붙임2

연구논문/작품 중간보고서

2021학년도 제 8학기

제목	머신러닝 기반의 데이터 프리패처	○ 작품( ) 논문(○) ※해당란 체크
평가등급	지도교수 수정보완 사항	팀원 명단
A	○ 본 논문에서 제안하는 데이터 프리패처를 다양한 벤치마크에 대해 평가하세요. 특히, 그래프 분석 벤치마크 (GAP)에 대한 성능 평가를 포함시켜주세요. ○ 세 종류의 모델을 제안했는데 각 모델의 성능을 다양한 벤치마크에 대해 비교 평가해주세요. 또한 기존 연구 (머신러닝 기반 데이터 프리패처)와의 정량 또는 정성적 평가가 필요합니다. ○ 논문에 사용된 그림은 직접 그려야 하며 모든 오타를 찾아서 수정해야 합니다.	박 준 혁 (학번:2016312946) 이승태 (학번:2017313107)

2021 년 9월 24일

지도교수 : 홍 석 인 서명

■ 요약

프로세서와 메모리간 간극은 점점 더 커저만 가는데 이를 해소하기 위한 기법이 데이터 프리패치 기법으로 미리 사용할 데이터를 캐시메모리에 올려다 놓는 행위를 뜻한다. 우리는 머신러닝 기법을 이용하여 데이터 프리패치를 구축했다. 이전에 데이터들을 page별로 delta 값으로 저장시켜 놓고 메모리 접근이 나타날 때 그에 따른 delta 값과 현재 접근한 주소를 LSTM을 통해 concatenate시켜 학습을 진행하고 다음에 접근할 메모리 주소를 예측하게 된다. 또한 예측할 때에 똑같은 값을 예측하여 성능저하를 일으킬 수 있는 중복을 막아 프리패 성능향상에 기여했다. 제안한 모델을 평가하기 위해 ChampSim simulator와 SPEC2006 밴치마크를 사용하였고 프리패가 없는 프로세서의 IPC에 비해 20%의 향상을 보였다.

■ 서론

1970년대에 처음 마이크로프로세서가 세간에 등장한 이후 시간이 지날수록 급격한 성장세를 보여왔다. 시간이 지남에 따라 마이크로프로세서가 처리해야 할 데이터의 양은 방대해지고 이를 빠르게 처리하는 것이 필수 요점이었다. 데이터의 양이 방대해지기 때문에 메모리는 대량의 데이터를 저장할 수 있도록 발전하게 되었고 프로세서는 이를 빠르게 처리할 수 있도록 발전해 나갔다. 프로세서의 속도는 계속 빨라져 갔지만 메모리는 대량의 데이터를 저장하기 때문에 처리속도가 이를 따라가지 못해서 메모리와 프로세서사이의 처리속도 간극이 계속 벌어지게 되었다.

프로그램을 실행하거나 데이터를 참조하기 위해서 용량은 크지만 속도가 느린 메인 메모리에서 사용할 데이터를 처리속도가 빠른 프로세서로 들고와야 한다. 프로세서에서 처리하는 속도에 비해 메인 메모리에서 데이터를 가져오는 속도가 현저히 느리기 때문에 프로세서는 오랜 시간 데이터를 기다리게 되고 속도의 격차로 인한 컴퓨터의 성능 저하를 초래한다.

이를 해결하기 위해서 메모리를 계층화 시키는데 이때 프로세서와 메인 메모리 사이에 적당한 속도와 적당한 용량을 가진 캐시메모리를 추가하여 속도 차이를 해결하게 된다. 프로세서에서 필요한 데이터를 캐시 메모리에서 찾게 되면 메인 메모리에 비해 속도차이가 작기 때문에 메모리 병목현상을 줄일 수 있게 된다.

대부분의 프로그램은 공간적, 시간적 지역성을 한번 사용한 데이터를 다시 사용할 가능성이 높고, 그 주변의 데이터도 곧 사용할 가능성이 높기 때문에 이러한 공간적, 시간적 지역성을 활용하여 메인 메모리에 있는 데이터를 캐시 메모리로 불러와서 캐시메모리로 두게 된다. 이렇게 한다면 프로세서가 필요로 하는 데이터를 느린 메인 메모리에서 찾는 것 보다 더 빨리 데이터를 불러올 수 있게 되고 이는 시스템 성능 향상으로 이어진다.

프로세서서가 데이터를 요청하여 캐시 메모리에 접근했을 때 캐시 메모리가 해당 데이터가 없다면 이를 캐시 미스라고 칭하게 된다. 캐시 미스가 발생 된다면 먼저 메인 메모리 같은 메모리 계층의 하부에 접근을 하여 그 데이터를 가져와서

캐시에 저장시켜서 그 데이터를 사용하게 됨으로 캐시 미스가 많을수록 메모리 레이턴시가 늘어난다고 할 수 있다.

이러한 캐시 미스가 발생하지 않게 미리 사용될 가능성이 높은 데이터를 예측하여 캐시에 저장해 놓을 수 있다면 프로세서는 데이터를 요청하고 기다리는 시간이 줄어들게 되는데 이러한 방식을 데이터 프리패치라고 한다. 프로세서는 처리해야 할 데이터를 빠르게 받아서 처리할 수 있기 때문에 프로세서는 쉬지 않고 일을 할 수 있게 되어 메모리 레이턴시를 감출 수 있게 된다.

머신러닝은 인공지능의 한 분야로, 컴퓨터가 학습할 수 있도록 하는 알고리즘과 기술을 개발하는 분야이다. 머신 러닝 기술은 명시적으로 프로그래밍하는 것이 아니라 데이터로 부터의 학습을 통해 작업을 수행하는 방법을 컴퓨터에게 가르치는 것이다. 이를 통하여 데이터를 분석하고 스스로 학습하는 과정을 거쳐 패턴인식에 장점을 가지게 된다. 즉 방대한 양의 데이터 가운데 비슷한 것끼리 묶어내고 서로 관계있는 것들의 상하구조를 인식하여 이것을 바탕으로 앞으로 행동을 예측하게 된다. 미래 예측 분야에 다양한 활용성을 보이기 때문에 시스템 성능 향상에 큰 영향을 발휘하고 있다.

특히 RNN(Recurrent Neural Network)은 순서, 시간 등의 시계열 데이터를 다루는 데 강점을 보이고 input과 output 길이를 다양하게 적용시킬 수 있다. 이는 프로세서에서 처리할 데이터들의 메모리 접근 패턴을 분석하고 예측하는데 유용하게 사용될 수 있다. 메모리들은 접근은 일정한 시간, 공간적 지역성을 띄기 때문에 이러한 메모리 접근을 input으로 사용하고 이를 통해 다음 사용될 데이터를 output으로 도출할 수 있다.

LSTM(Long Short-Term Memory)은 위와 같은 장점을 가지고 있는 RNN에 문제점인 시퀀스가 너무 길 경우 앞 쪽 타임 스텝의 정보가 뒤에 있는 타입 스텝까지 충분히 전달되지 못하는 문제인 vanishing gradients 문제점을 해결한 모델이다. 이로 인해 오래전 혹은 최근의 데이터 값을 기억할 수 있는 모델로 오래전 정보도 기억할 수 있는 long-term dependency를 지니고 있기 때문에 vanishing gradients 문제를 해결했다. cell state가 추가되어 오래전 정보 또한 사용할 수 있기 때문에 최근 메모리 접근 패턴 이외에도 필요한 오래전 메모리 접근 정보를 활용하여 다음 패턴 값을 예측하는데 활용할 수 있게 된다.

하지만 RNN에 비해서 LSTM은 과거의 정보를 기억해야 하기 때문에 더욱 복잡한 계산구조를 지니고 이를 위한 하드웨어의 복잡도가 높아서 기존 LSTM의 구조를 조금 더 간단하게 개선한 모델인 GRU(Gated Reccurent Unit)가 등장했다. 이는 3개의 gate를 사용하는 LSTM에 비해 2개의 gate만을 사용해서 구현했기 때문에 하드웨어 복잡도가 낮고 사용해야 할 계산구조가 단순해지게 된다. 성능 면에서 LSTM과 비슷하거나 약간 낮은 결과를 보여주기 때문에 우리는 먼저 LSTM을 이용한 데이터 프리패 구현에 관하여 나타내보려 한다.

