The POP Detector: A Lightweight Online Program Phase Detection Framework

Karl Taht

Department of Computer Science University of Utah

Salt Lake City, USA [taht@cs.utah.edu](mailto:taht@cs.utah.edu)

James Greensky

Intel Labs Intel

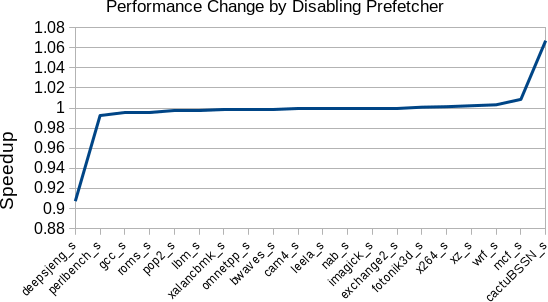
Hillsboro, USA [james.j.greensk](mailto:james.j.greensky@intel.com)[y@intel.com](mailto:y@intel.com)

Rajeev Balasubramonian Department of Computer Science University of Utah

Salt Lake City, USA [rajee](mailto:rajeev@cs.utah.edu)[v@cs.utah.edu](mailto:v@cs.utah.edu)

***Abstract*—**실시간 위상 검출은 다른 프로그램 동작에 기반한 시스템의 동적 적응을 가능하게 한다. 많은 위상 검출 기술이 제안되었으며, 가장 성공적인 단계는 응용 프로그램 코드와 관련된 것입니다.온라인 위상 검출의 범위 내에서, 기법들은 위상 검출 프레임워크의 오버헤드를 완화하기 위해 샘플링을 사용한다. 위상 간격이 충분히 길면 샘플링 접근법은 잘 수행된다**.**

우리는 오버헤드와 위상 검출기 성능 사이의 트레이드오프에 대한 심층 분석을 수행함으로써 위상 간격 길이에 대한 문제를 다시 시작한다. 본 논문에서는 위상 간격 길이, 위상 안정성, 위상 수 사이의 통계적 상충을 포착하는 새로운 메트릭을 제시한다.우리는 온라인 최적화의 맥락에서 더 짧은 단계가 가장 잘 수행되지만, 기존의 구현은 짧은 간격 크기에서 성능 저하와 오버헤드로 어려움을 겪는다는 것을 발견한다.

이러한 차이를 해결하기 위해 POP(Precise Online Phase) 검출기를 제안한다. POP 검출기는 성능 카운터를 사용하여 서명을 작성하는데, 이는 보다 미세한 입도에서 사실상 손실이 없다.두 번째 순서로, 검출기의 단순성은 10M 및 100M 명령 간격에서 각각 런타임 오버 헤드를 1.35% 및 0.09%로 줄입니다.

평균적으로 속도 향상이 거의 1에 가깝지만 S-Curve는 양수 이상치와 음수 이상치를 모두 드러낸다. 왼쪽에서는 많은 워크로드가 프리 페처를 비활성화하는 반면 오른쪽에서는 많은 워크로드가 변경을 선호합니다.간단한 접근방식은 작업부하별로 피쳐를 조정하는 것입니다.

그러나 이러한 접근방식은 실행 중에 프로그램의 동적 동작을 무시할 것이다. 따라서, 우리는 그림 2에서 Spec2017 [1]에서 lbm s의 개별 프로그램 단계에 대한 추가 S-Curve를 계획합니다. 글로벌 관점에서 볼 때, lbms는 공간 프리페처를 활성화하여 한계 성능 향상을 보여준다.그러나, 단계별로 보면, 이 같은 프로그램이 긍정적인 이상치와 부정적인 이상치를 모두 가지고 있어, 프로그램의 특정 간격이나 단계에 적응할 필요가 있다는 것을 알 수 있다..

1. INTRODUCTION

프로그램은 시간 가변적인 동작을 나타내고 단계 간 이동을 나타냅니다. 이는 언제든지 시스템이 해당 단계의 요구 사항에 대해 최적으로 구성되지 않을 수 있음을 의미합니다. 이에 따라 건축가는 특정 응용 프로그램 단계 및 조건에 더 잘 대응할 수 있는 적응 시스템을 개발하고 계속 개발할 것입니다. 가장 낮은 수준에서 분기 예측기 및 데이터 프리 페처와 같은 하드웨어 기능은 변경된 명령어 및 데이터 스트림을 지속적으로 모니터링하여 휴리스틱 예측 변수를 업데이트합니다.이러한 온라인 학습 에이전트는 주기 내의 변화하는 행동을 탐지할 수 있을 뿐 아니라 수백 주기 내의 반응을 업데이트할 수 없는 경량 알고리즘을 사용합니다.

특히 이러한 에이전트에 대한 업데이트 규칙은 많은 워크로드에서 세심한 분석을 통해 선택되어야 합니다. 한 가지 분석 기법은 많은 워크로드에서 피쳐를 변경하는 효과를 캡슐화하는 그래프인 S-Curve를 생성하는 것입니다. 높은 수준의 목표는 간단합니다. 모든 워크로드에서 속도를 극대화합니다. S-Curve는 평균을 왜곡할 수 있는 특이치를 식별하는 데 도움이 된다. 그림 1에서는 S-Curve를 사용하여 L2 공간 프리페처를 인텔 시스템에 비활성화하는 효과를 보여줍니다.

Fig. 1. Spec2017 performance change by disabling spatial L2 prefetcher, using training dataset.

신중한 오프라인 위상 분석 기술에서부터 경량 온라인 위상 검출기 및 최근에는 제어 이론 [2] - [6]에 이르기까지 동적 최적화의 필요성을 해결하기 위해 많은 작업이 수행되었습니다. 이러한 각각의 접근법은 장점이 있다. 예를 들어, 기본 블록 벡터(BBV)는 위상 검출을 위한 시간의 테스트를 서 있는 훌륭한 기준선임이 입증되었다. 하지만, 이 알고리즘은 많은 양의 데이터를 필요로 하기 때문에 온라인 알고리즘에서는 실현할 수 없다.이와 같이, 여러 선행 연구들은 원래 이론적인 하드웨어 지원을 통해 BBV의 근사를 시도했고, 최근에는 정밀 이벤트 기반 샘플링 (PEBS) [3], [4]을 사용하여 실제 시스템에서의 근사를 시도했다.

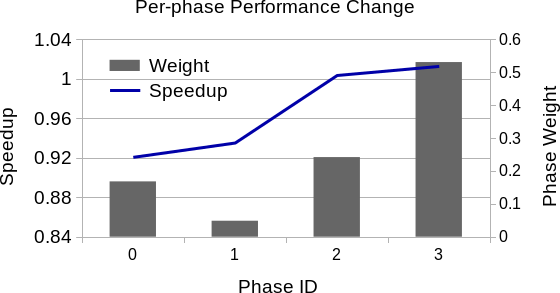


Fig. 2. The lbm s performance change (denoted by the S-Curve) per phase via disabling spatial L2 prefetcher, with respective phase weights denoted by bars. The phase weight refers to the amount of time the program spends in a particular phase, relative to the program as whole. As a result, the sum of phase weights is always 1.

궁극적으로 목표는 개별 단계를 이해하고 단계별로 특징을 최적화하는 것으로 남아 있으며, 이는 요구 사항 집합으로 정의할 수 있습니다.

기능적으로, 모든 위상 검출기는 유사하고 반복적인 행동의 위상을 생성해야 한다. 그러나 온라인 성능 최적화를 위해 프로그램 단계를 사용하는 것은 성공적인 위상 검출 알고리즘을 위한 많은 요구 사항이 필요하다. 첫째, 위상 검출기는 성능 변화와 독립적으로 작동해야 한다. 위상 검출기가 서로 다른 하드웨어 구성을 분석할 수 있도록 하기 위해서는 프로그램 위상은 서로 다른 구성으로 변경되지 않아야 한다. 즉, 위상 검출기는 성능 노브에 불변성을 유지해야 한다. 즉, 위상 A는 성능 노브의 변경 후에도 여전히 위상 A로 라벨이 붙여져 있다.SimPoint와 ScarPhase는 모두 코드 실행에 기반을 두고 있기 때문에 이 요구 사항을 모두 충족시킵니다.

