Received September 25, 2019, accepted October 20, 2019, date of publication October 24, 2019, date of current version November 8, 2019.

Digital Object Identifier 10.1109/ACCESS.2019.2949483

SoCodeCNN: Program Source Code for Visual CNN Classification Using Computer Vision Methodology

SOMDIP DEY[1], (Student Member, IEEE), AMIT KUMAR SINGH[1], (Member, IEEE), DILIP KUMAR PRASAD[2], (Senior Member, IEEE), AND KLAUS DIETER MCDONALD-MAIER[1], (Senior Member, IEEE)
[1]Embedded and Intelligent Systems Laboratory, University of Essex, Colchester CO4 3SQ, U.K.
[2]Department of Computer Science, UiT The Arctic University of Norway, 9019 Tromsø, Norway

Corresponding author: Somdip Dey (somdip.dey@essex.ac.uk)

This work was supported in part by the UK Engineering and Physical Sciences Research Council EPSRC [EP/R02572X/1 and EP/P017487/1].

ABSTRACT Automated feature extraction from program source-code such that proper computing resources could be allocated to the program is very difficult given the current state of technology. Therefore, conventional methods call for skilled human intervention in order to achieve the task of feature extraction from programs. This research is the first to propose a novel human-inspired approach to automatically convert program source-codes to visual images. The images could be then utilized for automated classification by visual convolutional neural network (CNN) based algorithm. Experimental results show high prediction accuracy in classifying the types of program in a completely automated manner using this approach.

INDEX TERMS Classification, intermediate representation, LLVM, MPSoC, resource management, program, source code, computer vision, energy consumption, resource optimization, dynamic power management, machine learning.

I. INTRODUCTION

Recently we could see the emergence of several machine learning based methodologies to map and allocate resources such as CPU, GPU, memory, etc. to applications on embedded systems in order to achieve energy efficiency, performance, reliability, etc. Several studies, which are focused on extracting features from source code of an application and then utilizing several machine learning models [1]–[4] such as Support Vector Machines (SVMs), Nearest Neighbor, etc. to classify different set of applications and then deciding the resources that need to be allocated to such applications. Using such methodologies also have their own disadvantages. Depending on feature extraction such as number of code blocks, branches, divergent instructions, and then utilizing machine learning on them usually requires accurate identification of features from the training data and then feeding them to the model. Extracting features from a source code of a program and then feeding to the machine learning model so that further inference could be made is difficult in many ways.

Our observations have shown that with an addition of simple load & store instruction in a "Hello, World" program can lead to 16.98% difference in the platform-independent LLVM intermediate representation (IR) code [5], [6], which is a platform-independent low-level programming language. This proves that there is a scope to find and learn the pattern from the program source code to build more intelligent information systems such that autonomy and ability to demonstrate close-to-human like intelligence could be demonstrated by the computing system. LLVM IR is a strongly typed reduced instruction set, very similar to assembly level language, used by the LLVM compiler to represent program source code. Fig. 1 shows the histogram of the IR code of a "Hello, World!" program written in C (see Program 1) and another C program with an addition of integer variable initialization code (see Program 2). When we represented the source code of these programs into visual images using our *SoCodeCNN* (Program **So**urce **Code** for visual Convolutional Neural Network (**CNN**) classification) approach and

The associate editor coordinating the review of this manuscript and approving it for publication was Peng Liu.

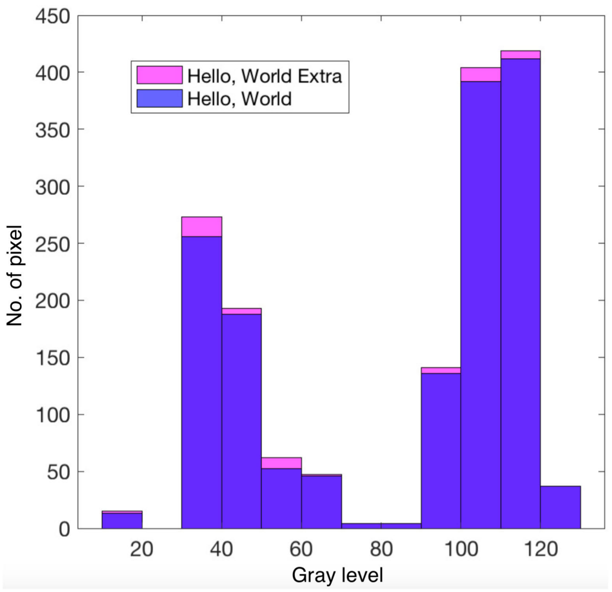


FIGURE 1. Histogram of source-code of "Hello, World" program vs Histogram of source-code of it with an additional integer variable initialization (Gray level vs Number of pixels).

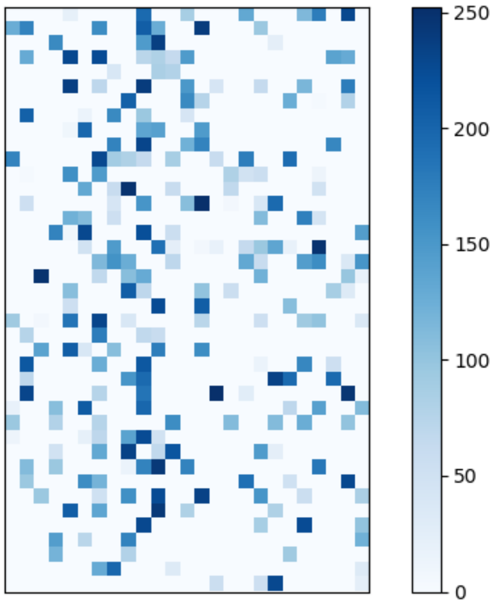


FIGURE 2. Differences in activation of neurons represented in shades of blue colour encoding.

passed them through a visual CNN based model, VGG16 [7], pre-trained with ImageNet dataset [8], [9], we observed that there was 14.77% difference in activation of neurons in the last fully connected layer consisting of 1000 neurons. We evaluated the difference between the activation of neurons for two different programs by converting the activation of neurons for each program into visual images and then compared using Quality of Index methodology (Q) [10]. Fig. 2 shows the differences in activation of 1000 neurons in the last fully connected layer of VGG16. In this figure, each cell in the matrix is represented as a colour ranging from 0 to 255, where each value ranges from white to different shades of blue. If the value is closer to 255 then the colour will be the darkest shade of blue whereas, the shade of blue fades away as the value is closer to 0. For the cells with white colour means that there was no difference (value equal to 0) in activation of neuron in that place for both the image representations of the programs (Program 1 and Program 2). However, if the cell has a colour other than white means that there is a difference between the activation of neurons in that place and the strength of the difference is represented by the darker shade of blue as mentioned earlier. The way we evaluate the difference of neuron values through blue colour representation is by finding the difference in the neuron values and then normalizing the value ranging from 0 to 255 (similar to ASCII values), where each value represents a shade of blue as mentioned above. More details on evaluating the difference between two images as an image representation of different shades of blue is provided in Sec. IV.

Program 1 Pseudo-code for "Hello World!"	**Program 2** Pseudo-code for "Hello World!" with additional load/store instruction

print("Hello, World!");	integer a = 3;
	print("Hello, World!");

In this paper, we took the inspiration from the human being's ability to learn from its surrounding visually [11]–[15]. In [11], [12] it is evident that humans learn and interact with their surroundings based on their visual perception and eyes playing an important role in the process and have grown to be one of the complex sensory organs with millions of years evolution. In fact, most humans start to learn and educate based on the visual representation of knowledge, may that be in the form of languages in written form or associating words with the visual representation of objects. Most scientists have also adopted this ideology and tried to extract patterns so that machines could be taught in the same manner. This gave rise to the interdisciplinary research between computer vision and natural language processing (NLP) in the field of artificial intelligence [16], where the main essence of the study is to teach computers to recognize, understand and learn human (natural) languages in the form of images. However, the trend in this interdisciplinary research is to understand patterns from human languages and then impart the knowledge to computers. For example, in order to teach computers to understand the digit '7', features from several human written forms of '7' are extracted and then imparted to the computer [17]. This method of learning could be synonymous to the example where a non-English speaking foreigner learns English by first associating the English words to their mother tongue and then remembering the word to learn English [18], [19]. Let's call this *learning approach 1*. In contrary, if we consider the example of how most human babies learn a language is through the process of associating phonetic words with the visual representation of objects first and then understanding the differences in features of different objects and remembering the associated words [14], [20], [21]. Let's call this *learning approach 2*. While it could be very intuitive to just take a picture of the program source-code (Program 1 & 2) and use NLP and visual CNN to classify the program, this approach would be similar to the *learning approach 1*. However, *learning approach 1* has its own limitations, especially when complex language frameworks are used in programs (more about this is discussed in Sec. II).

Although it should be kept in mind that the learning process in a human being is much more complex than covered by just two examples mentioned above and includes knowledge and information gathered from all sensory organs [15], [21] such as eyes, ears, tongue, skin, and nose.

In this paper, we have adopted the same ideology of learning through visual representation (*learning approach 2*) by converting the program source code into machine understandable intermediate representation and then trying to extract patterns and learning from them. To this extent the main contributions of this paper are as follows:

1) Propose **SoCodeCNN**, a way to convert program source code into more machine understandable visual representation (images) such that it makes the process of feature extraction from such images completely automated by utilizing the inherent feature extraction of visual deep convolutional neural network (DCNN) based algorithms, taking the expert skillful human effort out of the context.
2) Propose a new metric index named **Pixelator**: **Pixel** wise differenti**ator**, to understand the differences between two images pixel by pixel in a visually representative way.
3) Provide an exemplar case-study to utilize **SoCodeCNN** for practical applications in embedded devices. The exemplar application uses **SoCodeCNN** based classification to predict the types (*Compute intensive, Memory intensive, Mixed workload*) of different benchmark applications along with their probability of being a certain type, and then utilizing our heuristic based power management technique to save power consumption of the embedded device (Samsung Exynos 5422 multiprocessor system-on-a-chip [22]). To the best of our knowledge, this is the first work to convert program source code to a more machine understandable visual image and then classify into the type of program using CNN model in order to optimize power consumption.

