"PERF"를 사용하여 CPU 통계 수집 및 그래프 NSO의 툴

목차

소개

이 문서에서는 NSO 호스트에서 성능 툴을 사용하여 성능 문제를 조사하는 방법에 대해 설명합니다 .

사전 요구 사항

요구 사항

다음 주제에 대한 지식을 보유하고 있으면 유용합니다.

- 기본 Linux/Unix 명령줄 사용법
- NSO(Network Services Orchestrator) 시스템 아키텍처 및 운영
- CPU 프로파일링 및 분석 개념
- 성능 문제 해결 워크플로에 익숙함

사용되는 구성 요소

이 문서의 정보는 다음 소프트웨어 및 하드웨어 버전을 기반으로 합니다.

- 지원되는 Unix/Linux 호스트에 NSO 시스템 또는 로컬 설치
- Ubuntu, Debian, Fedora 또는 RedHat 파생물과 같은 Linux 배포판
- perf 툴(Linux 성능 분석 툴)

이 문서의 정보는 특정 랩 환경의 디바이스를 토대로 작성되었습니다. 이 문서에 사용된 모든 디바이스는 초기화된(기본) 컨피그레이션으로 시작되었습니다. 현재 네트워크가 작동 중인 경우 모든

명령의 잠재적인 영향을 미리 숙지하시기 바랍니다.

배경 정보

Perf는 Linux에서 주로 CPU 프로파일링에 사용되는 강력한 성능 분석 툴입니다. 하위 수준 기능의 로드를 캡처하고 분석하여 CPU가 현재 작업 중인 작업에 대한 통찰력을 제공합니다. 따라서 어떤 기능이나 프로세스가 CPU를 점유하고 있는지 식별할 수 있으며 성능 병목 지점을 정확히 찾아내는 데 필수적입니다.

Perf는 또한 프로그램 중 CPU 시간을 가장 많이 사용하는 부분을 시각적으로 나타내는 특수 차트인 플레임 그래프를 생성할 수 있습니다. 플레임 그래프는 최적화가 필요한 코드의 영역을 더 쉽게 찾아낼 수 있도록 합니다.

중요한 것은 NSO BU(Business Unit)에서 권장하는 OOM(Out of Memory) 사례의 기본 데이터 수집 체크리스트에도 perf가 포함되어 있다는 점입니다. OOM 트러블슈팅에 대한 자세한 지침은 Cisco TAC에 문의하십시오.

NSO 성능 문제에 대한 성능 사용량 문제 해결

이 섹션에서는 성능 문제를 해결하기 위해 NSO 호스트에서 perf 툴의 데이터를 설치, 사용 및 분석하는 포괄적인 워크플로를 제공합니다.

성능 설치

1단계: Linux 배포판에 perf를 설치합니다. OS에 적절한 명령을 사용합니다.

우분투의 경우:

```
apt-get update && apt-get -y install linux-tools-generic
```

데비안의 경우:

```
apt-get update && apt-get -y install linux-perf
```

Fedora/RedHat 파생물의 경우:

```
dnf install -y perf
```

perf를 설치하는 동안 알려진 주의 사항에 대한 자세한 내용은 Cisco TAC 팀에 문의하십시오.

데이터 샘플링

1단계: 기본 NSO 프로세스를 식별합니다.

아래 명령을 사용하여 NSO 프로세스(ncs.smp)를 찾습니다.

```
ps -ef | grep ncs\.smp
```

출력 예:

```
root    120829       1  16 13:23 ? 00:11:08 /opt/ncs/current/lib/ncs/erts/bin/ncs.smp -K true -P 277140
root    121424   120604  0 14:30 pts/0 00:00:00 grep --color=auto ncs.smp
```

2단계: 또는, 특히 Java 작업에 초점을 맞춘 경우 NSO에 연결된 기본 Java 프로세스의 PID를 사용해야 합니다. 실행:

```
ps -ef | grep NcsJVMLauncher
```

출력 예:

```
root    120903   120833  6 13:32 ? 00:03:40 java -classpath :/opt/ncs/current/java/jar/* -Dhost=127.0.0
root    121435   120604  0 14:33 pts/0 00:00:00 grep --color=auto NcsJVMLauncher
```

3단계: 문제가 있는 테스트 사례 또는 활용 사례를 실행하여 성능 시나리오를 검증합니다.