실현가능성 측면에서는 최소한의 프로그램과 시스템 변화을 가진 상품 하드웨어에서 위상 검출 기법이 실현되어야 한다. 심포인트의 입력인 BBV를 수집하는 것은 실시간에서 불가능하다. 이와 같이, ScarPhase와 같은 검출기는 이 오버헤드를 완화하기 위해 샘플링을 사용한다. 그러나 100M의 큰 명령어 구간에서도 이러한 기법의 오버헤드가 3% 이상이며, 세밀한 입도가 요구되면 오버헤드가 기하급수적으로 증가한다는 것을 알 수 있다.이러한 오버헤드는 온라인 성능 최적화의 범위 내에서 바람직하지 않습니다.

본 논문에서는 위상 검출 오버헤드를 0.09%로 대폭 감소시키면서 기존의 최첨단 위상 검출 성능과 경쟁적으로 수행하도록 하는 POP 검출기를 제안한다.구체적으로, 우리는 다음과 같이 우리의 기여를 개략적으로 설명한다.:

* 본 논문에서는 위상 간격 길이, 위상 안정성 및 위상 수의 상충을 포착하는 새로운 위상 검출기 메트릭인 통계 시간 분석 기준선 (STAB)을 제안한다..
* 정밀 온라인 상(POP) 검출기를 소개한다.  
  기존 접근법보다 오버헤드가 현저히 낮은 정밀한 미세입상 검출을 위해 성능 카운터를 활용하는 실시간 위상 검출 알고리즘

.

* 우리는 POP 검출기와 ScarPhase 검출기 사이의 상세한 경쟁 분석을 수행하며, 기존의 메트릭과 제안된 메트릭을 사용한다..

우리는 논문을 다음과 같이 정리합니다. II장에서는 POP 검출기를 포함한 논문 전반에 걸쳐 논의된 서로 다른 위상 검출 알고리즘에 대해 논의한다. POP 검출기의 주요 식별 인자는 선택된 특정 성능 카운터이며, 섹션 III에서 자세히 설명합니다. IV절에서는 새로운 메트릭인 STAB를 포함한 위상 검출 알고리즘을 평가하기 위한 메트릭을 설명한다. 그리고 Section V에서 위상 간격 크기에 대한 사례 연구를 수행한 후 Section VI에서 ScarPhase와 POP 검출기 간의 광범위한 분석을 수행하였다.마지막으로, 제 7절에서 관련 작업에 대한 간략한 조사를 제공하고 제 8절에서 결론을 내린다..

1. PHASE DETECTION ALGORITHMS

본 논문에서 제안하는 POP 검출 알고리즘을 포함하여 논문 전체에 사용되는 위상 검출 알고리즘을 간략하게 설명한다..

1. *SimPoint: Offline Baseline*

심포인트는 프로그램인을 단계[7]로 분류하기 위해 코드 실행을 추적한다. 원래 작업은 기본 블록의 개념에서 비롯됩니다. 정확히 하나의 항목과 하나의 출구가 있는 코드 영역입니다. 기본 블록 벡터(BBV)는 일정 시간 동안 각 기본 블록 내에서 실행되는 명령의 수를 프로파일링함으로써 생성된다. 심포인트는 BBV에 대한 대략적인 K-Means 클러스터링을 수행하고, BIC(Bayesian Information Criterion)를 이용하여 프로그램에 대한 최적의 단계 수를 선택한다. 그런 다음 시뮬레이션 포인트 역할을 하는 다수의 BBV를 출력한다.우리는 각 시뮬레이션 포인트를 클러스터 센터로 취급합니다.클러스터 센터는 가장 가까운 시뮬레이션 포인트에 따라 각 실행 간격을 레이블링할 수 있게 해줍니다.

1. *ScarPhase: Online Baseline*

ScarPhase는 프로그램 단계를 탐지하기 위해 조건부 분기 명령 포인터 (IPs)를 샘플링합니다 [3]. 프레임워크는 Linux perf 서브시스템을 이용하여 구축되며 사용자-공간 프로그램으로 운영된다. 인텔의 정밀 이벤트 기반 샘플링(PEBS)은 25K 지침당 최대 1회까지 세분화하여 IP를 직접 샘플링할 수 있다. 분기 IP는 각 간격에 대한 서명을 구성하기 위해 벡터로 해시된다. 서명은 프로그램 단계를 식별하기 위해 온라인 리더-폴로워 클러스터링 알고리즘에 공급된다. 오버 헤드를 완화하기 위해 ScarPhase는 커널이 간격 입도에서만 처리를 허용하는 버퍼를 직접 작성하는 버퍼를 사용합니다. 또한 ScarPhase는 안정적인 실행 단계에서 샘플 속도를 줄이는 동적 샘플링 주파수 방법론을 사용합니다. 동적 샘플 속도 자체가 예측하기 때문에 낮은 오버 헤드를 위해 위상 검출 정확도를 약간 저하시킵니다.ScarPhase와 관련하여보고된 모든 데이터는 GitHub [8]에서 사용할 수 있는 저자의 원래 코드를 사용합니다.

1. *The POP Detector*

POP 검출기는 ScarPhase와 관련된 사소한 오버 헤드뿐만 아니라 100M 미만의 위상 길이를 사용할 때 시트 성능 저하에 의해 동기 부여되었습니다. 이를 해결하기 위해 POP 디텍터는 성능 카운터만 사용하여 프로그램 단계를 탐지합니다. 성능 카운터를 사용하는 근거는 두 가지입니다. 첫째, 인터럽트는 성능 카운트를 수집하기 위해 간격의 끝에서만 필요합니다. 이것은 우리가 보여줄 것처럼 오버헤드를 급격히 감소시킨다. 둘째로, 수는 표본이 아니라 실제 측정입니다.이는 점점 미세한 입자상 검출을 수행할 때 기본 데이터가 추가 손실을 발생시키지 않는다는 것을 의미한다..

성능 카운터를 이용한 위상 검출의 품질은 성능 관련 메트릭이 아닌 코드 실행 경로를 캡처하는 특정 카운터를 선택하는 데 의존합니다. 카운터가 성능과 관련되어 있으면 피드백 루프를 작성합니다.예약자 예제를 고려하십시오.이 예에서는 피쳐를 사용 가능으로 설정할지 사용하지 않을지 선택하는 것이 목표입니다. 서명을 생성하는 데 사용되는 카운터 중 하나가 캐시 미스 수인 경우 프리 페처를 비활성화하면 캐시 미스 카운트에 영향을 줄 수 있습니다. 결과적으로 서명이 변경되고 잠재적으로 잘못된 위상 변경을 트리거할 수 있다. 따라서 성능 카운터는 시스템 변경에 불가지론적이기 위해 코드 실행 경로와 구체적으로 관련되어야 한다.우리는 기계 학습을 통해 최적의 카운터 선택을 목표로 하는 새로운 접근법에 III절을 바친다..

