韓國 컴퓨터情報學會 論文誌
第13卷 第4號, 2008. 7.

2008-13-4-1-3

명령어 연관성 분석을 통한 가변 입력 gshare 예측기

곽 종 욱*

Variable Input Gshare Predictor based on Interrelationship Analysis of Instructions

Kwak, Jong Wook *

요 약

분기 히스토리는 분기 예측기의 주된 입력 요소로 사용된다. 따라서 적절한 분기 히스토리의 사용은 분기 예측의 정확도 향상에 큰 영향을 미친다. 본 논문에서는 분기 예측의 정확도를 향상시키기 위한 방법의 하나로, 명령어의 연관성 분석을 통한 선별적 분기 히스토리 사용 기법을 제안한다. 우선, 본 논문에서는 명령어의 연관성을 분석하는 세가지 서로 다른 알고리즘을 제안한다. 제안된 기법은 명령어의 레지스터 쓰기 연산에 기반하는 방법, 분기 명령어의 참조 레지스터에 기반하는 방법, 그리고 이들 두가지 방식을 상호 결합하는 방법이다. 또한, 제안된 세가지 알고리즘의 실질적 구현을 위해 이를 적용할 수 있는 가변 입력 gshare 예측기를 제안한다. 본 논문에서는 모의실험을 통해 세가지 알고리즘의 특징 및 장단점을 비교 분석한다. 특히, 기존의 고정된 입력을 사용하는 방식과 비교하여 제안된 기법의 성능 향상의 정도를 분석하며, 사전 프로파일링을 통해 얻어진 최적의 입력에 대한 성능상의 차이도 소개한다.

Abstract

Branch history is one of major input vectors in branch prediction. Therefore, the proper use of branch history plays a critical role of improving branch prediction accuracy. To improve branch prediction accuracy, this paper proposes a new branch history management policy, based on interrelationship analysis of instructions. First of all, we propose three different algorithms to analyze the relationship: register-writing method, branch-reading method, and merged method. Then we additionally propose variable input gshare predictor as an implementation of these algorithms. In simulation part, we provide performance differences among the algorithms and analyze their characteristics. In addition, we compare branch prediction accuracy between our proposals and conventional fixed input predictors. The performance comparison for optimal input branch predictor is also provided.

▸ Keyword : 분기 예측(Branch Prediction), 분기 예측 정확도(Branch Prediction Accuracy), 명령어 연관성(Instruction Relation), 분기 히스토리(Branch History), 가변 입력(Variable Input), gshare 예측기(Gshare Predictor)

• 제1저자 : 곽종욱
• 접수일 : 2008. 5. 1, 심사일 : 2008. 5. 28, 심사완료일 : 2008. 7. 25.
* 영남대학교 전자정보공학부

Ⅰ. 서 론

오늘날의 프로세서 환경은 더욱 세분화되어 가는 파이프라인의 사용과 다수개의 명령어들이 동시에 실행되는 슈퍼스칼라 방식, 그리고 Tomasulo 기법과 같은 동적인 스케쥴링 방식의 사용 등으로 특징지을 수 있다. 이와 같은 프로세서 환경에서는, 프로그램 상에서 분기 명령어가 나타날 때, 이에 대한 최종적인 실행 결과가 결정될 때까지 프로그램의 수행을 중단(stall)시키지 않는다[1]. 오히려, 예상되는 분기 경로로의 수행을 투기적으로 진행(Speculative Execution)하여 그에 해당하는 다음 명령어를 인출(fetch)해 오는 방식을 취한다. 이러한 수행 환경은 분기 예측 실패로 인한 예측 실패 비용(Misprediction Penalty)을 증가 시키는 원인으로 작용하기 때문에 보다 더 정확한 분기 예측 기법이 요구된다 [2][3]. 아울러, Sprangle 등은 분기 예측 실패율이 전체 시스템 상에서 단일 구성 요소로서는 가장 큰 시스템 성능 제약 요소라는 사실을 보이고 있다[4].

오늘날까지, 분기 예측의 정확도를 높이기 위한 여러 가지 다양한 형태의 연구들이 진행되고 있다. 가장 기본적인 bimodal 예측기에서부터, 최근 들어 제시되는 다중 복합 예측기(Hybrid Predictor)에 이르기 까지 그 방법과 형태도 다양하다. 제안된 여러 분기 예측 기법 가운데 가장 대표적이며, 이후의 분기 예측기 설계에 가장 큰 영향을 준 기법이 Yeh and Patt의 이단계 적응형 분기 예측 기법(Two-level Adaptive Branch Prediction)이다[5]. 이 방식은 각 분기 명령어들 사이에 존재하는 상호 연관성(correlation)을 이용하여 분기 방향을 예측한다. 특히 이와 같은 이단계 적응형 분기 예측 기법의 소개 이후로, 분기 명령어들의 과거 수행 기록인 분기 히스토리(History)는 분기 명령어들의 주소값(PC, Program Counter)과 함께 오늘날의 분기 예측에 있어서 가장 중요한 입력 요소 가운데 하나로 자리 잡았다.

분기 예측에 사용되는 이러한 입력 요소들은, 다양한 형태의 분기 예측기 만큼이나 다양한 방식의 차별적 사용법들이 제안되어 왔으며, 이와 같은 차별성이 각 분기 예측기들의 고유한 특징이 되었다. 이 때의 차별성이란, 분기 예측에 사용되는 입력 요소의 새로운 형태 변화, 독자적인 분기 예측 입력 요소의 추가적 사용, 새로운 방식의 분기 예측 테이블(Branch Prediction Table) 접근 메커니즘, 그리고 분기 예측 테이블로의 접근을 제어하는 차별적인 입력 함수의 사용 등으로 특징지을 수 있다.

본 논문에서는, 이와 같은 분기 예측에 필요한 입력 요소들을 효과적으로 활용하고자 하는 노력의 연장선상에서 명령어 연관성 분석을 통한 가변 입력 gshare 예측기를 제안한다. 우선, 실행 결과를 예측하고자 하는 분기 명령어와 연관된 명령어들의 상호 관계를 파악한다. 이를 바탕으로 명령어 연관성에 기반 한 분기 예측 입력값의 선별적 사용 기법을 소개한다. 이 때, 연관성을 분석하는 방법에 따라 (1) 분기 명령어와 연관된 레지스터 쓰기 연산 명령어에 기반 한 연관성 분석, (2) 분기 명령어의 참조 명령어에 기반 한 연관성 분석, 끝으로 (3) 전술한 두가지 방법의 혼합 기법을 소개하며 해당 방식들이 기존의 방식과 가지는 성능상의 차이를 비교 분석한다.

이하 본 논문의 구성은 다음과 같다. 2절에서는 배경지식에 대해 설명하며, 관련연구로 분기 예측 입력값을 다양하게 조절하고자 하는 기존의 연구들에 대해 소개한다. 3절에서는, 명령어 연관성에 기반 한, 세가지 형태의 분기 예측 입력값의 동적인 조절 알고리즘을 소개하며 그 실행 예를 제시한다. 그리고 4절에서는 모의실험을 통해 제안된 방식과 기존 방식과의 성능상의 차이를 비교 분석하며, 5절에서 결론을 맺는다.