현재 데이터 프리패는 주로 머신러닝기법이 사용되지 않은 하드웨어적인 구

현을 통한 프리패가 통용되고 있다. 우리는 현재 서서히 연구가 진행 되어 가고 있는 머신러닝 기법을 데이터 프리패치에 접목 시켜 구현해 보려 한다. 위에 말했다시피 머신러닝 기법 중 RNN계열 모델은 시간, 공간 데이터의 패턴예측에 유리하기 때문에 이전에 메모리 접근 패턴을 통해 다음 어떤 메모리를 프로세서가 사용할 지를 예측하는데 효율적이라 판단했다.

머신러닝을 사용한 데이터 프리패는 사용될 데이터를 미리 메인 메모리에서 캐시 메모리로 옮겨놔야 하는데, 어떠한 프리패가 성능적인 측면에서 좋은지에 대한 평가 기준이 있어야 한다. 이러한 기준은 총 3가지로 ACCURACY, COVERAGE, IPC 이다. 먼저 ACCURACY는 총 프리패치 해온 데이터 중 몇 개의 데이터가 캐시 미스를 해결했는지에 대한 비율을 표한다. 다음 COVERAGE는 총 캐시 미스 중에 프리패치로 인해 사라진 캐시미스에 대한 비율을 뜻한다. 마지막으로 IPC는 instruction per cycle로 한 사이클 당 완료 가능한 명령어의 개수를 뜻한다. IPC와 COVERAGE는 어느 정도 비례 관계에 있다고 볼 수 잇는데, 총 캐시미스 중에 프리패치로 인해 사라진 캐시미스 비율이 증가한다면 프로세서는 그 전에 비해 데이터를 기다리는 시간이 줄어들게 될 것이고 이로 인해 한 사이클당 처리할 수 있는 명령어의 수가 늘어나게 된다.

위 평가 기준을 보면 만약 다음에 올 확률이 아주 작은 데이터 값이라도 일정 수준이상의 양을 프리패치를 해오게 된다면 ACCURACY는 총 프리패치 해온 데이터의 수가 늘어나지만 프리패치로 인해 해결한 캐시미스의 수는 소량 늘어나게 될 것 임으로 ACCURACY는 줄어들게 된다. 그러나 일정 수준 이상 양의 데이터 값을 프리패치 해오게 된다면 동일한 수의 캐시미스 내에서 프리패치로 인하여 해결한 캐시미스의 수가 증가하게 됨으로 COVERAGE는 증가하게 된다. 이는 IPC의 증가를 의미하게 된다. 또한 만약 한 번에 초 대량의 데이터를 프리패치 해오게 된다면 이는 오히려 캐시미스 수 자체를 늘리게 됨으로 프리패치 해올 데이터의 수를 2개의 프리패치 set으로 제한하여 기존의 타 하드웨어 프리패치 모델들과 우리가 제안한 모델과 비교할 것이다.

기존에 주로 사용되어 지고 있는 머신러닝을 사용하지 않은 하드웨어적 모델과의 차이를 위해 우리는 RNN계열의 머신러닝 모델을 구현하여 위에서 나타낸 제한조건 내에서 비교를 할 것이다. ACCURACY, COVERAGE, IPC 총 3가지의 측정 기준이 있는데, 우리는 IPC를 첫 번째 기준으로 판단할 것이다 IPC는 프로세서의 성능을 나타낼 수 있는 지표이기도 하기 때문에 IPC의 상승은 컴퓨터 시스템 성능향상이다. 이를 비교하기 위해선 우리는 TRACE 기반의 ChampSim Simulator를 사용하고 SPEC06, GAP trace set으로 성능을 비교할 것이다.

머신러닝을 이용한 데이터 프리페쳐 IPC/COVERAGE/ACCURACY 대한 설명과 중점

우리의 머신러닝을 사용한 데이터 프리패는 크게 3번의 모델 변화가 있었다. 기본적으로 LSTM을 이용하여 데이터를 training하는 데이터 input을 어떻게 가공하여 사용할지 그리고 output을 나타내기 위해서 기능적으로 추가하였다. 기본적

인 모델을 필두로 성능을 확인하며 점차적으로 본 프리패를 발전시켜 나갔기 때문에 크게 3가지 모델을 구현하였고 현재는 마지막 모델에서 약간의 요인들만 변화시켜 더욱이 좋은 결과를 나타내려 한다.

먼저 같은 Page 내에 현재 메모리 접근을 offset을 통해 delta값 들을 table에 저장하게 된다. 이후 delta 값들에 대한 정보와 현재 접근한 Load address의 주소를 2진법으로 나타내 이를 LSTM을 통해 concatenate 하여 추후 접근할 메모리 주소로 다시 변환하여 예측하게 된다. 현재 프리패를 사용하지 않은 것에 비해 IPC가 평균 20% 가량 향상되었다. 그러나 전통적인 프리패들의 IPC 향상률에 비해서는 낮은 비율을 보여 hyperparameter 혹은 약간의 모델 변화가 필요하다.

본 보고서에서 관련연구에서는 현재 연구되고 있는 프리패와 모델을 구성하기 위해서 참고했던 논문들에 대해서 소개한다. 제안 작품 소개에서는 현재 모델을 구성하는 요소와 함께 실제 제안된 모델에 대한 구현에 대해서 나타낸다.

■ 관련연구

1) Sangam : A Multi-component Core Cache[1]

본 프리패는 하드웨어적 구현을 통한 프리패로 컴파일러나 프로그래머의 개입 없이 하드웨어가 상황에 따라서 데이터 프리패치를 수행하게 된다. 현재 나타나 있는 Sangam은 머신러닝을 사용하지 않고 데이터 접근을 판단하여 다음에 사용될 가능성이 높은 메모리 주소를 판단하여 프리패치하여 시스템 성능을 향상시키게 된다.

Sangnam 프리패는 총 3번의 과정을 통해서 어떠한 메모리 주소들을 미리 캐시메모리로 들고 올지에 대해서 판단하는데, 3번의 과정에 필요한 하드웨어는 IP-Delta-based Seqence Predictor, IP-based Stride Prefetcher, Next-line Prefetcher 이다. 첫 번째인 IP-Delta-based Seqence Predictor는 현재 접근한 주소의 offset과 마지막을 접근한 access의 offset간의 차이를 delta로 정의한다. IP table과 IP-delta table을 가지고 프리패치 과정을 진행하게 되는데, IP table의 특징은 마지막으로 본 d+1개의 delta를 FIFO로 저장하고 delta를 저장하기 위한 마지막 접근 주소의 offset과 tag,valid bit, LRU state를 저장한다. 그다음 IP-delta table은 table의 indexing을 위하여 사용된 현재 사용된 delta 뒤의 d deltas들을 sequence하게 저장하고, 사용하기 위한 2-bit confidence counter, tag, valid bit, LRU state를 저장한다.

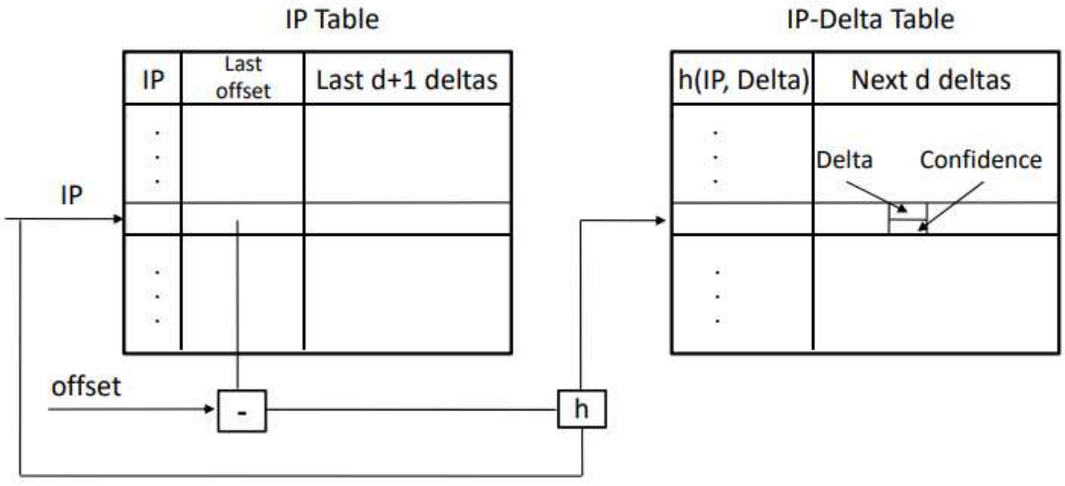

그림 1 Sangnam predict과정

IP-Delta-based Seqence Predictor는 다음 사용될 데이터를 예측하기 위해서 총 2가지의 과정을 거치게 되는데, 먼저 predict과정이다. predict 과정에서 먼저 IP Table에서 현재 IP값과 일치하는 값을 찾아서 그 행에 저장되어 있는 Last offset과 현재 IP에 대한 offset의 차이 값을 통해 delta 값을 구한다. 그 delta 값과 현재 IP를 가지고 IP-Delta Table에 일치하는 값을 찾게 된다. 이 때 NEXT d deltas에서 delta값들을 가지고 프리패치를 진행하게 되는데 여기서 Confidence counter를 판단하게 된다. 설정된 일정수준 이하의 confidence counter 값이 나오는 delta값 전까지의 sequence를 프리패치 하게 된다.