이 키 세트의 성능 카운터를 사용하여 POP 검출기는 카운터 값을 주기적으로 측정합니다. ScarPhase와 유사하게, 이는 사용자 공간 프로그램에서 Linux perf 서브 시스템을 활용함으로써 수행됩니다. 카운터 값은 간격의 서명으로 처리되고 온라인 리더 - 추종자 클러스터링 알고리즘에 입력으로 제공됩니다. 이 경량 온라인 알고리즘은 새로운 지점을 클러스터링하기 위해 10,000 사이클 또는 약 5μsat 2GHz만 소요된다. 리더-폴로워 알고리즘은 최신 데이터 점과 기존 클러스터 센터 간의 유사성이 너무 큰 경우 자동으로 새 클러스터를 생성합니다. 평균 백분율 차이를 이용하여 거리를 측정하고, 유사도 임계값을 20%(ScarPhase와 동일)로 설정한다. 우리는 POP 검출기가 32개의 독특한 위상을 동시에 추적할 수 있도록 한다.32개의 고유한 위상이 검출된 후 새로운 클러스터가 검출되면 LRU 대체 정책을 사용하여 클러스터를 제거합니다.

1. PERFORMANCE COUNTER SELECTION

이 절에서는 반복적인 행동보다는 반복적인 코드 세그먼트와 관련된 프로그램 단계를 캡처할 수 있는 핵심 성능 카운터를 선택하는 프로세스를 설명합니다.우리의 2단계 접근법은 200개 이상의 후보 카운터를 검토하고, 궁극적으로 기계 학습 기법과 심포인트 단계의 기본 진리를 사용하여 8개의 하위 집합을 제공한다..

1. *Performance Counter Collection*

코어별 이벤트로 검색을 제한하는 것 외에 성능 카운터 세트를 줄이기 위한 도메인 전문 지식을 제공하지 않는다.

|  |  |
| --- | --- |
| **Counter** | **Normalized Score** |
| UOPS RETIRED.RETIRE SLOTS | 100 |
| UOPS RETIRED.ALL | 93.28 |
| BR INST RETIRED.ALL BRANCHES | 79.29 |
| BR INST RETIRED.NOT TAKEN | 77.16 |
| MEM UOPS RETIRED.ALL STORES | 74.58 |
| BR INST RETIRED.NEAR TAKEN | 69.17 |
| MEM UOPS RETIRED.ALL LOADS | 64.75 |
| UOPS EXECUTED.CYCLES GE 4 UOPS EXEC | 64.60 |

TABLE I

THE HIGHEST RANKED COUNTERS USING THE GINI IMPURITY METRIC.

이것은 우리의 Haswell 기반 Intel 시스템의 프로파일에 대한 200개 이상의 성능 카운터 목록을 만들어낸다.하드웨어가 8개의 이벤트만 동시에 수집할 수 있도록 하기 때문에, 가장 일반적인 접근방식은 그룹을 만들고 대략적인 카운트에 시간 다중화를 사용하는 것이다.그러나, 우리의 연구의 목표는 샘플링 오류를 피하는 것이기 때문에, 우리는 표본 오류의 가능성을 없애기 위해 각각 실제 카운터 값을 가진 50개 이상의 개별적인 흔적을 수집한다..

1. *Ensuring Counter Invariance*

첫 번째 패스는 프로그램 실행보다는 시스템 동작과 관련된 성능 카운터를 제거하는 것이다. 언뜻 보기에 이것은 직관적으로 보이지 않을 수 있다. 유사한 시스템 동작은 유사한 코드 실행을 암시하기 때문이다. 하지만 이 말은 사실이지만, 프로그램의 지시를 입력으로, 시스템의 행동을 결과적으로 생각하는 것이 가장 좋다. 좋은 위상 검출기는 유사한 입력을 가진 간격을 상관시킨다. 이를 통해 성능 등 출력을 서로 다른 시스템 구성으로 분석할 수 있다.즉, 성능과 관련된 성능 카운터를 제거하려고 합니다.

시스템 동작보다는 코드 실행과 관련된 일련의 성능 카운터를 찾기 위해 시스템 재구성의 결과로 변화를 측정하기 위한 테스트를 수행합니다. 본 시스템은 모델 특정 레지스터(MSR)[9]를 통해 사용 가능하거나 비활성화할 수 있는 4개의 하드웨어 프리페처를 가지고 있다. 디폴트 구성, 프리페치 MSR = 0은 모든 프리페처를 활성화하는 반면, 비활성화된 구성인 프리페치 MSR = 15는 모든 프리페처를 비활성화한다. 각 성능 카운터의 평균 백분율 차이를 구간별로 측정합니다.N개의 구간을 실행하기 위해, 우리는 프리페처 구성 0(ci,0)과 15(ci,15)에서 간격 i의 카운트를 비교함으로써 차이를 측정한다.:

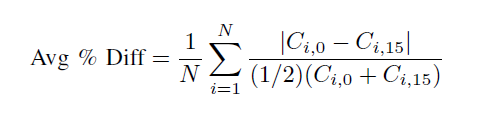


그림 3은 프리페처 구성의 변화에 가장 적게 반응하는 상위 50개 성능 카운터의 순위를 보여 잠재적인 후보자가 됩니다. 많은 저분산 카운터는 예상대로 명령 혼합과 관련이 있다.프로그램 카운터가 다음 단계로 진행되기 위해서는 프리페처 재구성으로 인해 평균 1% 미만의 차이가 필요합니다.

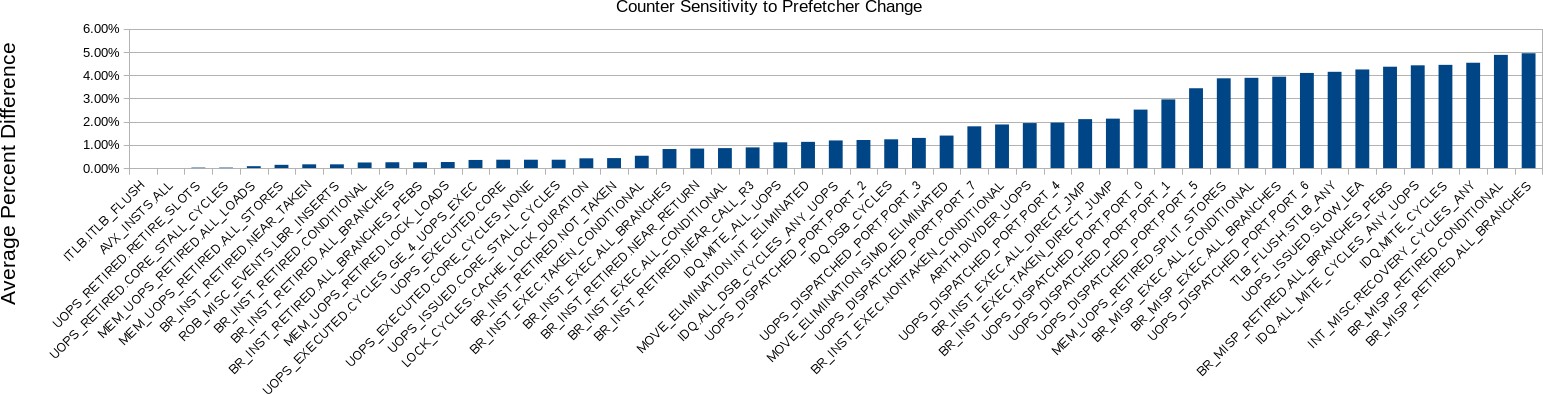


Fig. 3. Interval-to-Interval average percentage difference from varying prefetchers. Data collected using Spec2006 benchmarks.

1. *Counter Pruning*

우리의 마지막 단계는 코드 실행에 기초하여 단계를 정확하게 예측하는 능력에 의해 프로그램 카운터를 순위를 매긴다. 기본 블록 벡터(BBV)를 수집하고 SimPoint를 사용하여 지상 진리 프로그램 단계를 결정한다. 순진한 접근법은 8개의 카운터의 가능한 모든 조합을 사용하여 일련의 예측 변수를 훈련시키고 최상의 예측 정확도로 카운터를 선택하는 것입니다. 불행하게도, 이것은 단지 20명의 최고의 후보들 중에서 8개의 카운터를 선택하는 것이 125,000개 이상의 조합을 테스트하는 것을 포함하기 때문에 계산적인 관점에서는 실현할 수 없다. 또한, 이러한 방법은 카운터의 순위에 대한 통찰력을 제공하지 않는다.

