머신러닝 기법을 적용한 하드웨어 데이터 프리페치 기법 구현

Implementation of Hardware Data Prefetching Technique Using Machine Learning Technique

저자
(Authors)

송경환, 김강희, 최상방
KyungHwan Song, KangHee Kim, SangBang Choi

출처
(Source)

전자공학회논문지 56(3), 2019.3, 11-23(13 pages)
Journal of the Institute of Electronics and Information Engineers 56(3), 2019.3, 11-23(13 pages)

발행처
(Publisher)

대한전자공학회
The Institute of Electronics and Information Engineers

URL

http://www.dbpia.co.kr/journal/articleDetail?nodeId=NODE08000438

APA Style

송경환, 김강희, 최상방 (2019). 머신러닝 기법을 적용한 하드웨어 데이터 프리페치 기법 구현. 전자공학회논문지, 56(3), 11-23

Journal of The Institute of Electronics and Information Engineers Vol.56, NO.3, March 2019 https://doi.org/10.5573/ieie.2019.56.3.11
ISSN 2287-5026(Print) / ISSN 2288-159X(Online)

논문 2019-56-3-2

머신러닝 기법을 적용한 하드웨어 데이터 프리페치 기법 구현

(Implementation of Hardware Data Prefetching Technique Using Machine Learning Technique)

송 경 환*, 김 강 희*, 최 상 방**

(KyungHwan Song, KangHee Kim, and SangBang Choi©)

요 약

프리페치 기법은 프로세서와 메모리 사이에서 발생하는 메모리 지연시간을 숨기고 성능 격차를 해소하기 위한 방법 중 하나로 프로세서가 자주 사용하는 데이터나 앞으로 사용할 데이터를 예측하고 캐시에 미리 올린다. 이는 프로세서의 메모리 접근 패턴이 길고 복잡해질수록 하드웨어의 구조가 복잡해지고, 매우 많은 저장 공간을 요구한다. 머신러닝 기반의 프리페치 기법인 LSTM (Long Short-Term Memory) 프리페치 기법은 LSTM 머신러닝 알고리즘을 이용하여 학습과 예측을 수행한다. 그러나 LSTM이 예측을 위해 필요로 하는 파라미터의 수가 많기 때문에 많은 파라미터 저장 공간을 요구한다. 본 논문에서는 프리페치의 예측기로서 RNN의 변형 알고리즘인 GRU (Gate Recurrent Unit)을 사용하고 워크로드의 메모리 접근 패턴을 GRU에서 학습하여 다음 접근 주소를 예측하는 프리페치 기법을 제안한다. 제안하는 기법은 우수한 예측 성능을 가진다. 파라미터의 개수를 줄였기 때문에 프리페처 설계시 파라미터를 읽고 쓰기 위한 저장 공간을 줄일 수 있다. 이는 다이면적을 크게 줄여 높은 에너지 소비 효율을 얻게 한다. 또한 예측을 위한 연산과정과 프리페치 주소를 생성하는 시간을 크게 줄인다.

Abstract

The prefetch is a way to hide the memory latency between the processor and the memory and to resolve the performance gap, predicting and prepopulating the processor's frequently used or future data. As the processor's memory access pattern becomes longer and more complicated, the hardware structure becomes complicated and requires a lot of storage space. The LSTM (Long Short-Term Memory) prefetch technique, which is a prefetch method based on machine learning, performs learning and prediction using LSTM machine learning algorithm. However, since LSTM requires a large number of parameters for prediction, it requires a large amount of parameter storage space. In this paper, we propose a prefetch method that predicts the next access address by using GRU (Gate Recurrent Unit) which is a transformation algorithm of RNN as a predictor of prefetch and learning the memory access pattern of workload in GRU. The proposed method has excellent prediction performance. Reducing the number of parameters reduces the storage space for reading and writing parameters in the prefetcher design. This greatly reduces die area, resulting in high energy consumption efficiency. It also greatly reduces the computation time for the prediction and the time to generate the prefetch address.

Keywords : hardware, prefetch, cache, RNN, GRU

Ⅰ. 서 론

오늘날의 마이크로프로세서는 처음 등장한 1970년대 이후로 무어의 법칙에 맞추어 지속적으로 성장하였다. 마이크로프로세서가 처리해야 할 데이터는 늘어났고 이를 더욱 빠르게 처리할 수 있도록 요구하였다. 이런 요구사항을 충족시키기 위하여 마이크로프로세서는 동작 속도를 높이는 방향으로 발전하였고, 메모리는 많은 양의 데이터를 담을 수 있도록 용량을 늘리는 방향으로 발전하였다. 그 결과 프로세서의 동작 속도와 메모리의 동작 속도는 더욱 벌어지게 되었다.

프로세서는 처리할 데이터와 명령어를 메인 메모리로부터 인출해 오며, 메인 메모리는 보조기억 장치에 저장되어 있는 데이터를 일부 인출하여 보유한다. 프로세서가 메인 메모리에서 데이터와 명령어에 대한 인출 과정을 수행할 때 프로세서의 동작 속도보다 메인 메모

* 학생회원, ** 평생회원, 인하대학교 전자공학과
(Dept. of Electronic Engineering, Inha University)
© Corresponding Author(E-mail : sangbang@inha.ac.kr)
※ 이 논문은 2010년도 정부(교육부)의 재원으로 한국연구재단의 지원을 받아 수행된 기초연구사업임(2010-0020163)
Received ; August 2, 2018 Revised ; February 19, 2019
Accepted ; February 25, 2019

리의 동작 속도가 훨씬 느리기 때문에 프로세서는 메인 메모리에서 데이터를 인출하는 동안 수 백 사이클을 대기하게 된다. 이것은 메인 메모리에 접근하는 경우가 많을수록 성능이 떨어지게 되고 컴퓨터 시스템 성능저하의 원인이 된다[1].

캐시 메모리에는 메인 메모리에 저장된 데이터 중 프로세서가 인출을 시도했던 데이터를 저장한다. 캐시에 인출 데이터가 존재한다면 메인 메모리를 거치지 않고 프로세서에 데이터를 제공하기 때문에 프로세서의 대기시간을 크게 줄일 수 있다.

캐시 메모리를 이용한 계층 구조는 메모리 접근시간의 단축과 프로세서와 메모리 간의 성능 격차를 줄이는데 크게 일조하였다. 그러나 최신의 프로그램들은 복잡·다양하고 매우 긴 메모리 접근 패턴을 갖게 되었기 때문에 프로세서에서 필요로 하는 데이터가 캐시가 가지는 용량 이상을 요구하게 된다면 캐시에 저장된 데이터가 필요가 없어지거나 재사용될 확률이 낮아지게 된다. 이때 프로세서가 요구하는 데이터를 다시 메인 메모리에서 불러와서 캐시에 저장해야 하는 지연이 발생한다. 이 작업이 수행되는 동안 프로세서는 대기 상태로 있어야 한다.

지연을 줄이기 위한 여러 기법이 연구 중 하나는 프리페치 기법이다. 프리페치 기법이란, 프로세서가 미래에 사용할 가능성이 높은 데이터의 주소를 예측하고 프로세서가 이 주소에 대한 인출을 요청하기 전에 해당 주소의 데이터를 캐시에 미리 인출하는 것이다. 여러 가지 프리페치 기법 중 Memory Access Address를 이용한 델타 프리페치 기법이 있다. 이것은 Memory Access Address들 간의 차이 값을 Delta로 두고 이것을 프리페처가 학습을 수행한다. 학습이 수행된 프리페처는 예측을 수행할 때는 현재 들어온 Memory Access Address에 예상 Delta를 내놓고 이 둘을 더하여 다음에 접근할 Memory Access Address를 예측한다.

