Journal of The Institute of Electronics and Information Engineers Vol.56, NO.12, December 2019 https://doi.org/10.5573/ieie.2019.56.12.27
ISSN 2287-5026(Print) / ISSN 2288-159X(Online)

논문 2019-56-12-4

혼합 중요도 시스템에서 안전 필수 응용의 성능 고립화 방법

(Method for Performance Isolation of Safety-Critical Applications in Mixed-Criticality Systems)

김 명 선*

(Myungsun Kim©)

요 약

자율주행 자동차는 대표적인 혼합 중요도(mixed-criticality) 시스템이다. 이러한 시스템에서는 안전 운행에 밀접하게 관련되고 동시에 실시간성이 보장되어야 하는 안전 필수(safety-critical) 응용들과 일반(non-safety-critical) 응용들이 CPU 혹은 메모리와 같은 시스템 자원을 공유하며 수행된다. 객체 탐지(object detection)는 가장 중요한 안전 필수(safety-critical) 응용이며 DNN을 사용하여 높은 정확도를 나타낸다. 하지만 많은 연산량과 메모리 사용으로 다른 응용들과 시스템 자원을 공유할 경우 성능 간섭을 피할 수 없다. 본 연구에서는 안전 필수 응용들이 시작되면 일반(non-safety-critical) 응용들은 시스템 자원 사용률이 억제되는 방법을 제안한다. 제안된 기법을 NVIDIA사의 Jetson AGX Xavier에 탑재하고 그 효용성을 검증하였다. 실험을 통하여 확인한 결과, YOLOv3-tiny가 메모리 집중적인 응용들에게 성능 간섭을 받는 경우에도 제안된 기법을 적용할 때 성능이 최대 39% 향상되어 원래의 성능과 비슷한 결과를 나타내었다.

Abstract

Autonomous vehicles are representative mixed-criticality systems. In such systems, safety critical applications which are closely related to safe operation and must be guaranteed in real time, are performed by sharing system resources such as CPU or memory with non-safety-critical ones. Object Detection is the most important safety-critical application and shows high accuracy using DNN technology. However, due to the large amount of computation and memory usage, performance interference is inevitable when sharing system resources with other applications. In this paper, we propose a method to reduce system resource utilization of non-safety-critical applications when safety-critical ones are get started. The proposed scheme is running on top of NVIDIA Jetson AGX Xavier and its effectiveness is verified. Experimental results show that YOLOv3-tiny improves by up to 39% when the proposed technique is applied, even under being interfered by heavy memory-intensive applications.

Keywords : Deep neural network, Object detection, Mixed-critical system, Performance isolation

Ⅰ. 서 론

시스템에서 사용되는 응용들이 여러 수준의 중요도를 가지고 있을 때 이러한 시스템을 혼합 중요도(mixed-criticality) 시스템이라고 부른다[1]. 자율주행 자동차는 혼합 중요도 시스템의 대표적인 예이다. 이 시스템에서 사용되는 응용들은 1) 운행의 안전을 담당하는 안전 필수(safety-critical) 응용과 2) 일반 응용(non-safety-critical)으로 나뉠 수 있다.

안전 필수 응용은 영상 정보를 실시간으로 분석하는 능력을 갖추어야 한다. 카메라, 적외선, 라이다 등의 센서를 사용하여 수집한 차량 주변의 객체나 사람 등의 영상 정보를 정확하고 빠르게 분석할 수 있어야 한다. 최근 DNN(deep neural network) 기술의 발달로 매우 높은 정확도로 객체 탐지(object detection)가 가능하다[2~4]. 따라서 자율주행 자동차의 안전 필수 응용에 사용되는 기술의 핵심은 전장과 같은 임베디드 시스템에서 실시간으로 DNN 연산을 수행할 수 있는 컴퓨팅 능력이라 할 수 있다.

*정회원, 한성대학교 IT융합공학부(Department of IT Convergence Engineering, Hansung University)
©Corresponding Author(E-mail : kmsjames@hansung.ac.kr)
※ “본 연구는 한성대학교 교내학술연구비 지원과제임.”
Received ; August 14, 2019 Revised ; September 14, 2019
Accepted ; November 11, 2019

일반 응용과 안전 필수 응용이 동시에 수행되면 DNN 연산은 일반 응용과 메모리, CPU 자원 등을 공유하게 된다. DNN은 큰 규모의 MAC(mmultiplication and accumulation) 연산을 하고 많은 양의 데이터를 사용하므로 이러한 자원 공유 상황에서는 DNN 연산이 다른 응용들로부터의 성능 간섭을 피하기 어렵다. 따라서 객체 탐지와 같은 안전 필수 응용의 실시간성을 보장하기 위해서는 일반 응용으로부터의 성능 고립화(performance isolation)가 필요하다.