따라서 우리는 심포인트 단계를 예측하기 위해 의사결정 숲 분류기를 사용한다. 결정 숲을 훈련한 후, 지니 불순물 메트릭을 사용하여 측정된 각 변수와 관련된 중요성을 볼 수 있습니다 [10].Spec2006 벤치마크[11]에 대한 이 접근법을 사용하여, 우리는 표 I에 표시된 변수를 순위화한다..

마지막으로 거리 시각화 실험을 수행하여 카운터가 실제 기본 블록 벡터와 유사한 데이터를 캡처하는 것을 검증한다. 셔우드 외의 원작에 의해 동기 부여되었다.[12] 우리는 반복 패턴을 찾기 위해 서로 다른 간격으로 서명 간의 유사성을 측정합니다.

1. PHASE DETECTION EVALUATION

이 절에서는 위상 검출 알고리즘을 평가하는 데 사용되는 여러 메트릭을 제시합니다. 우리는 모든 메트릭을 계산하는 기초로 CPI를 사용한다. 예를 들어, μi는 i로 표시된 모든 구간 중 CPI의 표준 편차를 나타냅니다. 제시된 모든 계산은 프로그램 위상 검출기를 비교하기 위한 수단으로 사후 실행된다는 점에 유의하십시오.이러한 지표들 중 많은 것들이 온라인에서 계산되도록 적응될 수 있지만, 선행 기술과 일관성을 유지하고 공정한 경쟁 분석을 위해, 우리는 이러한 지표들을 후기적으로 활용한다.

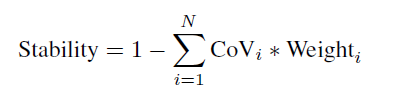
1. *Phase Stability*

위상 검출 알고리즘의 품질을 측정하는 방법은 안정성( 백분율)을 살펴보는 것이다. 안정성은 각 단계 내에서 변동 계수(CoV)를 측정하는 통계적으로 근거가 있는 메트릭이다.CoV는 단순히 분산을 계산하는 정규화된 방법일 뿐이며, 보다 공식적으로,

CoV = *σ*

*µ*

안정성 측정은 또한 각 단계에서 사용되는 프로그램의 지속 기간 또는 단계의 가중치를 고려합니다.N상의 경우 안정성은 다음과 같이 계산할 수 있습니다. :



어두운 색은 더 큰 유사성을 나타내며, 검은 색은 정확한 일치다. 마찬가지로 흰색은 완전히 일치하지 않는다. mcf 작업량에 대한 결과는 그림 4에 나와 있다.이 그림은 프로그램 실행 과정에서 서명 유사성만 캡처하고 프로그램 단계에 대한 개념을 포함하지 않습니다

.

mcf 시각화를 조사함으로써 성능 카운터가 BBV와 동일한 패턴을 캡처하는 것이 분명합니다. 그러나 BBV는 완전한 불일치를 나타낼 수 있지만, 성능 카운터는 항상 유사성의 일부 요소를 유지한다(백색 대신 회색으로 표시). 결과적으로, 성능 카운터 서명에 의해 주어진 단계는 BBV 단계보다 더 일반화된다.그 차이가 유의미해 보일 수도 있지만, 우리는 나중에 결과 섹션에서 그 차이가 IP 기반 접근법에 비해 약간 더 나쁜 안정성을 초래한다는 것을 보여줄 것이다.

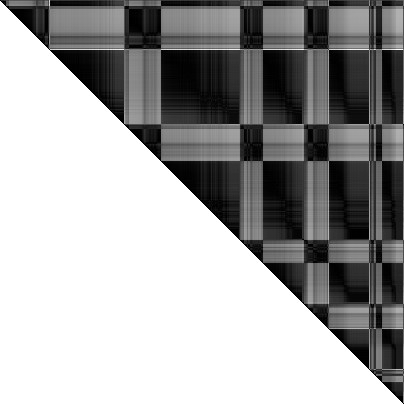
그러나 순진한 위상 검출 알고리즘은 각 간격을 고유한 위상으로 분류하여 안정성을 100%로 밀어낼 수 있다. 이를 바로잡기 위해 Sembrant et al. [3]는 변이 계수 (CCoV)를 제안했습니다. CCoV는 기본 CoV 계산을 조정하여보다 독특한 위상을 검출하기 위한 알고리즘을 벌칙화합니다. CCoV는 가상 단계의 일부로 이웃과 다른 모든 단계를 배치합니다. 가상 단계의 모든 샘플에는 프로그램 전체의 CoV가 주어집니다.이것은 사소한 경우, 단일 단계 또는 모든 고유한 단계에서 CCoV가 동일한 값으로 수렴되는 바람직한 특성을 제공합니다.

Counter Similarity Matrix *mcf*

BBV Similarity Matrix *mcf*

따라서 충분한 신뢰를 가지고 충분한 오차범위를 설정하기 위해서는 구성 A와 B를 여러 번 샘플링해야 할 수도 있다..

명확성을 위해, 우리는 특정한 예를 설명한다. 프로그램이 100M 명령어 간격 대 10M 명령어 간격에서 3 배 더 안정성을 가지고 있다고 가정합니다.주어진 신뢰 구간과 특정 오차 허용치를 설정하는 데 필요한 샘플 수와 안정성 사이에는 2차 관계가 있습니다 [13]. 따라서 100M 명령 구간에서 *x*2 샘플이 필요한 반면 10M 명령 구간에서는 (3*x*)2 샘플이 필요합니다. 이것은 100M 명령 간격에 대해 기본선을 설정하는 100 *x*2 명령어를 사용하지만 10M 명령 간격에 대해서는 90 *x*2 명령어만 사용하게 됩니다.결과 섹션에서 보여 주듯이, 이것은 짧은 위상을 가진 비선형 트레이드오프로 이어진다..

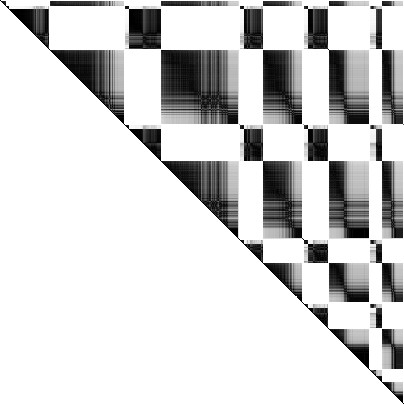


Most Similar

Least Similar

×

우리는 주어진 신뢰 구간과 오차 허용에 필요한 샘플의 수를 공식으로 계산하여 STAB의 공식화를 시작합니다:



Most Similar

Least Similar

Num Samples =

*Z* ∗ *CoV* 2

*Error*

. Σ

여기서 Z는 신뢰 구간과 관련된 요인이다(예:). 95% 신뢰 구간 Z-값이 1.96)[13]이다. STAB는 주어진 신뢰 요구 사항 및 위상 당 오차 허용 오차에 필요한 샘플 수를 계산합니다. 그런 다음 이 샘플은 성능 기준선을 설정하는 데 필요한 총 샘플 수를 제공하기 위해 보이는 각 단계에 대해 요약됩니다.따라서 N단계를 갖는 프로그램에 대한 것이다.:

Baseline Samples =

*N*

Σ*i*=1

*Z* ∗ *CoVi* 2

*Error*

. Σ

Fig. 4. Similarity matrices show the similarity between interval signatures, where the diagonal of the matrix represents the program’s execution. A darker color indicates greater similarity, as measured by percentage distance. We show the similarity matrix for counters (top) and BBVs (bottom) for the Spec2006 *mcf* benchmark.