머신러닝은 패턴 예측에 매우 효과적인 솔루션 중 하나다. 음성인식, 영상인식, 자연어 처리, 미래 예측 등 다양한 영역에서 두루 사용되고 있다. 이러한 머신러닝은 컴퓨터 아키텍처 디자인에도 적극적으로 활용되어 시스템의 성능 향상에도 크게 기여하고 있다. 이중 RNN (Recurrent Neural Network)은 일반적인 머신러닝으로 대응하기 어려운 시계열 데이터를 다루는데 널리 사용되는 머신러닝이며, 자연스럽게 시간 순서에 따라 나타나는 메모리 접근 패턴을 다루는데 효과적인 솔루션이 될 수 있다. LSTM (Long Short-Term Memory) 프리페처는[2] 메모리 접근 패턴을 학습하고 다음에 접근할 주소를 예측하기 위하여 기계 학습의 일종인 RNN을 개선한 LSTM을 사용한다. 그러나 LSTM은 예측을 수행하기 위한 계산과정이 길고 구현을 위한 하드웨어의 복잡도가 높다. 기계 학습 기반 프리페치 기법의 하드웨어를 구현할 때 필요한 리소스를 줄이고 Delta를 예측하기 위한 과정을 단순화해야 한다. 이에 머신러닝 프리페처의 예측기로서 LSTM 대신 GRU (Gated Recurrent Unit)[3]를 예측과정에 도입하여 기존의 LSTM 프리페치 기법이 가지는 성능과 이점을 유지하면서 에너지 효율과 응답속도를 개선하는 기법을 제안한다.

GRU는 LSTM이 가지는 매우 많은 수의 파라미터와 복잡한 연산과정을 단순화한 RNN 알고리즘이다. LSTM이 가지는 많은 장점을 그대로 수용하면서 연산에 필요한 동작을 간략화하였기 때문에 LSTM과 같이 예측기의 성능이 매우 우수하며 예측기의 학습시간이 LSTM보다 빠르게 끝난다. 연산에 필요한 메모리 공간도 LSTM보다 적기 때문에 실제 구현 시 다이 사이즈를 줄여 전력소모율도 줄일 수 있다. 프로세서에서 내려오는 Memory Access Address가 프리페처에 도달하게 되면 프리페처 안에 있는 Delta Stream을 Page 별로 관리 및 저장하는 Local History Table을 참조하여 GRU Unit에서 새로운 Delta를 예측하고 이를 바탕으로 다음에 접근할 메모리 접근주소를 예측한다.

본 논문은 다음과 같이 구성된다. 2장에서는 기존의 하드웨어 프리페치 기법들에 간략하게 소개한다. 3장에서는 제안하는 프리페치 기법에 대하여 설명하며, 4장에서는 실험결과에 대해서 다루고, 마지막으로 5장에서는 연구내용에 대한 결론을 기술하며 끝맺는다.

Ⅱ. 하드웨어 프리페치 기법

하드웨어에 의한 프리페치 기법은 컴파일러나 프로그래머가 개입하지 않더라도 프로세서가 필요로 하는 데이터를 하드웨어가 파악하고 프리페치 요청을 수행한다. 프로세서가 필요로 하는 데이터가 무엇인지 알아내기 위하여 하드웨어는 프로세서와 캐시를 포함한 메모리들 사이에서 일어나는 캐시 Hits/Miss와 같은 이벤트들을 관찰하여, 프리페치 하려는 메모리 주소를 결정한다.

본 장에서는 널리 사용되는 전통적인 프리페치 기법들과 최신의 하드웨어 프리페치 기법들을 소개한다.

1. 순차적 프리페치 기법

순차적 프리페치 기법은 프로그램의 접근 패턴을 찾지 않는다. 대신 공간 지역성을 이용하기 위하여 프로세서가 현재 접근한 주소를 기점으로 다음 주소의 N개의 캐시 블록 ($N = 1, 2, 3 \cdots$) 에 대해 프리페치를 수행한다. 예를 들어, A 라는 주소를 가지는 블록에 대한 프로세서의 접근이 발생하면 $A+1$, $A+2$, $A+3$ ... $A+N$ 의 주소에 해당하는 블록을 프리페치 한다. 순차적 프리페치 기법은 별도의 학습 과정을 수행할 필요가 없기 때문에 하드웨어가 매우 단순해지고 프로그램의 접근 패턴이 공간 지역성이 높고 분기 수행을 하지 않는다면 높은 프리페치 성능을 기대할 수 있다. 그러나 N값이 커지면 불필요한 데이터를 가져올 수 있기 때문에 캐시 오염이 발생할 수 있다. 이를 해결하기 위하여 FIFO(First In First Out) 큐 형태의 스트림 버퍼를 추가하기도 한다[4]. 이는 프리페치한 데이터를 바로 캐시에 넣지 않고 스트림 버퍼에 먼저 저장한다. 그리고 스트림 버퍼에 있는 내용에 있는 내용이 접근되면 해당 블록을 캐시로 옮긴다. 순차적 프리페치 기법은 캐시에서 발생하는 이벤트들을 고려하지 않기 때문에 접근 패턴이 순차적이지 않고 복잡하고 다양할 경우에는 비효율적이며 성능저하를 불러올 수 있다.

2. 스트라이드 프리페치 기법

스트라이드 프리페치 기법[5]은 프로세서가 현재 접근하는 주소와 이전에 접근한 주소 값의 차이를 찾아낸다. 이를 스트라이드라 한다. 찾아낸 스트라이드를 현재 접근한 주소에 더하여 만들어낸 주소를 예측 주소로 삼아 해당 주소를 프리페치 한다.

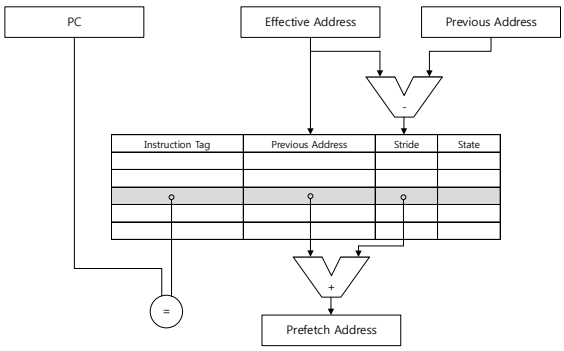


그림 1. 스트라이드 프리페처의 구조
Fig. 1. The stride prefetcher structure.

그림 1 은 스트라이드 프리페처의 구조이다. 스트라이드 프리페처는 RPT (Reference Prediction Table)라 불리는 검색 테이블을 사용한다. 검색 테이블의 엔트리는 Instruction Tag 필드, Previous Address 필드, Stride 필드, State 필드로 구성되어 있다. 프로세서의 메모리 접근이 발생하였을 때 RPT 테이블에 메모리 참조 명령어인 Load/Store의 PC(Program Counter)의 값을 Instruction Tag 필드에 저장한다. Previous Address 필드는 이전에 실행된 Load/Store 명령어의 Memory Access Address를 저장하며, State 필드는 Previous Address와 Effective Address의 차이 값을 저장한다. State 필드는 Init, Transient, Steady, No-Pred의 총4개의 상태를 가지며 해당 필드의 상태에 따라 프리페치의 수행 여부를 결정하게 된다. 프리페치를 수행할 주소는 Previous Address 필드와 Stride 필드의 값을 더하여 계산되며, State 필드가 Steady이고, 스트라이드 값이 0이 아니라면 프리페치를 수행한다. 이전에 프리페치를 수행한 주소와 현재 접근한 메모리 주소가 동일하면 주소 예측에 성공한 것이고 동일하지 않으면 주소 예측에 실패한 것으로 판단한다.

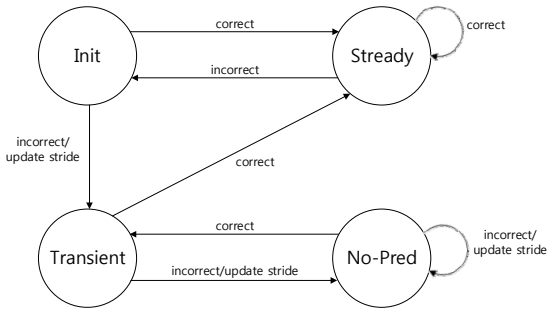


그림 2. State 필드의 FSM (Finite State Machine)
Fig. 2. The state field FSM. (Finite State Machine)

그림 2 는 스트라이드와 State 값에 따라 State 필드의 변화를 나타낸다. 각 상태별 동작은 4 가지로 구분된다. Init는 Load/Store 명령어가 처음 실행될 때의 상태를 나타낸다. Transient는 현재 계산한 스트라이드 값과 이전에 저장된 스트라이드 값이 한 번 일치한 상태를 나타낸다. Steady는 연속적으로 두 번 이상 스트라이드 값이 일치하여 프리페치 명령을 요청한 상태를 나타낸다. No-Pred는 스트라이드 값이 일치하지 않은 상태이다.