II. MOTIVATIONAL CASE STUDY

A. TRADITIONAL FEATURE EXTRACTION FROM SOURCE CODE

Let us discuss the traditional approach of using machine learning on program source code with an example. We could assume that there is a simple program, which is capable of executing on several CPUs using OpenMP [23] programming framework. If we consider the following programs in Program 3 and Program 4 then if a skillful human without a knowledge of OpenMP is given the task of extracting features such as how many parallel executions of for loops are there or how many for loops are there in the programs, that person would classify both the programs (3 & 4) as the same, having two *for* loops in each algorithm. Whereas, Program 3 has one general *for* loop and one parallel *for* loop capable of executing on multiple threads. Therefore, the human being has to have special technical skills in order to understand such differences. In the study [2], the authors proposed a heuristic based deep learning methodology for allocating resources to executing applications by utilizing feature extraction from source code and then training a deep neural network (DNN) model to take decisions on resource allocation. Their proposed methodology requires special technical skill-set as described earlier.

Program 3 An OpenMP example of partial program

```
#pragma omp parallel
{
   #pragma omp for
   for (i = 0; i<N; i++) {
      c[i] = a[i] + b[i];
   }
}
for (i = 0; i<M; i++) {
   d[i] = a[i] + b[i];
}
```

Program 4 An example of partial program

```
for (i = 0; i<N; i++) {
   c[i] = a[i] + b[i];
}
for (i = 0; i<M; i++)
{
   d[i] = a[i] + b[i];
}
```

On the other hand if we consider that a program consists of 1000 features such as number of code blocks, branches, divergent instructions, number of instructions in divergent regions, etc. and each feature extraction requires 3 seconds for a human being then such program would consume 3000 seconds or 50 minutes for complete feature extraction so that those features could be further used in machine learning algorithm. Since feature extraction from program source-code using NLP is heavily utilized in compiler optimization and thus someone could argue that the field of compiler optimization has improved a lot in past couple of years [24], [25]. However, given the emergence of specialized frameworks and directives such as OpenMP, OpenCL, OpenVX, modern automated code feature extraction methods [1]–[4] are still lacking in pace in terms of accurately extracting such features in a completely automated manner. Therefore, human intervention for improved accuracy in feature extraction is always required. However, if we utilize our **SoCodeCNN** methodology of converting the program source code to images and then utilize them in visual convolutional neural networks (CNNs) then it does not require any human intervention in the process and could end up saving 50 minutes in manual feature extraction such as the case for the example mentioned above.

B. FILLING UP THE GAP

Instead of identifying features from source code by the user and then feeding them to machine learning models as in the conventional approaches, with our approach the machine learning model is able to understand and learn from the

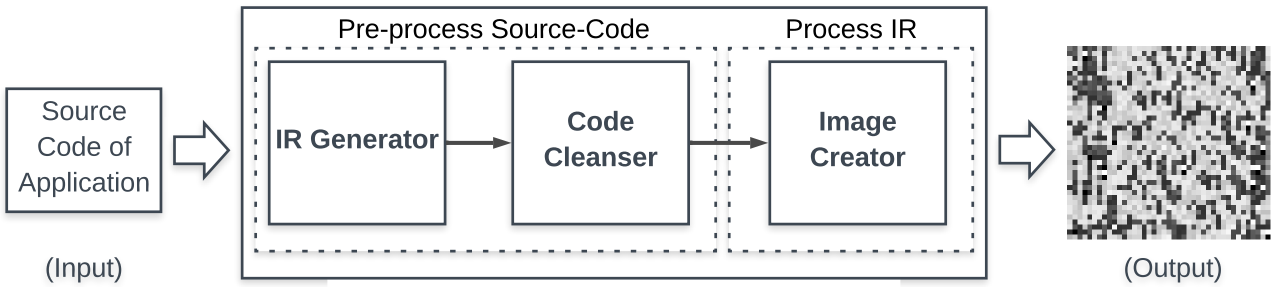


FIGURE 3. Block diagram of *SoCodeCNN*.

patterns in the source code of the program by themselves. One of our important observations which has led to the proposal of our methodology, "SoCodeCNN", human evolution inspired approach to convert program **So**urce **Code** to image for **CNN** classification/prediction, is that when we compared the two different source codes (Program 1 & 2), where the difference is only of that of an additional load/store instruction, there was a difference of 16.98% between the images using the Quality of Index methodology (Q) [10], and a Mean Squared Error (MSE) value of 7864.3. In Fig. 1 we also show the histogram of two different aforementioned source codes, which highlights the fact that even for a minute difference such as introducing a simple instruction is capable of creating a different pattern. The main motivation of this study is to fill up the gap in the usual conventional approaches by employing "SoCodeCNN" to automate feature extraction from program source-codes and using visual based machine learning algorithm to understand the inherent differences in patterns of source codes of different programs so that further learning and classification could be performed on such programs. In our proposed methodology, we introduce an effective way of converting source codes to visual images so that they could be fed to different computer vision based Deep Convolutional Neural Network for training and classification purposes.

III. SOCODECNN: HOW IT WORKS

Many human beings are not able to read or write using written languages, yet intelligent capacity of human brain and sophistication of visual capacity make the same human being intelligent enough to learn about the surrounding through visual representation of every object. For example, a human being might not be able to read or write "car" or "truck", yet when he/she sees one, the person instantly can differentiate between a car and a truck based on obvious visual features of each of these objects.

We try to impart the same kind of intelligence to a computer by representing each source-code of applications in the form of visual images. *SoCodeCNN* is not just a methodology but also a software application that processes source-codes to be represented as visual images, which is understandable by computing machines. It has two parts (*Pre-process Source-Code* and *Process Source-Code IR*), which are achieved through three distinct modules (refer to Fig. 3) accomplishing separate tasks on their own in order to achieve the accumulated goal. The three modules are as follows: *IR Generator, Code Cleanser & Image Creator*. *IR Generator* and *Code Cleanser* pre-process the source-code of the application in order to generate platform-independent intermediate representation code so that the visual image could be created, whereas the *Image Creator* actually processes the intermediate representation code of the program source-code to create a visual image. The overview of *SoCodeCNN* is provided in Algorithm 5. Next, we provide more details of steps *Pre-process Source-Code* and *Process Source-Code IR*.

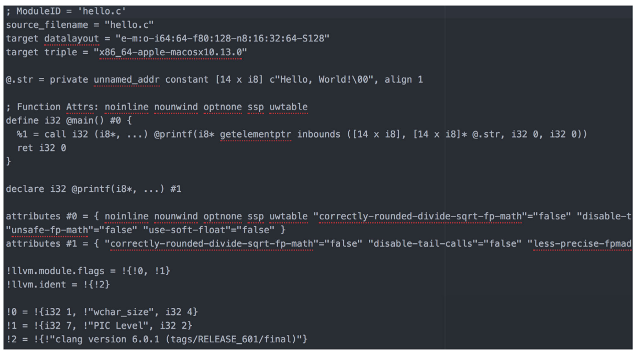
A. PRE-PROCESS SOURCE-CODE

The algorithm of this part is provided in Algo. 5 (from line 4 to 11).

1) IR GENERATOR

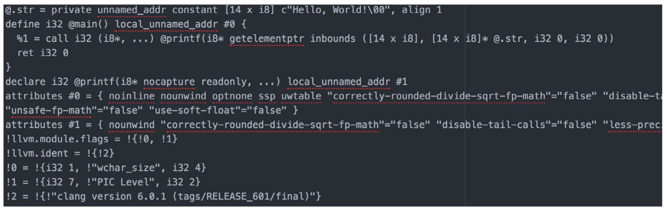
In this intermediate step, the LLVM intermediate representation (IR) [5], [6] of the source code of an instance of an application (App_i) is generated. LLVM IR is a low-level programming language, which is a strongly typed reduced instruction set computing (RISC) instruction set, very similar to assembly level language. The importance of converting to LLVM IR is that the code is human readable as well as easily translatable to machine executable code, which is platform-independent. This means that LLVM code could be used to build and execute an application instance (App_i) on any operating system such as Windows, Linux, Mac OS, etc. LLVM also provides a methodology to create optimized IR codes, where the IR code is optimized even further such as not including unused variables, memory optimization, etc. The *IR Generator* generates the optimized IR code from the program source-code for further processing.

For Example: When we convert the Program 1 to LLVM IR code we achieve the IR code as shown in the snapshot in Fig. 4.(a). We could notice that the first four lines consist of meta-data about the program itself such as the name of the program, related meta-information, etc. Although it should be noted that regardless of the target platform and the platform OS information is available in the LLVM IR code as meta-information, however, the variables and other instructions generated as part of the IR code is platform-independent.



```
; ModuleID = 'hello.c'
source_filename = "hello.c"
target datalayout = "e-m:o-i64:64-f80:128-n8:16:32:64-S128"
target triple = "x86_64-apple-macosx10.13.0"

@.str = private unnamed_addr constant [14 x i8] c"Hello, World!\00", align 1

; Function Attrs: noinline nounwind optnone ssp uwtable
define i32 @main() #0 {
  %1 = call i32 (i8*, ...) @printf(i8* getelementptr inbounds ([14 x i8], [14 x i8]* @.str, i32 0, i32 0))
  ret i32 0
}

declare i32 @printf(i8*, ...) #1

attributes #0 = { noinline nounwind optnone ssp uwtable "correctly-rounded-divide-sqrt-fp-math"="false" "disable-t
"unsafe-fp-math"="false" "use-soft-float"="false" }
attributes #1 = { "correctly-rounded-divide-sqrt-fp-math"="false" "disable-tail-calls"="false" "less-precise-fpmad

!llvm.module.flags = !{!0, !1}
!llvm.ident = !{!2}

!0 = !{i32 1, !"wchar_size", i32 4}
!1 = !{i32 7, !"PIC Level", i32 2}
!2 = !{!"clang version 6.0.1 (tags/RELEASE_601/final)"}
```

(a) Snapshot of LLVM IR Code of Program 1 generated by *IR Generator* module



```
@.str = private unnamed_addr constant [14 x i8] c"Hello, World!\00", align 1
define i32 @main() local_unnamed_addr #0 {
  %1 = call i32 (i8*, ...) @printf(i8* getelementptr inbounds ([14 x i8], [14 x i8]* @.str, i32 0, i32 0))
  ret i32 0
}
declare i32 @printf(i8* nocapture readonly, ...) local_unnamed_addr #1
attributes #0 = { noinline nounwind optnone ssp uwtable "correctly-rounded-divide-sqrt-fp-math"="false" "disable-t
"unsafe-fp-math"="false" "use-soft-float"="false" }
attributes #1 = { nounwind "correctly-rounded-divide-sqrt-fp-math"="false" "disable-tail-calls"="false" "less-prec
!llvm.module.flags = !{!0, !1}
!llvm.ident = !{!2}
!0 = !{i32 1, !"wchar_size", i32 4}
!1 = !{i32 7, !"PIC Level", i32 2}
!2 = !{!"clang version 6.0.1 (tags/RELEASE_601/final)"}
```

(b) Snapshot of LLVM IR Code of Program 1 after *Code Cleanser* module processed the IR code

FIGURE 4. Processing of LLVM IR Code of Program 1 by *IR Generator* and *Code Cleanser* modules.