다음으로, 우리는 이 기준선을 수립하는 데 필요한 총 명령 수를 결정하기 위해 위상 간격 크기를 설명한다. 이것은 단순히 기준선 샘플에 위상 간격 크기를 곱한 것이다.마지막으로, 기준선 분석에 소요되는 통계 시간을 비율로 계산할 수 있다.:

1. *Phase Interval Length*

이전 메트릭은 위상 간격 길이에 영향을 받을 수 있지만, 더 작거나 더 큰 간격을 선호하기 위해 가중 메트릭으로 사용하지 마십시오. 따라서, 우리는 새로운 메트릭인 통계 시간 분석 기준선 (STAB)을 제안합니다.이 메트릭은 위상 수, 안정성 및 위상 간격 길이 사이의 균형을 조사합니다.STAB는 실시간 최적화를 위한 위상 검출 알고리즘을 염두에 둔 메트릭이다..

시스템을 동적으로 조정하려면 주어진 구성에서 성능을 평가해야 합니다. 기본 구성 A와 새 구성 B 중 하나를 선택해야하는 알고리즘을 고려하십시오.알고리즘이 구성 B로 이동하려면 성능 차이가 측정 오차를 초과해야 변경이 합리적입니다.

STAB = Baseline Samples ∗ Interval Size

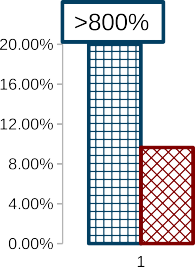
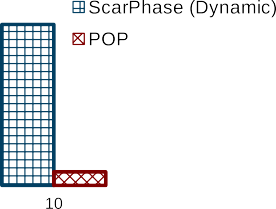
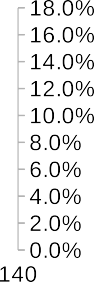
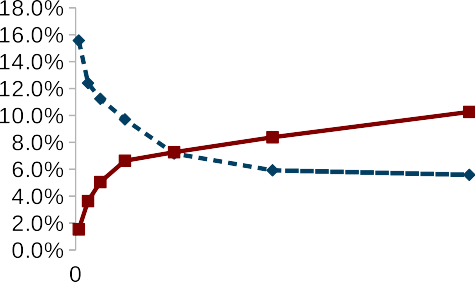
Total Instructions

이 계산의 결과는 직관적인 비율입니다. 주어진 오차 마진으로 신뢰 구간을 설정하는 데 얼마나 많은 명령어 ( 백분율로)를 사용할 것인지입니다. 그러나 이 비율은 바람직하지 않은 인공물인 프로그램의 총 지속 기간에 영향을 받는다. 예를 들어 Spec2017 omnetpp은 단 하나의 위상을 가지며 50개의 샘플이 훈련 또는 참조 데이터 세트를 사용하여 기본 성능을 설정해야 합니다.그러나 훈련 세트가 작고 참조 세트(1)보다 지시 사항이 적기 때문에 STAB의 정규화되지 않은 버전은 훈련 데이터 세트의 경우 7.3%이지만 참조 데이터 세트의 경우 0.91%에 불과합니다.

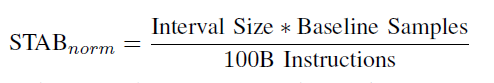
1This measurement was performed using ScarPhase with 100M instruction intervals and dynamic sampling enabled.

Fig. 5. Trade off between CoV and STAB using various interval sizes. While CoV suffers at smaller intervals, many more intervals are available for analysis. As a result, the STAB overhead reduces at smaller intervals.



따라서, 우리는 프로그램의 총 실행 길이가 STAB에 영향을 미치지 않도록 고정된 수의 명령으로 정규화를 선택한다.

Fig. 6. Wall-clock Overhead comparison. Results averaged across Spec2017 benchmarks using the training dataset.

COV는 4.5%에서 15% 이상으로 증가하지만 사용 가능한 간격 수는 128개 증가합니다.그 결과, STAB95,5 메트릭은 128M 명령어 구간에서 10%에서 1M 명령어 구간에서 1.9%로 감소합니다.

STAB 메트릭은 더 작은 입상 단계를 사용하는 이점을 전달하지만 오버 헤드를 고려하는 것은 소홀합니다. 우리는 오버헤드를 모니터링 프레임워크가 없는 위상 검출 프레임워크 대 위상 검출 프레임워크로 실행하는데 필요한 추가 시간으로 정의한다. Wall-clock 시간을 이용하여 1M, 10M, 100M 명령어 구간 크기에서 ScarPhase와 POP 위상 검출기의 오버헤드를 분석하고 그 결과를 그림 6에 표시한다.

 이 공식을 사용하여 omnetpp은 훈련 또는 참조 데이터 세트와 함께 5%의 STAB 규범을 가집니다. 본 논문의 나머지 부분에서는 정규화 인자로 100B 명령어를 사용하여 정규화된 STAB 버전을 활용한다. 이 선택은 임의적이며 값을 선형으로 축척하기 때문에 최종 결과와 관련이 없습니다.우리는 주어진 신뢰 구간 c에 대해 정규화된 STAB를 보고하고, STABc,e로 오차 허용 오차를 보고한다.예를 들어 신뢰 구간 95%와 오차 허용 오차 5%는 STAB95,5일 것이다..

1. CASE STUDY: INTERVAL SIZE

STAB 메트릭을 이용하여 다양한 간격 크기에서 위상 검출의 트레이드오프를 이해하기 위한 분석을 수행한다. 정보 손실이나 오버헤드 발생이 없는 공정한 기준선을 만들기 위해 II-A 절에서 설명한 대로 SimPoint를 사용하여 추적 기반 분석을 수행한다. 우리는 풀 시스템 에뮬레이터인 QEMU [14]를 사용하여 정확하고 완전한 장기 프로그램 추적을 수집합니다.명령 추적이 수집되면 원하는 간격 크기에 대해 BBV 및 해당 위상 레이블을 생성할 수 있습니다.

CoV와 STAB를 이용하여 분석을 수행하려면 성능 데이터, 특히 각 구간에 대한 IPC가 필요하다. 워크로드당 50개의 성능 트레이스를 수집하고 구간당 평균 데이터를 수집하여 런 투 런 오류를 완화합니다. 훈련 데이터 세트를 사용하여 19개의 Spec2006 워크로드를 사용하여 1M에서 128M까지의 명령 구간에서 SimPoint 분석을 수행합니다.그 결과는 그림 5에 나와 있다.

첫째, CoV와 STAB는 모두 더 낮은 메트릭이라는 점에 유의하십시오. 예상대로 간격이 작을수록 CoV가 점점 더 높다.그러나 분석에 사용할 수 있는 간격 수에 대해서는 비선형이다.

100M 명령 구간에서, 우리는 Spec2017 속도 벤치마크에서 평균 3.19%의 오버헤드를 발생시키기 위해 ScarPhase를 측정한 반면, POP 검출기는 0.09%의 오버헤드를 발생시킨다.10M 명령 구간에서 이 오버헤드는 ScarPhase의 경우 바람직하지 않은 16.12%로 증가하지만 POP 검출기는 1.35%에 불과하다. STAB 메트릭이 더 작은 간격으로 계속 개선되었지만,보다 효율적인 POP 검출기는 1M 명령어 간격에서 실행 시간이 9.65% 증가합니다.POP 검출기의 목표는 오버헤드가 낮기 때문에 1M 명령 간격을 제공하지 않으므로 다음 섹션에서 결과를 얻을 수 있습니다.

1. RESULTS
2. *Methodology*

To assess the proposed POP detector, we compare it to the prior art, ScarPhase, utilizing the Spec2017 benchmark suite on an Intel® system2. Both phase detectors are configured with a similarity threshold of 20% for their clustering algorithm, the default specification for ScarPhase. To obtain consistent, repeatable results we evaluate single-threaded applications via *speed* benchmarks. It is also known that single threaded applications typically have more phase variability, making them more applicable to our analysis [4]. Furthermore, we apply a few command line boot parameters to ensure that results are not skewed. Table II details each of our Intel® Haswell evaluation node’s specifications.