스트라이드 프리페치 기법은 최초의 스트라이드만으로 프리페치를 수행하지 않는다. 2 회 이상의 동일한 스트라이드 값이 발생하였을 때 프리페치를 수행한다. 이는 불규칙한 메모리 접근 패턴에 의한 프리페치 오류를 줄이기 위함이다. 스트라이드 프리페치 기법은 정방

향과 역방향의 프리페치를 지원하며 스트림 프리페처보다 다양한 거리의 프리페치가 가능하다. 그러나 복잡하고 불규칙한 패턴에 대해서는 스트라이드 값이 계속 변하기 때문에 2회 이상 동일한 스트라이드 값을 얻기가 어려워 프리페치가 수행되지 않는다.

3. 글로벌 히스토리 버퍼 기법

글로벌 히스토리 버퍼 기법[6]은 엄밀하게 프리페치 기법을 말하지는 않는다. 프리페치를 위한 테이블의 구성 방식을 의미한다. 이것은 기존의 프리페치를 위한 테이블의 구성방법과 달리 테이블의 인덱싱과 프리페치를 수행하기 위한 정보 부분을 나눈 것이다. 그림 3 은 글로벌 히스토리 버퍼 기법을 적용한 프리페처의 구조를 보여주고 있다.

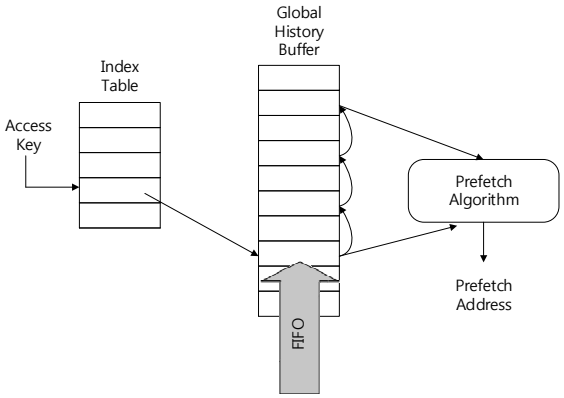


그림 3. 글로벌 히스토리 버퍼를 적용한 프리페치 구조
Fig. 3. Prefetch structure with Global History Buffer.

글로벌 히스토리 버퍼 기법은 인덱싱을 위한 인덱스 테이블 (Index Table), 순환 버퍼의 형태로 만들어진 FIFO 테이블인 글로벌 히스토리 버퍼 (Global History Buffer)로 구성된다.

인덱스 테이블은 기존의 프리페치 기법에서와 동일하게 Load/Store의 PC값이나 혹은 Memory Access Address 또는 이들의 조합을 이용하여 테이블에서 원하는 정보를 찾는 키 값으로 사용한다. 인덱스 테이블의 각 엔트리는 키 값과 함께, 같은 키 값을 가지는 글로벌 히스토리 버퍼의 가장 최근 엔트리를 가리키는 포인터를 기록한다.

글로벌 히스토리 버퍼는 프리페치의 알고리즘에 따라 Memory Access Address 간의 차이 값인 델타나, 오프셋 혹은 학습한 데이터를 시간순으로 기록한다[7]. 이러한 구조는 프리페치 구성 시 정확성을 개선하는데 사용될 수 있으며, 하드웨어 크기도 줄일 수 있는 이점을 가진다.

4. Feedback Directed Prefetching 기법

FDP(Feedback Directed Prefetching)은 프리페치가 동작하는 과정에 맞추어서 프리페치 정확도, 지연도, 캐시 오염도를 평가하고 프리페치의 파라미터를 조정하여 프리페치를 수행한다[8]. FDP는 프로그램이 동작하면서 정확도, 지연도, 캐시 오염도를 실시간으로 특정 간격마다 수집하고 아래 그림 4에 따라 수집한 척도들을 평가하기 위한 카운터 값을 조절한다.

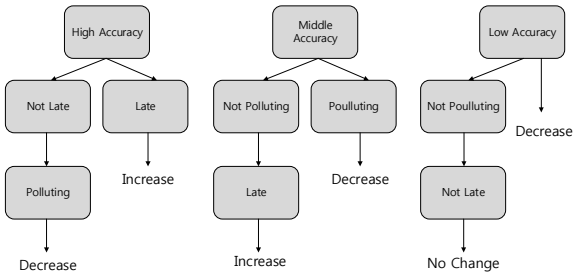


그림 4. 수집한 정확도, 지연도, 캐시 오염도를 평가하는 과정
Fig. 4. The process of assessing collected Accuracy, Latency, and Cache Pollution.

평가되어 나온 값은 아래 표 1의 Count에 따라 현재 Memory Access Address에서 떨어진 거리만큼을 정의하는 파라미터인 Distance와 프리페치 과정을 반복하는 횟수를 결정하는 파라미터 Degree를 조절하고 프리페치를 수행한다.

표 1. FDP의 동적 카운터 설정
Table1. Dynamic counter settings for FDP.

Count		Distance	Degree
1	Very Conservate	4	1
2	Conservate	8	1
3	Middle of the Road	16	2
4	Aggressive	32	4
5	Very Aggressive	64	4

Count가 1 이라면 프리페치 과정은 매우 보수적으로 수행해야 함을 의미한다. 따라서 적극적으로 프리페치 동작을 수행하지 않는다. Count가 5 가 된다면 프리페치를 적극적으로 수행하게 된다.

4. BestOffset 프리페치 기법

BO(BestOffset)은 Memory Access Address에 대해 하위 주소의 오프셋을 평가하여 적절한 오프셋을 선택 후 프리페치를 수행한다[9]. 프리페치가 동작하는 동안

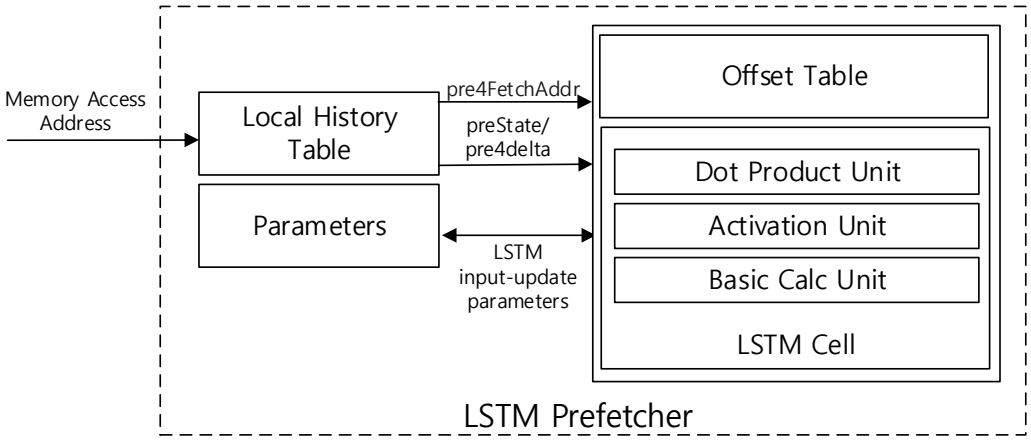


그림 5. LSTM기반 프리페치 기법의 구성
Fig. 5. Composition of LSTM based prefetch technique.

페이지가 가질 수 있는 모든 오프셋 값에 대해서 평가 테이블을 구성하고, Memory Access Address에서 하위 비트의 오프셋을 얻고 해당 오프셋 값에 따라 평가 테이블의 점수 증감을 조절한다. 이때 BO는 이러한 평가를 Memory Access Address가 발생하는 시점에 바로 수행하지 않고 프리페치가 수행되고 프리페치를 통해 필요한 데이터가 캐시에 올라오는 시점에 평가를 수행한다. 이것은 프리페치가 캐시 메모리에서부터 캐시에 올라올 때까지의 응답시간을 고려하기 위한 동작이다. BO는 메모리 지연시간을 고려하면서 캐시에서 자주 발견되는 오프셋 값을 찾아내어 앞으로 필요할 데이터를 찾아낸다.