2) CODE CLEANSER

The main job of the *Code Cleanser* module is to get rid of the redundant part of the IR code, which does not add any value in the process of understanding the implementation of the application. Such redundant part of the code consists of initializing the name of the application and on which platform the LLVM IR code is built for or comments in the IR, etc. Once the redundant part of the LLVM IR code is removed the IR is ready to be utilized for visual image creation by the *Image Creator*.

For example: Fig. 4.(b) shows the IR code after *Code Cleanser* processes the IR code which is generated from the *IR Generator* module.

B. PROCESS SOURCE-CODE IR

The algorithm of this part is provided in Algo. 5 (from line 12-34).

IMAGE CREATOR

The *Image Creator* module first gets the total number of characters in the file consisting of LLVM IR code and the number of characters ($SizeOf(IR)$ in line 14 of Algo. 5) is denoted by $totalSize$. The $totalSize$ would be used to evaluate the height and width of the visual image to be created and the relationship between the height, width and $totalSize$ is provided in Eq. 1. The height and width of the image are determined such that $|height - width|$ (see Eq. 1) is the least from all the possibilities of a set (F) of factors, $F = \{f_1, f_2, \ldots f_n\}$ (where $f_1, f_2, \ldots f_n$ are all possible factors of $totalSize$), of $totalSize$, and $height$ and $width$ belong to the set F.

$$totalSize = height \times width \tag{1}$$

When the $height$ and $width$ is evaluated, the *Image Creator* module creates an instance of an empty image matrix (Img_i) as $0_{M_{height \times width}}$. The *Image Creator* then parses through the file containing the LLVM IR code and reads the file character by character and fetches the ASCII value (a) of those characters. Since each unique character will have a unique ASCII value (number), we would be converting the IR code to their equivalent number representatives, which are correspondingly processed by the computing system. After fetching the ASCII value (a_j) of the character at position j of $totalSize$, the value at the corresponding position on the image matrix (Img_i) is replaced with the ASCII value of the character (as shown in line 28 to 31 in Algo. 5) since $totalSize$ follows a relationship with $height$ and $width$ as shown in Eq. 1. The image matrix could be denoted by the formula portrayed in Eq. 2.

$$Img_i = (a_{height,width}) \in \mathbb{R}^{heigth \times width} \quad \text{and}$$
$$a_{height,width} = ASCII(\, a_j \,) \quad \forall j \in \mathbb{R} \;\&\; 0 \leq j \leq totalSize \tag{2}$$

For Example: After *Image Creator* processes the optimized LLVM IR code of Program 1, we get a visual image as the Output in Fig. 3.

IV. PIXELATOR: PIXEL WISE DIFFERENTIATOR OF IMAGES

A. OVERVIEW OF PIXELATOR

We have also designed a special algorithm, which is capable of showing pixel wise difference between two separate images in the form of different colour shades. We call this algorithm as Pixelator view (**Pixel** wise differenti**ator** view). In the Pixelator view, two images are compared pixel by pixel where each difference in the pixel value is evaluated using Eq. 3 and the difference is shown in a cell in the matrix representation. Each pixel of first image (P^1), which is being compared, is converted into its equivalent integer value. Since each pixel of the image has a Red-Green-Blue (RGB) value associated with it, we use ($R \times 2^{16} + G \times 2^8 + B$) formula to convert the associated RGB value of the pixel of the first image to its corresponding integer, and then compared with the integer value (RGB to integer) of the corresponding pixel in the second image (P^2), where the difference in the value only ranges from 0 to 255 similar to ASCII values. Each value, ranging from 0 to 255, represents a shade of blue. Since most of the program source codes are written in the English language where each character in the code could be represented by a unique ASCII value ranging from 0 to 255, henceforth, we chose the range of difference between the

Algorithm 5 SoCodeCNN: The Methodology

```
Input:
$S(App)$: set of source-code of applications, where
source-code of each application instance is represented
as $App_i$
Output: $I$: set of visual images, each representing each
        $App_i$ in $S(App)$
Initialize:
$height_{imageMatrix} = 0$;
$width_{imageMatrix} = 0$;
Preprocess Source-code:
foreach $App_i$ in $S(App)$ do
    Generate LLVM IR using IR Generator module;
    Generate LLVM Optimized IR using IR Generator
    module;
    Strip all the program related metadata using Code
    Cleanser module;
    Strip all the comments using Code Cleanser module;
    Store the IR in a set $S(IR)$;
end
Process Source-code IR using Image Creator module:
foreach $IR_i$ in $S(IR)$ do
    $totalSize = SizeOf(IR_i)$;
    foreach Byte in $IR_i$ do
        Store in $imageArray[totalSize]$ as an integer
        value;
    end
    $lengthOfFactorArray$ = Total number of factors of
    $totalSize$;
    Factorize $totalSize$ and store in
    $factorArray[lengthOfFactorArray]$;
    foreach Factor, $f$,
    in $factorArray[lengthOfFactorArray]$ do
        $divisor = totalSize/f$;
        if $(f - divisor)$ is least then
            $height_{imageMatrix} = f$;
            $width_{imageMatrix} = divisor$;
        end
    end
    Create an image matrix, $Img_i$ with height,
    $height_{imageMatrix}$, and width, $width_{imageMatrix}$;
    foreach ASCII Integer value, $a_i$,
    in $imageArray[totalSize]$ do
        foreach Cell, $c_i$, in $Img_i$ do
            Store $a_i$ in $c_i$;
        end
    end
    Store $Img_i$ in $I$;
end
return $I$;
```

pixels to be within that.

$$int(P') = |int(P^1_{i,j}) - int(P^2_{i,j})| \; mod \; 255 \tag{3}$$

If we assume that h and w are the height and width of the referenced (original) image respectively then the *Pixelator* also integrates (adds) the difference between corresponding pixels to quantify the difference between two separate images using the Eq. 4. In 4, the h and w corresponds to the height and width of the image (P') respectively. We have given the index in Eq. 4 the same name as the approach itself for ease of naming convention. If the value of the index, *Pixelator*, using Eq. 4 is high then it means that the difference between the two images is also high and directly proportional.

Note: If the size of the images (P^1, P^2) are different then the pixel value of the smaller image (P^{small}, where$P^1 \leq P^{small} \leq P^2$) is compared with corresponding pixel value of the larger image (P^{large}, where$P^1 \leq P^{large} \leq P^2$) till the difference of all the pixel values of dimension ($h^{small} \times w^{small}$) of P^{small} are evaluated, where h^{small}, w^{small} corresponds to the height and width of P^{small} respectively.

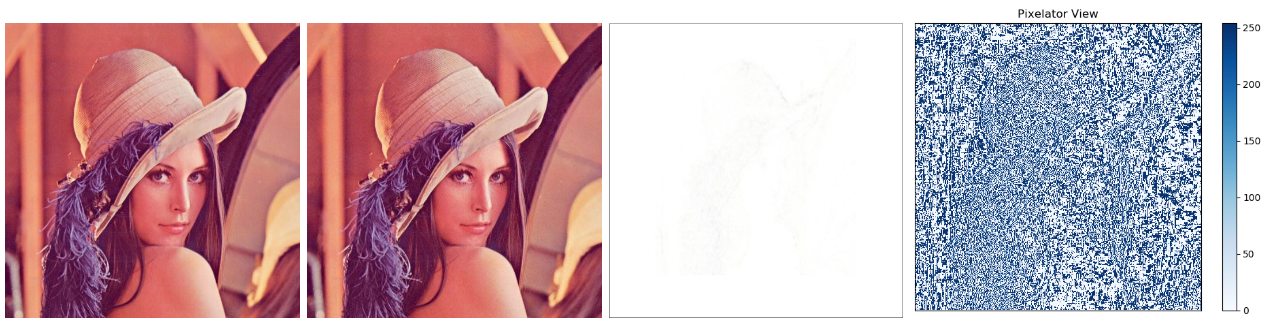
The reason to have both a quantitative value and a visual image to understand the difference between two images, pixel by pixel, is to make it easier for both human and machine to understand the differences between the images. Although it is easier and intuitive to understand the differences of the images just by visualizing and inspecting the difference by a human being, however, for a machine it is not easy to achieve such level of capability without some complex computation. Whereas, machines can process numbers faster and hence having a quantitative value (number) associated to measure the difference between two images is more readily understandable by the machine.

$$Pixelator = \sum_{1,1}^{h,w} int(P') \tag{4}$$

B. IMPORTANCE OF PIXELATOR

Some of the popular approaches for assessing perceptual image quality to quantify the visibility of errors (differences) between a distorted image and a reference image includes Mean Squared Error (*MSE*) [26], Quality of Index (Q) methodology [10] and Structural Similarity Index (SSIM) [27] for measuring image quality.