Ⅱ. 배경 지식 및 관련 연구

분기 예측에 있어서 사용되는 대표적 입력값인 분기 히스토리는 기존의 경우 고정된 길이로 사용되었다. 하지만 이전의 여러 연구에서 보여주듯이, 각 분기 명령어가 사용하는 히스토리는 최적의 길이를 가질 수 있으며, 많은 정보, 즉 긴 길이의 히스토리를 사용한다고 항상 최선의 결과를 보이는 것은 아니었다. 분기 명령어의 주소값은 각 분기 명령어를 유일하게 구분 짓는 요소이지만, 분기 히스토리는 분기 명령어들 사이에 존재하는 상호 연관성을 반영하는 요소이기 때문이다. 결국, 다양한 형태로 존재하는 분기 명령어 별 최적의 상호 연관성을 추출하는 것이 문제 해결의 핵심이다.

히스토리 길이를 조절하기 위한 연구로서, DHLF (Dynamic History Length Fitting)[6], Elastic History Buffer를 이용한 히스토리 길이 조절 기법[7], 가변 길이 경로 분기 예측 기법(Variable Length Path Prediction)[8] 등을 들 수 있다. 또한 다중 분기 예측기(Hybrid Branch Predictor)의 경우, 다양한 부속 예측기(Sub-Predictor)들에서 사용되는 입력 히스토리의 길이를 각 예측기 별로 서로 다르게 구성하여 그 결과를 선택적으로 사용(voting)하는 방법[9] 역시 간접적인 형태이긴 하나 히스토리 길이의 조절 효과를 나타낼 수 있다. 한편, 분기 예측에 있어서 명령어들 사이의 연관성 정보를 추출하여 이를 활용하고자 하는 노력도 진행되었다. Thomas 등은 자료 흐름 기반의 분석 방법(Dataflow based Identification)을 사용하여, 충분히 긴 전역

(global) 히스토리 내에서 각 분기 명령어들 사이의 상호 연관성을 효율적으로 반영할 수 있는 방식을 소개 하였다[10]. 저자들은, 현재 예측하고자 하는 분기 명령어에 큰 영향을 주는 분기 명령어들만을 선별적으로 파악하여 그 결과를 반영하는 것이 분기 예측의 정확도 향상에 보다 더 효율적이라고 주장한다. Chen 등은 동적으로 명령어들 사이의 연관성을 추적할 수 있는 새로운 하드웨어 구조를 소개하고, 이를 분기 예측기의 구성에 적용한 구현 사례를 제시하였다[11]. 제안된 방법은 값에 기반을 둔 분기 예측 기법(Value-based Branch Prediction)과 경로에 기반을 둔 분기 예측 기법(Path-based Branch Prediction)의 혼합된 형태 중 하나라 할 수 있다.

본 논문에서는 분기 명령어와 연관된 명령어들 사이에 존재하는 최적의 연관성을 추출해 내기 위해, 이들 사이에 존재하는 상호 연관성을 세가지 서로 다른 방식으로 분석한다. 이를 활용하여 각 분기 명령어 별 최적의 히스토리 길이를 찾고, 이를 수행 시간에 동적으로 조절하여 분기 예측의 정확도 향상에 기여하고자 한다.

III. 명령어 연관성 분석을 통한 가변 입력 예측기 설계

3.1 명령어 연관성 분석

본 논문에서는 제안된 알고리즘의 적용 사례에 대한 설명의 편의를 위해 그림 1에 제시된 예제 제어 흐름도를 기준으로 설명한다. 우선, 그림 1은 Basic Block(BB)들로 구성되어 있으며, 각 Basic Block들은 하나의 분기 명령어와 제어의 흐름을 변경시키지 않는 다수의 명령어들로 구성되어 있다. 그림 1의 마지막 Basic Block인 BB10의 if(R4)가 현재 우리가 예측하고자 하는 분기 명령어이다. if(R4)의 Global 히스토리는 그림 1에서 보여 주듯이 (…TNNTTN) 이다.

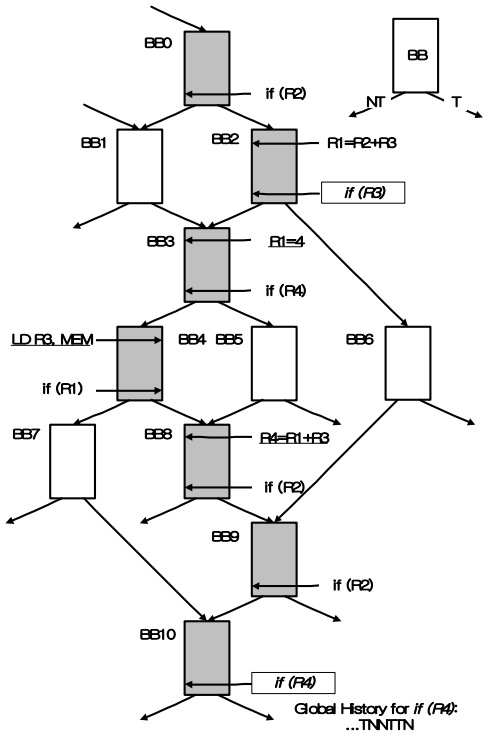


그림 1. 분기 예측을 위한 예제 제어 흐름도
Fig 1. Data Flow Graph for Branch Prediction

한편, 본 논문에서 소개될 세가지 서로 다른 명령어 연관성 추출 알고리즘 가운데 하나로, 그림 2는 레지스터 쓰기 연산에 기반하는 명령어 연관성 분석 방법이다. 참고로 본 논문에서는 그림 2에 제시된 기법을 "RW기반" 알고리즘이라 명명한다. 아울러, 그림 3은 해당 알고리즘을 활용하기 위해 추가적으로 제안된 하드웨어 모듈이다. 주어진 테이블은 실제 프로세서가 지원하는 레지스터 수의 엔트리를 가지며, 각 엔트리들은 분기 예측 테이블인 PHT(Pattern History Table)가 지원하는 최대 히스토리 개수만큼의 비트 폭을 가진다. PHT는 각 분기 명령어 별로 이전 히스토리를 기록하고 있는 테이블이다.

```
if (Conditional Branch Instruction)
{
Length_Indicator <= Br_RDT(Reg_src1) ( | Br_RDT(Reg_src2) |...);
All Br_RDT entry << 1;}

else if (Register Writing Instruction)
Br_RDT(Reg_dest) <= 1 ( | Br_RDT(Reg_src1) | Br_RDT(Reg_src2)
| ...);

( ...) depends on the number of source operands in the branch
instruction.
```