5. LSTM기반 프리페치 기법

LSTM 기반 프리페치 기법은 프로세서와 메모리 사이의 이벤트 중 Memory Access Address들의 간격을 델타로 보고 델타를 수집하여 시계열 머신러닝 알고리즘인 LSTM에 학습을 수행한다[2]. 학습이 수행된 프리페처는 다음에 프로세서에서 접근이 예상되는 주소의 델타값을 LSTM을 통하여 예측을 수행한다. LSTM을 이용한 델타 학습은 기존의 글로벌 히스토리 버퍼의 방식을 이용한 프리페치 기법들과 비교할 때 글로벌 히스토리 버퍼가 가지는 크기의 제약에서 자유롭다. 글로벌 히스토리 버퍼보다 많은 수의 시계열 델타를 기억하고 저장 할 수 있기 때문에 매우 오랜 시간이 지난 접근 패턴에 대한 델타 값에 대해서도 높은 예측성능을 기대할 수 있다. LSTM은 학습에 들어가는 패턴들이 테이블의 저장 공간에 기억되지 않고 연산과정으로 기억 및 예측을 수행하게 된다.

그림 5는 LSTM기반 프리페처의 구성도이다. LSTM 기반의 프리페치 기법은 크게 3 개의 테이블과 LSTM 유닛으로 구성되어 있다.

Local History Table은 페이지 번호와 접근한 주소의 페이지 오프셋 값 (preAddr), 델타의 스트림을 저장하는 (pre4Delta), 최근에 발행된 프리페치 주소 4개의 오프셋 (pre4FetchAddr), Cell 상태와 은닉 상태를 저장하는 항목(preState)으로 구성되어 있다. 프로세서에서 메모리 접근이 발생하면 해당 이벤트에 대한 Memory Access Address가 Local History Table에 전달된다. 프리페처는 Local History Table을 검색하여 해당 Memory Access Address에 해당하는 페이지 번호를 검색한다. Local History Table에 페이지 번호에 해당하는 Column이 존재한다면 Memory Access Address의 오프셋과 Column의 preAddr를 바탕으로 델타를 계산하고 pre4Delta에 저장한다. Memory Access Address에 해당하는 페이지 번호가 Local History Table에서 검색되지 않았다면 Local History Table의 Column 들 중 사용량이 가장 적은 항목을 제거하고 들어온 Memory Access Address를 바탕으로 새로운 항목을 생성한다.

Offset Table은 페이지의 오프셋 하나당 가질 수 있는 델타 값을 저장하는 단일 항목 하나로 구성되어 있다. Offset Table은 Memory Access Address가 페이지에 대하여 최초로 들어왔을 때 해당 접근주소의 오프셋을 바탕으로 테이블을 검색하고 델타를 출력한다. Offset Table의 델타 값 갱신은 Memory Access Address의 동일 페이지가 두 번 발생하였을 때, 즉 Local History Table의 특정 Column의 접근이 두 번 이루어져 preAddr와 pre4Delta값이 한 개 존재할 때 갱신이 이루어진다.

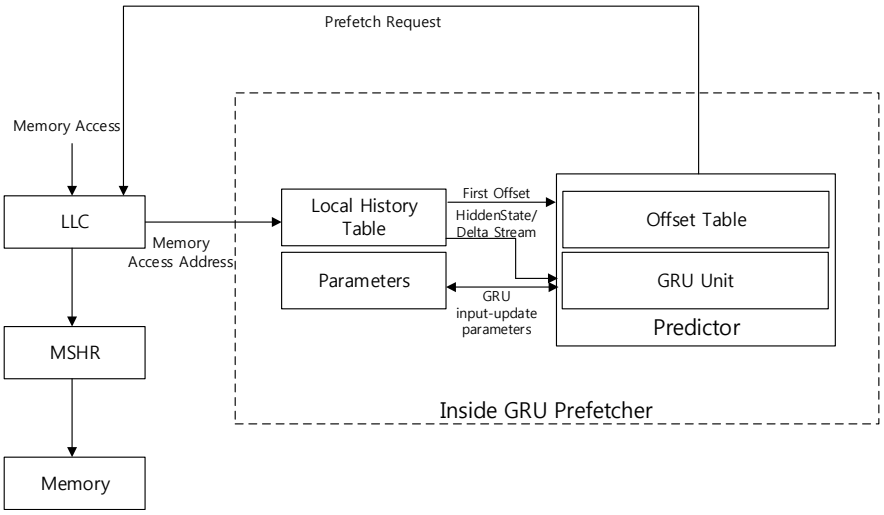


그림 6. GRU 머신러닝 알고리즘을 이용한 델타 상관관계 프리페치 기법의 구성도
Fig. 6. Diagram of Delta Correlation prefetch technique using Machine Learning Algorithm GRU.

파라미터 테이블은 LSTM 유닛이 입력값을 바탕으로 델타를 계산하기 위하여 사용되는 Weight와 바이어스들이 저장되는 테이블이다. 학습 과정이 수행될 때마다 해당 테이블들은 갱신이 수행된다.

LSTM 기반의 프리페치 기법의 예측 동작은 다음과 같다. 프로세서에서 메모리 접근을 수행하게 해당 Memory Access Address가 프리페처로 전달된다. 전달된 주소는 페이지 주소를 바탕으로 Local History Table을 검색한다. 페이지 주소에 해당하는 Column이 존재할 경우 해당 Column들의 정보를 통하여 델타를 계산하고 Column의 정보를 새로이 갱신한다. 갱신이 끝나면 Column의 pre4Delta와 preState, LSTM Unit에 전달되고 LSTM Unit은 전달받은 데이터를 바탕으로 델타를 예측하고 현재 Memory Access Address와 델타 값을 더하여 프리페치를 수행한다. Local History Table에 해당 Column이 존재하지 않는다면 새로이 Column를 추가하고, Offset Table에서 Memory Access Address의 오프셋을 가지고 Offset Table을 검색한다. Offset Table의 Column에 델타값이 들어있다면 해당 델타값과 Memory Access Address를 더하여 프리페치를 수행한다.

Ⅲ. 제안하는 프리페치 기법

본 논문에서는 높은 성능을 얻고 파라미터 수를 줄이기 위하여 머신러닝 기법 중 GRU (Gated Recurrent Unit)를 이용한 델타 상관관계 프리페치 기법을 제안한다. 그림 5는 제안하는 프리페치 기법의 구성도이다. 프리페치를 수행하기 위한 주소의 예측기로 GRU를 사용함으로써 기존의 머신러닝 기반 프리페치 기법보다 예측성능이 향상되고 파라미터가 감소하여 동작에 필요한 리소스를 절감하고 구현 하드웨어 크기를 줄인다.

본 장에서는 GRU를 예측기로 사용하는 하드웨어 프리페치 기법의 구조와 동작 과정을 설명한다.

1. Local History Table 엔트리 생성 및 관리

Local History Table은 최근에 접근한 Memory Access Address의 델타 기록을 관리한다. 델타 기록은 Delta Stream 필드에 저장되며 GRU는 델타 기록을 바탕으로 앞으로 사용할 것으로 예측되는 데이터의 접근주소를 생성하는 데 사용된다.

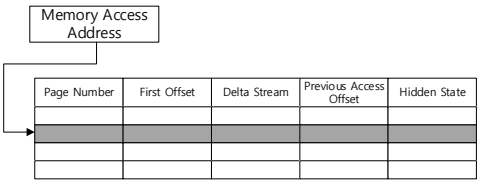


그림 7. Local History Table의 엔트리 구조
Fig. 7. Structure of Local History Table entries.

그림 7 은 Local History Table의 엔트리 구조를 보여주고 있다. Local History Table 엔트리의 각 필드에는 (1) Page Number, (2) First Offset, (3) Delta Stream, (4) Previous Access Offset, (5) Hidden State 로 구성되어 있다.

(1) Page Number는 Memory Access Address에서 얻을 수 있는 페이지 주소가 저장된다. (2) First Offset은 엔트리가 처음 생성되었을 때 Memory Access

Address에서 얻을 수 있는 오프셋 값을 저장한다. (3) Delta Stream은 현재 메모리 접근주소에서 얻을 수 있는 오프셋 값과 Previous Access Offset 필드 값의 차이 값인 델타들을 순서대로 저장한다. (4) Previous Access Offset은 Memory Access Address를 통해 해당 엔트리에 접근될 때마다 얻을 수 있는 오프셋 값을 저장한다. 해당 필드의 갱신은 Delta Stream 필드의 갱신이 수행된 후에 갱신된다. (5) Hidden State 는 GRU의 상태 값 h_t가 저장된다.