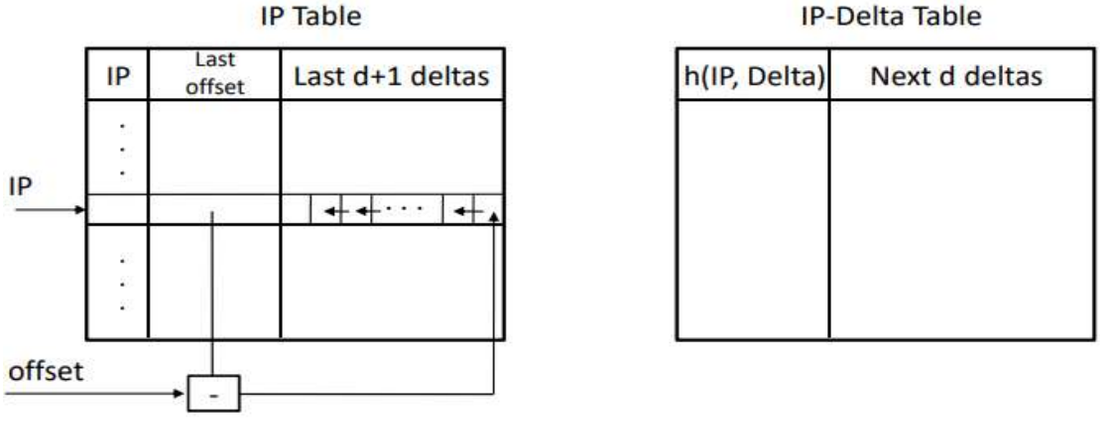

그림 2 Sangnam IP Table learning 과정

두 번째는 learning 과정 즉 두 table을 update하는 과정이다. 예측과정이 끝나고 IP Table에서 update 과정이 일어나는데 이 때 Last offset과 deltas의 update가 필요하다. 기존에 있는 Last offset 값과 현재 들어온 IP값에 offset으로 delta값을 구해 deltas list에 FIFO으로 제일 끝 쪽에 넣게 되고 Last offset 값을 현재 들어온 IP값의 offset으로 변경하게 된다.

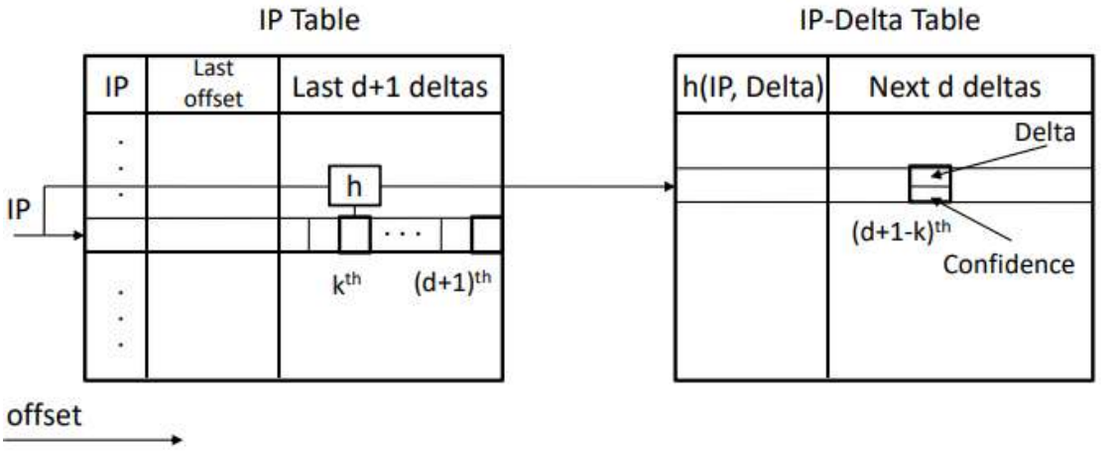

그림 3 Sangnam IP-Delta Table learning 과정

그 다음은 IP-Delta Table을 update하는 과정이다. IP Table의 deltas list가 변했기 때문에 IP-Delta Table에서 관련된 값들을 전부 갱신해야한다. 먼저 IP Table의 delta list들 중 왼쪽부터 k번째라고 했을 때 그 k번째에 있는 delta 값과 현재 IP 값으로 찾은 IP-Delta Table에서 Next d deltas list의 d+1-k번째 값이 새로 들어온 delta 값으로의 갱신이 필요한데 만약 이 delta 값이 list의 d+1-k번째에 있던 값이랑 동일하다면 Confidence counter값을 올려주고 만약 새로운 값이면 갱신 후 Confidence값을 reset해주게 된다.

IP-based Stride Prefetcher는 IP-Delta-based Seqence Predictor가 miss가 많이 발생하거나 낮은 confidence로 인해 예측이 불가능 할 때 IP Table의 d deltas를 다시 이용하여 이전 delta 값을 stride로 이용하여 그 다음 값을 예측하는 방식이다. 마지막으로 Next-line Prefecther는 위 두 방식으로 사용이 불가능 할 때 NL buffer를 이용하여 hit고 insertion counter를 두어 이를 비교하여 다음에 사용될 데이터를 탐색할 시간을 줄이는 방식이다.

2) Accurately and Maximally Prefetching Spatail Data Access with Bingo[2]

본 프리패는 머신러닝을 사용하지 않고 하드웨어적인 구현을 통해 만들어진 프리패이다. 메모리 접근 패턴의 유사성을 활용하여 메모리를 참조하는 프리페칭 기법으로 <event,footprint> 쌍을 history table에 저장하게 되는 방식이다. 이 때 event로 사용될 수 있는 패턴은 PC+Address, PC+Offset, Address, PC, Offset으로 longest event를 저장하게 된다면 프리패치 했을 때 정확성이 높게 되지만 그만큼 프리패치 될 확률이 줄어들게 되는 것이다. 반대로 shortest event를 저장하면 정확성이 낮지만 프리패치 될 확률이 높아지는 것이다. footprint는 접근한 page 안의 사용된 블록을 1로 표기하여 나타낸다.

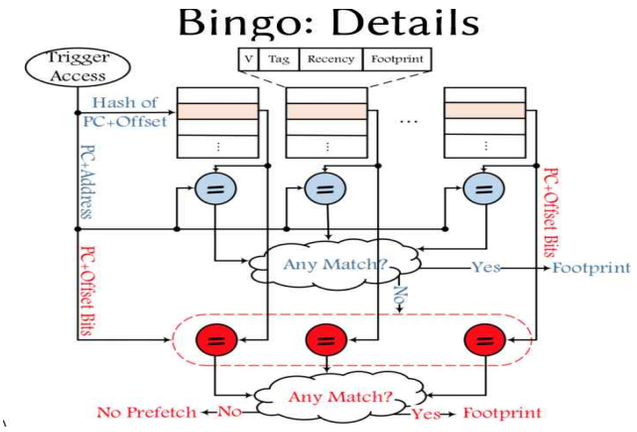

그림 4 Bingo predict 과정

기존에는 프리패치를 위해 PC+Address, PC+Offset 2개의 history table이 존재해 1개에는 PC+Address 정보 나머지는 PC+Offset 정보를 저장했지만 BINGO 모델에서는 PC+Address 정보를 이용하여 PC+Offset의 정보까지 알 수 있음으로 table을 1개만 사용할 수 있게 된다. 먼저 short event를 통해 history table을 indexing하여 어떠한 set을 찾게 되고 그 set중에 long event를 tag로 사용해서 찾게된다. 만약 일치하는 것이 있다면 그 데이터를 프리패치 해온다. 만약 tag되는게 없다 하더라도 short event로 프리패치 하게 된다.