Note that both the POP detector and ScarPhase are im- plemented as user-space programs which tap into the Linux

2Performance results are based on testing as of February 13, 2019 and may not reflect all publicly available security updates.

|  |  |
| --- | --- |
| Base Board | Intel® Server Board S2600WT2 |
| CPU | 2x Intel® Xeon® CPU E5-2699 v3 @ 2.30GHz |
| BIOS | SE5C610.86B.01.01.0027.071020182329 |
| DRAM | 24x 16G DDR4 ECC DIMM @ 1600 MHz |
| OS | CentOS 7.5 w/ Linux 4.14.58 |
| Kernel Cmdline | nmi watchdog=0 transparent hugepages=never  cpuidle.off=1 intel idle.max state=0 cpufreq.off=1 intel pstate=disable procesor.max state=0 isolcpus=0,1 rcu nocbs=0,1 rcu nocb poll  nohz full=0,1 skew tick=1 |
| Hyperthreading | Off |

TABLE II

EVALUATION SYSTEM

*perf* subsystem, allowing them to collect counter and branch IP information. This means that no super-user privileges are required to run either detector. While kernel-module imple- mentations can offer superior performance, the user space implementation allows us to perform a fair comparison to the prior art. As Sembrant et al. [3] mention, context switches to this user space implementation can incur significant overhead at smaller time scales but is often more desirable for system administrators.

1. *Phase Detector Performance*

We perform an in-depth comparison of ScarPhase and the POP detector. As discussed in Section II-B, ScarPhase has an optional dynamic sampling mode which reduces overhead while slightly degrading phase detection performance. To remain succinct and capture ScarPhase’s optimal phase detec- tion ability, we focus our discussion of results on ScarPhase without dynamic sampling. We report results for both 100M and 10M instruction intervals, the smallest granularity possible without significant overhead. We include all *speed* benchmarks in the Spec2017 benchmark suite, which includes some out- liers. As such, we utilize the geometric mean to draw final conclusions for various metrics. Note that when we use the term “average”, we are referring to the geometric mean unless otherwise stated.

* 1. *Stability:* We begin by examining phase stability in Figure 7. The POP detector finds 0.51% more stable phases at 100M instruction intervals. In particular, POP performs significantly better on lbm and fotonik3d, where its stability is more than 10% higher than ScarPhase at 100M instruction intervals. These programs are among the most variable with respect to IPC changes in the Spec2017 benchmark suite, but ScarPhase inadequately separates the phases. In the case of fotonik3d at 100M instruction intervals, ScarPhase generates 112 unique phases, but more than 85% of intervals reside in just 3 phases. As a result, the phases are poorly separated and phase stability is just 80.5%.

At 10M instruction intervals, ScarPhase is able to identify phases which are 0.52% more stable compared to the POP detector. Nevertheless, for half of the benchmarks tested the POP detector is within a 2% margin compared to ScarPhase, and never performs more than 7% worse than ScarPhase. Also worth emphasizing is that this result is with ScarPhase’s dynamic sampling mode disabled. Enabling dynamic sampling

mode nets a 2% decrease in average stability for ScarPhase, placing it behind the POP detector in terms of phase stability.

* 1. *Number of Phases:* While the two detectors perform similarly from a phase stability standpoint, they begin to diverge when analyzing the number of phases detected, as shown in Figure 8. At 100M instruction intervals, ScarPhase detects 13.7 phases compared to 18.5 with the POP detector. Yet the reverse is true at 10M instruction intervals, where ScarPhase detects 214 unique phases compared to POP’s 86, a reduction of nearly 60%. Specifically, for 12 of the 20 benchmarks tested, ScarPhase finds more than 100 unique phases. While more unique phases at finer granularity is intu- itive, the divergence of ScarPhase and the POP detector can be traced back to the fundamental difference of sampling versus measurement. ScarPhase’s signature vector is more susceptible to noise in the case of finer granularity due to its lack of sufficient samples. This result is particularly prominent in x264 and wrf, where the number of phases detected increases by more than an order of magnitude.
  2. *CCoV:* Sembrant et al. proposed the “Corrected” Coefficient of Variation as a means to account for the intrinsic trade-off between number of phases and stability. We compute the CCoV and report the results in Figure 9. The CCoV metric most emphasizes the effectiveness of the POP detector at smaller phase intervals. ScarPhase’s CCoV degrades by 83% at finer granularity due to the large number of phases detected. The lower-is-better metric increases from 8.12% to 14.93% when moving from 100M to 10M instruction intervals using ScarPhase. By comparison, the POP detector experiences negligible deviation in CCoV, improving from 8.65% to 8.49% when moving from 100M to 10M instruction intervals.
  3. *STAB:* While CCoV balances number of phases and stability, the STAB metric applies statistics to also balance phase interval size. We report the normalized STAB metric using a 95% confidence interval and 5% error tolerance in Fig- ure 10. The overall winner here is the POP phase detector with 10M instruction intervals. ScarPhase’s optimal configuration using 100M instruction windows boasts a STAB95*,*5 of 26.5%,

2.1 worse than the POP detector’s 12.5% STAB95*,*5 with

×

10M instruction intervals. Based on our statistically grounded metric, this means use of the POP detection framework would net significantly quicker analysis to enable better online opti- mization.

1. *Summary of Results*

We provide a summary of results in Table III to highlight the key takeaways of the POP detector compared to the existing state-of-the-art, ScarPhase. All values reflect the geomean across all benchmarks tested; the same data is presented in Figures 6, 7, 8, 9, and 10. While the two phase detection frameworks are closely matched in the traditional phase detec- tion metrics, the two parameters that apply most directly to the success of deploying an online phase detection algorithm are execution overhead and STAB. For any potential performance gain, the optimization will have to overcome the overhead of the phase detection framework. Our experiments concluded

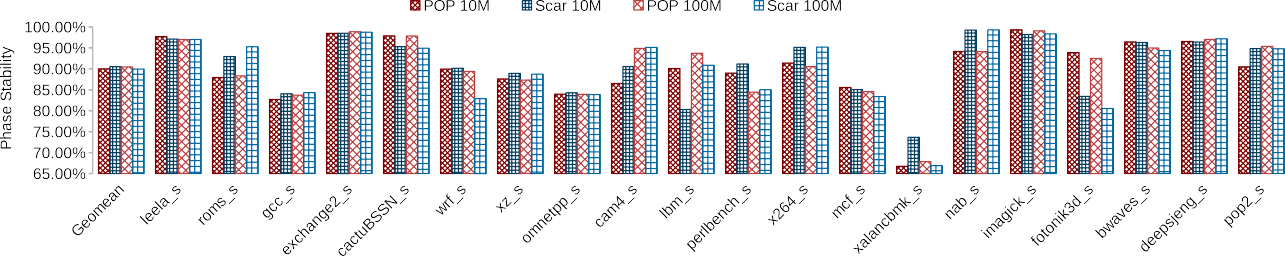


Fig. 7. Phase Stability metric for Spec2017 Speed benchmarks, Reference Dataset.