4단계: 다른 터미널 창에서 관련 PID(프로세스 ID)에 대해 perf를 실행합니다. 아래 지정된 명령 형식을 사용하여 XX,YY,ZZ를 위에서 얻은 PID로 바꿉니다.

```
perf record -F 100 -g -p XX,YY,ZZ
```

예를 들어 시스템 전체에 대해 프로필을 생성하고 특정 PID에 대해 99Hz의 통화 그래프를 수집하려면

```
perf record -a -g -F 99 -p 120829,120903
```

출력 예:

```
Warning:
PID/TID switch overriding SYSTEM
```

옵션 설명:

- -a: 모든 CPU; 모든 CPU의 시스템 전체 수집(대상이 지정되지 않은 경우 기본값).
- -g: 호출 그래프 캡처(스택 추적) 함수가 호출되는 위치를 식별합니다.
- -F: 샘플링 빈도(Hz). 주파수가 높으면 정밀도는 높아지지만 오버헤드가 추가됩니다.
- -p: 프로세스 ID를 지정합니다.

5단계: 샘플 수집이 완료되면 Ctrl+C를 사용하여 성능 유지를 중지합니다.

```
^C
[ perf record: Woken up 1 times to write data ]
[ perf record: Captured and wrote 0.646 MB perf.data (4365 samples) ]
```

이제 현재 디렉터리에 perf.data 파일이 표시됩니다.

6단계: 다음 명령을 사용하여 요약 보고서를 생성합니다.

```
perf report -n --stdio > perf_report.txt
```

옵션 설명:

- -n: 그룹화 없이 기호 표시(평면 보기)
- —stdio: 출력을 표준 출력(터미널)으로 강제 출력합니다.

이때 추가 분석을 진행하기 전에 두 파일(perf.data 및 perf_report.txt)을 모두 저장하고 지원 담당자와 공유해야 합니다.

캡처가 성공하면 perf_report.txt는 계층 호출 그래프를 나타내는 트리형 구조를 표시합니다. 백분율을 사용하면 대부분의 CPU 시간이 소모되는 핫스팟을 식별할 수 있습니다.

발췌 예:

```
# Children      Self       Samples  Command          Shared Object         Symbol
# ........  ........  ............  ...............  ....................  ..............................
#    30.61%     0.00%             0  C2 CompilerThre  libc.so.6              [.] start_thread
```

```
#            ---start_thread
#               thread_native_entry(Thread*)
#               Thread::call_run()
#               JavaThread::thread_main_inner()
#               CompileBroker::compiler_thread_loop()
#                           --30.58%--CompileBroker::invoke_compiler_on_method(CompileTask*)
#                                     --30.47%--C2Compiler::compile_method(ciEnv*, ciMethod*, int, bool
#                                     Compile::Compile(ciEnv*, ciMethod*, int, bool, bool, bool, bool, |
#                                               |--17.57%--Compile::Code_Gen()
#                                               |          |--12.46%--PhaseChaitin::Register_Allocate()
#                                               |          |          |--2.79%--PhaseChaitin::build_ifg_
#                                               |          |          |          --1.05%--PhaseChaitin
#                                               |          |--1.49%--PhaseChaitin::Split(unsigned int, |
#                                               |          |--1.26%--PhaseChaitin::post_allocate_copy_r
```

해석:

- 프로세스/스레드: C2 CompilerThree 스레드를 분석하는 중입니다.
- 총 CPU 사용량: 이 스레드는 CPU 시간의 30.61%를 담당합니다.
- 기능 흐름: 스레드는 `start_thread`로 시작하며 대리자는 여러 레이어에서 작동합니다. CPU 시간의 대량(30.47%)이 `C2Compiler::compile_method`에서 사용되어 잠재적인 핫스팟을 나타냅니다.