3) empirical evaluation of gated recurrent neural networks on sequence modeling[3]

우리는 다양한 RNN에 대하여 공부하기 위해 'empirical evaluation of gated recurrent neural networks on sequence modeling'이라는 논문을 참고했다. 위 논문에서는 vanilla RNN과 LSTM을 설명하면서 GRU라는 새로운 RNN을 소개한다.

① Vanilla RNN

Vanilla RNN은 가장 기본적인 형태의 rnn이며, tanh 함수를 이용한다.

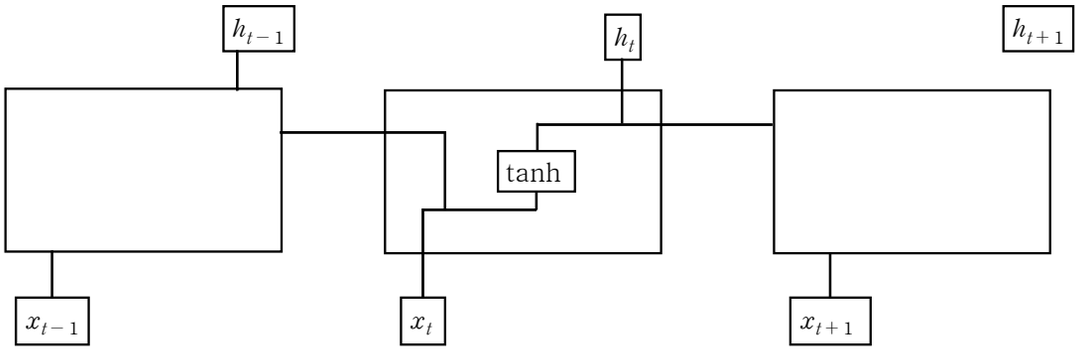

그림 5 Vanilla RNN

Vanilla RNN은 이전 히든 스테이트의 값(h_{t-1})을 받아 그 정보를 현재 입력에 반영하여 현재의 히든 스테이트 값과 아웃풋을 내는 형식이다. 이 때, 히든스테이트의 값과 아웃풋의 값은 $h_t = \tanh(W_{hh}h_{t-1} + W_{xh}x_t), y_t = W_{hy}h_t$로 결정된다. 하지만 Vanilla

RNN은 학습 데이터가 길어질수록 gradient 값이 0에 수렴하거나(vanishing gradient) 무한대로 발산하는(exploding gradient)문제가 발생하기 때문에 이를 해결하기 위해 LSTM이나 GRU를 이용하게 된다.

② LSTM(Long Short Term Memory)

LSTM은 크게 3개의 gate와 cell state 가지며 서로 영향을 주고받는 구조로 이루어져 있다. 3개의 gate는 Forget gate, Input gate, Output gate라는 이름을 가지고 있다. LSTM의 상세한 내부 구조는 그림과 같이 이루어져 있다.

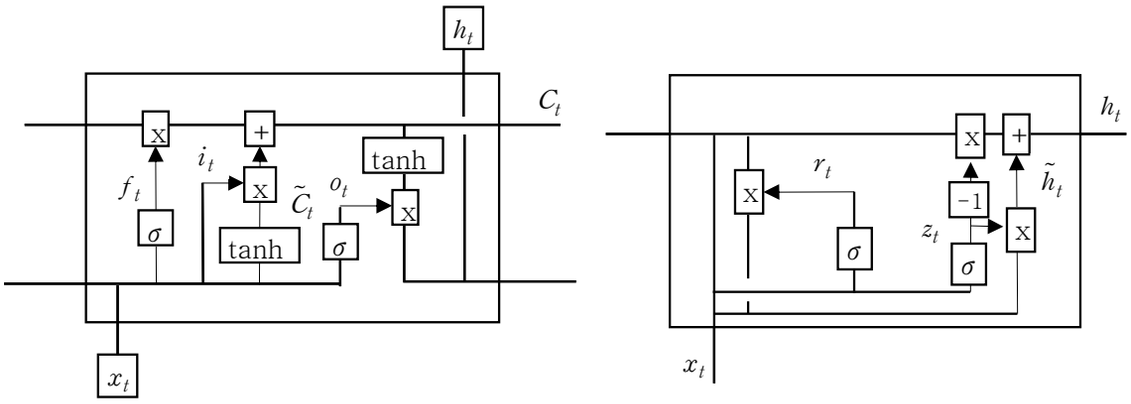

그림 6 LSTM(좌)과 GRU(우)의 세부구조

1) Forget gate: 앞에 들어온 정보 중에 어떤 것을 사용할 것인지 결정해준다.

$f_t = \sigma(W_f \cdot [h_{t-1}, x_t] + b_f)$

2) Input gate: input값 중에 어떤 값을 cell state에 업데이트할지를 결정하는 layer

$i_t = \sigma(W_i \cdot [h_{t-1}, x_t] + b_i)$

$\tilde{C}_t = \tanh(W_C \cdot [h_{t-1}, x_t] + b_C)$

3) cell state update: forget gate와 input gate에서 구한 값을 이용하여 cell state의 값을 update한다.

$C_t = f_t * C_{t-1} + i_t * \tilde{C}_t$

4) output gate: cell state 와 인풋 값을 이용하여 output값을 도출해낸다.

$o_t = \sigma(W_o[h_{t-1}, x_t] + b_o)$

$h_t = o_t * \tanh(C_t)$

③ GRU(gated recurrent units)

GRU는 LSTM보다 parameter의 개수가 약 3/4정도이고, LSTM과 비슷한 성능이라고 알려져 있으며 LSTM과 비슷하게 reset gate, update gate가 존재한다.

1) reset gate: 과거의 정보를 어느정도 reset시켜준다.

$r_t = \sigma(W_r[h_{t-1}, x_t] + b_r)$

2) update gate: LSTM의 input gate와 output gate를 합쳐놓았다고 생각하면 좋다.

과거와 현재의 정보를 적당히 비율로 섞어 나타낸다.

$u_t = \sigma(W_u[h_{t-1}, x_t] + b_u)$

3) Candidate: 은닉층의 정보를 그대로 사용하지 않고, reset gate의 결과를 이용하여 값을 얻어낸다.

$\tilde{h}_t = \tanh(Wh_{t-1} * r_t + Ux_t + b)$

$h_t = (1-u_t) \otimes \tilde{h}_t + u_t \otimes h_{t-1}$

4) 머신러닝 기법을 적용한 하드웨어 데이터 프리패치 기법 구현[4]

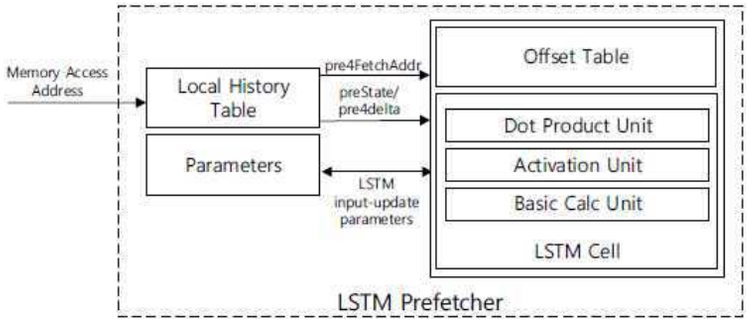

그림 7 LSTM Prefetcher 모델 구성도

LSTM 기반 프리패는 머신러닝 알고리즘인 LSTM을 통해 메모리 접근 주소들의 간격을 델타라고 한다면 이를 수집하여 학습을 수행한다. 즉 다음 프로세서가 접근할 주소를 LSTM을 통해 델타값을 예측한다. LSTM은 학습에 들어가는 패턴들이 테이블의 저장 공간에 기억되지 않고 연산 과정으로 기억 및 예측을 수행하게 된다.