그림 8 은 Local History Table의 동작을 보여준다. Memory Access Address가 발생하면 Local History Table의 Page Number 엔트리를 검색한다. 해당하는 Page Number가 존재하지 않는다면 비어있는 엔트리를 검색해서 해당 엔트리에 새로운 Memory Access Address로 생성한 Page Number와 오프셋을 Page Number 필드와 First Offset 필드 그리고 Previous Access Offset 필드에 저장한다. 비어있는 엔트리가 존재하지 않을 경우 Local History Table에 생성된 엔트리 중 가장 오랫동안 접근이 없었던 엔트리를 지우고 새로 내용을 채운다. 이후 새로운 Memory Access Address가 발생하면 엔트리를 검색하게 되고 테이블에서 해당하는 엔트리를 발견하게 되면 Delta Stream 필드와 Previous Access Offset 필드를 갱신한다. Delta Stream 필드는 현재 들어온 Memory Access Address에서 얻은 오프셋 값과 Previous Access Offset 필드의 오프셋 값을 빼서 새로운 델타 값을 생성한다.

Delta Stream 필드에 시간순으로 저장된 델타 값들 중 가장 오래된 델타 값을 제거하고 새로 얻은 델타 값을 델타 값들의 가장 뒤에 집어넣는다. Delta Stream 필드에 Delta가 시간순으로 +4,-1,+4,-1 가 저장되어 있다면 새로운 델타 값 +4를 계산하고 앞쪽의 +4 를 제거한 후 새로운 델타 값 +4 를 가장 뒤에 삽입하여 -1,+4,-1+4 를 구성한다. Previous Access Offset 필드는 Delta Stream 필드의 갱신이 끝나면 현재 Memory Access Address의 오프셋 값으로 필드의 값을 갱신한다.

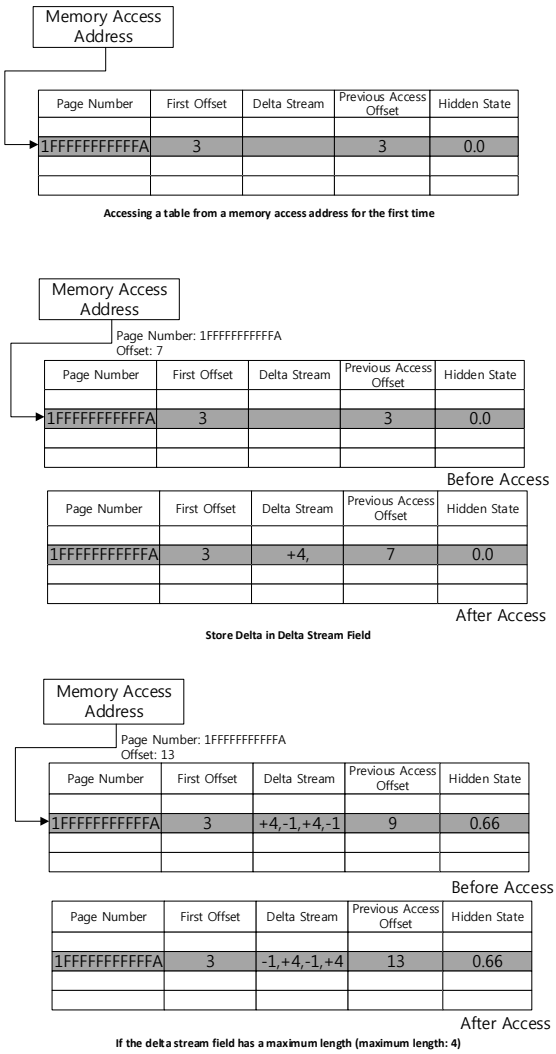


그림 8. 로컬 히스토리 테이블의 델타 저장 및 갱신 과정
Fig. 8. Delta store and update process of Local History Table.

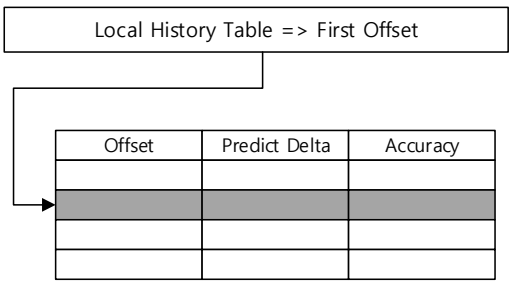


그림 9. 오프셋 테이블의 엔트리 구조
Fig. 9. Structure of Offset Table entries.

2. Offset Table을 이용한 예측 및 관리

Offset Table은 Local History Table에서 만들어진 엔트리에서 Delta Stream 필드의 내용이 없을 경우에 동작한다. GRU Unit은 시계열 순으로 저장된 Delta Stream의 정보로 동작하기 때문에 처음 새로이 엔트리가 생성되었을 경우 오프셋만 존재하고 Delta Stream이 존재하지 않으면 GRU Unit에 의한 예측은 수행되지 않는다. Offset Table은 Local History Table에서 Delta Stream이 없을 경우의 프리페치 동작을 수행하도록 고안되었다. 그림 8은 Offset Table의 엔트리 구조이다.

Offset Table 엔트리의 각 필드에는 (1) Offset, (2) Predict Delta, (3) Accuracy 로 구성되어 있다.

(1) Offset은 페이지에서 나타날 수 있는 오프셋 값을 전부 기록하게 된다. (2) Predict Delta는 오프셋 필드에 대응하는 델타 값이 기록된다. (3) Accuracy는 Offset 필드에 대응되는 Predict Delta의 값이 얼마나 신뢰성이 있는지를 평가한다. 해당 필드의 값이 0 이 된다면 해당 엔트리의 Predict Delta 필드의 값은 프리페치 하기에 부정확한 델타로 판단한다.

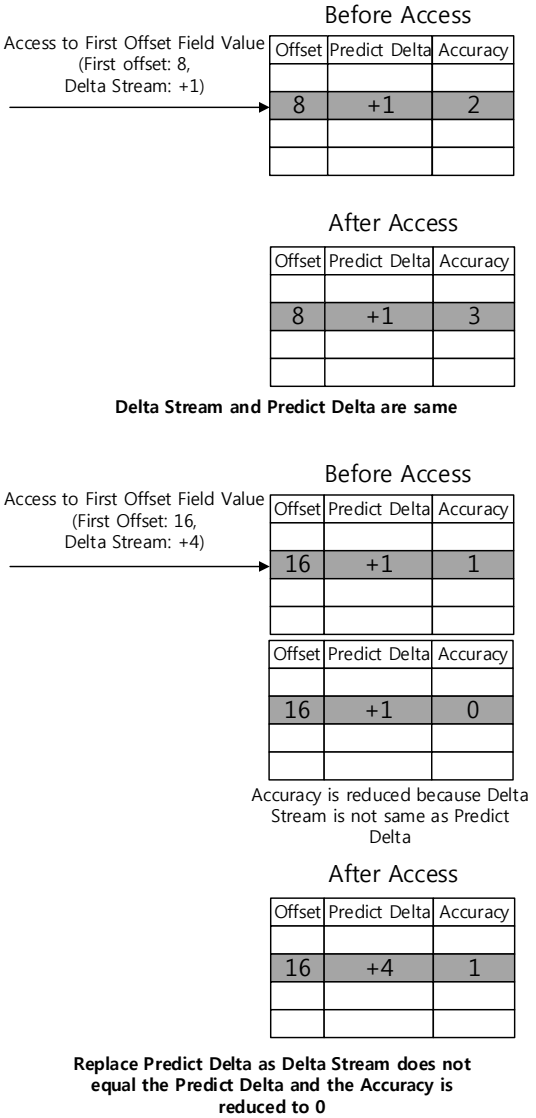


그림 10. 오프셋 테이블 갱신과정
Fig. 10. Updating the Offset Table.

Local History Table의 엔트리 갱신이 끝나면 Offset Table에서는 해당 엔트리의 Delta Stream필드에 델타가 존재하는지를 확인하게 된다. Delta Stream에 델타가 존재하지 않을 경우 해당 엔트리의 First Offset 필드의 값이 Offset Table에 전달된다. Offset Table은 전달받은 First Offset 값을 테이블에서 검색하고 검색된 엔트리에 있는 델타 값을 예측 값으로 삼고 해당 델타 값과 메모리 오프셋 주소를 더하여 프리페치 주소를 생성하고, 해당 주소에 위치하는 데이터를 프리페치 한다. 검색된 엔트리에 델타 값이 없거나 Accuracy의 값이 0에 도달하여 있다면 프리페치 과정을 수행하지 않는다.