플레임 그래프 생성

1단계: 정의된 간격(예: 60초) 동안 모든 CPU 및 프로세스에서 성능 샘플을 생성합니다.

```
perf record -a -g -F 99 sleep 60
```

출력 예:

```
[ perf record: Woken up 32 times to write data ]
[ perf record: Captured and wrote 10.417 MB perf.data (67204 samples) ]
```

2단계: 이 perf.data 파일을 플래미그래프 템플릿 저장소를 다운로드할 수 있는 호스트에 복사하거나 전송합니다.

3단계: perf.data 파일을 텍스트 형식으로 변환합니다.

```
perf script > data.perf
```

4단계: FlameGraph GitHub 리포지토리를 복제하고 data.perf를 다음 디렉토리에 배치합니다.

```
cp data.perf $PWD/FlameGraph/.
```

5단계: Flamegraph 처리를 위해 스택 추적을 축소합니다.

<#root>

```
cat data.perf | ./stackcollapse-perf.pl > data.perf-folded
```

6단계: 화염 그래프 SVG 파일을 생성합니다.

<#root>

```
./flamegraph.pl data.perf-folded > data.svg
```

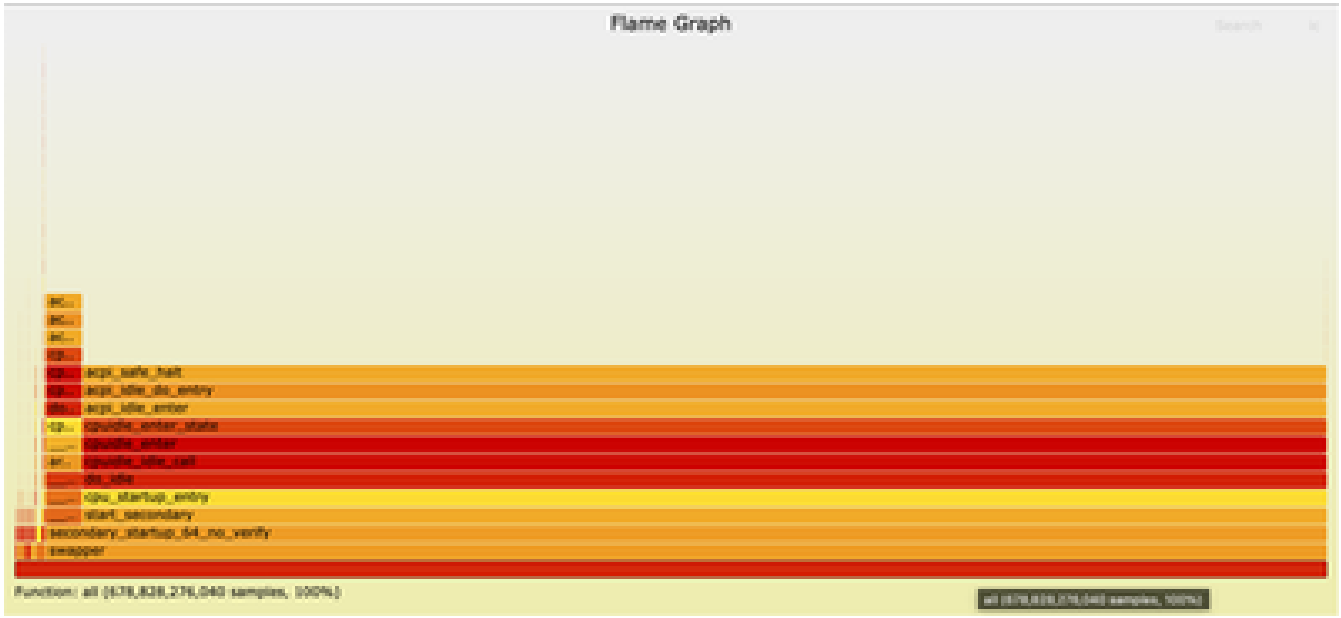
참고: CentOS 또는 RHEL에서 "open.pm in @INC을 찾을 수 없음" 오류가 발생하면 필요한 Perl 모듈을 설치합니다.

```
yum install perl-open.noarch
```