이는 위 그림을 보면 Local History Table, Offset Table, Parameter Table과 LSTM 유닛으로 구성되어 있다. LSTM 기반의 프리패는 프로세서에서 메모리 접근을 수행하기 위해서 메모리 접근 주소가 프리패로 전달된다. 전달된 주소는 페이지 주소를 바탕으로 Local History Table을 검색한다. 페이지 주소에 해당하는 Column이 존재할 경우 해당 Column들의 정보로 델타를 계산하고 Colum의 정보를 갱신한다. 그 후 Column의 pre4Delta와 preState, LSTM Unit에 전달되고 LSTM Unit은 전달받은 데이터를 바탕으로 델타를 예측한다. 그 다음 현재 메모리 접근 주소와 델타값을 더하여 프리패치를 수행한다. 만약 Local History Table에 해당 Column이 존재하지 않는다면 새로 Column을 추가하고 Offset Table에서 메모리 접근 주소의 오프셋을 가지고 Offset Table을 검색한다. Offset Table의 Column에 델타값이 들어있다면 해당 델타값과 메모리 접근 주소를 더해 프리패치를 수행한다.

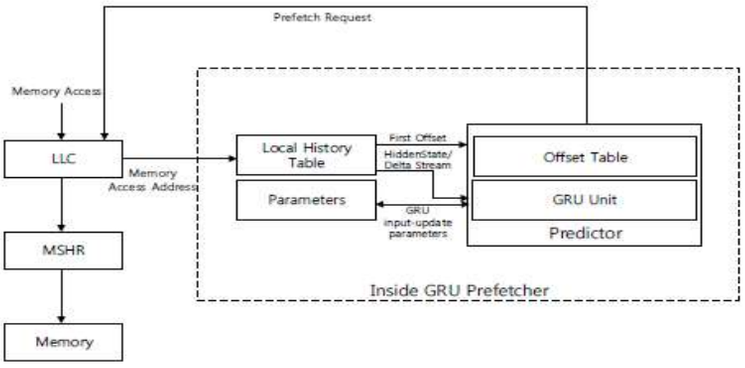

그림 8 GRU Prefetcher 모델 구성도

GRU 기반 프리패는 위의 LSTM기반의 프리패를 개선하여 만든 것으로 LSTM을 개선한 GRU 알고리즘을 채택한 방식으로 GRU는 LSTM의 장기 의존성 문제에 대한 해결책을 유지하면서 은닉 상태를 업데이트하는 계산을 줄였다. 즉 LSTM은 3개의 게이트가 존재했는데, GRU는 2개의 게이트만이 존재하게 되어 GRU는 LSTM과 성능은 유사하나 복잡했던 LSTM의 구조를 단순화했다.

GRU를 주소의 예측기로 사용함으로써 LSTM 기반의 프리패치 기법보다 예측성능이 향상되고 파라미터가 감소하여 동작에 필요한 리소스를 절감하고 하드웨어 크기를 줄이게 되었다.

먼저 위 그림을 보면 Local History Table에서 최근에 접근한 메모리 접근 주소의 델타 기록을 관리한다. 델타 기록은 Delta Stream 필드에 저장되며 GRU는 델타 기록을 바탕으로 앞으로 사용할 것으로 예측되는 데이터의 접근 주소를 생성할 때 사용된다. 그 다음 Offset Table을 이용하는데 이는 Local History Table에서 만들어진 엔트리에서 Delta Stream 필드의 내용이 없을 경우에 동작한다. GRU Unit은 시계열 순으로 저장된 Delta Stream의 정보로 동작하기 때문에 처음 새로 엔트리가 생성되었을 경우 오프셋만 존재하고 Delta Stream이 존재하지 않으면 GRU Unit에 의한 예측이 수행되지 않기에 이를 위해 고안되었다.

GRU Unit을 통하여 예측 및 관리하는데, 이는 Delta Stream 필드의 델타가 한 개 이상 있을 때 동작한다. GRU Unit에서 Local History Table 엔트리의 Delta Stream과 Hidden State 필드의 값과 Parameter Table의 Parameter가 입력되면 delta stream을 시간 순서대로 연산을 거쳐 예측한 델타 값을 출력하게 된다.

5) MPMLP: A Case for Multi-Page Mulit-Layer Perceptron Prefetcher[5]

MPMLP은 머신러닝 기반의 hybrid 프리패로 머신러닝을 사용한 프리패와 기존의 프리패를 동시에 사용하는 기법이다. 전통적으로 사용되는 BO(best offset) 프리패는 지배적인 메모리 접근 패턴이 있을 때 데이터 프리페칭에 효율을 보인다. 즉 전통적인 프리패들은 자신 주변의 데이터의 패턴에 의존하게 되는 방식이다. 그러나 현대 데이터들은 물론 기본적으로 시간, 공간적 지역성을 보이지만 데이터

bandwidth가 넓은 데이터 또한 많이 보이고 있다. 다시 말하자면 전통적인 프리패는 경향성이 데이터 전역으로 나타나는 심지어 랜덤하게 보이는 복잡한 데이터 set에 대해 낮은 효율성을 보이게 된다. 그에 반해 머신러닝 기법은 지배적인 데이터 패턴이 없더라도 데이터 전역의 통계적인 트렌드 분석이 가능하기 때문에 전통적인 프리패의 단점을 보완할 수 있게된다.

위 프리패치 방식에는 총 3가지의 구성요소가 존재하는데 이는 BO prefetcher, MLP-based prefetcher, Page Transition Table이다. 이 때 MLP-Based prefetcher는 1개의 input layer, 2개의 hidden layer와 1개의 output label로 구성되어 있으며 위 그림과 같이 먼저 현재 메모리 접근을 기준으로 총 h개를 input으로 사용한다. 다음 l개 뒤의 메모리 주소를 프리패치 하는데 이는 프리패치에 시간적 고려를 위해서인데, 프리패치 요청이 들어오고 프리패치가 완료될 때까지 시간이 지나서 이미 프리패치가 끝난 후에는 그다음 메모리의 접근이 있게 된다. 이를 고려해주기 위해서 l개의 메모리 주소 뒤의 메모리 k개를 프리패치 하게 된다.(논문에서 제시한 값은 h=4, l=5)

Page Transition Table은 메모리가 접근하는 page의 transition에 대해 기록해 놓는 table이다. 즉 IP가 page x에 접근한 후 다음 page y에 접근했다면 (x,y)의 형태로 IP를 기록하는 것이다. 이는 다른 page로의 transition은 다음 번 x에 접근이 다시 y로의 접근하는 반복할 가능성이 높기 때문에 이러한 page transition을 기록해 놓는 것이다.

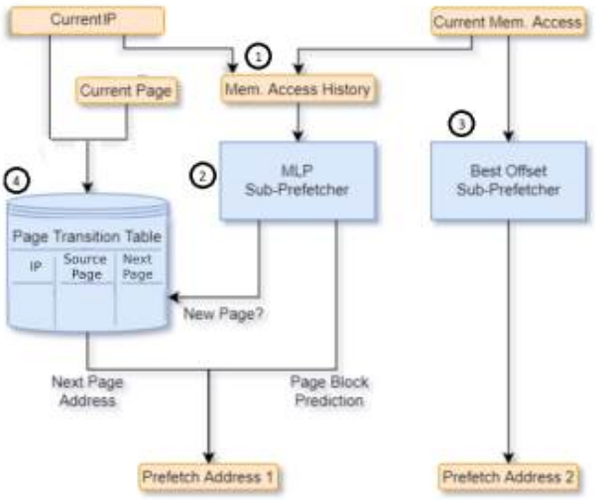

그림 9 MPMLP Prefetcher 모델 구성도

MPMLP 프리패는 총 2개의 프리패치를 진행하게 되는데, 먼저 메모리 접근 요청이 있을 경우 MLP를 가지고 1개, BO를 가지고 1개 총 2개의 메모리 접근 주소를 프리패치하게 된다. 이렇게 한다면 BO의 단점인 지배력이 낮은 메모리에 접근에 대한 보안으로 MLP를 사용하게 된다. MLP로 프리패치 할 offset을 찾게 되고 Page Transition Table을 통해 page transition 있었는 지를 확인하게 된다. 1개를 더 프리패치 함으로 높은 확률로 프리패치를 성공할 수 있게 되어 BO만 사용했을

때에 비해 IPC 측면에서 유리하다. 그러나 ACCURACY 측면에서 필요 없는 프리패치가 될 수 있기 때문에 기존 BO만 사용했을 때에 비해서 어느 정도 감수가 필요하게 된다.

■ 제안 작품 소개

실제로 구현에 대해 소개합니다. 자세하면 자세할수록 좋습니다. 이론적 배경과 시스템구성은 필수입니다.