A widely adopted assumption in image processing is that the loss of perceptual quality is directly related to the visibility of the distorted image. An easy implementation of this concept is visualized in the MSE [26], where the differences between the distorted image and reference image is quantified objectively. But two distorted images with the same MSE may have very different types of distortion, where some of the distortions are much more visible than others. To overcome this issue in the study, Quality of Index (Q) methodology [10], Wang et al. developed an approach, which would quantify the distortion by modeling it as a combination of three factors: loss of correlation, luminance distortion, and contrast distortion. Hence, quantifying distortion in images as a number does not truly reflect the exact area on the image where distortion happened nor reflects the kind of distortion that



(a) Lena Original (b) Lena Distorted (c) SSIM Map (d) Pixelator View

FIGURE 5. **Highlighting differences between distorted Lena and reference Lena images using SSIM and Pixelator.**

took place. In contrast in SSIM [27] approach, Wang et al. developed a framework for quality assessment based on the degradation of structural information by computing three terms: the luminance, the contrast and the structural information. The overall SSIM is a multiplicative combination of the aforementioned three terms and represents the structural distortion appropriate for human visual perception. However, for minuscule distortions in one of the colour channels out of the RGB channels of the image, SSIM fails to represent such minimal distortion which could be differentiated for human visual perception (see example in Sec. IV-B).

In order to overcome the drawbacks of MSE, Q and SSIM we have developed *Pixelator*, which is not just able to quantify the distortion but at the same time represent the exact distortion area in an image representation, which is suitable and comprehensive for human visual perception. We developed *Pixelator*, especially to understand differences in images (example result as Output in Fig. 3) created from our *SoCodeCNN* approach such that we could understand and visualize minuscule modifications in these visual images due to minuscule changes in the program source-code, which might not be visualized easily in general.

AN EXAMPLE DEMONSTRATING THE IMPORTANCE OF PIXELATOR

We choose the Lena image (refer to Fig. 5.(a)), which is popularly utilized in image processing, to demonstrate the effectiveness of *Pixelator* over approaches such as MSE, Q and SSIM. We choose only one of the colour channels of the Lena image and for the pixel values in that channel representing 94, 95, 96, 97 and 220, we increment the corresponding values by 2 (distorted Lena image is shown in Fig. 5.(b)). The reason to choose the aforementioned pixel values is that from the histogram of the image we got to know that these pixel values were the most frequently occurring values in the chosen Red channel. When we evaluated the difference between distorted Lena image and the referenced Lena image using MSE, Q, SSIM and Pixelator, we could visualize that Pixelator is able to quantify and reflect the differences in the form of an image with respect to human visual perception (refer to Fig. 5.(d)), and at the same time outperforms the popular approaches. By differentiating the distorted and referenced images MSE gave a value of 9.7221, Q index evaluated to be 0.99984 (approx.), SSIM was evaluated to be 0.9959 (approx.) and *Pixelator* was evaluated to be 30533.43457. However, *Pixelator* is able to represent the differences more prominently in the form of visual representation image than SSIM, which is shown in Fig 5 as SSIM map. We could notice that regardless of having a SSIM value of 0.9959 (approx.), the approach is not able to represent the differences visually in SSIM map (refer to Fig. 5.(c)), whereas *Pixelator* is able to highlight the difference in each pixel wherever there is one.

Therefore, using *Pixelator* we are able to both visualize the difference between the original and distorted image, and quantify the difference at the same time, which could not be achieved by other popular methodologies such as MSE, Q and SSIM.

V. EXPERIMENTAL AND VALIDATION RESULTS

A. EXPERIMENTAL SETUP

We ran several sets of experiments to evaluate the potential and efficacy of utilizing *SoCodeCNN* and scale its usability. The first experiment denoted as *Exp. 1* was performed to see how much difference could be there in the visual images with the slightest modification in the program source code. We have chosen several simple programs with slight modifications to convey the efficacy of utilizing the *SoCodeCNN* methodology. In this experiment, the base program (denoted as *1st program*) is a "Hello, World!" program, which just prints out "Hello, World!" on the terminal (see Program 1). We added an additional load/store instruction in the base program (1st program), where we initialized an integer value into a variable and this program is denoted as the *2nd program* (see Program 2). In the next program, we added an additional code to the base program to print out three integer values and we denote this program as *3rd program* (see Program 6). In the *4th program*, also denoted as the same name in figures and tables, has some additional load/store codes to initialize three integer variables whereas one of the variables is the sum of the other two and the result of the summation is printed out on the terminal (see Program 7). We used *SoCodeCNN* to convert these program source-codes to visual images and compared the differences using histogram, Mean Squared Error (*MSE*) [26] and Quality of Index (Q) methodology [10].

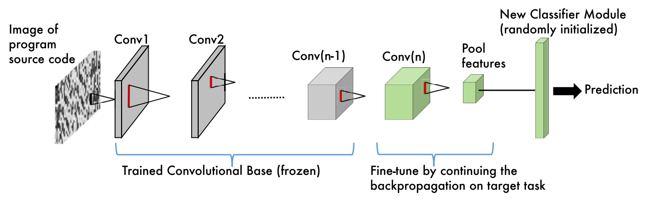


Image of program source code
Conv1
Conv2
Conv(n-1)
Conv(n)
Pool features
New Classifier Module (randomly initialized)
Prediction
Trained Convolutional Base (frozen)
Fine-tune by continuing the backpropagation on target task

FIGURE 6. Network architecture used for fine-tuning.

Program 6 3rd program pseudo-code

```
print("Hello, World!");
print(1 2 3);
```

Program 7 4th program pseudo-code

```
integer a = 3, b = 4, c;
print("Hello, World!");
c = a + b;
print("Sum of " + a + " and " + b + " is " +
c);
```

In the second set of experiments, denoted by *Exp. 2* we chose VGG16 [7] Imagenet trained model with our custom classifier having only three classes: *Compute, Memory, Mixed*. According to several studies [28], [29] different workloads could be classified as compute intensive, memory intensive, and mixed (compute and memory intensive) based on the number of instructions per cycle or memory accesses. The purpose of *Exp. 2* is to show the efficacy of existing CNN models to classify programs based on images generated by *SoCodeCNN*. The classes (*Compute, Memory, Mixed*) of our classifier reflects the different types of workloads and hence denotes the type of program application. The class *Compute* refers to the programs, which are very compute intensive, but has low memory transactions (read/write, data sharing/exchange) in comparison, whereas the class *Memory* represents the programs, which have really high memory transactions in comparison to the computation performed in such programs. The class *Mixed* represents programs, which are both compute intensive and memory intensive. We converted all the benchmarks of the PARSEC [30] benchmark suit using our *SoCodeCNN* and passed the corresponding images through the pre-trained VGG16 to fetch the Deep Dream [31] images from the last fully connected layer of the model for each of the three classes to compare the visual differences between these classes if there is any.

In the third set of experiments (*Exp. 3*) we utilized *SoCodeCNN* to convert the program source-codes of all benchmarks from the PARSEC, SPLASH-2 [32] and MiBench [33] benchmark suit. The purpose of *Exp. 3* is to show scalability of *SoCodeCNN*'s application with CNN model by classifying programs from some of the popular benchmark suits. There were 28 individual images in total created from PARSEC and SPLASH-2 including P-thread and serial version of some of the benchmarks, and were segregated into three different classes (Compute, Memory & Mixed) based on the study [34] comparing each benchmark with respect to their different number of instructions and memory transaction. To train and test the images for classification purposes VGG19 CNN model is used instead of VGG16 since it produced improved classification accuracy due to its deeper architecture. We fine-tuned VGG19 CNN model by adding our a new randomly initialized classifier, and training the last fully connected layer by freezing all the layers of the base model (frozen layers represented with gray colour in Fig. 6) and unfreezing the last fully connected layer (unfrozen layers represented with green colour in Fig. 6). In this way, only the weights of the last fully connected layer is updated and the classifier is trained with our images (see Fig. 6 for the CNN architecture used for fine-tuning). The source-code of benchmarks of MiBench are used for cross-validation purpose and testing the trained VGG19 CNN with our defined classes. The 28 images from PARSEC and SPLASH-2 were utilized to train the classifier and the last fully connected layer of the VGG19 pre-trained CNN using transfer learning [35] so that during prediction we could classify a program source-code image using a visual based CNN model such as VGG16/19. The *Compute* and *Mixed* classes have 10 images each, and the *Memory* class has 8 images for training. Due to the imbalance in the training dataset, weights of the classes were set accordingly to facilitate fair training.

Program 8 Execute power of 2 for 100,000 numbers iteratively on 4 different threads

```
function evaluatePowerOf2()
{
foreach $i$ in 100, 000 do
    Compute $i^2$;
end
}
Execute evaluatePowerOf2() on Thread 1;
Execute evaluatePowerOf2() on Thread 2;
Execute evaluatePowerOf2() on Thread 3;
Execute evaluatePowerOf2() on Thread 4;
```

Program 9 Execute power of 2 for 100,000 numbers iteratively and add the result with itself in a separate variable

```
function doubleSumOfPowerOf2()
{
  foreach $i$ in 100, 000 do
      z = Compute $i^2$;
      y = z;
      x = y + z;
  end
}
```

Since most of the benchmarks from MiBench are mixed load and, sometimes the benchmark programs are complicated to be segregated into either of the three different classes. Hence, to show the efficacy of using *SoCodeCNN* approach in CNN based algorithm we wrote simple programs, which would directly reflect either compute intensive or memory intensive or mixed workloads, and evaluate the classification outcome of such programs in *Exp. 3*. We wrote a simple program (see Program 8), which computes power of 2 for 100,000 numbers iteratively on 4 different threads and hence it could be classified as 'Compute'. We also slightly modified the program to make it more memory intensive by initializing the value of the power of 2 in a separate variable and then adding the result with itself in another separate variable (see

Program 9). In Prog. 9 instead of executing the computation function on four different threads, we execute it only on one thread, making it more memory intensive. We further slightly modified Prog. 9 to make it mixed workload (compute and memory intensive) by executing the memory intensive function iteratively on 4 separate threads (see Program 10). We fed the source-code of the Prog. 8, 9 and 10 to the trained CNN in order to verify the output classification.