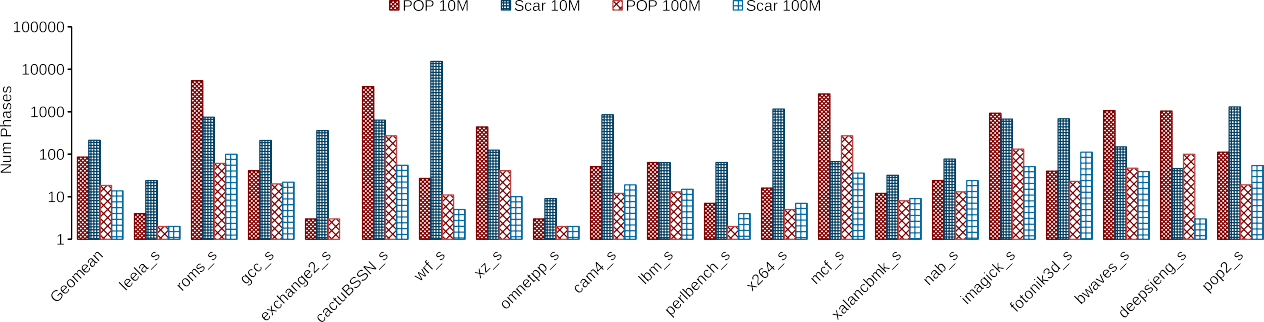


Fig. 8. Number of Phases detected for Spec2017 Speed benchmarks, Reference Dataset.

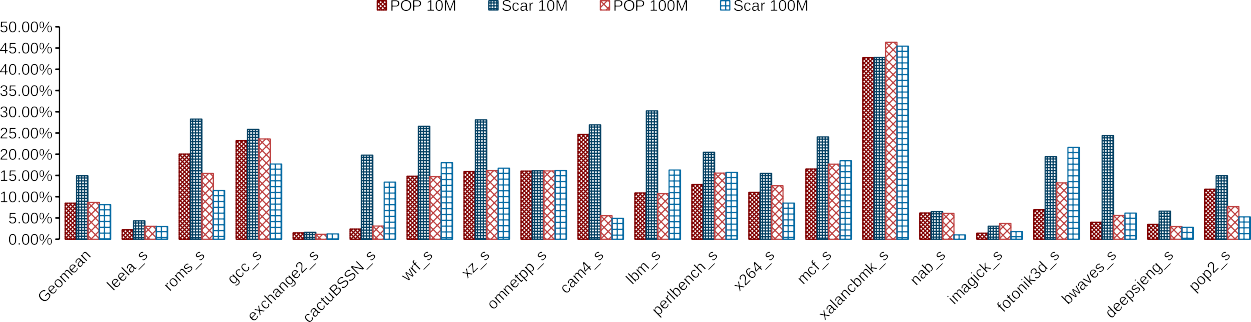


Fig. 9. Corrected Coefficient of Variation (CCoV) metric for Spec2017 Speed benchmarks, Reference Dataset.

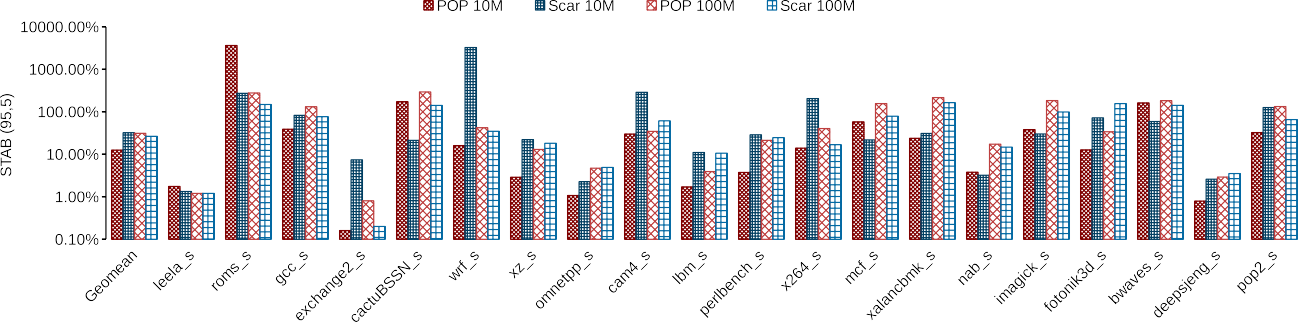


Fig. 10. Normalized STAB95*,*5 metric for Spec2017 Speed benchmarks, Reference Dataset.

that even in the most optimistic scenario with dynamic sam- pling enabled and larger 100M instruction intervals, ScarPhase still incurs a 3.19% overhead. POP detection incurs just 1.35% execution overhead at shorter 10M instruction intervals, and just 0.09% with 100M instruction intervals.

In addition to overhead, an adaptive system must adequately analyze performance before applying a final configuration. Motivated by this, we propose the STAB metric to statically quantify the amount of time an adaptive system must spend analyzing behavior to gain a particular level of confidence with a given error tolerance. This metric captures trade-offs of phase stability, number of phases, *and* phase interval size. STAB quantifies that POP detection enables 2.1 faster performance analysis compared to ScarPhase.

×

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Geomean** | **Overhead** | **Stability** | **Phases** | **CCoV** | **STAB** |
| **POP 10M** | 1.35% | 89.99% | 85.8 | 8.49% | 12.53% |
| **Scar-D 10M** | 16.12% | 88.39% | 195.2 | 16.72% | 53.30% |
| **Scar 10M** | 17.85% | 90.51% | 213.9 | 14.93% | 31.98% |
| **POP 100M** | 0.09% | 90.45% | 18.5 | 8.49% | 31.18% |
| **Scar-D 100M** | 3.19% | 88.62% | 12.7 | 9.57% | 35.04% |
| **Scar 100M** | 9.01% | 89.94% | 13.7 | 8.12% | 26.53% |

TABLE III

SUMMARY OF RESULTS FOR PHASE DETECTORS AT BOTH 10M AND 100M INSTRUCTION WINDOW SIZES. SCARPHASE WITH DYNAMIC SAMPLING IS LISTED AS *Scar-D*, WHILE WITH FIXED SAMPLING AS JUST *Scar*.

OVERHEAD IS REPORTED IN WALL-CLOCK EXECUTION TIME, AND THE *STAB* METRIC REFERS TO 100B INSTRUCTION NORMALIZED *STAB* WITH A CONFIDENCE INTERVAL OF 95% AND ERROR TOLERANCE OF 5%, AS

DESCRIBED IN IV-B.

1. RELATED WORK

Sherwood et al. began to define the standards for program phase detection in their early work which utilized basic blocks. Their initial work did an in-depth dive to understanding the cyclical behavior of a program via frequency analysis [12]. They later expanded this work to SimPoint, which attempts to cluster basic vectors to define phases [15]. SimPoint’s main objective is to speed up simulation work by drastically reducing the amount of a program that must be simulated to gain an accurate representation. SimPoint identifies phases and their respective weights such that a sample from each phase can can be used for simulation. More than a decade later, this technique is still proven to be effective as shown in [16].

While random projections and approximate K-means clus- tering help to speed up SimPoint, the fundamental idea still hinged on collecting basic blocks which are expensive to pro- file. Dhodapkar and Smith attempted to mitigate this overhead by proposing a hardware mechanism which constructs signa- tures hashing branches into an *n*-bit vector [17]. Sherwood et al. later refine this proposal by adding in the amount of time spent in each branch as well as a phase predictor [2]. Dhodapkar and Smith then provided an evaluation framework utilizing sensitivity, false-positives, and stability which lay the foundation and motivation for CCoV and STAB [18]. In their comparison work, they find that while BBVs provide the best phase detection, utilizing branch counts nets a phase detection with 80% of the performance. In many ways, this is the motivation for our work.

Shen et al. explore the notion of phases as repeating, but non-uniform behavior [19]. They utilize wavelet filtering and allow for variable phase sizes. While their technique performs comparably to manual-code injection, it requires a separate training step to determine program phases. Nagpurkar et al. also approach the problem utilizing dynamic phase sizes by suggesting the notion of stable and transition periods [20]. They use an adaptive trailing window policy to detect phases and instrument a new binary. However, their technique is only applicable to Java applications. Nevertheless, both of these approaches are particularly applicable when the goal is optimizing software as the program can be run and profiled many times. Since these publications, Linux *perf* has improved dramatically, and supports such analysis. Gregg has an ex- cellent blog which includes many tutorials for such software optimization using *perf* and other tools [21].