그림 2. RW 기반 알고리즘
Fig 2. Register-Writing Based Algorithm

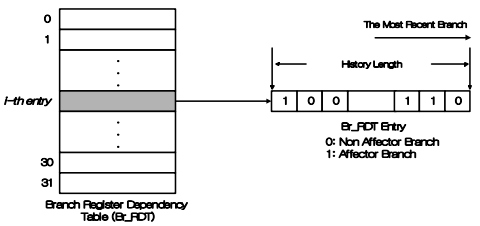


그림 3. 분기 레지스터 종속 테이블
Fig 3. Branch Register Dependency Table

그림 2에 제시된 연관성 분석 알고리즘은 다음과 같은 사실을 기반으로 제안되었다. 우선, 그림 1에 나타나 있는 예제 제어 흐름도의 명령어 연관성을 분석해 보면, BB10의 분기 명령어 if (R4)의 분기 결과는 레지스터 R4의 값과 밀접한 관련이 있다는 것을 알 수 있다. 또한 R4의 값은 BB8에서 R1과 R3의 값에 의해 결정되며(R4=R1+R3), 연속해서 R1의 값은 BB3에서 재결정된다 (R1=4). 물론 R1의 경우 BB2의 R2와 R3에 의해 다시 재결정되지만 (R1=R2+R3), BB3의 R1=4 연산에 의해 새롭게 갱신되었기 때문에 이전의 명령어 연관성과는 단절된다. 따라서, BB10의 if (R4) 분기 입장에서는 R1의 값이 BB3에서 결정되었다고 할 수 있다. 한편 R4와 연관되어 있는 BB8에 나타난 또 다른 레지스터인 R3은, BB4의 LD R3, MEM 연산에 의해 새롭게 갱신됨을 알 수 있으며, 주어진 그림 1에서는 더 이상 R3과 연관된 명령어 연관성을 발견할 수 없다.

이처럼 BB10의 if (R4)의 분기 예측은 BB3의 "R1=4"와 BB4의 "LD R3, MEM"과 같은 명령어 연관성을 가지는 Basic Block의 해당 레지스터 값에 의해 크게 좌우되며, 결국 BB3과 BB4의 R1=4와 LD R3, MEM의 수행 여부가 BB10의 if (R4)의 예측에 결정적이라 할 수 있다. 이를 Basic Block들의 관점에서 살펴보면, BB3과 BB4의 수행 여부가 BB10에 나타난 분기 예측에 큰 영향을 미치게 됨을 의미한다. 더 나아가, 이와 같은 BB3의 수행 여부는 BB2의 if (R3) 분기의 분기 예측 결과에, 그리고 BB4의 수행 여부는 BB3에 포함된 if (R4)분기의 분기 예측 결과에 큰 영향을 받는다는 것을 알 수 있다. 하지만 BB4의 수행에 영향을 주는 BB3의 수행 여부는 결국 BB2의 수행 여부에 따라 재결정되기 때문에, 분석 결과 보다 더 멀리 떨어져 있는 Basic Block에 근본적인 영향을 받는다고 할 수 있다.

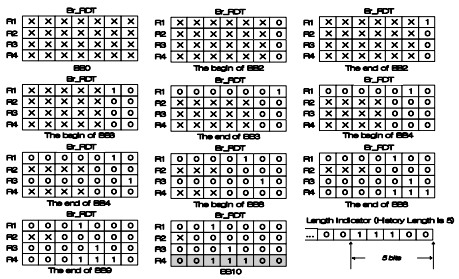


그림 4. RW 기반 알고리즘 구성 예제
Fig 4. RW based Algorithm Example

그림 4는 그림 3에서 제안된 하드웨어 모듈을 활용하여 그림 2에 소개된 알고리즘을 각 단계별로 적용하였을 경우데 대한 결과를 보여준다. 적용 결과 RW 기반 알고리즘에 의해 동적으로 조절된 히스토리 길이는 5가 됨을 알 수 있다.

한편, 명령어 연관성을 분석하는 두 번째 방식으로 본 논문에서는 분기 명령어의 참조 명령어에 기반한 명령어 연관성 분석 알고리즘 제안한다. 본 논문은 이를 BR 기반 알고리즘이라 명명하며, 그림 5에 해당 알고리즘이 소개되어 있다.

```
if (Conditional Branch Instruction) {
Length_Indicator <= Br_RDT(Reg_src1) ( | Br_RDT(Reg_src2) |
... );
All Br_RDT entry << 1;

Br_RDT(Reg_src1) <= 1 | Br_RDT(Reg_src1) ;
Br_RDT(Reg_src2) <= 1 | Br_RDT(Reg_src2);
(...)
}

else if (Register Writing Instruction)
Br_RDT(Reg_dest) <= Br_RDT(Reg_src1) ( | Br_RDT(Reg_src2)
| ... );

( ... ) depends on the number of source operands in the
branch instruction.
```

그림 5. BR 기반 알고리즘
Fig 5. Branch-Reading based Algorithm

그림 5에 제시된 알고리즘은 그림 1에 소개된 예제 제어 흐름도에서 다음과 같은 분석을 통해 제안되었다. 우선, 분기 명령어 if (R4)의 분기 결과는 레지스터 R4의 값과 밀접한 관련이 있다는 것을 알 수 있다. 그리고 이러한 R4의 값은 BB8에서 R1과 R3의 값에 의해 결정(R4=R1+R3)된다. 이상은 앞서 언급한 바와 동일하다. 한편, BB10의 if (R4) 분기에 영향을 주는 레지스터 R1과 R3은 BB2의 if (R3) 분기 명령어와 BB4의 if (R1) 분기 명령어에서 각각 사용된다. 따라서, 기본적으로 이들 분기 명령어들은 BB10의 if (R4) 분기 명령어와 강한 상호 연관성을 가질 수 있다. 즉, R1과 R3의 실제 값에 따라서 BB2의 if (R3)과 BB4의 if (R1)의 분기 방향이 결정되며, 해당 두 분기 명령어의 분기 결과에 따라 BB8의 R4값의 변경 여부가 결정된다. 그리고 그 결과에 따라 결정된 R4의 값은 BB10의 if (R4)에 의해 분기 예측에 사용된다. 즉, R4와 상호 연관이 있는 레지스터들인 R1과 R3을 사용하는 분기 명령어들이 R4를 사용하는 분기 명령어와 강한 상호 연관성을 가질 수 있다. 하지만, 두 레지스터 R1과 R3이 BB10의 분기 명령어인 if (R4) 분기와 계속적으로 명령어 연관성을 가지는 것은 아니다. 우선 주어진 제어 흐름도에서, BB2의 if (R3) 분기에서 사용된 레지스터 R3의 값은 BB4의 LD R3, MEM 연산에 의해 이전과 존재하던 명령어 연관성이 단절되고 새롭게 그 값이 정의된다는 것을 알 수 있다. 따라서 BB2의 if (R3) 분기의 레지스터 R3의 값은 사실상 BB10의 if (R4) 분기와 상호 연관성이 결여된다. 동일한 방법으로, BB4의 if (R1) 분기 명령어에서도 계속적으로 명령어 연관성을 추적하여 거슬러 올라 갈 수 있다. 주어진 예제에서는 BB3의 R1=4 연산이 상호 명령어 연관성을 가지는 명령어이다. 하지만 BB3의 R1=4의 연산에 있어서는, R1의 값이 다른 레지스터들과 연관된 값을 가지는 것이 아니라 독자적으로 상수 4에 의해 그 값이 새롭게 재정의 되었다. 즉 레지스터 R1의 경우, 이전 Basic Block과의 연관성이 BB3의 R1=4에 의해 단절되었다는 것을 알 수 있다. 이상의 사실을 통해 BB10의 if (R4) 분기는, 분기 명령어들 사이에 사용된 레지스터들의 명령어 연관성 관점에서 분석한 결과, R1과 R3 레지스터와 상호 연관성을 가지며 BB4의 if (R1) 분기와 특히 강한 상호 연관성이 있다는 사실을 알 수 있다.