본 논문에서는 객체 탐지와 같은 DNN 기반 응용이 성능 고립화가 보장될 수 있는 기법을 제안한다. 제안된 기법은 먼저 일반 응용들의 태스크들과 전체 시스템의 나머지 태스크들을 각각 그룹화 한다. 안전 필수 응용이 시작되면 리눅스 커널의 cgroup 제어기는 일반 응용들의 태스크들로 구성된 그룹의 최대 시스템 자원 사용율을 일정 수준 이하로 제한한다. 이를 통하여 안전 필수 응용들은 상대적으로 높은 사용율을 보장받게 되어 성능 고립화를 달성할 수 있다. 이후 안전 필수 응용이 종료되면 해당 그룹의 제한되었던 사용율은 원래의 값으로 복원된다.

Ⅱ. 대상 시스템 및 문제 정의

본 절에서는 본 논문에서 대상으로 하는 시스템을 설명하고 이를 모델링한다. 이어서 본 연구에서 풀고자 하는 문제를 정의한다.

1. 대상 시스템

그림 1은 본 논문에서 대상으로 하는 시스템을 보여준다. 객체 탐지와 같은 안전 필수 응용은 DNN을 주로 사용하며 이는 CPU와 같은 범용 프로세서가 아닌 GPU(graphics processing unit) 혹은 DLA(deep learning accelerator)와 같은 DNN 연산에 특화된 프로세서를 사용한다.

GPU를 DNN에 사용하기 위하여 응용프로그램들은 CUDA 혹은 OpenCL과 같은 프로그래밍 방식을 사용한다. 또한 cuDNN[5]과 같은 DNN 라이브러리를 사용하여 최적화된 DNN연산이 가능하게 하고, 런타임을 DNN 프레임워크 내부에 위치하게 하여 GPU 드라이버를 호출할 수 있게 한다. 이를 통하여 병렬처리에 특화된 GPU를 접근할 수 있고 CPU보다 훨씬 많은 단위 시간당 MAC 연산을 수행할 수 있다.

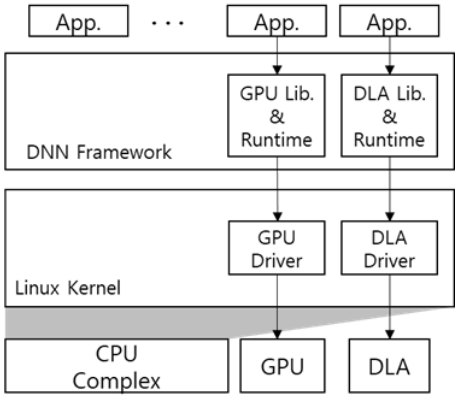

그림 1. 대상 시스템
Fig. 1. Target system.

DLA는 주로 FPGA(field programmable gate array)나 전용 하드웨어 블록으로 개발된다. NVIDIA의 NVDLA는 대표적인 예이다. GPU와 마찬가지로 제조사에서 제공하는 라이브러리, 런타임, 드라이버 등을 통하여 DLA에 접근할 수 있다. GPU와 다른 점은 주로 특정 DNN에 특화된 구조를 갖추어 상대적으로 다양한 DNN을 소프트웨어로 프로그래밍한 후 탑재하기 어려운 점이다. 하지만 에너지 및 연산 효율적인 측면에서는 GPU보다 훨씬 높은 성능을 나타낸다[6~7].

전체 시스템 관점에서 보면 그림 1에 나타낸 것처럼 기본적으로 모든 소프트웨어 스택이 CPU Complex(복수의 CPU로 구성)에서 수행된다. CPU 위에서 수행되는 응용프로그램들이 런타임과 드라이버를 호출한 후 커널이라는 소프트웨어적 수행단위의 일을 GPU 혹은 DLA에 전달한다. 그 후 전달된 커널의 MAC 연산을 병렬적으로 하드웨어를 사용하여 가속한다. 이후 그 결과를 CPU에게 전달한다.

정리하면, 본 연구에서 다루는 대상 시스템은 안전 필수 응용이 수행하는 DNN 연산을 위하여 CPU는 호스트 프로세서로, GPU와 DLA는 보조 프로세서(coprocessor)로 동작한다.