Program 10 Execute power of 2 for 100,000 numbers iteratively and add the result with itself in a separate variable on 4 different threads

```
function doubleSumOfPowerOf2() {
foreach i in 100, 000 do
    z = Compute $i^2$;
    y = z;
    x = y + z;
end
}
Execute doubleSumOfPowerOf2() on Thread 1;
Execute doubleSumOfPowerOf2() on Thread 2;
Execute doubleSumOfPowerOf2() on Thread 3;
Execute doubleSumOfPowerOf2() on Thread 4;
```

B. EXPERIMENTAL RESULTS

Table 1 shows the result from *Exp. 1*, where different programs are compared using MSE and Q methodologies. From the table, it is evident that the visual representation of different program source-codes have different image representation and hence a potential playground for pattern recognition using visual CNN and image processing methodologies. Fig. 7 shows the histograms of four different programs (Program 1, 2, 6 and 7) of *Exp. 1*, where the X-axis represents the gray level of the visual images of the corresponding program and Y-axis represents the number of pixels for the corresponding gray level. Fig. 8 shows the Pixelator view of the differences in the visual images of Program 1 and Program 2.

Fig 9 shows the Deep Dream images of three different classes (Compute, Memory & Mixed) of program source-codes from *Exp. 2*, which proves that each class has different features that could be extracted to differentiate between program source-code in a visual manner. Fig. 2 shows the difference in activation of neurons of the VGG16 CNN when 1st Program and 2nd Programs are passed through the CNN model of *Exp. 2*.

In *Exp. 3* after training the VGG19 CNN the CNN achieved a validation prediction accuracy of 55.56% and when we passed the program source-code of Program 8, the trained CNN was able to classify the program as Compute intensive with the confidence probability of each class as shown in Table 2. From Table 2 we could also notice that although the CNN classified Program 8 as compute intensive but the probability of memory intensive is also high and that is because when the power of 2 is computed iteratively, the values are still stored in the memory and hence has moderately high memory transaction as well. Table 3 shows the classification prediction for Program 9 and Table 4 shows the classification prediction for Program 10. In order to verify whether Program 8, 9 & 10 are Compute, Memory intensive and Mixed workload respectively we used

TABLE 1. MSE and Q values of ***SoCodeCNN*** of different programs compared to 1st program.

Frequency Levels	MSE	Q
$1st\ Program$	0.0	1.0
$2nd\ Program$	7864.280746459962	0.830200472952753
$3rd\ Program$	6436.206069946288	0.871193634636040
$4th\ Program$	8134.392196655273	0.833703260745370

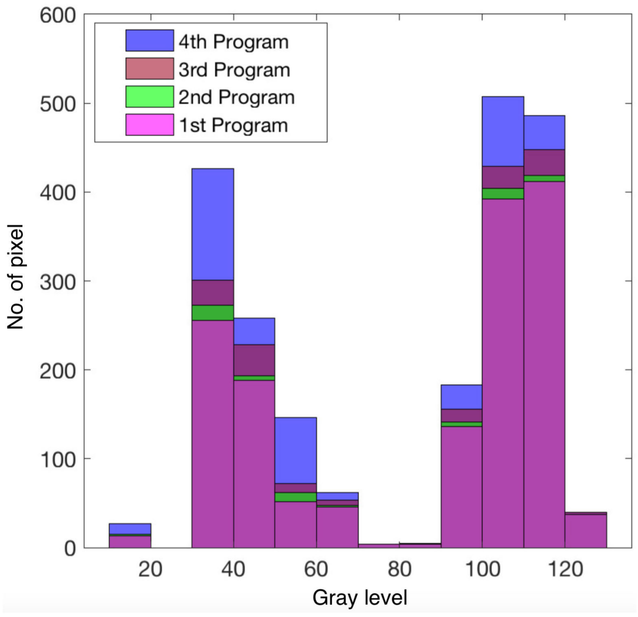


FIGURE 7. Histogram of source-code of 1st, 2nd, 3rd and 4th Program.

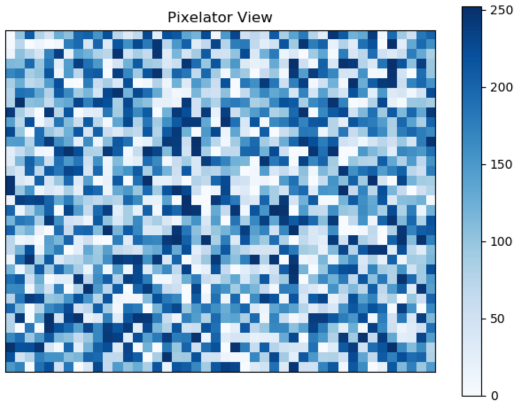


FIGURE 8. Differences in pixel of 1st and 2nd Program using Pixelator view.

TABLE 2. Classification probability of program 8 for different classes: compute, memory & mixed.

Class Name	$Probability$
Compute	**0.5661**
Memory	0.3085
Mixed	0.1253

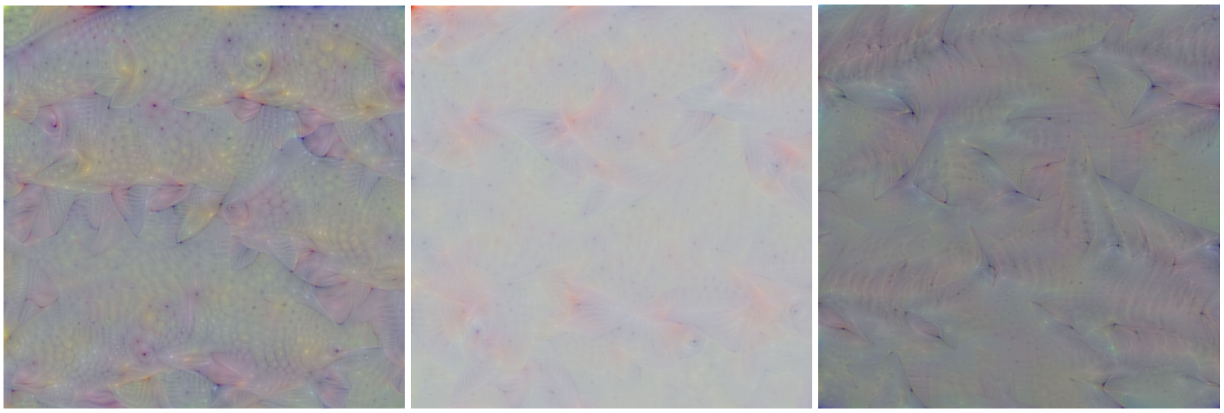


FIGURE 9. Deep Dream Images of three different types of program source-codes: Compute, Memory & Mixed.

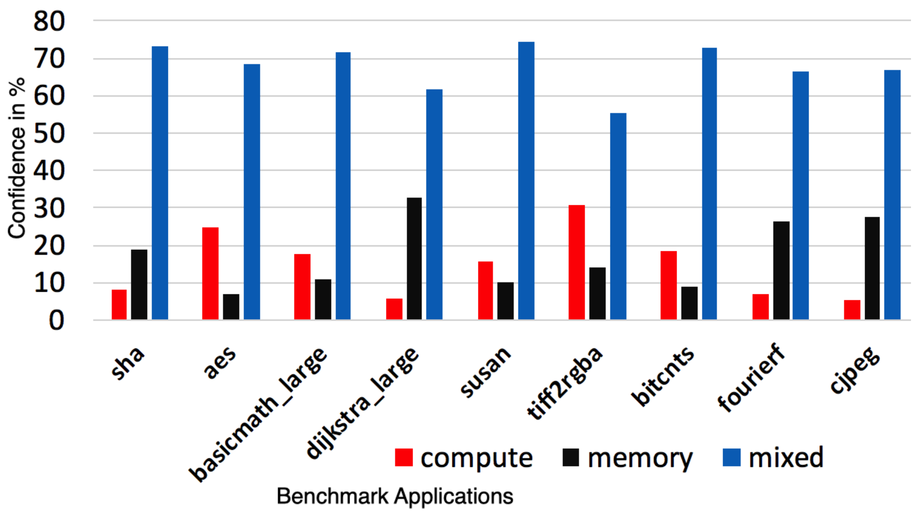


FIGURE 10. Classification of MiBench [33] benchmark suits (Benchmark vs Confidence in % for a specific class).

TABLE 3. Classification probability of program 9 for different classes: compute, memory & mixed.

Class Name	*Probability*
Compute	0.0774
Memory	**0.8236**
Mixed	0.0990

TABLE 4. Classification probability of program 10 for different classes: compute, memory & mixed.

Class Name	*Probability*
Compute	0.0071
Memory	0.1339
Mixed	**0.8590**

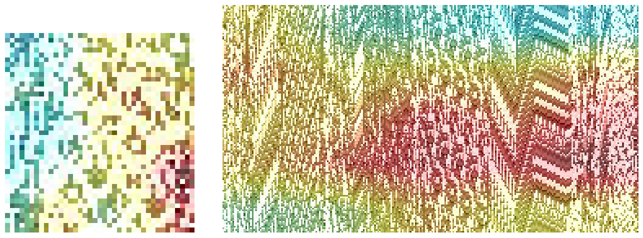


FIGURE 11. Grad-Cam visualization.

MRPI [28] methodology for cross-validation. In [28] workloads are classified based on **M**emory **R**eads **P**er **I**nstruction metric ($MRPI = \frac{\text{L2 cache read refills}}{\text{Instructions retired}}$). The workload is quantified by MRPI, where high value of MRPI signifies low workload on the processing core and vice-versa. For Program 8, 9 & 10 MRPI values were 0.018, 0.031 and 0.028 on average respectively, proving the correctness of workload classified by our trained CNN model. Therefore, for our chosen programs (Program 8, 9 & 10) we could notice that the CNN classifier is able to predict the label for each program source-code with high probability. Fig. 10 shows the classification (confidence in percentage) of some of the chosen popular benchmarks from MiBench benchmark suits.