그림 10 은 Offset Table의 갱신과정을 보여주고 있다. Offset Table의 갱신에는 Local History Table의 갱신이 끝나고 해당 엔트리의 Delta Stream 필드의 델타가 하나 존재할 경우에 수행된다. 해당 Local History Table의 First Offset 필드의 값을 가지고 Offset Table을 검색한다. 검색 후 나오는 엔트리의 Predict Delta 필드의 값이 해당 Local History Table의 엔트리에 있는 Delta Stream 필드안에 있는 델타 값과 동일하다면 Accuracy를 1 씩 증가시킨다. 델타 값이 동일하지 않다면 Accuracy는 1 씩 감소시키며, 이때 Accuracy가 0 이 된다면 Offset Table의 Predict Delta 필드의 값은 Delta Stream 필드 안에 들어있는 델타 값으로 교체된다.

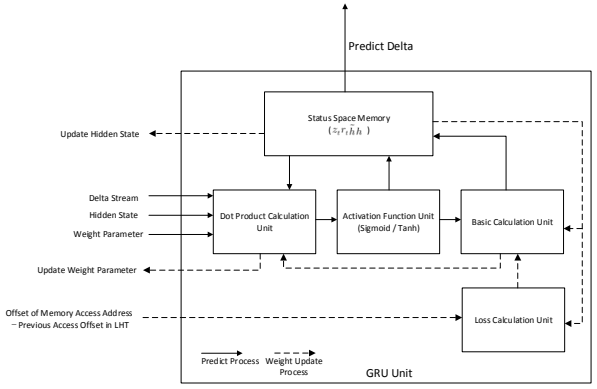


그림 11. GRU Unit
Fig. 11. GRU Unit.

3. GRU Unit를 이용한 예측 및 관리단계

GRU Unit은 GRU Cell을 구현한 Unit으로 Local History Table에서 만들어진 엔트리에서 Delta Stream 필드의 델타가 한 개 이상 있을 때 동작한다. GRU Unit이 동작하기 위해서는 Local History Table 엔트리의 Delta Stream과 Hidden State 필드의 값과 Parameter Table의 Parameter들이 필요하다.

그림 11은 GRU Unit의 구조를 보여주고 있다. GRU Unit에 Delta Stream, Hidden State 필드 값, Parameter가 입력되면 GRU Cell에서는 입력된 Delta Stream을 시간 순서대로 연산을 거쳐 예측 Delta 값을 출력하게 된다.

GRU Unit은 Dot Product Calculation Unit, Activation Function Unit, Basic Calculation Unit, Loss Calculation Unit, Status Space Memory로 구성된다.

Local History Table에서 넘어온 Delta Stream에서 시간상 가장 오래된 델타가 Hidden State 필드 값과 같이 Dot Product Calculation Unit에 들어가고 Weight, 바이어스와 같이 계산이 이루어진 후 Activation Function Unit에서 Activation Function를 거친 값 z_t, r_t, $\tilde{h}$ 가 출력된다. 이후 Basic Calculation Unit에서 h가 계산된다. 계산된 h는 Status Space Memory에 저장되고 Delta Stream에서 다음으로 오래된 델타가 h와 같이 Dot Product Calculation Unit에 들어가고 출력되는 과정이 반복된다. 이것은 Delta Stream의 델타가 전부 사용될 때까지 반복된다. 최종적으로 출력되는 h값은 예측 델타 값으로 사용되며 이후 Memory Access Address와 예측 델타 값을 더하여 프리페치 주소를 생성하고 프리페치를 수행한다.

GRU의 Weight 생성 및 갱신은 Memory Access Address가 프리페치에 들어오고 Local History Table을 갱신하기 이전에 미리 사전 수행되어 있어야 한다. Local History Table에서 Memory Access Address의 Page Number를 바탕으로 검색을 수행하고 검색되었을 경우 Delta Stream 필드에 델타가 1 개 이상 존재하는지 확인한다. 검색에서 존재하지 않거나 Delta Stream 필드에서 검색이 되지 않을 경우 갱신은 수행되지 않는다. Delta Stream 필드에서 델타가 1 개 이상 존재하면 해당 Delta Stream과 Hidden State 필드의 값과 Parameter Table의 Parameter들이 GRU Unit에 전달된다. Local History Table에서 넘어온 Delta Stream에서 시간상 가장 오래된 델타가 Hidden State 필드 값과 같이 Dot Product Calculation Unit에 들어가고 Weight, Bias와 같이 계산이 이루어 진 후 Activation Function Unit에서 Activation Function를 거친 값 z_t, r_t, $\tilde{h}$ 가 출력된다. 이후 Basic Calculation Unit에서 h가 계산된다. 계산된 z_t, r_t, $\tilde{h}$, h 는 Status Space Memory에 저장되고, 첫 번째 반복과정에서 생성된 h는 Local History Table의 Hidden State 에 기록되어 갱신된다. Delta Stream에서 다음으로 오래된 델타가 h와 같이 Dot Product Calculation Unit에 들어가고 출력되는 과정이 반복된다. 이것은 Delta Stream의 델타가 전부 사용될 때까지 계속 반복된다. 최종적으로 출력되는 h값은 Memory Access Address의 오프셋과 Local History Table의 Previous Access Offset 필드를 빼서 얻은 델타 값과 비교된다. 차이가 난다면 Loss Calculation Unit과 Status Space Memory의 값들을 이용하여 Weight를 갱신한다.

Ⅳ. 결과 분석

이 장에서는 LSTM 기반의 프리페치 기법과 본 논문에서 제안한 프리페치 기법에 대한 성능을 검증하기 위하여 LSTM과 GRU의 학습 데이터들의 수렴 속도를 비교하고 학습 및 예측을 수행하는 RNN들이 요구하는 저장 공간의 크기를 비교하였다. 마지막으로 실제 프리페치 성능을 비교하기 위하여 ChampSim Simulator[10]와 Tensorflow를 사용하여 프리페치 기법들을 구현 하였다.

1. 시뮬레이션 환경

ChampSim Simulator은 C++로 작성된 트레이스 기반 시뮬레이터로 벤치마크 프로그램의 실행 파일로부터 Pin Tool을 이용하여 trace를 추출한 뒤, 추출한 trace를 입력으로 받고 LSTM 및 GRU 예측기에는 Tensorflow로 구현하며 Weight와 State는 시뮬레이션 상에서는 32bit floating Point를 사용한다. ChampSim Simulator와 Tensorflow는 IPC로 데이터를 주고 받으며 시뮬레이션을 수행한다. 시뮬레이션을 위하여 하나의 워크로드당 10 억개의 명령어들을 실행하였다. 본 논문의 시뮬레이션 실험을 위해 사용된 ChampSim Simulator의 시뮬레이션 시스템 구성은 표 2와 같다.

표 2. 시뮬레이션 시스템 구성
Table2. Simulation system configuration.

Core	X86, 1core, Out-of-Order, 4Ghz
Branch Predictor	Type: Gshare, History Length: 14, PHT Counter Bits: 2
Front end	Fetch Width: 4, Fetch Queue: 32
ROB Size	256
Private I-Cache	32KB, 8-way, 4 cycles
Private D-Cache	32KB, 8-way, 4 cycles
Private L2 Cache	256 KB, 8-way, 12 cycles
Shared LLC Cache	2MB , 16-way, 20 cycles
Main Memory	4GB, hit: 55Cycles, miss: 165 Cycles

시뮬레이션을 수행하기 위하여 사용한 벤치마크 프로그램은 SPEC 2006의 워크로드들을 사용하였다. SPEC 2006의 프로그램을 빌드하여 실행파일을 생성 한 뒤,

Valgrind[11]와 SimPoint[12]로 빌드한 실행파일을 분석하여 시뮬레이션 결과에 유의미한 시뮬레이션 포인트를 찾아낸 후[13], 해당 시뮬레이션 포인트를 바탕으로 trace를 추출하여 시뮬레이션을 수행하였다.

GRU 프리페치 기법을 구현하기 위한 테이블과 파라미터의 스토리지 사용공간은 표 3 과 같다.

표 3. GRU 스토리지 오버헤드
Table3. GRU Stroage Overheads.