2. 시스템 모델링 및 문제 정의

호스트 프로세서로 동작하는 CPU Complex는 n개의 CPU로 이루어져 있다. 각각의 CPU는 런큐(run queue)[8]를 하나씩 가지고 있다. 시스템 전체적으로 n개의 런큐가 존재하고 각각의 역할은 현재 CPU를 점유하고 있는 태스크가 CPU 사용을 끝내면 새로운 태스크가 CPU를 점유하게 되는데 이때 점유 가능한 태스크들의 집합을 나타낸다[8].

시스템의 모든 런큐에 속할 수 있는 태스크들은 세 가지로 정의할 수 있다. GPU 혹은 DLA를 사용하고 DNN을 연산하는 태스크들은 안전 필수 태스크로 정의하고 $T^C = \{\tau_1^C, \tau_2^C, \cdots\}$로 표시한다. 일반 응용들의 태스크들은 $T^N = \{\tau_1^N, \tau_2^N, \cdots\}$, 리눅스 커널이 자체적으로 사용하는 시스템 데몬이나 서버와 같은 커널 태크스들은 $T^L = \{\tau_1^L, \tau_2^L, \cdots\}$로 표시한다. $\tau_i^L \in T^L$를 만족하는 태스크 τ_i^L는 어떤 이벤트를 기다리다가 발생하면 동작하는 태스크로, 장시간 CPU를 점유하지 않고 짧은 시간 동안 이벤트를 처리하고 슬립(sleep) 상태로 천이한다.

시스템에서 사용하는 스케줄러는 리눅스 커널의 CFS(completely fair scheduler)[8]를 사용한다. 따라서 $\tau_i \in \{T^C \cup T^N \cup T^L\}$를 만족하는 어떤 태스크 τ_i는 자신의 가상 실행시간(virtual runtime[8])크기에 따라서 CPU를 점유하는 순서가 결정되고 자신의 가중치(weight[8])에 따라서 CPU를 점유할 수 있는 시간이 정해진다.

본 연구에서 풀고자 하는 문제는 $\tau_i^C \in T^C$를 만족하는 태스크 τ_i^C가 $\tau_i^N \in T^N$를 만족하는 태스크 τ_i^N에 의하여 CPU와 메모리 자원을 사용함에 있어서 최대한 방해받지 않게 하고, 나아가 CPU에서 수행중인 τ_i^C가 더 많은 시간 동안 GPU 혹은 DLA를 보조 프로세서 형태로 사용할 수 있게 함에 있다. GPU나 DLA를 통해서 DNN 연산을 가속하는 태스크 τ_i^C는 단위 시간당 CPU를 점유하는 시간에 비례하여 GPU나 DLA에 드라이버를 통하여 단위 시간당 커맨드를 전송할 수 있다. 따라서 리눅스 CFS가 τ_i^C에 더 많은 단위 시간당 CPU 점유 시간을 할당하면, 단위 시간당 GPU나 DLA에 전달되는 커맨드 전송 횟수가 증가한다. 결과적으로 τ_i^C는 더 많은 DNN 연산을 가속 수행할 수 있게 되어 안전 필수 응용의 성능이 향상된다.

Ⅲ. 제안 방법

본 절에서는 앞 절에서 정의한 문제를 풀기 위하여 제안하는 방법을 설명한다. 구체적으로 솔루션의 구조와 이 구조를 이루고 있는 세부 기법들을 기술적으로 설명한다.

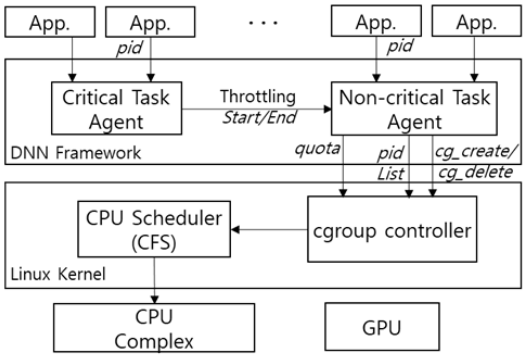

그림 2. 솔루션 구조
Fig. 2. Solution architecture.

1. 솔루션 구조

그림 2는 본 연구에 사용된 솔루션의 전체적인 구성을 나타낸다. Ⅱ 절의 대상 시스템에서 기술한 DNN 연산에 특화된 프로세서로 GPU를 사용한다. 솔루션에 사용된 핵심 내용은 T^C에 속한 태스크들이 수행될 때 T^N에 속한 태스크들의 CPU 점유율을 제한시키는 것이다. 이때 리눅스 커널의 cgroup 제어기를 동작시켜 T^N에 속한 태스크들의 CPU 점유율을 조정한다.