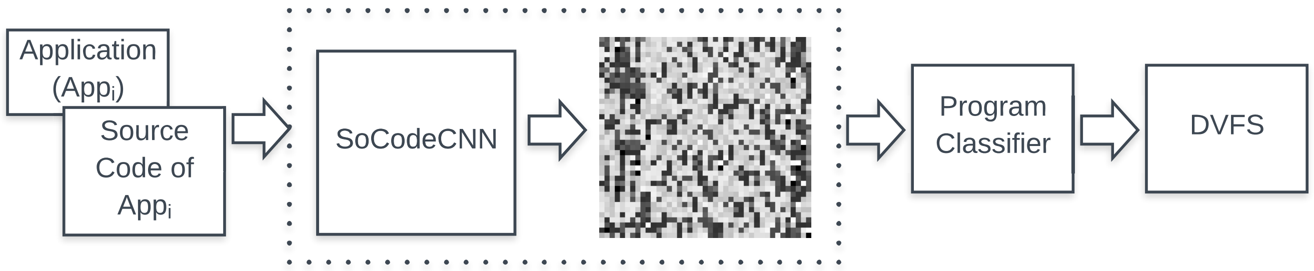


FIGURE 12. Block diagram of power management agent, APM, using SoCodeCNN.

From the aforementioned experiments we noticed that compared to conventional methodologies [1], [3], where skilled human is required to extract features from program source-code to determine whether a program is compute intensive or memory intensive or mixed, our approach is able to avoid such manual feature extraction and still able to classify programs into their corresponding classes accurately in an automated manner using visual based CNN algorithm.

C. WHERE THE CNN IS LOOKING

In order to verify whether the VGG19 CNN of *Exp. 3* is extracting the correct features to be able to predict the type of the application (program) we utilized Grad-Cam [36] to visualize which area of the program-source code image is the model focusing on to make a decision (predict). In Grad-Cam the visual-explanation of decision made by the CNN model is provided by using the gradient information flowing into the last convolutional layer of the CNN model to understand the importance of each neuron for a decision made.

When we utilized Grad-Cam for classification of Program 10 and SHA benchmark application of MiBench by CNN we got Fig. 11.a and Fig. 11.b respectively to notice which regions are highlighted, reflecting the regions focused by the CNN to make the prediction decision. In Fig. 11.a and Fig. 11.b the regions highligted as red are the most important feature-extraction regions by the CNN, whereas the yellow regions are less significant and the blue ones are the least significant regions influencing the prediction decision.

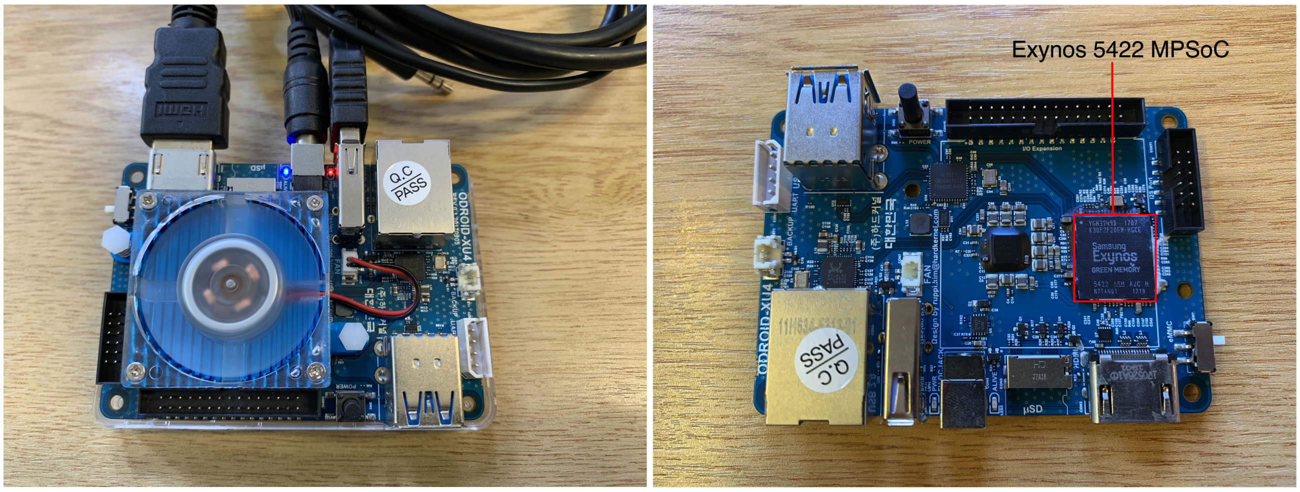
When we referred back the red-highlighted regions of Fig. 11.a and Fig. 11.b we noticed that the CNN is focusing on the code for separate thread executions of Program 10, and parts of the functions named *sha_transform* and *sha_final* of SHA benchmark of MiBench. Upon inspecting Grad-Cam visualization of Program 10 and SHA benchmark it re-instated our confidence in the performance of the CNN in order to make a prediction decision since the aforementioned code regions in those programs are actually the important code-regions which are required to deduce the type of the application.

VI. EXEMPLAR APPLICATION OF SOCODECNN

To prove the efficacy of utilizing SoCodeCNN we use the CNN model from the *Exp. 3* mentioned in Section V-A to develop an automated power management agent, which uses the CNN model to decide the operating frequency of the processing elements (CPU) based on the type of the program being executed on the computing system. The procedure of reducing dynamic power consumption ($P \propto V^2 f$) by reducing the operating frequency of the processing elements is known as *dynamic voltage frequency scaling* (*DVFS*) [37], [38]. Since dynamic power is proportional to the operating frequency of the processing elements as shown in the aforementioned equation, executing the application on a reduced operating frequency leads to a reduced power consumption of the system. In order to cater for performance and reduced power consumption several resource mapping and partitioning mechanisms using DVFS [37]–[42] has been proposed. To implement the automated power management agent we chose Odroid XU4 [43] development board (see Fig. 13), which employs the Samsung Exynos 5422 [22] multiprocessor system-on-a-chip (MPSoC) platform. The Exynos 5422 MPSoC is used in several modern Samsung smart-phones and phablets including Samsung Galaxy Note and S series devices.

A. HARDWARE AND SOFTWARE INFRASTRUCTURE

Nowadays heterogeneous MPSoCs consist of different types of cores, either having the same or different instruction set architecture (ISA). Moreover, the number of cores of each type of ISA can vary based on MPSoCs and are usually clustered if the types of cores are similar. For this research, we have chosen an Asymmetric Multicore Processors (AMPs) system-on-chip (AMPSoC), which is a special case of heterogeneous MPSoC and has clustered cores on the system. Our study was pursued on the Odroid XU4 board [43], which employs the Samsung Exynos 5422 [22] MPSoC (as shown in Fig.13.b). Exynos 5422 is based on ARM's big.LITTLE technology [44] and contains cluster of 4 ARM Cortex-A15 (big) CPU cores and another of 4 ARM Cortex-A7 (LITTLE) CPU cores, where each core implements the ARM v7A ISA. This MPSoC provides dynamic voltage frequency scaling feature per cluster, where the big core cluster has 19 frequency scaling levels, ranging from 200 MHz to 2000 MHz with each step of 100 MHz and the LITTLE cluster has 13 frequency scaling levels, ranging from 200 MHz to 1400 MHz, with each step of 100 MHz. Additionally, each core on the cluster has a private L1 instruction and data cache, and a L2 cache, which is shared across all the cores within a cluster.



(a) Odroid XU4 in action (b) Exynos 5422 MPSoC on Odroid XU4 development board

FIGURE 13. Odroid XU4 development board and Exynos 5422 MPSoC.

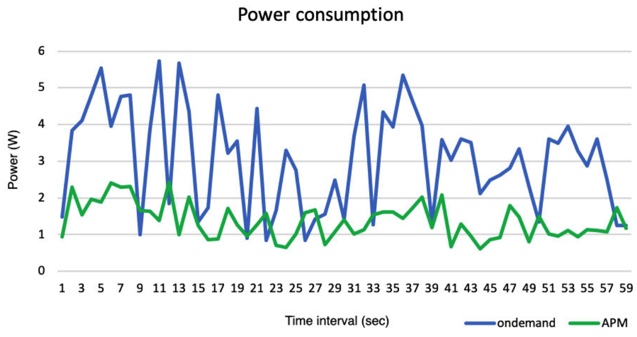


FIGURE 14. Power consumption of executing Program 9 on ondemand vs APM.

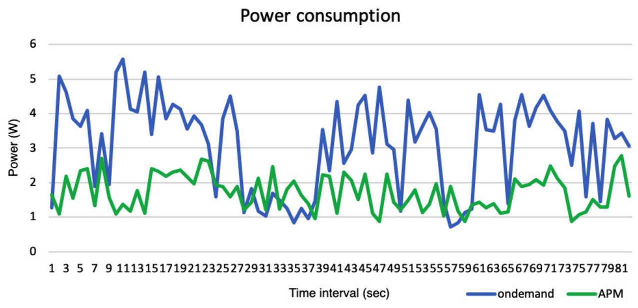


FIGURE 15. Power consumption of executing Program 10 on ondemand vs APM.

Since Odroid XU4 board does not have an internal power sensor onboard, hence an external power monitor [45] with networking capabilities over WIFI is used to take power consumption readings. Although the ARM Cortex-A7 (LITTLE) CPU cores on Odroid XU4 do not have temperature sensor but our intelligent power management agent approach is scalable and works for heterogeneous cluster cores. We have run all our experiments on UbuntuMate version 14.04 (Linux Odroid Kernel: 3.10.105) on the Odroid XU4.

B. DVFS USING SOCODECNN IN MPSOCS

Fig. 12 shows the block diagram of the implementation of the automated power management agent. When an instance of an application (App_i) is executed, the program source code of the application is fed to the *SoCodeCNN* to create the image representing the platform-independent IR code of App_i, which will be used by the CNN model (called as "*Program Classifier*" in Fig. 12) for classification purpose. If App_i has been executed before on the platform then the image representation created by *SoCodeCNN* during its first execution is already saved on the memory and used only for classification purpose for future executions. The *Program Classifier* will classify based on what type of application is being executed at the moment such as App_i is of compute intensive or memory intensive or mixed load, and *DVFS* module is used to set the operating frequency of the CPUs of the Odroid as required by the type of executing application.