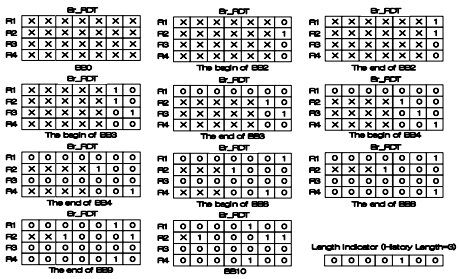


그림 6. BR 기반 알고리즘 구성 예제
Fig 6. BR based Algorithm Example

지금까지 분석한 방법을 이용하여 현재 예측하고자 하는 분기 명령어와 강하게 연관되어 있는 레지스터를 파악할 수 있다. 그림 1의 흐름도에서 이상의 분석 방식을 적용한 예제가 그림 6에 제시되어 있다. 특히 언급된 방식은, 레지스터들 사이의 명령어 연관성을 분기 명령어들이 사용하는 소스 레지스터들 사이에서 추적해 나가는데 그 특징이 있다고 할 수 있다. 즉, 모든 레지스터들 사이에 존재하는 쓰기 연산에 대해 명령어 연관성을 추적해 나가는 방식 대신, 분기 명령어가 실제 사용하는 레지스터들 사이의 명령어 연관성만을 추적해 나가는 방식이다. 그림 6에서와 같이, 제안된 BR 기반 방식의 동적으로 조절된 히스토리 길이는 3이 됨을 알 수 있다.

끝으로, 세 번째 제안될 명령어 연관성 분석 알고리즘은 다음과 같다. 이는 앞서 제시된 두 가지 알고리즘의 상호 결합된 형태이다. 다시 말해, 분기 명령어의 소스 오퍼랜드에 영향을 주는 레지스터를 파악하고 해당 레지스터에 대한 쓰기 연산을 포함하고 있는 Basic Block으로의 분기를 가능하게 하는 직전 분기 명령어를 찾아내는 기법(RW 기반)과 함께, 분기 명령어가 가지고 있는 소스 레지스터들 사이의 명령어 연관성을 추적해 가면서 해당 소스 레지스터를 사용하는 이전 분기 명령어들의 명령어 연관성을 추적하는 기법(BR 기반)을 상호 결합한다. 본 논문에서는 이를 Merged 기법이라 명명한다. 그림 7에 해당 알고리즘이 소개되어 있으며, 그림 8에 해당 알고리즘을 수행 시켰을 경우의 구성 예제가 나타나 있다.

```
if (Conditional Branch Instruction) {
Length_Indicator <= Br_RDT(Reg_src1) ( | Br_RDT(Reg_src2) |
... );
All Br_RDT entry << 1;

Br_RDT(Reg_src1) <= 1 | Br_RDT(Reg_src1) ;
Br_RDT(Reg_src2) <= 1 | Br_RDT(Reg_src2);
(…)
}

else if (Register Writing Instruction)
Br_RDT(Reg_dest) <= 1 | ( Br_RDT(Reg_src1) |
Br_RDT(Reg_src2) | ... );

( … ) depends on the number of source operands in the
branch instruction.
```

그림 7. Merged 알고리즘
Fig 7. Merged Algorithm

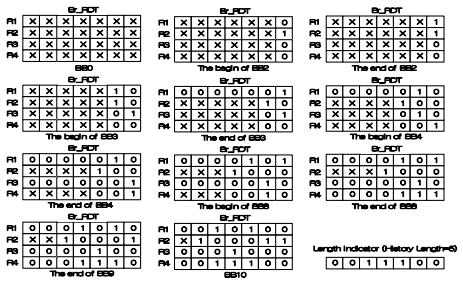


그림 8. Merged 알고리즘 구성 예제
Fig 8. Merged Algorithm Example

3.2 가변 입력 예측기 설계

이 절에서는 제안된 세가지 가변 입력 알고리즘을 적용할 수 있는 구현 사례로서, 가변 입력 gshare 예측기를 제안한다. 그림 9는 가변 입력 기능을 지원하는 gshare 기법으로, 해당 분기 예측기의 회로 레벨의 구현도를 보여준다. 주어진 구현도는 분기의 주소값으로 10 비트, 히스토리 값으로 8 비트를 사용하는 일례이다. 우선, 그림 9의 상단은 기존의 gshare 예측기의 회로 레벨 구현도이다[12].

이에 비해 그림 9의 하단은, 가변 입력 히스토리가 허용되는 gshare 예측기를 보여준다. 기본적으로 xor 함수의 입력 가운데 하나인 Branch 히스토리를 선별적으로 반영하기 위해, 입력 함수의 경로상에 and 함수를 추가한 형태이다. 그리고 Length_Indicator의 값을 제어하는 Global History Enabler (GHE)가 1의 입력 값을 가지는 Length_Indicator의 최상위 MSB 위치를 기준으로 하위 비트를 모두 1로 변경시키게 된다. 그리고 이는 and 함수의 입력으로 반영된다. 즉, GHE에 의해 0의 입력이 출력된 경우, 히스토리 값이 주소값과 xor 되는 것을 방지하는 역할을 하여, 해당 영역은 분기의 주소값만이 사용되게 한다.

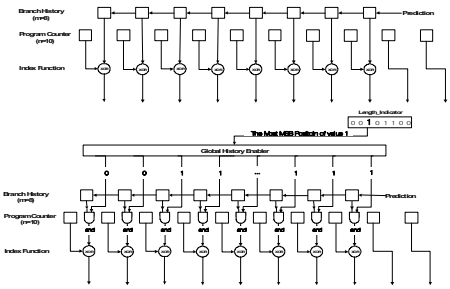


그림 9. 가변 입력 gshare 예측기
Fig 9. Variable Imput Gshare Predictor

주어진 그림 9의 동작 메커니즘은 다음과 같다. 임의의 분기 예측 시점 c 에서 분기 명령어 B_i 의 분기 결과를 예측한다고 할 때, 분기 명령어 B_i 의 예측을 위한 현재 Global 히스토리 Register의 내용을 GHR_i 라 하자. 이때의 GHR_i 는 과거 m 개의동적인 수행 경로상의 다음과 같은 분기 명령어들의 히스토리를 가지게 된다.