이를 위하여 먼저 CTA(critical task agent)와 NTA(non-critical task agent)를 DNN 프레임워크에 포함시킨다. 이 들은 현재 시스템의 런큐에 등록되어 있거나 CPU를 점유하고 있는 안전 필수 태스크와 일반 응용의 태스크들의 pid를 수집하고 각각을 그룹화 한다. NTA에서 전달된 정보들을 이용하여 리눅스 커널의 cgrouop 제어기는 CPU 점유를 제한받아야 하는 그룹에 속한 태스크들과 이들의 점유율을 결정하여 CFS[8]에게 전달한다.

2. 솔루션의 동작 방법

CTA는 $\tau_i^C \in T^C$를 만족하는 태스크 τ_i^C의 pid를 입력으로 받고 NTA에는 $\tau_i^N \in T^N$를 만족하는 태스크 τ_i^N의 pid가 입력된다. CTA가 *start* 신호를 NTA에 전송하면 이는 τ_i^C가 현재 수행을 시작하고 있다는 것을 의미한다. NTA는 $\tau_i^N \in T^N$를 만족하고 현재 시스템의 n개의 CPU 중 하나를 점유하고 있거나 혹은 해당 CPU의 런큐에 속해있는 τ_i^N와 같은 태스크들의 pid 값(그림 2의 *pid List*)들을 리눅스 커널의 cgroup 제어기에 전달한다.

전달된 *pid List*는 일반 응용 태스크들의 그룹이며 NTA가 *cg_create* 커맨드를 전송하면 이 그룹은 리눅

스 커널 내부에 하나의 *cgroup*으로 형성된다. 그 후 NTA는 cgroup 제어기에게 그림 2에 나타낸 *quota* 값을 전달하고 이는 형성된 *cgroup*이 사용할 수 있는 최대 CPU 점유율을 나타낸다.

그림 3은 전달된 *quota* 값에 의하여 CPU 사용율을 제한받게 되는 k개의 T^N에 속한 태스크들이 *cgroup2*의 이름으로 그룹화되고 그 외 시스템의 다른 태스크들은 *cgroup1*로 형성되어있는 모습을 나타낸다.

CPU complex 내부에 있는 n개의 CPU의 전체 점유율을 100%라고 할 때 각 CPU는 $\frac{100}{n}$%의 점유율을 갖게 된다. 태스크 τ_i^N가 임의의 한 시점에서 한 개의 CPU를 점유할 수 있는 비율을 $R_i^N(0 < R_i^N \leq 1)$이라고 정의하면 *quota*와 R_i^N는 다음과 같은 관계로 나타낼 수 있다.

$$\sum_{i}^{k} \frac{100}{n} R_i^N = quota \tag{1}$$

식 (1)에서 알 수 있듯이 *quota*는 전체 CPU 중 일반 응용들의 태스크들이 점유할 수 있는 비율을 나타내고 CFS는 위 식을 만족하는 R_i^N를 사용하여 τ_i^N의 CPU 점유시간을 계산한 후 스케줄링하게 된다. 결과적으로 객체 탐지에 사용되는 안전 필수 태스크들은 CPU 사용율을 더 높게 유지할 수 있어서 성능저하의 가능성이 줄어들게 된다.

이후 CTA로부터 *End* 신호를 NTA가 수신하게 되면 *cg_delete* 커맨드를 통하여 리눅스 커널에 존재하는 *cgroup2*를 해제하여 일반 응용들의 태스크들이 원래의 CPU 점유시간을 회복하게 한다.

3. 솔루션 비교

중요한 태스크들에게 CPU 점유 시간을 더 할당하여 시스템 자원을 더 많은 시간 사용할 수 있게 하는 방법으로 Ⅱ-2에서 설명한 태스크들에게 부여된 가중치(weight)를 조절하는 방법이 있다[9]. 태스크 τ_i가 CPU를 점유할 수 있는 시간(타임 슬라이스)은 CFS에서 아래 식 (2)로 정의된다[9].

식 (2)에서 S는 CPU당 한 개씩 할당된 런큐 내부의 모든 실행 가능한 태스크들의 집합을 나타내고 $W(\tau_i)$는 τ_i의 가중치, P는 CFS에서 사용하는 상수이다. 즉

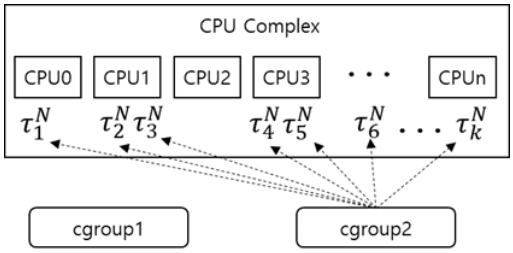

그림 3. 일반 응용 태스크들의 그룹화
Fig. 3. Grouping the non-safety-critical tasks.