1. data set의 구성, 가공 및 용어 정리

```
11, 145, 28e837c87040, 406bc9, 0 ····· ①
12, 160, 28e837c89f00, 406bd0, 0 ····· ②
31, 375, 28e837c87000, 407040, 0 ····· ③
27, 407, 28e837c86240, 406d87, 0 ····· ④
```

그림 10 data set의 구성

(1) data set의 구성

data set은 위의 그림 10과 같이 구성되어 있다. 왼쪽부터 Unique Instr Id, Cycle Count, Load Address, Instruction Pointer of the Load, LLC hit/miss를 나타내며, Load address와 Instruction Pointer of the Load의 값은 16진수로 표현되어 있다. rnn_prefetcher는 현재까지 접근한 data set을 이용하여 미래에 올 Load address의 값들을 예측하게 된다.

(2) data set의 가공

Load address는 항상 64로 나누어 떨어지므로 rnn_prefetcher에 값을 넣을 때에는 64로 나눈 값이 사용된다. 예를 들어 ①의 Load address를 사용한다면 0x28e837c87040값을 사용하는 것이 아닌 64로 나눈 값인 0xA3A0DF21C1를 사용하게 된다. 앞으로 Load address라는 용어는 실제 접근한 주소가 아닌 64로 나눈 값이라는 뜻으로 사용하겠다.

(3) 델타(delta): 현재 접근한 Load address와 그 전에 접근한 Load address의 차이이다. 마지막 프리패인 rnn_prefetcher_3에서는 이 의미가 조금 달라지지만 비슷한 의미를 가지고, rnn_prefetch_1과 rnn_prefetcher_2에서는 이를 따른다. 예를 들어 현재 ②의 Load address에 접근했고 ③의 Load address에 접근한다고 하면 delta의 값은 (0x28e837c87000(③의 실제 접근 어드레스) - 0x28e837c89f00(②의 실제 접근 어드레스)) / 64의 값인 –0xBC가 된다.

(4) seq_length: seq_length는 Rnn의 길이를 의미한다.

2. rnn_prefetcher_1 model

(1) 첫 번째 input, delta

가장 간단하게 생각할 수 있는 rnn_prefetcher_1은 delta값을 차례대로 LSTM에 넣어 다음 delta값을 예측하는 것이다. 다만 delta값이 음수가 나올 수 있으므로 실제 입력인 δ에는 500을 추가한 delta+500의 값이 들어간다. 또한 delta값이 –500 미만인 값에 대해서는 0을 넣고, delta값이 500 초과인 값에 대해서는 1000을 넣어 주었다. 이로써 delta값은 항상 0이상 1000이하의 값으로 매핑 된다. 이렇게 생각한 이유는 data들의 delta값들의 분포를 본 결과(그림 11) 거의 모든 delta값이 –500에서 500사이의 값에 존재하기 때문이다.

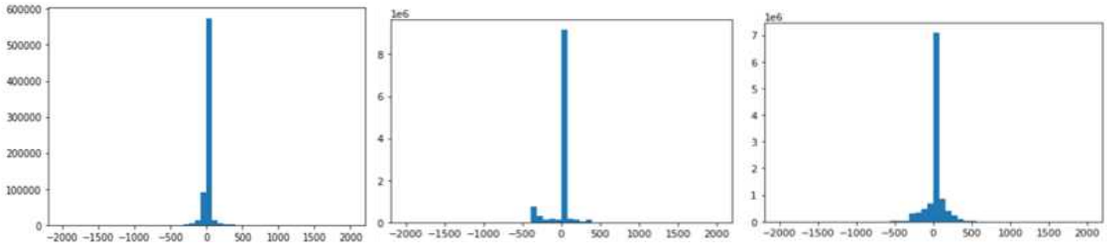

그림 11 473.astar-s0.txt, 410.bwaves-s0.txt, 459.GemsFDTD-s1.txt의 delta값의 분포
x축은 delta값, y축은 data set의 개수

(2) 모델의 구성

모델은 그림5와 같이 구현하였다. 순서대로 접근한 Load address를 이용하여 δ값을 구하면 다음 δ값을 예측해준다. output은 확률값으로 나타나며, 만약 p_0의 값이 가장 크다고 하면 다음에 나올 delta값은 0 – 500 = -500이 된다. 그 후, 다음 접근할 실제 address를 구하기 위해 (Load_address(현재 접근한 Load_address) – 500(delta 값)) * 64를 계산하여 구한다. 또한 이 프리패처는 항상 확률이 가장 높은 2개의 delta값을 프리패치해준다.

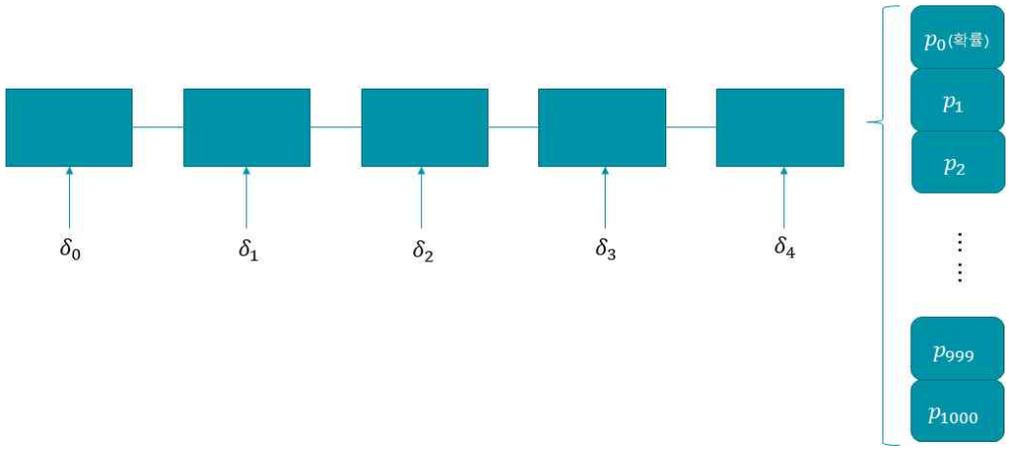

그림 12 rnn_prefetcher_1의 구성

(3) 장점과 단점

- 장점: 모델의 파라미터가 적어 비교적 단순하다.
- 단점: delta값이 –500과 500 사이에 있기 때문에 그보다 작아지거나 커지면 값을 예측이 불가능하다. 같은 delta값들을 입력으로 받아들이면 항상 같은 출력을 내놓게 되는데 다른 page에서 같은 delta입력을 같지만 다른 출력이 나올 수 있으므로 성능이 좋지 못하다.

3. rnn_prefetcher_2 model

(1) model의 구성

rnn_prefetcher_2는 rnn_prefetcher_1에 현재 접근한 Load_address정보를 넣어 개선한 모델이다. 모델의 구성은 그림 6과 같다.