$$GHR_i = H_{i,c-m}\, H_{i,c-m+1} \cdots H_{i,c-2}\, H_{i,c-1}$$

이를 히스토리 길이 조절 함수인 ϕ 대입하여 다음과 같이 변형된 히스토리 길이인 H_c를 얻는다. 함수 ϕ의 또 다른 입력 값인 $Length_indicator_i$는 그림4, 그림 6, 그리고 그림 8에 제시된 알고리즘의 수행 결과얻어진 값이다.

$$H_c = \phi(GHR_i, Length\_indicator_i)$$

또한, 분기 명령어 B_i는 다음과 같은 주소값 n $(n \geq m)$비트를 가지게 된다.

$$A_{i,n}\, A_{i,n-1} \ldots A_{i,2}\, A_{i,1}$$

이와 같이 주어진 두 입력 요소인 H_c와 $A_{i,n}\, A_{i,n-1} \cdots A_{i,2}\, A_{i,1}$은 PHT 인덱스 결정 함수인 ω에 대입되어 다음과 같은 Q_c를 얻게 된다.

$$Q_c = \omega\ (H_c, A_{i,n}\, A_{i,n-1} \ldots A_{i,2}\, A_{i,1})$$

그 결과에 따라, PHT_Q 엔트리의 카운터 값 S_c 는 분기 명령어 B_i의 분기 예측에 사용되게 된다. 이때의 예측 결과 Z_c는 분기 예측 결정 함수 λ에 의해 다음과 같이 결정된다.

$$Z_c = \lambda(S_C)$$

이와 같이 분기 예측이 수행되고 난 이후에는, 새로운 분기 히스토리 기록인 $H_{i,c}$가 GHR_i 상에서 LSB 위치에 추가되게 되고, 또한 PHT_{Qi}의 해당 엔트리 값을 갱신하게 된다. 그 후, GHR_i는 $H_{i,c-m+1} H_{i,c-m+2} \cdots H_{i,c-1} H_{i,c}$ 값을 가지게 되며, 이는 다음 분기 명령어 B_j의 예측에 사용되게 된다. 또한 분기 예측 결정 함수에 의해 사용되었던 카운터 값 S_C 는 카운터의 상태 전환 함수에 따라 다음 상태인 S_{C+1} 가 된다. 이때의 카운터 변환 함수를 δ라고 한다면, 새로운 상태는 함수 δ에 의해 다음과 같이 결정 된다. 즉, 이전 상태 S_C 와 분기 명령어 B_i 의 분기 결과인 $H_{i,c}$의 조합으로 결정된다.

$$S_{c+1} = \delta(S_C, H_{i,c})$$

끝으로, 본 논문에서 제안된 기법의 추가 하드웨어 요구량에 대해 설명하면 다음과 같다. 제안된 기법의 주된 하드웨어 복잡도는 그림 3에서 제시된 Br_RDT에서 기인한다. 그림 9의 GHE는 조합 회로 부분으로 메모리적 요소를 포함하는 순차 회로 부분과 비교할 때, 사실상 무시될 만한 수준의 하드웨어 요구량을 가진다. 한편, 그림 3에 제시된 Br_RDT 모듈은 추가적인 또 하나의 레지스터 파일 형태로 구현된다. 이를 하드웨어 요구량의 관점에서, 보다 더 세분화 하여 나타낸 형태가 그림 10에 제시되어 있다. 전체적 구조는 Br_RDT의 내용을 기록하는 메모리 부분과 멀티플렉서(MUX)와 디코더(decoder) 등으로 이루어져 있는 제어 회로 부분으로 이분화 할 수 있다. 이와 같은 그림 10을 바탕으로, Br_RDT를 구성하기 위한 하드웨어 복잡도는 식 (1)과 같이 묘사될 수 있다.

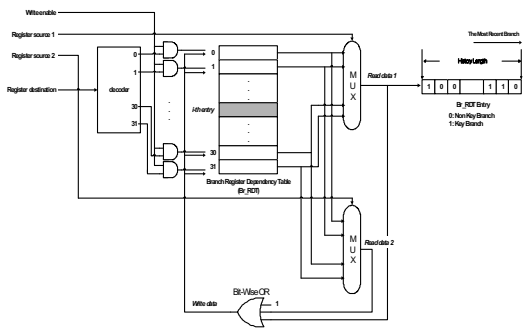


그림 10. Br_RDT 구현도
Fig 10. Br_RDT Implementation Diagram

$$\begin{aligned}
&Cost(Br_RDT)\\
&= Cost_{memory_element} + Cost_{control_logic}\\
&= Cost_{memory_element} + (Cost_{read_port} + Cost_{operation} + Cost_{write_port})\\
&= (N_{architectural_register} \times history_length)\\
&\quad + \{(2 \times N_{architectural_register} - by - 1_Mux + C_{read_port})\\
&\quad + C_{bit-wise-or}\\
&\quad + (N_{architectural_register} - to - 1_Decoder + C_{write_port})\}
\end{aligned}$$

그림 11. Br_RDT 하드웨어 복잡도
Fig 11. Br_RDT Hardware Complexity

주어진 Br_RDT의 주된 하드웨어 복잡도는, 그림 11에서 보이는 바와 같이, 이를 구성하기 위한 메모리 부분과 제어 로직의 구성 비용으로 구분될 수 있다. 우선, 메모리 부분의 구성 비용은 history_length 크기의 레지스터 Narchitectural_register개가 필요하게 된다. 한편, 제어 로직의 구성 비용은 Br_RDT를 레지스터 파일로 구성하기 위한 Costread_port와 Costwrite_port 그리고 실질적인 알고리즘의 수행을 위해 지원되어야 하는 Costoperation부분으로 구분될 수 있다. Costread_port의 경우는 2 포트로 구성되어야 하며, 특정 레지스터 값을 추출하기 위해 mux 가 사용되게 된다. 다음으로, Costoperation은, 알고리즘에서 요구되는 주요 연산인, bit-wiseor 연산을 위한 기본 비용 상수 Costbit_wise_or만큼의 추가 복잡도를 요구하게 된다. 끝으로, Costwrite_port의 경우는 알고리즘 수행 연산의 결과를 특정 레지스터에 대입하여 이를 다시 Br_RDT에 반영하기 위한 디코더가 필요하게 된다.