$W(\tau_i)$값을 크게/작게 할당하면 그만큼 긴/짧은 타임 슬라이스가 부여되어 CPU 점유 시간이 조절된다.

$$TS(\tau_i) = \frac{W(\tau_i)}{\sum_{\tau_j \in S} W(\tau_j)} \times P \tag{2}$$

위 식 (2)에서 알 수 있듯이 해당 런큐의 S에 속한 모든 태스크들에 의하여 어떤 태스크의 타임 슬라이스가 결정되기 때문에, T^N에 속한 태스크들이 그림 3처럼 n개의 CPU 런큐에 흩어져있을 경우, 식 (1)의 *quota*를 얻기 위하여 n개의 CPU 런큐에서 모든 $\tau_i^N \in T^N$에게 알맞은 타임 슬라이스를 계산하여야하는 복잡함이 있다. 따라서 본 연구에서는 시스템 전체적 관점에서 그룹화된 태스크들의 CPU 점유율을 쉽게 제어할 수 있는 cgroup을 이용하였다.

Ⅳ. 실 험

본 절에서는 제안된 솔루션의 효용성을 입증하기 위하여 수행한 실험 내용을 다룬다. 먼저 실험에 사용한 대상 시스템과 응용을 소개하고 이어서 실험 결과와 이의 분석 내용을 기술한다.

1. 실험에 사용된 시스템 및 응용

본 연구에서는 NVIDIA(사)의 Jetson AGX Xavier를 대상 시스템으로 한다[10]. 해당 시스템은 대표적인 자율 기기(autonomous machine)이며 배송 및 물류 로봇, 대형 산업용 UAV(unmanned aerial vehicle)로 사용이 가능하다[10]. 표 1은 이의 구체적인 사양을 보여준다.

실험에 사용된 일반 응용은 두 가지로 나눌 수 있다. 첫 번째 일반 응용은 인위적 합성 응용(synthetic workload)이며 이는 단순한 산술연산을 집중적으로 수행하는 CPU 집중적인 태스크와 힙영역(malloc으로 할당)에 어

떤 값을 집중적으로 읽고 쓰는 동작을 하는 메모리 집중적인 태스크로 구성된다. 두 번째 일반 응용은 SPEC CPU2017[11]을 사용하였다. 이 벤치마크에서는 IPC(instruction per cycle)와 메모리 사용량이 상대적으로 많은 638.imagick_s, 619.lbm_s 두 개의 벤치마크 프로그램을 선정하였다.

안전 필수 응용으로 DNN기반 객체 탐지 응용인 YOLOv3-tiny[2]를 사용하고 임의의 교통사고 장면을 담은 동영상을 입력으로 하였다. 이 안전 필수 응용에서는 매 프레임마다 자동차들을 인식하고 이들의 테두리를 그린다. 따라서 측정된 FPS(frame per second)가 높을수록 많은 프레임에서 자동차 인식을 수행했다는 뜻이고 이는 YOLOv3-tiny의 성능이 높다는 의미다.

표 1. Jetson AGX Xavier 사양
Table1. Jetson AGX Xavier specifications.

CPU	8-Core ARM v8.2 64-Bit CPU, 8MB L2 + 4MB L3
GPU	512-Core Volta GPU/64 Tensor Cores 11 TFLOPS (FP16), 22 TOPS (INT8)
Memory	16GB 256-bit LPDDR4x 2133MHz - 137GB/s
Storage	32GB eMMC 5.1
Linux kernel version	4.9.108

2. 실험 결과 및 분석

가. 인위적 합성 응용과 객체 탐지 성능의 관계

본 실험에서는 인위적 합성 응용을 사용하여 객체 탐지 응용이 겪는 성능 간섭을 알아본다. 인위적 합성 응용 없이 YOLOv3-tiny에 임의의 한 동영상을 입력 시 42초의 수행 시간과 23FPS(frame per second)가 측정되었다. 인위적 합성 응용의 태스크 숫자를 0에서 15로 변화시켜가며 그 성능 간섭 정도를 다르게 하였다.

