedded Systems", Proceedings of the 10th International Symposium on Hardware/Software Codesign, pp 73-78, 2002.

[4] Steven J. E. Wilton, Norman P. Jouppi, "An Enhanced Access and Cycle Time Model for On-Chip Caches", Western Research Laboratory Research Report 93/5.

[5] Wayne Wolf, Mahmut Kandemir, "Memory System Optimization of Embedded Software", Proceedings of the IEEE, Vol. 91, No. 1, January 2003.

저자소개

이 재 진

1991년 서울대학교 물리학과 학사
1995년 Stanford University Computer Science 석사
1999년 University of Illinois at Urbana-Champaign Computer Science 박사
1999년-1999년 University of Illinois at Urbana-Champaign, Department of Computer Science, 객원 강사
2000년-2002년 Michigan State University, Department of Computer Science and Engineering, 조교수
2002년-2004년 서울대학교 컴퓨터공학부 조교수
2004년-현 재 서울대학교 컴퓨터공학부 부교수
주관심분야 컴파일러, 임베디드 시스템, 고성능 컴퓨터 시스템