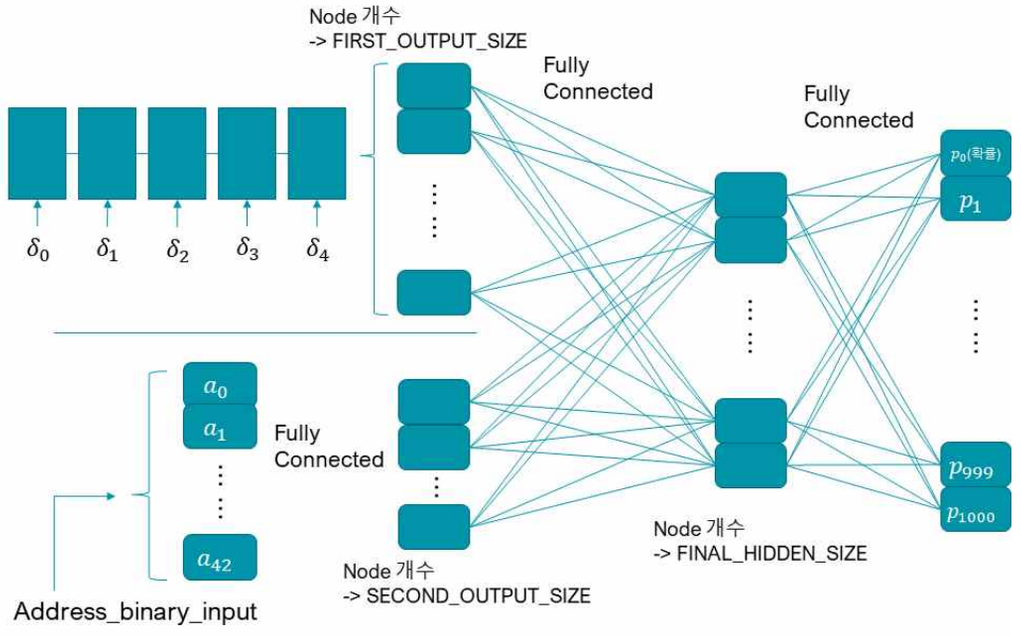

그림 13 rnn_prefetcher_2 모델의 시각화

(2) 2번째 input, address binary input

먼저, rnn_prefetcher_1과 다르게 address_binary_input이라는 input이 추가되었다. 이 input은 바로 전에 접근한 Load_address정보를 넣어주기 위해 Load_address를 2진법으로 만든 후, scale값(0.2)를 곱하여 넣어주었다. 이 작업을 해줌으로써 다른 page에서 같은 delta입력을 같지만 다른 출력이 나올 수 있는 상황에 대해서 보완해 줄 수 있다. 그 후, 기존의 rnn_prefetcher_1과 concatenate를 하고, hidden layer를 하나 더 쌓아 모델을 완성하였다. 하지만 모델의 복잡도가 증가함에 따라 tuning을 해줘야할 hyperparameter의 개수가 증가하였다. 마찬가지로 rnn_prefetcher_1과 동일하게 확률이 가장 높은 2개를 프리패치 해준다.

(3) 장점과 단점

- 장점: 다른 page에서 같은 delta입력을 같지만 다른 출력이 나올 수 있는 상황에 대해 비교적 정확한 값을 내놓을 수 있다.
- 단점: 여전히 delta값이 –500과 500 사이에 있기 때문에 그보다 작아지거나 커지면 값을 예측이 불가능하다. 모델이 rnn_prefetcher_1보다 더 복잡하다.

3. rnn_prefetcher_3 model

(1) model의 input과 output의 변경

delta값이 너무 커지거나 너무 작아지는 것에 대해 생각해주기 위해서 rnn_prefetcher_3를 구현하였다. 모델의 구조는 rnn_prefetcher_2와 매우 비슷하나 input과 output이 차이가 난다. 먼저 메모리를 적당한 값으로 쪼갠다. 이 때 적당한 값을 PageSize라고 부르도록 하겠다. 예를 들어, PageSize가 128이라면, Load address 0 ~ 127 (실제로는 0 ~ 127*64)이 첫 번째 페이지를 이루고 128 ~ 255 (실제로는 128*64 ~ 255*64)가 2번째 페이지를 이룬다. 그렇다면 만약 page 내부에서만 Load address값이 바뀐다면, 나올 수 있는 delta값은 –127(-PageSize + 1) ~ 127(PageSize – 1)이 될 것이다. 이를 이용하여, 그림 7과 같이 input을 변형시켰다. Page 내부에 있는 address에 접근하면, offset과 delta값이 하나씩 추가되는 형식이다. 각 블록에 대해 하나씩 살펴보자. Page에는 Page의 번호 값이 들어간다. Offset은 Page의 첫 번째 부분부터 얼마나 떨어져있는지를 나타낸다. 또 delta에는 현재 offset과 이전의 offset값의 차이가 들어가게 된다.

그림 14 rnn_prefetcher_3의 input

예를 들어 다음과 같은 상황을 가정해보자. PageSize가 100이라고 가정하고, 210, 250, 350, 230, 330, 810, 820, 220, 340(실제로는 210 * 64, 250 * 64, 350 * 64 .....인 곳에 접근했음)에 접근했다고 가정해보자. 처음에 input이 아무것도 없을 때, 210 Load address에 접근했을 경우, Page1에 해당하는 block에 3이라는 값이 쓰이면서 offset 1은 10, delta 1은 0이 쓰이게 된다. 그 후, 250에 접근했을 때 Page의 값은 3이고, Page 1에 3이 이미 존재하므로 Page 1의 offset 2에는 50이, delta 2에는 40 (50 – 10)이 쓰이게 된다. 350에 접근하게 되면, Page 2에는 4라는 값을 써주고 새로운 Page 정보를 만들게 된다. 그 후, Page 2에 해당하는 offset 1, delta 1에는 각각 50, 0이라는 값이 들어간다. 그 후의 동작은 다음과 같다.

- 230에 접근한 경우 3 값을 갖는 Page에 offset 3에 30, delta 3에 –20을 추가한다
- 330에 접근한 경우 4 값을 갖는 Page에 offset 2에 30, delta 2에 –20을 추가한다
- 810에 접근한 경우 9에 해당하는 Page가 존재하지 않으므로, 9에 해당하는 Page를 만들어주고, offset 1에 10, delta 1에 0을 추가한다.
- 820에 접근한 경우 9 값을 갖는 Page에 offset 2에 20, delta 2에 10을 추가한다
- 220에 접근한 경우 3 값을 갖는 Page에 offset 4에 20 delta 4에 –10을 추가한다
- 340에 접근한 경우 4 값을 갖는 Page에 offset 3에 40, delta 3에 10을 추가한다

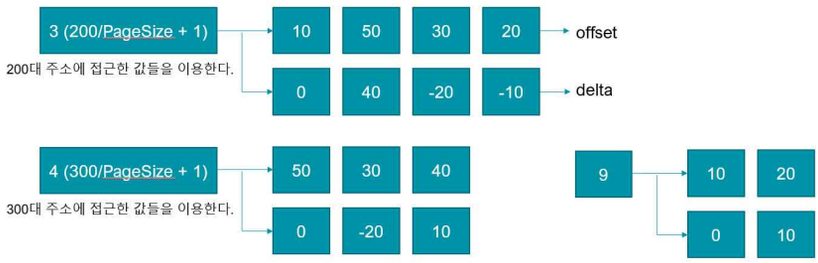

그림 15 rnn_prefetcher_3의 input예시

그렇다면, 실제로 model에 들어가는 input과 output을 살펴보자. rnn_prefetcher_3는 rnn_prefetcher_2와 비교했을 때, input과 output만 다르고, model의 구성은 동일하다. rnn_prefetcher_2에서 delta input과 address binary input이 있었는데, delta input에서는 그림 9와 같이 delta input을 차례대로 넣어주고, address binary input은 그 Page를 binary값으로 바꾼 값을 넣어주게 된다. output은 그 뒤의 delta값을 넣어주게 된다.

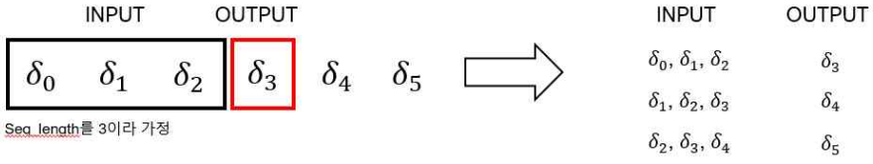

그림 16 model의 input과 output

(2) output의 개선

우리는 rnn_prefetcher_1,2를 구현하면서 outptut의 개선 필요성을 느꼈다. 왜냐하면, 우리가 넣어주는 실제 label에는 델타값을 한 개만 넣어주게 되는데, 실제로 프리패치를 할 때에는 2개를 프리패치하기 때문이다. 따라서 그림9에 나와있는 output의 형태를 그림16과 같이 고쳐, model이 한꺼번에 여러개의 output을 기억할 수 있게 만들어주었다. 하지만 이렇게 output을 구성하게 된다면, 비슷한 시간에 같은 값이 2번 이상 프리패치가 될 수 있는 문제가 발생할 수 있다. 이를 방지하기 위해 page별로 pre_addr이라는 array 변수를 둬서 이전에 접근한 address인지 확인하여 프리패치를 해주게 된다.

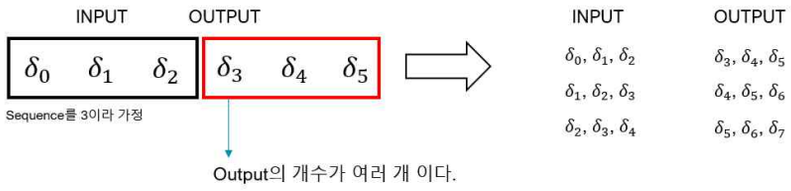

그림 17 여러개의 output

■ 구현 및 결과분석 (6페이지 내외)

소정의 결과를 소개하고 결과에 대한 분석을 적습니다.