그림 4는 CPU 집중적인 인위적 합성 응용과, 그리고 그림 5는 메모리 집중적인 인위적 합성 응용이 YOLOv3-tiny와 동시에 수행될 때의 성능을 나타낸다. 그림 4와 5의 x축은 YOLOv3-tiny와 동시에 수행되는 인위적 합성 응용의 태스크 수를 나타낸다. 그림 4와 5의 (a)는 YOLOv3-tiny의 FPS와 수행 시간을 나타내며 (b)는 그때 측정된 L1 캐시 미스 수와 시스템에 남아 있는 자유 메모리 공간의 크기를 나타낸다.

CPU와 메모리 집중적인 태스크들의 수가 증가할수록 FPS는 최대 52.1%까지 떨어지고 수행 시간 역시 약 52.5% 증가 되었다. 메모리 집중적인 태스크들이 동시

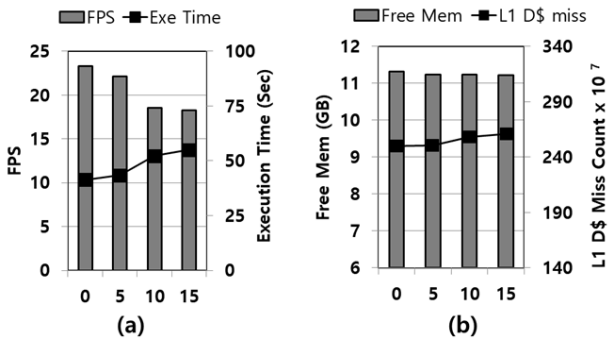

그림 4. CPU 집중적인 인위적 합성 응용과 객체 탐지 성능의 관계 : (a) FPS 및 수행시간, (b) 자유 메모리 크기 및 L1 데이터 캐시 미스 횟수

Fig. 4. Relation between the performance of object detection and CPU-intensive synthetic workloads: (a) FPS and execution time, (b) free memory size and the number of L1 data cache miss.

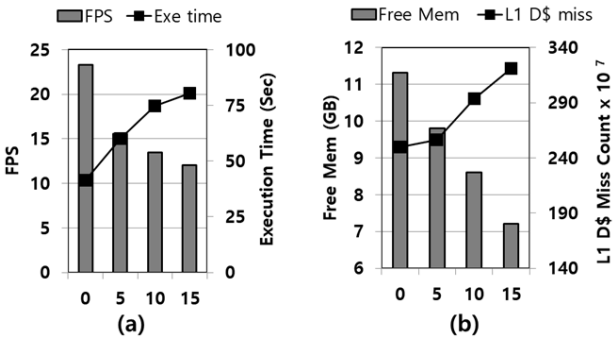

그림 5. 메모리 집중적인 인위적 합성 응용과 객체 탐지 성능의 관계 : (a) FPS 및 수행시간, (b) 자유 메모리 크기 및 L1 데이터 캐시 미스 횟수

Fig. 5. Relation between the performance of object detection and memory-intensive synthetic workloads : (a) FPS and execution time , (b) free memory size and the number of L1 data cache miss.

수행될 때 성능저하가 더 뚜렷한 이유는 그림 4(b)와 5(b)에서 태스크 수가 15일 때를 비교하면 알 수 있다. 그림에서 알 수 있듯이 메모리 집중 태스크들의 경우 L1 캐시 미스 수가 약 20% 더 많기 때문이다.

이때 한 가지 더 알 수 있는 점은 메모리 집중적인 태스크로 인한 성능저하가 메모리가 부족해서가 아니라는 점이다. 메모리 집중적인 태스크를 15개 수행해도 여전히 시스템에는 7.2GB의 자유 메모리 공간이 있다.

즉, 할당받을 메모리 공간의 문제가 아니고 다수의 태스크들이 한꺼번에 많은 메모리를 요구할 때, 메모리 제어기(memory controller)에서 겪게 되는 메모리 요구 병목현상이 추가적인 성능저하의 원인임을 간접적으로 알 수 있다.

나. 인위적 합성 응용 영향에서 성능 고립화

그림 6은 그림 4와 5에 나타낸 실험 결과 중 태스크들의 수가 10개인 환경에서, 본 논문에서 제안된 기법을 적용하기 전(w/o cgroup)과 후의 결과를 나타낸다. 기법을 적용 시에는 식(1)에서 나타낸 quota를 2.5%로 하였다. 그림 6.(a)와 6.(b)는 각각 CPU, 메모리 집중적인 인위적 합성 용용의 태스크를 YOLOv3-tiny와 동시에 수행하였을 때의 결과이다.