However, another application of phase detection is on- line hardware optimization, which requires lightweight online phase detection. In this scope, ScarPhase [3] represents the state-of-the-art. ScarPhase bridges the gap of previous phase detection techniques by designing a framework which works on present hardware, operates completely online without prior training, and has significantly lower overhead than previous approaches [3]. However, ScarPhase chooses 100M instruction window phases as a baseline by citing prior work. We differ from ScarPhase in that we explore trade-offs associated with different phase sizes. We show that while variance increases with finer resolution, the trade-off is nonlinear and has auxil- iary benefits.

1. CONCLUSION

본 논문에서는 온라인 성능 최적화의 사용 사례를 위한 프로그램 위상 검출을 연구하였다. 실시간 최적화를 수행하려면 알고리즘이 특정 구성에서 위상별 성능을 평가할 수 있어야 합니다. 위상 검출 알고리즘을 이용하여 위상별 성능이 더 빨라질 수 있으며, 빠른 시대 강화 학습 에이전트는 시스템을 적응시킬 수 있다. 이를 염두에 두고 위상 안정성, 위상 수 및 간격 크기를 가중시키는 새로운 메트릭인 STAB를 만듭니다.주어진 알고리즘이 위상별 성능을 확립하는 데 필요한 시간을 통계적으로 구한다..

STAB를 이용하여 SimPoint를 이용한 신탁 연구를 수행하고, 학습 오버헤드를 최소화하기 위해 더 작은 단계가 바람직하다는 것을 보여준다. 그러나, 우리는 기존의 최첨단 위상 검출기인 ScarPhase가 미세 결정립 위상 분류를 시도할 때 과도한 런타임 오버헤드를 겪고 있다는 것을 발견한다. 오버헤드 외에도 제한된 정보로 인해 ScarPhase의 샘플링 기법이 고장나 수백 개의 고유한 위상이 확인된다.이 공백을 메우기 위해 온라인 위상 탐지를 위한 성능 카운터의 사용을 탐구합니다.

우리는 성능 카운터의 핵심 부분 집합을 선택하기 위해 2 단계 프로세스에서 통계 및 기계 학습 기술을 사용합니다. 이러한 카운터를 사용하여 POP 위상 검출기를 구축하여 1.35%의 오버헤드를 발생시키면서 미세 입도에서 위상을 정확하게 검출할 수 있다. 온라인 최적화의 맥락에서 POP 검출기는 기준 성능 설정을 위해 2.1개의 지침을 덜 필요로 합니다. 더 낮은 오버헤드가 필요한 경우, POP는

100M 명령 간격 단계를 사용할 때 0.09% 오버헤드만 부과한다.향후 우리는 POP 검출기의 효능을 더욱 검증하기 위해 더 많은 벤치마크와 시스템 구성으로 테스트하고 싶다..

×

ACKNOWLEDGEMENTS

We thank the anonymous reviewers for their helpful sug- gestions. This work was funded in part by Intel.

REFERENCES

1. Standard Performance Evaluation Corporation, “SPEC CPU 2017,” [https://www](http://www.spec.org/cpu2017/).spec.or[g/cpu2017/.](http://www.spec.org/cpu2017/)
2. T. Sherwood, S. Sair, and B. Calder, “Phase tracking and prediction,” in *ACM SIGARCH Computer Architecture News*, vol. 31, no. 2. ACM, 2003, pp. 336–349.
3. A. Sembrant, D. Eklov, and E. Hagersten, “Efficient software-based online phase classification,” in *Workload Characterization (IISWC), 2011 IEEE International Symposium on*. IEEE, 2011, pp. 104–115.
4. A. Sembrant, D. Black-Schaffer, and E. Hagersten, “Phase behavior in serial and parallel applications,” in *2012 IEEE International Symposium on Workload Characterization (IISWC)*. IEEE, 2012, pp. 47–58.
5. R. P. Pothukuchi, A. Ansari, P. Voulgaris, and J. Torrellas, “Using multiple input, multiple output formal control to maximize resource efficiency in architectures,” in *Computer Architecture (ISCA), 2016 ACM/IEEE 43rd Annual International Symposium on*. IEEE, 2016, pp. 658–670.
6. R. P. Pothukuchi, S. Y. Pothukuchi, P. Voulgaris, and J. Torrellas, “Yukta: multilayer resource controllers to maximize efficiency,” in *Proceedings of the 45th Annual International Symposium on Computer Architecture*. IEEE Press, 2018, pp. 505–518.
7. E. Perelman, G. Hamerly, M. Van Biesbrouck, T. Sherwood, and

B. Calder, “Using simpoint for accurate and efficient simulation,” in *Proceedings of the 2003 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, ser. SIGMETRICS ’03. New York, NY, USA: ACM, 2003, pp. 318–319. [Online].

Available: <http://doi.acm.org/10.1145/781027.781076>

1. A. Sembrant, D. Eklov, and E. Hagersten, “Scarphase,” https://github.com/uart/scarphase, 2012.
2. “Disclosure of H/W prefetcher control on some Intel processors,” https://software.intel.com/en-us/articles/disclosure-of-hw-prefetcher- control-on-some-intel-processors, accessed: 2018-10-12.
3. P. Dolla´r, Piotr and L. C. Zitnick, “Structured forests for fast edge detection,” in *Proceedings of the IEEE International Conference on Computer Vision*, 2013, pp. 1841–1848.
4. Standard Performance Evaluation Corporation, “SPEC CPU 2006,” [https://www](http://www.spec.org/cpu2006/).spec.or[g/cpu2006/.](http://www.spec.org/cpu2006/)
5. T. Sherwood, E. Perelman, and B. Calder, “Basic block distribution anal- ysis to find periodic behavior and simulation points in applications,” in *Parallel Architectures and Compilation Techniques, 2001. Proceedings. 2001 International Conference on*. IEEE, 2001, pp. 3–14.
6. G. D. Israel, “Determining sample size,” 1992.
7. F. Bellard, “Qemu, a fast and portable dynamic translator,” in *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ser. ATEC ’05. Berkeley, CA, USA: USENIX Association, 2005, pp. 41–41. [Online]. Available: <http://dl.acm.org/citation.cfm?id=1247360.1247401>
8. T. Sherwood, E. Perelman, G. Hamerly, and B . Calder, “Automatically characterizing large scale program behavior,” *ACM SIGARCH Computer Architecture News*, vol. 30, no. 5, pp. 45–57, 2002.
9. Wu, Qinzhe and Flolid, Steven and Song, Shuang and Deng, Junyong and John, Lizy K, “Hot Regions in SPEC CPU2017,” in *Invited Paper, Special Session on Hot Workloads, IEEE International Conference on Workload Characterization (IISWC)*, 2018.
10. A. S. Dhodapkar and J. E. Smith, “Managing multi-configuration hardware via dynamic working set analysis,” in *Computer Architecture, 2002. Proceedings. 29th Annual International Symposium on*. IEEE, 2002, pp. 233–244.
11. A. S. Dhodapkar and J. E. Smith, “Comparing program phase detection techniques,” in *Proceedings of the 36th annual IEEE/ACM International Symposium on Microarchitecture*. IEEE Computer Society, 2003, p. 217.
12. X. Shen, Y. Zhong, and C. Ding, “Locality phase prediction,” *ACM SIGPLAN Notices*, vol. 39, no. 11, pp. 165–176, 2004.
13. P. Nagpurkar, P. Hind, C. Krintz, P. Sweeney, and V. Rajan, “Online phase detection algorithms,” in *Code Generation and Optimization, 2006. CGO 2006. International Symposium on*. IEEE, 2006, pp. 13–pp.
14. B. Gregg, *Systems performance: enterprise and the cloud*. Pearson Education, 2013.