특히, 주어진 제어 로직의 복잡도 가운데 Costread_port와 Costwrite_port의 경우는 레지스터 파일의 구성을 위해 요구되는 일반적으로 하드웨어 복잡도와 동일하다. 따라서, 본 논문에서 제안된 Br_RDT 구조에서 요구하는 추가적인 고유 복잡도는 Costoperation 부분과 Costmemory_element부분이라 할 수 있으며, 이 가운데에서는 상대적으로 Costmemory element 부분이 하드웨어 복잡도의 지배적 요소라 할 수 있다. 이와 같은 Costmemory_element 부분에 대한 하드웨어 복잡도를 예를 들어 설명하면 다음과 같다. 가령, 2 비트 카운터를 가지는 1024개 엔트리의 PHT를 사용하며, 프로세서가 요구하는 구조 레지스터의 개수가 32개라고 가정하면, 이 경우, 2048 비트(1024 X 2 비트)의 PHT에 추가적으로 320 비트(32 registers X 10 histories)의 하드웨어 복잡도가 요구된다. 이를 그림 10에서 제시된 바와 같이, 메모리 구성 요소와 제어 회로 구성 요소에서 각각 사용되는 실제 논리 게이트의 수(gate count)적인 측면에서 분석해 보아도 유사한 복잡도를 유추할 수 있다. 가령, 2 비트 카운터로 구성된 1024개 엔트리의 PHT의 경우 대략 12,288 게이트 수를 필요로 한다. 한편, Br_RDT를 구성하기 위한 게이트 수는 메모리 구성 요소에 대략 1,536 게이트 수,

제어 회로 및 기타 구성 요소에 1,197 게이트 수가 필요하여, 전체적으로 22.2%의 추가 복잡도를 요구하게 된다.

이와 같은 추가 복잡도는 비용 효율적(Cost-Effective)이라는 것을 확인할 수 있으며, PHT의 크기가 증가할수록 상대적인 복잡도의 비율은 더욱 줄어들게 된다. 이와 유사한 관련 연구의 추가 하드웨어 요구량과 비교해 볼 때도, 본 논문에서 제시된 기법의 추가 복잡도는 적당하다(modest)고 할 수 있다[10][11]. 또한, CMOS 및 Gate-Level의 트랜지스터 개발 기술의 발전과 함께, 분기 예측기에 투자되는 하드웨어 자원이 점차 늘어나는 추세라는 점을 함께 감안하면, 이 같은 추가 복잡도의 상대적 비율은 추후 보다 더 감소하리라 예상된다.

Ⅳ. 성능 평가

이 절에서는 본 논문에서 제안된 세가지 알고리즘에 대한 성능 분석 및 가변 입력 gshare 예측기에 대한 성능 타당성을 검증하기 위해 모의실험을 수행한다. 우선 모의실험에서 사용될 실험 환경 및 벤치마크 프로그램을 소개하며, 이어서 기존 방식과 제안된 방식과의 성능을 비교 분석한다.

4.1 실험 환경 및 벤치마크 프로그램

본 논문에서의 모의실험은 구동 기반 시뮬레이터인 SimpleScalar로 진행되었다[13]. 이벤트 구동형 시뮬레이터(Event-Driven Simulator)인 SimpleScalar는 빠른 모의실험과 높은 정확도의 결과를 보장하는 강력한 실험 환경으로, 컴퓨터 구조적인 측면에서의 성능 측정을 위해 세계적으로 널리 사용되는 대표적인 실험 환경이다. 표 1에 실험 환경을 요약하였다.

표 1. 모의 실험 인자
Table 1. Simulation Parameters

Parameter	Value
Fetch Queue	4 entries
Fetch, Decode Width	4 instructions
ROB entries	16entries
LSQ entries	8entries
Functional Units(integer)	4 ALUs, 1 Mult/Div
Functional Units(floating point)	4 ALUs, 1 Mult/Div
Instruction & Data TLB	64(16 X 4-way) entries & 128(32 X 4-way)entries, 4K pages, 30 cycle
Predictor Style	gshare
BTB entries	2048(512 ✕ 4-way) entries
RAS entries	8 entries
Extra Branch Miss-prediction Penalty	3 cycles
L1 I-Cache	16 KB, direct map, 32B line, 1 cycle
L1 D-Cache	16 KB, 4-way, 32B line, 1 cycle
L2 Cache(unified)	256 KB, 4-way, 64B line, 6 cycles
Memory Latency	first_chunk=18 cycles, inter_chunk=2 cycles

한편, 모의실험에 사용된 벤치마크 프로그램은 SPEC에서 제공되는 CPU 성능 측정 프로그램인 SPEC CINT2000 프로그램들로써, 이 가운데 무작위로 선별된 정수형 프로그램들이다[14]. 일반적으로 SPEC에서 제공하는 CPU 성능 평가 프로그램은 정수형 프로그램인 CINT와 실수형 프로그램인 CFP로 구분되는데, CFP 프로그램의 경우는 과학 계산 형태의 응용 프로그램이 주를 이루며, 이들은 매우 정형화되어 있는 형태의 코드를 가진다. 이 같은 경우, 분기 예측의 정확도가 매우 높게 나타나기 때문에, 분기 예측의 개선으로 인한 성능 향상의 정도를 효율적으로 관찰하기에는 곤란하다. 따라서 CFP 프로그램들은 분기 예측과 관련된 연구에서 일반적으로 제외된다.

4.2 실험 결과

이절에서는 앞서 소개된 가변 입력 gshare 예측기를 위한 세가지 알고리즘의 성능상의 차이를 비교 분석한다. 우선 실험 결과에서 소개될 비교 대상 결과값들은 다음과 같다. 우선, 프로파일링을 통해 사전에 얻어진 각 응용 프로그램별 최적의 입력에 대한 분기 예측 실패율인 Profiled Best와 주어진 PHT의 크기에 맞게 사용된 기존의 고정 길이 방식인 Fixed Length를 비교 대상으로 삼는다. 그리고 본 논문에서 소개된 세가지 길이 조절(Length Adjustment) 알고리즘으로 그림 2의 Length Adjustment (RW), 그림 5의 Length Adjustment (BR), 그리고 그림 7의 Merged (RW + BR) 기법을 각각 비교한다. 각각의 경우에 있어서, PHT는 0.5K에서 4K까지의 크기

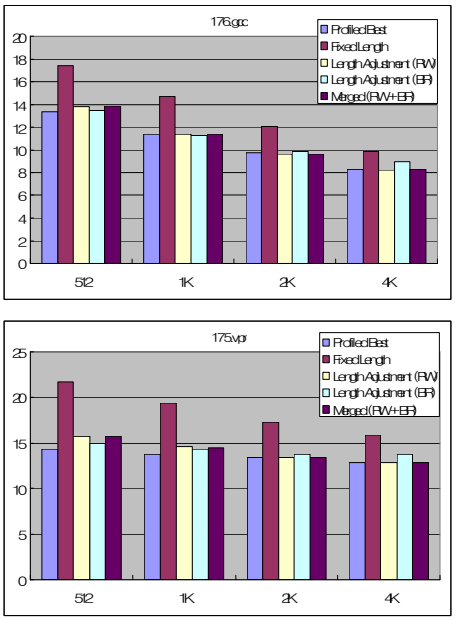
변화를 주었으며, 이들의 분기 예측 실패율(%)을 각각의 응용 프로그램 별로 나타내었다. 아울러, 그림 13은 모든 응용 프로그램들에 대한 PHT 크기 별 평균 결과이다.