Offset Table	64×(6+7+2) = 960bit
Local History Table	64×(52+6+28+224) = 19840bit
Weight Matrices	3×7×14×32 = 8736bit
State Vectors	16×4×7×32 = 14336bit

Offset Table은 64개의 엔트리를 가지며 Offset 6bit, 예측 델타 7bit, 정확도 2bit의 공간을 차지한다. Local History Table은 64개의 엔트리를 가지며 하나의 엔트리 당 56bit 길이의 Page Number, First Offset, 6bit, 4개의 델타값을 모은 Delta Stream 28bit, Hidden State 224bit 의 공간을 차지한다. Weight Matrices은 3개의 Weight 행렬과 7bit 크기를 가지는 예측 델타 값, 32bit floating point에 의하여 8376bit의 공간을 차지한다. State Vectors는 16개의 은닉층과 4가지의 상태, 7bit의 크기를 가지는 예측 델타값, 32bit floating point에 의하여 14336bit의 공간을 차지한다.

Tensorflow의 학습에는 AdamOptimizer를 이용하여 학습시간을 제어한다.

2. 시뮬레이션 결과

시뮬레이션 결과를 분석하기 위하여 기존의 델타 기반 프리페치 기법인 VLDP와 LSTM 기반 프리페치 기법 그리고 본 논문에서 제시하는 프리페치 기법의 IPC, 커버리지, Tensorflow로 생성된 파라미터 크기, 사전 학습시간을 비교하고 분석하였다. 실험결과의 평균값에는 기하평균으로 계산하였다.

가. IPC비교

VLDP와 LSTM 기반 프리페치 기법 그리고 본 논문이 제시하는 GRU 기반 프리페치 기법에서의 IPC (Instruction Per Cycle)를 비교하였다.

캐시와 프리페치를 통하여 메인 메모리의 동작 속도가 충분하게 숨겨진다면 IPC의 성능 향상이 나타나게 된다. 본 논문이 제안하는 프리페치 기법은 VLDP 프리

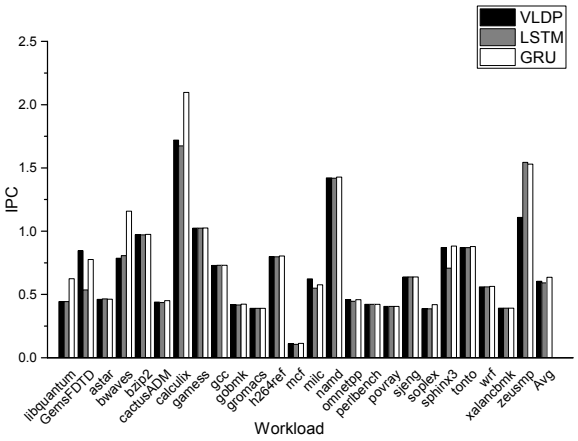


그림 12. 프리페치 기법의 IPC 비교
Fig. 12. IPC comparison of prefetch technique.

페치 기법과 비교하여 평균 5.3% 성능이 향상되었으며 LSTM 기반 프리페치 기법보다 평균 7.5% 성능이 향상되었다. 특히 calculix와 bwaves는 LSTM 기반 프리페치 기법과 비교할 때 각각 25.3%, 43.7% 성능 향상을 보였다.

나. 커버리지 비교

VLDP와 LSTM 기반 프리페치 기법 그리고 본 논문이 제시하는 GRU 기반 프리페치 기법에서의 커버리지를 비교하였다.

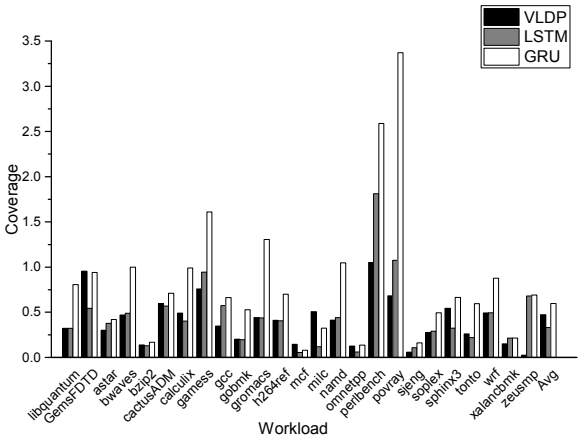


그림 13. 프리페치 기법의 커버리지 비교
Fig. 13. Coverage comparison of prefetch technique.

커버리지는 프리페치 기법을 통하여 캐시의 누락을 얼마나 억제하였는지 알아보는 척도이다. 본 논문이 제시하는 프리페치 기법은 VLDP 프리페치 기법과 LSTM 기반 프리페치 기법보다 높은 성능 향상을 보여주고 있다. VLDP와 비교 하여 평균 92.1%, LSTM 기반 프리페치 기법과 비교 하여 평균 80.3%의 향상을 보여주었다.

다. 프리페치 파라미터 크기 비교

LSTM 기반 프리페치 기법과 본 논문에서 제시하는 프리페치 기법의 파라미터의 크기를 비교하였다. 파라미터는 Tensorflow를 통하여 학습된 Weight와 State Vector를 모두 포함한다.

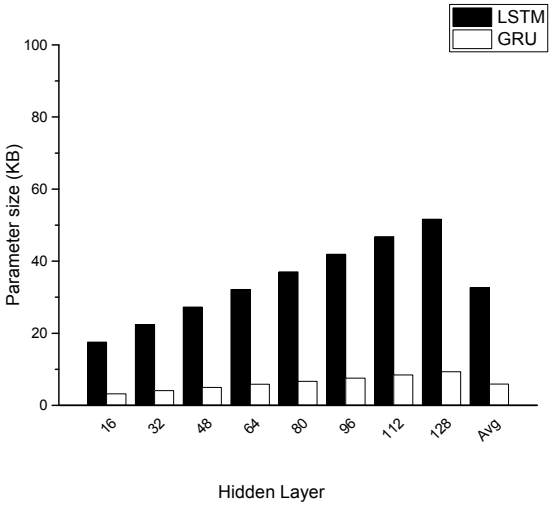


그림 14. LSTM기반 프리페처와 제안 프리페치 기법의 파라미터 크기 비교

Fig. 14. Comparison of parameter sizes of LSTM-based prefetchers and proposed prefetch schemes.

GRU는 LSTM 보다 Cell의 구조상 적은 파라미터의 크기를 가진다. 워크로드들을 학습한 LSTM과 GRU의 은닉층 개수를 변경해 가면서 프리페처의 LSTM과 GRU의 파라미터의 크기를 비교하였으며, GRU가 평균 81.9% 파라미터를 적게 사용한다.

라. 프리페치 사전 학습시간 비교

LSTM 기반 프리페치 기법과 본 논문에서 제시하는 프리페치 기법의 워크로드 수행을 위한 사전 학습시간을 비교하였다. GRU는 LSTM보다 적은 파라미터를 가지기 때문에 학습수행에 필요한 연산시간이 적게 들어간다. 이를 확인하기 위하여 워크로드들 별로 Tensorflow 상에서 GRU와 LSTM이 학습을 수행하고 모델을 생성하는 시간을 측정하고 비교하였다. GRU를 사용한 프리페치 기법이 비교 대상 프리페치 기법보다 학습을 끝내는데 더 빠른 속도를 보여줌을 알 수 있다.

전체 평균으로는 12.58% 학습시간이 감소하였다. 이는 GRU가 LSTM보다 빠르게 학습을 수행함을 알 수 있다.

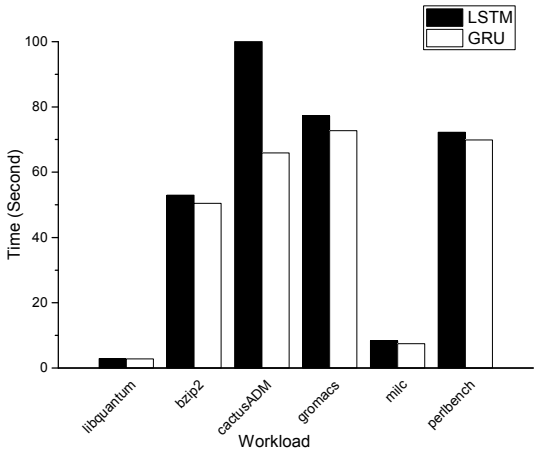


그림 15. LSTM기반 프리페처와 제안 프리페치 기법의 사전 학습시간 비교 1

Fig. 15. Comparison of pre-learning time of LSTM based prefetcher and proposed prefetch method techniques 1.