We refer to our automated power management agent as *APM*. In our APM implementation we specify if an application is compute intensive then the operating frequency of the big CPU cluster of the Odroid should be set to 2000 MHz and the LITTLE cluster's frequency to 1400 MHz, whereas if the application is memory intensive then the operating frequency of the big cluster to be set to 900 MHz and the LITTLE cluster's frequency to 600 MHz. If the executing application is of mixed workload then the operating frequency of the big cluster to be set to 1300 MHz and the LITTLE cluster's frequency to 1100 MHz. Through our experiments we have found that if an application is memory intensive or mixed load then most of the time running the CPUs at high frequency only wastes energy while not utilizing the maximum cycles per second capacity of the CPUs. Hence, we chose the associated operating frequencies as mentioned earlier through several experimentations. In the next sub-section we show the power consumption difference between execution of Program 9 and Program 10 on UbuntuMate's (Linux) ondemand governor and on our APM implementation in a graphical representation. We also evaluate the difference

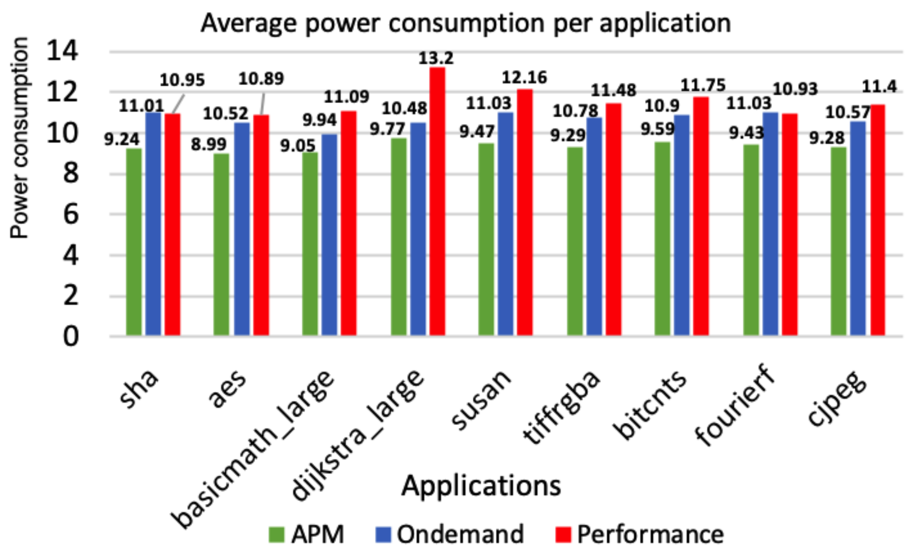


FIGURE 16. **Average power consumption of different benchmark programs on ondemand vs APM vs performance.**

in terms of power consumption while executing several benchmark applications of MiBench using Linux's ondemand, performance and APM.

C. RESULTS

Fig. 14 and Fig. 15 show the power consumption over time of execution of Program 9 and Program 10 respectively while executing on Linux's ondemand governor and on our APM. In the figures, the Y axis is denoted by power consumption in watts (W) vs time interval in seconds. In Fig. 14, using APM we are able to save 49.52% of power on average over the time period (APM power consumption: 1.372 W vs ondemand power consumption: 2.718 W). Using APM we only sacrificed 1.8 secs of execution time compared to ondemand's execution time of 58.2 secs while achieving 49.52% more power consumption reduction. In Fig. 15, using APM we are able to save 43.48% of power on average (APM power consumption: 1.716 W vs ondemand power consumption: 3.036 W). Using APM we only sacrificed 3.1 secs of execution time compared to ondemand's execution time of 80.4 secs while achieving 43.48% more power consumption reduction.

When we evaluated the power consumption of executing several benchmark applications of MiBench using ondemand, performance governors and APM, we noticed that APM is able to achieve more than 10% power saving on average over the time period compared to ondemand and performance while sacrificing only less than 3% of performance on average in terms of execution time. Fig. **??** shows the average power consumption of different benchmarks while using ondemand, performance and APM. In Fig. **??** the X-axis denotes the name of the benchmark and the Y-axis denotes the average power consumption.

It should also be noted that when a new application is executed on the platform, the average time taken to create the visual image from the source-code of the application using *SoCodeCNN* is less than 2 seconds (depending on the size of the program). Image creation is only performed once if the new application is executed for the first time using APM, otherwise, the inference of the image for classification and setting the operating frequency appropriately takes less than 150 milliseconds.

D. ADVANTAGE OF SOCODECNN BASED DVFS

In the methodology proposed by Taylor et al [1], the authors define a machine learning based system which selects the appropriate processing elements and the desired operating frequency of the processing elements by extracting the features from the program source code and then evaluating the feature values by a predictor. However, the features from the program source code has to be manually selected by skilled people having experience with the programming language framework. In another study by Cummins et al. [2], the authors utilize similar feature extraction methodology from source code to be fed to a DNN model to make decisions and this approach also requires the intervention of a skilled person to perform the manual feature extraction. On the other hand, studies [28], [46], [47] which include hybrid scheduling/resource mapping where the methodology is partly dependent on offline and online training of the executing application to decide the appropriate processing elements and their operating frequencies, also has its own limitations. In case a new application is being executed on the device, we need to perform an offline training on this new application in order to achieve an improvement on the main objective of scheduling/resource mapping to optimize performance, energy efficiency, etc.

From the exemplar application of APM using *SoCodeCNN* to use DVFS of the processing elements we could notice that we do not require a skilled person to extract features manually from the source code to be fed to the software agent to decide the operating frequency of the system. At the same time in case a new application is installed and executed on the system then the APM is capable of classifying the application using *SoCodeCNN*'s image conversion methodology and trained CNN model, and then appropriately deciding the operating frequency of the processing elements based on the type of application being executed. The most advantage of utilizing *SoCodeCNN* is that we can design power and thermal management agents which are automated in nature with an overhead of at most 150 ms during classification and setting the operating frequency.

VII. SOCODECNN: A FUTURE DIRECTION

Usually to train Deep Convolutional Neural Networks (DCNNs) we require a huge dataset consisting of thousands to millions of data (image) or else there might be an issue of overfitting by the trained DCNN model, which in turn could lead to inaccurate predictions. However, from the experimental results, the trained DCNN models using *SoCodeCNN*'s code to image conversion methodology led to accurate results in all the predictions.

Given the efficacy of this proposed approach, we would expect more researchers utilizing this concept to understand, study and learn from the visual patterns of images of the source-codes of applications and continue contributing to building a large dataset of such code repositories so that it could be used to train DCNN models in the future.

VIII. CONCLUSION

In this paper, we propose *SoCodeCNN* (Program **So**urce **Code** for visual **CNN** classification) capable of converting program source-codes to visual images such that they could be utilized for classification by visual CNN based algorithm. Experimental results also show that using *SoCodeCNN* we could classify the benchmarks from PARSEC, SPLASH-2, and MiBench in a completely automated manner and with high prediction accuracy for our chosen test cases. We also demonstrate with an example the use of *SoCodeCNN* for DVFS in MPSoC.

IX. CODE AVAILABILITY

The source-code for *SoCodeCNN* is available from https://github.com/somdipdey/SoCodeCNN, and that of *Pixelator* is available from https://github.com/somdipdey/Pixelator-View.

REFERENCES

[1] B. Taylor, V. S. Marco, and Z. Wang, "Adaptive optimization for OpenCL programs on embedded heterogeneous systems," *ACM SIGPLAN Notices*, vol. 52, no. 5, pp. 11–20, 2017.

[2] C. Cummins, P. Petoumenos, Z. Wang, and H. Leather, "End-to-end deep learning of optimization heuristics," in *Proc. IEEE 26th Int. Conf. Parallel Archit. Compilation Techn. (PACT)*, Sep. 2017, pp. 219–232.

[3] M. Allamanis, E. T. Barr, P. Devanbu, and C. Sutton, "A survey of machine learning for big code and naturalness," *ACM Comput. Surv.*, vol. 51, no. 4, p. 81, Jul. 2018.

[4] A. H. Ashouri, W. Killian, J. Cavazos, G. Palermo, and C. Silvano, "A survey on compiler autotuning using machine learning," 2018, *arXiv:1801.04405*. [Online]. Available: https://arxiv.org/abs/1801.04405

[5] C. A. Lattner, "LLVM: An infrastructure for multi-stage optimization," Ph.D. dissertation, Dept. Comput. Sci., Univ. Illinois at Urbana–Champaign, Champaign, IL, USA, 2002.

[6] Y. Ko, B. Burgstaller, and B. Scholz, "LaminarIR: Compile-time queues for structured streams," *ACM SIGPLAN Notices*, vol. 50, no. 6, pp. 121–130, 2015.

[7] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," 2014, *arXiv:1409.1556*. [Online]. Available: https://arxiv.org/abs/1409.1556

[8] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "ImageNet classification with deep convolutional neural networks," in *Proc. Adv. Neural Inf. Process. Syst.*, 2012, pp. 1097–1105.

[9] O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, and M. Bernstein, "ImageNet large scale visual recognition challenge," *Int. J. Comput. Vis.*, vol. 115, no. 3, pp. 211–252, Dec. 2015.

[10] Z. Wang and A. C. Bovik, "A universal image quality index," *IEEE Signal Process. Lett.*, vol. 9, no. 3, pp. 81–84, Mar. 2002.

[11] D. Marr, *Vision—A Computational Investigation into the Human Representation and Processing of Visual Information*. Cambridge, MA, USA: MIT Press, 1982.

[12] P. Messaris, *Visual Literacy: Image, Mind, and Reality*. Boulder, CO, USA: Westview Press, 1994.

[13] P. Dallow and J. Elkins, "The visual complex: Mapping some interdisciplinary dimensions of visual literacy," in *Visual Literacy*. Evanston, IL, USA: Routledge, 2009, pp. 99–112.