그림 12에서 주어진 실험 결과를 분석하면 다음과 같다. 우선 평균적인 관점에서 보았을 때, 본 논문에서 제시된 세가지 방식이 기존의 Fixed Length 기법과 비교하여 성능상 모두 우위에 있다는 것을 확인 할 수 있다. 게다가, RW 기반 방식과 더불어 Merged 방식의 경우에 있어서는, 사실상 프로파일링을 통해 확인한 최적의 결과라 할 수 있는 Profiled Best의 결과 값과 비교해서 아주 근소한 성능상의 차이를 나타내며, 경우에 따라서는 오히려 더 우수한 결과를 보인다는 사실을 확인 할 수 있다.

Profiled Best의 결과값은 어디까지나 실험적으로만 얻을 수 있는 최적의 결과값으로, 실제 시스템으로의 구현은 불가능한 어디까지나 최적치를 위한 비교대상으로만 사용될 수 있는 값이다. 이러한 점을 감안할 때, 본 논문에서 제안된 기법들이 제공하는 성능 향상의 정도가 최적의 결과와 사실상 일치하거나 보다 더 우수 할 수 있다는 사실은 의미하는 바가 크다. 기존의 관련 연구도 그들의 제안에 대한 성능 평가를 사전 프로파일링에 의한 최적의 값과 비교하여 어느 정도 최적치에 유사하게 근접하는가를 보이고 있으나, 기존의 어떠한 관련 연구도 최적의 결과치보다 우수한 결과를 생성해 내지는 못 하였다. 특히, 최적의 결과보다 더 우수할 수 있는 결과를 제공한 경우는 본 논문에서 제안된 기법이 각 분기 명령어 별로의 가변 입력을 제어하기 때문이다. 기존의 관련 연구에서는 가변 입력에 대한 제어를 각 응용 프로그램별로 제공한 것에 비해 보다 더 작은 단위인 응용 프로그램내부의 분기 명령어 별로 가변 입력에 대한 제어를 수행 했기 때문으로 판단된다[6][7].

한편 전체적으로 그림 2와 그림 5에 제시된 두 가지 방식에 있어서는, RW 기반 방식이 BR 기반 방식과 비교하여 다소간 성능상의 우위에 있다는 것을 알 수 있다. 하지만 각각의 결과는 응용 프로그램 별로 상이하게 나타난다. 세부적으로 분석해 보면 다음과 같다. 평균 결과를 통해서도 확인할 수 있듯이, 응용 프로그램 별로도 전체적으로 RW 기반 방식이 우위에 있는 경우가 많다. 181.mcf, 197.parser, 254.gap 그리고 256.bzip2가 이에 해당하는 응용들이다. 이들 응용에 있어서는, RW 기반 방식이 전반적으로 Fixed Length 기법보다 성능면에서 월등하게 우위에 있으며, Profiled Best 기법과 비교하여 매우 근소한 차이를 보인다는 것을 알 수 있다.한편, 175.vpr, 176.gcc, 186.crafty 그리고 255.vortex의 경우는 BR 기반 방식이 성능면에서 RW 기반 방식보다 우월하다. 하지만 이들 응용에 있어서도 공통적으로 나타난 결과는, PHT의 크기가 증가 할수록 RW 기반 방식이 성능면에서 다시 우월해 진다는 사실이다. 175.vpr, 176.gcc, 186.crafty는 2K 크기에서부터, 그리고 255.vortex의 경우는 4K 크기에서부터 RW 기반 방식의 결과가 다시 우월해 진다. 252.eon의 경우는 BR 기반 방식이 Fixed Length 기법과 비교해서도 성능이 매우 좋지 않게 나온 유일한 응용 프로그램이다.

이 같은 결과는 그림 4와 그림 6에 제시된 각각의 Br_RDT의 구성 예제를 통해서도 알 수 있듯이, 대체적으로 BR 기반 방식의 경우, 상대적으로 RW 기반 방식과 비교하여, 조절되는 히스토리의 길이가 짧다는 것을 알 수 있다. 이는 BR 기반 방식이 PHT의 크기가 작을 경우는 상대적으로 조절되는 히스토리 길이가 효율적일 수 있으나, PHT의 크기가 큰 경우에는 이 같은 방식이 효과적이지 못할 수 있다는 것을 의미한다. 이는 Br 기반 방식이 상대적으로 보다 더 적은 규모의 분기 예측 모듈이 사용되는 내장형 환경에 유리하게 적용될 수 있다는 것을 의미한다. 즉, 각 응용 프로그램의 수행 특성에 따른 두가지 방식의 성능 차이 이외에, BR 기반 방식과 RW 기반 방식 각각이 가질 수 있는 유리한 시스템 환경 설정의 힌트를 내포하고 있는 실험 결과라 할 수 있다.



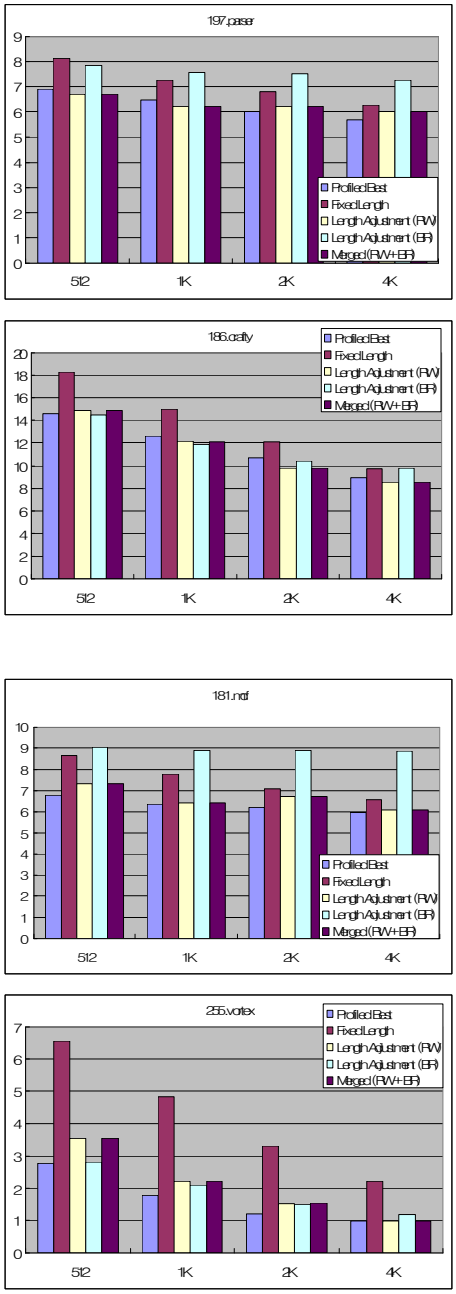


그림 12. 각 알고리즘별 분기 예측 실패율 (%)
Fig 12. Misprediction Rate (%) for Each Algorithm

한편, Merged 기법은 상대적으로 두가지 기법의 조절 결과로 생성되는 히스토리 가운데 조금 더 긴 히스토리를 생성해 내는 경향이 있다. 일반적으로 히스토리 테이블의 크기가 큰 경우 지나치게 짧은 히스토리는 오히려 분기 예측의 정확도를 감소시킬 수 있으며, Merged 기법은 이를 완충시켜 주는 역할을 한다.