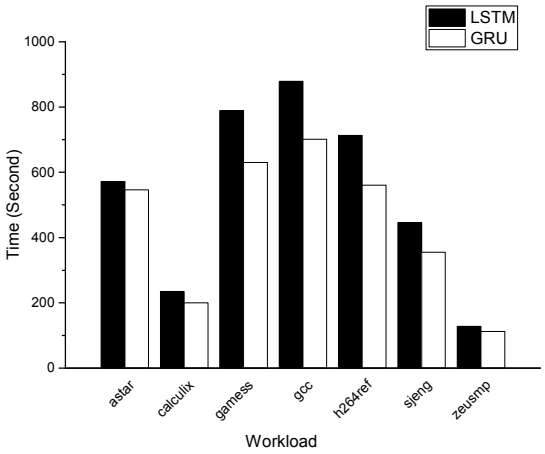


그림 16. LSTM기반 프리페처와 제안 프리페치 기법의 사전 학습시간 비교 2

Fig. 16. Comparison of pre-learning time of LSTM based prefetcher and proposed prefetch method techniques 2.

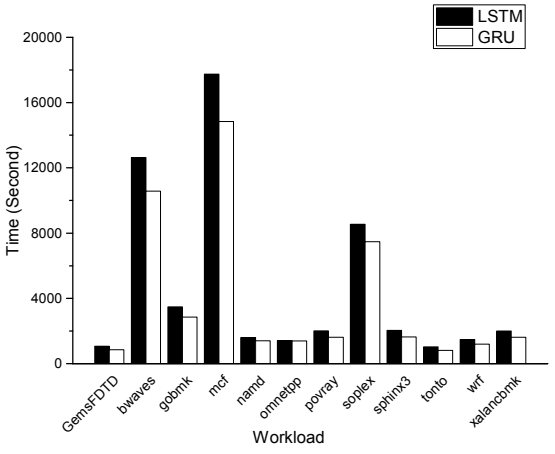


그림 17. LSTM기반 프리페처와 제안 프리페치 기법의 사전 학습시간 비교 3

Fig. 17. Comparison of pre-learning time of LSTM based prefetcher and proposed prefetch method techniques 3.

V. 결 론

프리페치 기법은 프로세서가 앞으로 사용할 가능성이 있는 데이터를 예측하여 프로세서가 해당 주소에 접근하기 전에 미리 캐시에 데이터를 올리는 기법이다.

머신러닝을 이용한 프리페치 기법에서는 프로세서와 메모리 사이에서 발생하는 이벤트 중 메모리 접근 패턴을 머신러닝을 이용하여 시계열 학습을 수행한다. 프리페치를 통한 예측을 수행할 때에는 학습이 수행된 모델을 이용하여 접근 데이터의 주소를 예측하게 된다.

기존의 머신러닝 기반의 델타 프리페치 기법인 LSTM 기반 프리페치 기법은 LSTM을 델타의 예측에 사용하는 프리페치 기법이다. 그러나 학습을 수행하고 프리페치를 예측하기 위하여 긴 대기시간을 요구하고 연산을 위하여 메모리 공간을 많이 사용한다.

본 논문에서 제안하는 프리페치 기법은 예측에 GRU라는 머신러닝 알고리즘을 사용하였다. 기존의 LSTM 기반의 프리페치 기법보다 학습과정과 예측과정이 짧으며 메모리 공간을 적게 소비하면서 LSTM의 예측성능과 근접한 성능을 보여준다. 본 논문에서 제안하는 프리페치 기법의 GRU 유닛은 LSTM보다 연산을 처리해야 할 횟수가 적고 파라미터 갯수가 적기 때문에 델타를 연산해 내는 과정이 빠르며, 프리페치 기법을 하드웨어로 구현할 때 필요한 저장 공간이 작기 때문에 칩 사이즈를 줄여 발열과 전력 소비를 개선 할 수 있다.

기존의 프리페치 기법과 제안하는 프리페치 기법의 비교 분석 결과 GRU를 이용한 델타 상관관계 프리페치 기법은 VLDP 프리페치 기법과 비교할 때 IPC 5.3%, 커버리지 92.1%의 성능 향상이 있었으며, LSTM 프리페치 기법과 비교할 때 IPC 7.5%, 커버리지 80.3% 향상이 있었고 파라미터 크기 81.9% 학습시간 12.58% 감소하였다.

REFERENCES

[1] J. L. Hennessy, and D. A. Patterson “Computer Architecture: A Quantitative Approach,” 5th Ed. Morgan Kaufmann Publishers, pp. 72-95, 2012.

[2] Y. Zeng, and X. Guo, “Long short term memory based hardware prefetcher,” in MEMSYS’17 proceedings of the International Symposium on Memory Systems, pp. 305-311, Alexandria, USA, Oct 2017.

[3] J. Chung, C. Gulcehre, K. Cho, and Y. Bengio “Empirical evaluation of gated recurrent neural networks on sequence modeling,” arXiv preprint arXiv:1412.3555. Dec. 2014.

[4] N. P. Jouppi. “Improving Direct-Mapped CachePerformance by the Addition of a Small Fully-Associative Cache and Prefetch Buffers,” In Proceedings of the 17th Annual International Symposium on Computer Architecture, pp. 364-373, Seattle, USA, May 1990.

[5] K. H. Kim, T. S. Park, K. H. Song, D. S. Yoon, and S. B. Choi, “Implementation of Hardware Data Prefetcher Adaptable for Various State-of-the-Art Workload,” Journal of The Institute of Electronics Engineers of Korea, vol. 53, No. 12, pp. 20-35, Dec 2016.

[6] K. Nesbit, and J. E. Smith, “Data Cache Prefetching Using a Global History Buffer,” In Software, IEE Proceedings, pp. 96-96, Madrid, Spain, Feb 2004.

[7] Y. S. Jeong, J. H. Kim, T. H. Cho, and S. B. Choi, “Instructions and Data Prefetch Mechanism using Displacement History Buffer,” Journal of The Institute of Electronics Engineers of Korea, vol. 52, No. 10, pp. 82-94, Oct 2015.

[8] S. Srinath, O. Mutlu, H. Kim, and Y. N. Patt, “Feedback Directed Prefetching: Improving the Performance and Bandwidth-Efficiency of Hardware Prefetchers,” 2007 IEEE 13th International Symposium on High Performance Computer Architecture, pp. 63-74, Scottsdale, USA, Feb 2007.

[9] P. Michaud, “Best-offset hardware prefetching,” 2016 IEEE International Symposium on High Performance Computer Architecture (HPCA), Barcelona, Spain, pp. 469-480, Mar 2016.

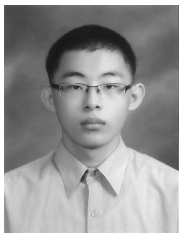
[10] The 2nd Cache Replacement Championship (CRC-2). “ChampSim” Available: http://crc2.ece.tamu.edu/. 2017.

[11] N. Nethercote, and J. Seward “Valgrind: A framework for heavyweight dynamic binary instrumentation,” Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation ser. PLDI ‘07, pp. 89-100, San Diego, USA, Jun. 2007.

[12] G. Hamerly, E. Perelman, J. Lau, and B. Calder “Simpoint 3.0: Faster and more flexible program phase analysis,” Journal of Instruction Level Parallelism, Vol. 7 no. 4 pp. 1-28, Sep 2005.

[13] V. M. Weaver, and S. A. McKee. “Using dynamic binary instrumentation to generate multi-platform simpoints: Methodology and accuracy,” International Conference on High-Performance Embedded Architectures and Compilers, pp. 305-319, Gothenburg, Sweden, Jan 2008.

저 자 소 개



송 경 환(학생회원)
2015년 한남대학교 컴퓨터공학과 학사 졸업.
2018년 인하대학교 전자공학과 석사 졸업.
<주관심분야 : 컴퓨터 구조, SoC, 임베디드 시스템>

최 상 방(평생회원)
1981년 한양대학교 전자공학과 학사 졸업.
1981년～1986년 LG 정보통신(주).
1988년 University of washinton 석사 졸업.
1990년 University of washinton 박사 졸업.
1991년～현재 인하대학교 전자공학과 교수
<주관심분야 : 컴퓨터 구조, 컴퓨터 네트워크, 무선 통신, 병렬 및 분산 처리 시스템>

김 강 희(학생회원)
2011년 인하대학교 전자공학과 학사 졸업.
2013년 인하대학교 전자공학과 석사 졸업.
2013년～현재 인하대학교 전자공학과 박사과정.
<주관심분야 : 컴퓨터 네트워크, 무선 센서 네트워크, SoC>