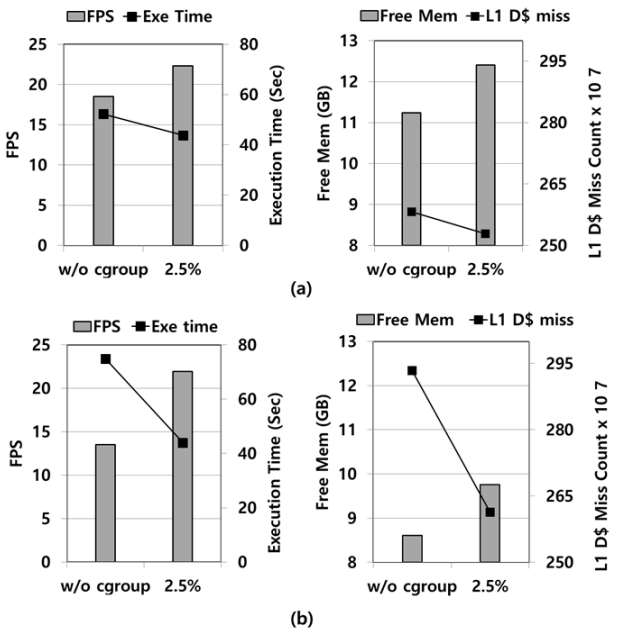

그림 6. 제안된 기법 적용 전후 성능 비교: (a) CPU 집중적인 태스크들과 수행, (b) 메모리 집중적인 태스크와 수행

Fig. 6. Performance comparison before and after applying the proposed methodology: (a) with CPU-intensive tasks, (b) with memory-intensive tasks.

CPU 집중적인 태스크들은 상대적으로 더 적은 메모리 요구를 하지만 10개의 태스크가 8개의 CPU에 골고루 흩어져 점유하게 된다. 이때 그림 6(a)의 왼편 그림에서 알 수 있듯이 10개의 태스크들이 전체 CPU의 2.5%만 점유할 수 있게 제어하면 약 17%의 FPS 향상되었다.

메모리 집중적인 태스크들은 많은 메모리 요구를 하고 동시에 10개의 태스크들이 8개의 CPU를 점유한다. 이때 2.5%로 CPU 점유율을 제한하면 시간당 수행할 수 있는 인스트럭션 수가 줄어들었음을 의미하고 동시에 그만큼 메모리를 요구하는 횟수도 줄어들게 되는 효과가 생긴다. 이에 따라 CPU 집중적인 태스크들의 경우보다 성능 향상이 뚜렷하여, 그림 6(b)에서 볼 수 있듯이 FPS는 약 39% 향상되고 수행 시간은 41.2%, L1 데이터 캐시 미스 수는 11% 감소하였다.

다. SPEC CPU2017 영향에서 성능 고립화

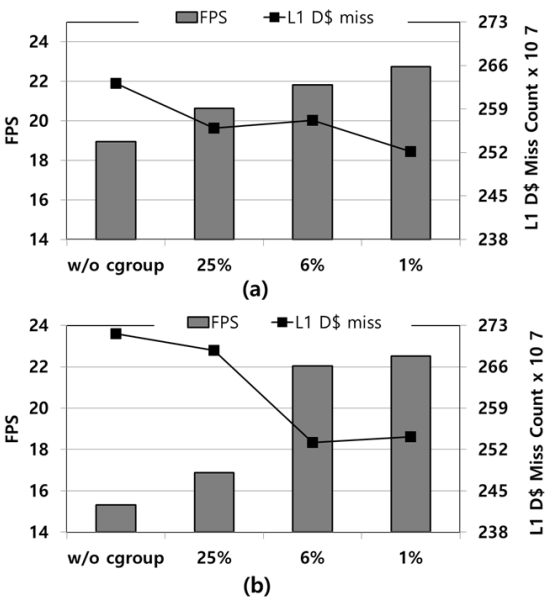

그림 7. SPEC CPU2017 벤치마크와 객체 탐지 성능: (a) 638.imagick_s와 수행, (b) 619.lbm_s와 수행

Fig. 7. Performance of object detection with SPEC CPU2017 benchmark suites: (a) with 638.imagick_s, (b) with 619.lbm_s.

그림 7(a)와 7(b)는 각각 SPEC CPU2017의 638.imagick_s와 619.lbm_s를 8개 CPU의 수만큼 생성한 후 이들 태스크들이 YOLOv3-tiny와 동시 수행될 때의 결과를 나타낸다. x 축은 제안된 기법을 적용하지 않을 때(w/o cgroup)와 식 (1)에서 나타낸 quota를 25%, 6%, 1%로 적용했을 때를 나타낸다.

두 경우 모두 quota를 작게 할수록 YOLOv3-tiny의 성능이 원래의 값인 23FPS에 가까워지고 동시에 L1 캐시 미스 수도 감소함을 알 수 있다.