우리는 결과를 분석하기 위해서 ChampSim simulator를 활용하여 우리의 모델을 평가했다. 또한 현재 까지는 SPEC2006 trace를 이용하여 우리 모델의 성능에 대해서 평가했다. 또한 각각의 벤치마크에 대해서 첫 1억개의 instruction을 통해서 train했고 다음 1억개의 instruction을 통해서 evaluate했다.

다음 표는 spec06의 trace file들을 이용하여 accurcay, coverage, MPKI, IPC를 측정한 표이다. 여기서 가장 중요한 값은 IPC로 컴퓨터의 성능에 직접적인 영향을 미친다. 현재 프리패치가 없는 것에 비해 약 20%의 IPC성능 향상이 있었다.

표 1 trace file에 따른 성능 비교표

Trace	Accuracy	Coverage	MPKI	MPKI_Improvement	IPC	IPC_Improvement
437.leslie3d-s2	52.69592	28.99386	18.15462	28.64852	0.619209	32.66481
462.libquantum-s1	99.91357	19.64112	22.9218	19.6411	0.716962	17.28941
410.bwaves-s1	94.0409	32.45737	15.41611	32.45752	0.786984	26.59518
473.astar-s0	43.85824	29.8031	1.134677	22.27781	1.05778	4.955152
470.lbm-s0	99.96805	8.251139	29.1071	8.25433	0.634808	9.899866
470.lbm-s1	99.97639	8.379703	27.4703	8.382504	0.67219	11.16789
482.sphinx3-s0	66.6023	42.0644	7.30003	41.60392	0.853692	29.04948
437.leslie3d-s1	59.58174	31.61458	18.08743	31.3715	0.644161	30.81031
433.milc-s2	30.54392	18.63837	16.6768	18.61401	0.643591	13.27696
433.milc-s0	27.94464	20.83051	19.29603	20.82921	0.546944	15.75118
473.astar-s1	52.55039	46.01626	1.275017	45.09209	0.479587	7.088437
410.bwaves-s0	85.61507	31.76448	15.57426	31.76463	0.774072	24.89101
433.milc-s1	47.00328	30.92213	18.40261	30.91166	0.542219	13.97908
437.leslie3d-s0	59.15551	28.88175	19.10317	28.7773	0.623958	30.15693

이는 전통적인 프리패에 비해 낮은 IPC 상승률을 보이고 있다. 위와 같은 결과는 아래와 같은 hyperparameter의 조정이 아직 끝나지 않아서 나타난 것으로 보인다.

우리가 조정해야 할 hyperparameter는

1) seq_length: RNN의 길이를 나타내며, delta값이 한번에 몇 개나 들어가는지를 알려주는 변수이다.

2) FINAL_HIDDEN_SIZE: 마지막 hidden layer의 node개수를 의미한다. 다른 hidden layer의 node개수를 변화시키는 것보다 이를 변화시키는게 accuracy변화가 컸으므로, 중요한 hyperparameter라고 생각된다.

3) LSTM과 GRU의 선택: LSTM은 안정적이지만, GRU가 parameter의 수가 더 적고, 특별한 경우 GRU가 더 좋은 모델이므로 이 또한 고려해야 될 요소라고 생각된다.

4) cut off: output의 값은 항상 0과 1사이의 값이 나타나고 프리패치er는 가장 큰 값, 그 다음으로 큰 값을 선택하게 된다. 만약 cut off value를 둔다면, 0과 1사이의 값을 두어 cut off value이상일 때 그 값을 프리패치하여 필요 없는 프리패치 값들은 줄어들게 되고, accuracy가 올라가고 coverage, ipc가 감소하는 효과를 가져올 것이다. 그러므로 이를 잘 조절하는 것이 필요하다.

5) output의 node개수: output node의 개수가 증가한다면, 그만큼 더 많은 데이터가 필요하고, 모델도 더 복잡해질 것이다. 하지만 데이터만 충분히 많고 fit이 잘 되었다면 accuracy, coverage, ipc는 전체적으로 증가할 것이다. 반대로 output node의 개수를 줄인다면 모델은 단순화 될 것이고 데이터가 충분히 많지 않아도 잘 작동하게 될 것 임으로 이를 잘 조절해주는 것이 중요하다고 판단된다.

6) batch size: 많이 중요하진 않지만 어느 정도의 값 이상으로 잡아야 된다고 생각

되므로 이 기준을 정해주는게 중요하다.
7) epoch: epoch을 많이 해도 drop out을 설정해줘서 어느정도 잡아주긴 하지만, epoch이 크면 train시간이 길어져서 적당히 작은 값으로 골라서 설정해줘야 한다고 판단된다.
8) drop out rate: mlp에서 drop out을 해줌으로써 overfitting을 방지할 수 있는데 drop out을 하는 정도를 사람이 직접 결정해 줘야 한다.
이 외에도, RNN units, embedding dimension, FIRST_HIDDEN_SIZE, SECOND _HIDDEN_SIZE 등 다양한 파라미터들이 존재한다. 하지만 중요하지 않은 값들은 기본적인 값으로 설정해주고, 중요한 값들만 grid search나 random search를 이용하여 최적의 hyperparameter를 찾을 계획이다.

■ 결론 (1페이지 내외)

결론을 맺습니다.
가장 중요한 부분입니다.

데이터 프리패치는 컴퓨터 시스템 성능 향상을 위해서 크게 기여했다. 프로세서가 원하는 데이터가 캐시 메모리에 없다면 메인 메모리로의 요청이 필요했고 그에 따라 프로세서는 작업을 수행하기 위한 데이터를 오래 기다려야 했다. 그러나 프로세서가 사용할 가능성이 높은 데이터를 미리 캐시메모리로 올려 놓게 된다면 프로세서는 기다리는 시간 없이 작업을 수행할 수 있게 되고 IPC의 향상을 보이게 된다.

현재 주로 사용되는 프리패는 머신러닝 기법이 사용되지 않는 것이 일반적이다. 전통적인 프리패 기법들은 좋은 성능을 보이고 있지만 많은 프리패들이 개발되어 왔고 발전속도는 현재 더딘편이다.

시계열 데이터의 패턴을 다루는 데 강점을 보이는 RNN 계열의 머신러닝 기법은 현재 많은 분야에서 활용되고 있다. 우리가 다루는 데이터 또한 시간, 공간적 지역성을 보이는 시계열 데이터로 분류할 수 있는데, 패턴 분석에 능한 머신러닝을 사용하여 더욱 효율적으로 다음 메모리를 할 수 있게 되는데, 기존의 프리패치 기법들은 이전 데이터를 기반으로 데이터를 프리패치 하게 되지만 머신러닝을 사용하게 된다면 다음에 사용될 확률이 높은 데이터를 가져 올 수 있게 된다.

우리가 제안한 머신러닝 프리패치 기법은 LSTM 기반의 프리패로 현재 접근주소 기준으로의 delta값을 활용하고 현재 접근한 주소를 2진법으로 사용하여 예측 확률을 높혔다. 그리고 output값의 중복을 막게 되어 캐시가 불필요한 데이터를 프리패치 하지 못하도록 막았다. 또한 output을 여러개 가져올 수 있게 만들어 IPC 향상을 도모했다.

현재 우리가 제안한 모델은 프리패가 없는 모델을 기준으로 IPC가 평균 20% 정도의 성능향상이 있었다. 이는 전통적으로 사용되던 프리패의 평균IPC 상승률 보다는 아쉬운 수준이다. 그러나 아직 hyperparameter을 조정해야 하고 필요하다면 모델의 간략한 수정할 수 있기 때문에 모델 성능의 발전 가능성이 남아 있다고 생각한다.

■ 참고문헌 (1페이지 내외)

[1] "3rd data prefetching championship." [Online]. Available: https://dpc3.compas.cs.stonybrook.edu/?final_programs.

[2] "3rd data prefetching championship." [Online]. Available: https://dpc3.compas.cs.stonybrook.edu/?final_programs.

[3] Junyoung Chung, Caglar Culcehre, KyungHyun Cho, Yoshua Benglo, "Empirical Evaluation of Gated Recurrent Neural Networks on Sequence Modeling," arXiv dec 2014

[4] 송경환 외 3명, "머신러닝 기법을 적용한 하드웨어 데이터 프리패치 기법구현." 대한전자공학회, 2019

[5] "ML-Based Data Prefetching Competition." [Online]. Available: https://sites.google.com/view/mlarchsys/isca-2021/ml-prefetching-competition.