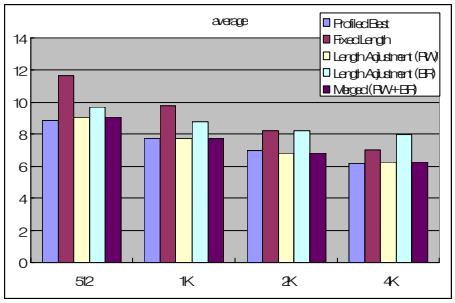


그림 13. 분기 예측 실패율 (평균)
Fig 13. Misprediction Rate (Average)

각 응용 프로그램별 실험 결과에 대한 관찰과 함께, 그림 13은 전술한 바와 같이 전체 응용 프로그램들에 대한 평균 결과를 분기 예측 테이블의 크기에 따라 정리하여 제시하였다. 결론적으로 보았을 때, BR 기반 방식은 대부분 Fixed Length 기법보다 우수한 결과를 보였으며, RW 기반 방식과 Merged 방식의 경우에 있어서는 Profiled Best 기법보다도 경우에 따라서 우수한 결과를 보였다. 각 기법 별로는, 응용에 따라서 BR 기반 기법이 우수한 경우도 있었으나, PHT의 크기가 증가 할수록 RW 기반 기법이 다시 우수한 결과를 보이기 시작하였다. 전반적으로 RW 기반 방식이 우월한 경우가 많았으며, 비교적 큰 크기의 PHT가 사용되는 환경에서는 모든 응용 프로그램에 있어서 RW 기반 방식과 Merged 기반 방식이 우월한 결과를 보였다. 끝으로, 임베디드 시스템과 같은 환경 하에서는 일반적으로 응용 프로그램들이 몇몇개로 국한되어 반복 수행되는 경향이 있다. 이러한 경우, 해당 응용 프로그램에서 보다 더 우수한 결과를 보이는 알고리즘을 사전 모의실험을 통해 선택하여 사용하는 것도 가능하리라 판단된다.

V. 결론

본 논문에서는 분기 예측의 정확도 향상을 위한 기법 가운데 하나로, 분기 명령어에 대한 입력 요소들의 효율적인 활용 및 관리 방법을 제안하였다. 우선, 분기 예측에 있어서 사용되는 대표적인 입력 요소인 분기 히스토리를 각 분기 명령어별로 조절 할 수 있는 가변 입력 히스토리 조절 알고리즘을 소개하였다. 제안된 기법은, 분기 명령어와 연관된 레지스터들 사이의 명령어 연관성을 추적하여, 최종적으로 관련이 있는 레지스터를 포함하도록 유도하는 분기를 파악한 후, 해당 분기 명령어들의 히스토리들을 선별적으로 사용하게 해 주는 방식이었다. 또한 이를 위해 본 논문에서는 대표적으로 사용되는 분기 예측 방식인 gshare 예측기를 변형한 가변 입력 gshare 예측기를 제안하였고, 이를 지원하기 위한 하드웨어 모듈을 제시하였다. 제안된 세가지 기법은 각각 레지스터 쓰기 연산과 연관된 분기 명령어의 명령어 연관성을 추적하는 기법, 분기 명령어가 사용하는 레지스터들의 명령어 연관성을 추적하는 기법, 그리고 이를 상호 결합한 형태의 기법이었다. 본 논문에서는 제안된 세가지 형태의 가변 입력 히스토리 조절 기법에 대한 성능을 모의 실험을 통해 상호 비교 분석 하였다. 전체적인 실험 결과 제안된 방식은, 기존의 고정 길이를 사용하는 방식에 비해 우수한 성능 향상을 가져왔으며, 응용 프로그램에 따라서는 사전 프로파일(prior-profiling)을 통해 알아낸 사실상의 최적 입력 히스토리에 대한 실험 결과와 비교해서도 성능상의 향상을 보였다.

참고문헌

[1] Patterson, D. A., and Hennessy, J. L. "Computer architecture: a quantitative approach" 4th edition, Morgan Kaufman, 2007

[2] J. Kwak and C. Jhon, "Recovery Logics for Speculative Update Global and Local Branch History", LNCS Vol. 4263, pp. 258-266, 2006.

[3] K. Skadron, M. Martonosi, and D. Clark. "Speculative updates of local and global branch history: A quantitative analysis", JILP, vol. 2, Jan. 2000.

[4] E. Sprangle and D. Carmean. "Increasing processor performance by implementing deeper pipelines". 29th Int'l Symp. on Computer Architecture, pp. 25-34, 2002.

[5] Yeh and Patt, "Alternative implementations of two-level adaptive branch prediction," In Proc. of the 19th ISCA, pp. 124-134, May, 1992.

[6] T. Juan, S. Sanjeevan, and J. J. Navarro, "Dynamic history length fitting: A third level of adaptivity for branch prediction", In Proc. 25th Int'l Symp. on Computer Architecture, pp. 155-166, 1998.

[7] M.-D. Tarlescu, K. B. Theobald, and G. R. Gao, "Elastic history buffer: A low-cost method to improve branch prediction accuracy", In Proc. Int'l Conf. on Computer Design, pp. 82-87, 1997.

[8] J. Stark, M. Evers, and Y. N. Patt, "Variable length path branch prediction", In Proc. 8th

Int'l Conf. on Architectural Support for Programming Languages and Operating Systems, pp.170-179, 1998.
〔9〕 A. Seznec, S. Felix, V. Krishnan, and Y. Sazeid`es. "Design tradeoffs for the ev8 branch predictor", In Proc. of the 29th ISCA, pp. 295-306, May 2002.
〔10〕 R. Thomas, M. Franklin, C. Wilkerson and J. Stark, "Improving Branch Prediction By Dynamic Dataflow-based Identification of Correlated Branches From a Large Global History", In Proc. of the International Symposium on Computer Architecture, pp. 314-323, 2003.
〔11〕 L. Chen, S.Dropsho, and D. H. Albonesi, "Dynamic data dependence tracking and its application to branch prediction", In Proc. 7th Int'l Symp. on High Performance Computer Architecture, pp. 65-76, 2003.
〔12〕 McFarling, S., "Combining branch predictors. Tech. Rep. TN-36m", Digital Western Research Lab., June, 1993.
〔13〕 SimpleScalar LLC, http://www.simplescalar.com/
〔14〕 SPEC CPU2000 Benchmarks, http://www.specbench.org.

저 자 소 개

곽 종 욱 (Kwak, Jong Wook)

1998년 경북대학교 컴퓨터공학과 졸업(학사)
2001년 서울대학교 대학원 컴퓨터공학과(공학석사)
2006년 서울대학교 대학원 전기컴퓨터공학부(공학박사)
2006년 - 2007년 삼성전자 SOC 연구소 책임 연구원
2007년 - 현 재 영남대학교 전자정보공학부 전임교수
관심분야 : 컴퓨터 구조, 내장형 시스템, 고성능 컴퓨팅