V. 결 론

자율주행 자동차와 같은 혼합 중요도 시스템은 안전 운행에 관련된 안전 필수 응용들과 일반 응용들이 CPU 혹은 메모리를 공유하면서 수행된다. 대표적인 안전 필수 응용인 객체 탐지는 DNN을 사용하여 높은 정확성을 보장하나 많은 수의 MAC 연산과 메모리를 요구하여 일반 응용들과 동시에 수행될 때 CPU와 메모리 사용에 있어서 경합을 피할 수 없다. 본 연구에서는 이러

한 시스템 자원 공유에서 생기는 문제들을 안전 필수 응용이 최대한 겪지 않게 하는 방법을 제시하였다. 각 응용들이 사용하는 태스크들을 그룹화한 후, 일반 응용의 태스크들로 구성된 그룹이 사용할 수 있는 시스템 자원 사용율을 제한하였다. NVIDIA사의 Jetson AGX Xavier에 제안된 기법을 적용하고 실험한 결과, 대표적인 객체 탐지 응용인 YOLOv3-tiny가 CPU/메모리 집중적인 응용들에게 성능 간섭을 받는 상황에서도 원래 성능과 비슷한 값으로 복원될 수 있음을 나타내었다.

REFERENCES

[1] E. Rolf, and M. D. Natale, “Mixed Criticality Systems－A History of Misconceptions?”, IEEE Design & Test Vol. 33, No. 5, pp65-74, 2016, DOI: 10.1109/MDAT.2016.2594790.
[2] R, Joseph, S. K. Divvala, R. B. Girshick, A. Farhadi, “You only look once: Unified, real-time object detection”, Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR), 2016, DOI: 10.1109/CVPR.2016.91.
[3] J. Lee, S. Lee and S. Yang, “Model Ensemble for Speed Enhancement in Object Detection”, Journal of the Institute of Electronics and Information Engineers, Vol. 56, No. 6, June 2019, DOI: 10.5573/ieie.2019.56.6.35.
[4] J. Choi, D. Hwang, J. An and J. Lee, “Object Detection Using CNN for Automatic Landing of Drones”, Journal of the Institute of Electronics and Information Engineers, Vol. 56, No. 5, May 2019, DOI : 10.5573/ieie.2019.56.5.82.
[5] https://developer.nvidia.com/cudnn.
[6] Y. Chen, T. Krishna, J. S. Emer, V. Sze, “Eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks,” IEEE Journal of Solid-State Circuits, Vol. 52, no. 1, pp. 127-138, Nov. 2017, DOI: 10.1109/ISSCC.2016.7418007.
[7] Y. Chen, T. Luo, S. Liu, S. Zhang, L. He, J. Wang, L. Li, T. Chen, Z. Xu, N. Sun, and O. Temam, “DaDianNao: A machine-learning supercomputer,” in Proceeding of the 47th Annual IEEE/ACM International Symposium on Microarchitecture, Cambridge, UK, pp. 609-622, 2014, DOI: 10.1109/MICRO.2014.58.
[8] S. Huh, J. Yoo, M. Kim and S. Hong, “Providing Fair Share Scheduling on Multicore Cloud Servers via Virtual Runtime-based Task Migration Algorithm”, Proceedings of the 32nd IEEE International Conference on Distributed Computing Systems (ICDCS), pp. 606-614, Jun 2012, DOI:10.1016/j.jss.2016.11.053.
[9] S. Huh, H. Yoo, S. Hong, “Cross-layer Resource Control and Scheduling for Improving Interactivity in Android”, International Journal of Software: Practice and Experience, Vol 45, Number11, Nov, 2015, DOI: 10.1002/spe.2285.
[10] https://developer.nvidia.com/embedded/jetson-agx-xavier-developer-kit.
[11] A. Limaye and T. Adegbija, “A Workload Characterization of the SPEC CPU2017 Benchmark Suite,” 2018 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS), Belfast, 2018, pp. 149-158. DOI: 10.1109/ISPASS.2018.00028.

저 자 소 개

김 명 선(정회원)
2000년～2002년 LG전자 전송연구소 주임연구원
2002년～2011년 삼성전자 DMC 연구소 책임연구원
2016년 서울대학교 전기컴퓨터공학부 박사 졸업.
2016년～2019년 삼성전자 SR연구소 수석연구원
2019년～현재 한성대학교 IT융합공학부 조교수
<주관심분야 : 인공지능 가속기, Linux kernel, HW/SW Co-desgin 등>