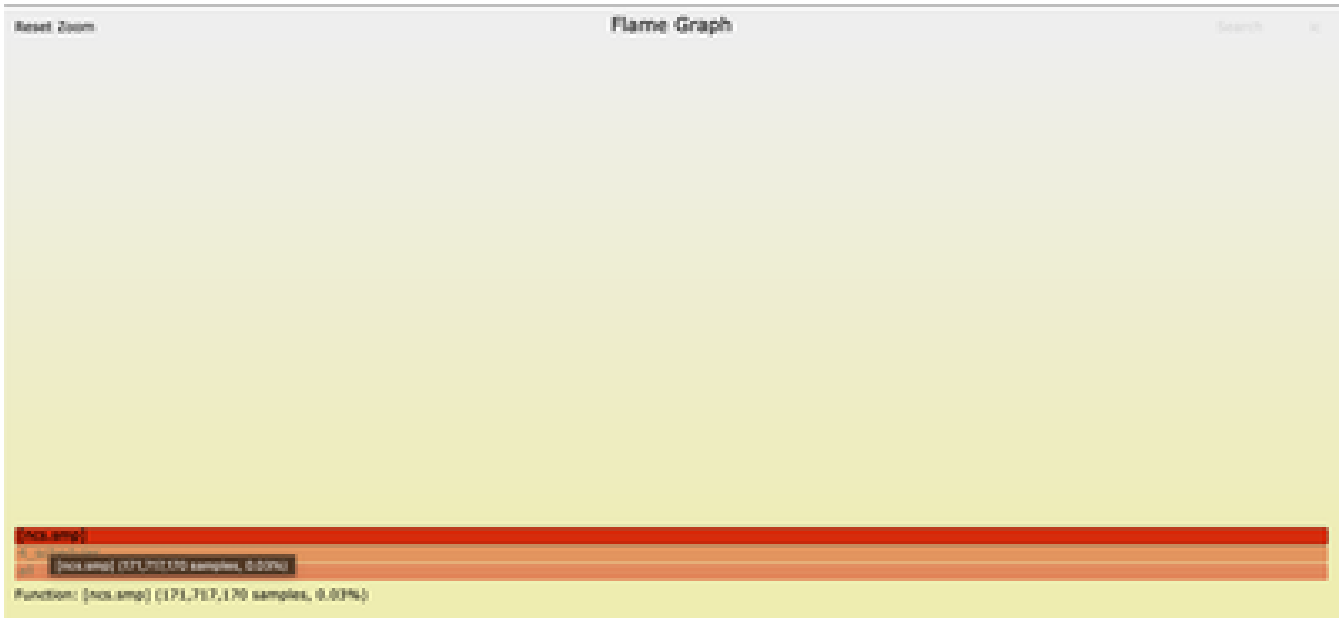
7단계: 원하는 웹 브라우저에서 data.svg 파일을 열어 불꽃 그래프를 시각화합니다.

Flame Graph 찾아보기

브라우저에서 화염 그래프 파일이 열리면 아무 상자나 클릭하여 해당 함수와 호출 스택을 확대하여 상호 작용할 수 있습니다. 각 상자의 길이는 해당 함수 및 해당 호출 스택에 소요된 CPU 시간을 나타냅니다. 이러한 시각화를 통해 최적화를 위한 핫스팟과 영역을 쉽게 식별할 수 있습니다.



ncs.smp에서 확대:



관련 정보

- Linux 성능 알려진 주의 사항
- Cisco 기술 지원 및 다운로드

이 번역에 관하여

Cisco는 전 세계 사용자에게 다양한 언어로 지원 콘텐츠를 제공하기 위해 기계 번역 기술과 수작업 번역을 병행하여 이 문서를 번역했습니다. 아무리 품질이 높은 기계 번역이라도 전문 번역가의 번역 결과물만큼 정확하지는 않습니다. Cisco Systems, Inc.는 이 같은 번역에 대해 어떠한 책임도 지지 않으며 항상 원본 영문 문서(링크 제공됨)를 참조할 것을 권장합니다.