[14] M. R. Dillon and E. S. Spelke, "Young children's use of surface and object information in drawings of everyday scenes," *Child Develop.*, vol. 88, no. 5, pp. 1701–1715, 2017.

[15] M. Ahissar, M. Nahum, I. Nelken, and S. Hochstein, "Reverse hierarchies and sensory learning," *Philos. Trans. Roy. Soc. London B, Biol. Sci.*, vol. 364, no. 1515, pp. 285–299, 2009.

[16] P. Wiriyathammabhum, D. Summers-Stay, C. Fermüller, and Y. Aloimonos, "Computer vision and natural language processing: Recent approaches in multimedia and robotics," *ACM Comput. Surv.*, vol. 49, no. 4, 2017, Art. no. 71.

[17] I. Goodfellow, Y. Bengio, A. Courville, and Y. Bengio, *Deep Learning*, vol. 1. Cambridge, MA, USA: MIT Press, 2016.

[18] L. Jancová, "Translation and the role of the mother tongue in ELT," Palacký Univ. Olomouc, Olomouc, Czech Republic, Tech. Rep., 2010.

[19] J. Krajka, "Your mother tongue does matter! Translation in the classroom and on the Web," *Teach. English Technol.*, vol. 4, no. 4, 2004.

[20] J. L. Shinskey and L. J. Jachens, "Picturing objects in infancy," *Child Develop.*, vol. 85, no. 5, pp. 1813–1820, 2014.

[21] R. A. Thompson, "Development in the first years of life," *Future Children*, vol. 11, no. 1, pp. 20–33, 2001.

[22] *Exynos 5 Octa (5422)*. Accessed: Jul. 23, 2018. [Online]. Available: https://www.samsung.com/exynos

[23] *The Open Standard for Parallel Programming of Heterogeneous Systems*. Accessed: Jul. 23, 2018. [Online]. Available: https://www.khronos.org/opencl/

[24] M. Namolaru, A. Cohen, G. Fursin, A. Zaks, and A. Freund, "Practical aggregation of semantical program properties for machine learning based optimization," in *Proc. ACM Int. Conf. Compilers, Archit. Synthesis Embedded Syst.*, 2010, pp. 197–206.

[25] H. Leather, E. Bonilla, and M. F. P. O'Boyle, "Automatic feature generation for machine learning–based optimising compilation," *ACM Trans. Archit. Code Optim.*, vol. 11, no. 1, 2014, Art. no. 14.

[26] J.-B. Martens and L. Meesters, "Image dissimilarity," *Signal Process.*, vol. 70, no. 3, pp. 155–176, 1998.

[27] Z. Wang, A. C. Bovik, H. R. Sheikh, and E. P. Simoncelli, "Image quality assessment: From error visibility to structural similarity," *IEEE Trans. Image Process.*, vol. 13, no. 4, pp. 600–612, Apr. 2004.

[28] A. K. Singh, M. Shafique, A. Kumar, and J. Henkel, "Mapping on multi/many-core systems: Survey of current and emerging trends," in *Proc. IEEE 50th ACM/EDAC/IEEE Design Autom. Conf. (DAC)*, May/Jun. 2013, pp. 1–10.

[29] B. K. Reddy, A. K. Singh, D. Biswas, G. V. Merrett, and B. Al-Hashimi, "Inter-cluster thread-to-core mapping and DVFS on heterogeneous multicores," *IEEE Trans. Multiscale Comput. Syst.*, vol. 4, no. 3, pp. 369–382, Jul./Sep. 2018.

[30] C. Bienia, "Benchmarking modern multiprocessors," Ph.D. dissertation, Dept. Comput. Sci., Princeton Univ., Princeton, NJ, USA, Jan. 2011.

[31] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich, "Going deeper with convolutions," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit.*, Jun. 2015, pp. 1–9.

[32] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta, "The SPLASH-2 programs: Characterization and methodological considerations," *ACM SIGARCH Comput. Archit. News*, vol. 23, no. 2, pp. 24–36, 1995.
[33] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown, "MiBench: A free, commercially representative embedded benchmark suite," in *Proc. IEEE Int. Workshop Workload Characterization (WWC)*, Dec. 2001, pp. 3–14.
[34] C. Bienia, S. Kumar, and K. Li, "PARSEC vs. SPLASH-2: A quantitative comparison of two multithreaded benchmark suites on chip-multiprocessors," in *Proc. IEEE Int. Symp. Workload Characterization (IISWC)*, Sep. 2008, pp. 47–56.
[35] S. J. Pan and Q. Yang, "A survey on transfer learning," *IEEE Trans. Knowl. Data Eng.*, vol. 22, no. 10, pp. 1345–1359, Oct. 2010.
[36] R. R. Selvaraju, M. Cogswell, A. Das, R. Vedantam, D. Parikh, and D. Batra, "Grad-CAM: Visual explanations from deep networks via gradient-based localization," in *Proc. IEEE Int. Conf. Comput. Vis.*, Oct. 2017, pp. 618–626.
[37] A. Aalsaud, R. Shafik, A. Rafiev, F. Xia, S. Yang, and A. Yakovlev, "Power–aware performance adaptation of concurrent applications in heterogeneous many-core systems," in *Proc. ACM Int. Symp. Low Power Electron. Design*, 2016, pp. 368–373.
[38] A. K. Singh, C. Leech, K. R. Basireddy, B. M. Al-Hashimi, and G. V. Merrett, "Learning-based run-time power and energy management of multi/many-core systems: Current and future trends," *J. Low Power Electron.*, vol. 13, no. 3, pp. 310–325, Sep. 2017.
[39] K. Chandramohan and M. F. P. O'Boyle, "Partitioning data-parallel programs for heterogeneous MPSoCs: Time and energy design space exploration," *ACM SIGPLAN Notices*, vol. 49, no. 5, pp. 73–82, 2014.
[40] R. Barik, N. Farooqui, B. T. Lewis, C. Hu, and T. Shpeisman, "A black-box approach to energy-aware scheduling on integrated CPU-GPU systems," in *Proc. ACM Int. Symp. Code Gener. Optim.*, 2016, pp. 70–81.
[41] A. K. Singh, A. Prakash, K. R. Basireddy, G. V. Merrett, and B. M. Al-Hashimi, "Energy-efficient run-time mapping and thread partitioning of concurrent OpenCL applications on CPU-GPU MPSoCs," *ACM Trans. Embedded Comput. Syst.*, vol. 16, no. 5s, p. 147, 2017.
[42] A. K. Singh, P. Dziurzanski, H. R. Mendis, and L. S. Indrusiak, "A survey and comparative study of hard and soft real-time dynamic resource allocation strategies for multi-/many-core systems," *ACM Comput. Surv.*, vol. 50, no. 2, 2017, Art. no. 24.
[43] *ODROID-XU4*. Accessed: Jul. 23, 2018. [Online]. Available: https://goo.gl/KmHZRG
[44] *ARM Big.LITTLE Technology*. Accessed: Jul. 23, 2018. [Online]. Available: http://www.arm.com/
[45] *Odroid SmartPower2*. Accessed: Jul. 23, 2018. [Online]. Available: https://www.odroid.co.uk/odroid-smart-power-2
[46] B. K. Reddy, G. V. Merrett, B. M. Al-Hashimi, and A. K. Singh, "Online concurrent workload classification for multi-core energy management," in *Proc. Design, Autom. Test Eur. Conf. Exhibit. (DATE)*, 2018, pp. 621–624.
[47] E. W. Wachter, G. V. Merrett, B. M. Al-Hashimi, and A. K. Singh, "Reliable mapping and partitioning of performance-constrained OpenCL applications on CPU-GPU MPSoCs," in *Proc. 15th IEEE/ACM Symp. Embedded Syst. Real-Time Multimedia*, Oct. 2017, pp. 78–83.

SOMDIP DEY was born in Kolkata, India, in 1990. He received the B.Sc. degree (Hons.) in computer science from St. Xavier's College (Autonomous), Kolkata, India, in 2012, and the M.Sc. degree in advanced computer science, with specialization in computer systems engineering, from The University of Manchester, U.K., in 2014. He has more than 10 years of industry experience, working on developing technologies, including working for Microsoft and Samsung Electronics. He is currently an Artificial Intelligence Scientist working on embedded systems at the University of Essex, U.K., and a Serial Entrepreneur, with a focus on social impact. His current research interests include affordable artificial intelligence, information security, computer systems engineering and computing resource optimization for performance, energy, temperature, reliability, and security in mobile platforms. He has also served as a Reviewer and TPC Member for several top conferences such as DATE, DAC, AAAI, CVPR, ICCV, IEEE EdgeCom, IEEE CSCloud, and IEEE CSE.

AMIT KUMAR SINGH (M'09) received the B.Tech. degree in electronics engineering from the Indian Institute of Technology (Indian School of Mines), Dhanbad, India, in 2006, and the Ph.D. degree from the School of Computer Engineering, Nanyang Technological University (NTU), Singapore, in 2013. He was with HCL Technologies, India, for a year and half, until 2008. He has a post-doctoral research experience for over five years at several reputed universities. He is currently a Lecturer with the University of Essex, U.K. His current research interests include system-level design-time and runtime optimizations of 2D and 3D multicore systems for performance, energy, temperature, reliability, and security. He has published over 80 papers in reputed journals/conferences, and received several Best Paper Awards, including the ICCES 2017, ISORC 2016, and PDP 2015. He has served on the TPC of the prestigious IEEE/ACM conferences DAC, DATE, CASES, and CODES+ISSS.

DILIP KUMAR PRASAD received the B.Tech. and Ph.D. degrees in computer science and engineering from the Indian Institute of Technology (ISM), Dhanbad, India, and Nanyang Technological University, Singapore, in 2003 and 2013, respectively. He is currently an Associate Professor at UiT The Arctic University of Norway. His current research interests include image processing, machine learning, and computer vision.

KLAUS DIETER MCDONALD-MAIER (S'91–SM'06) is currently the Head of the Embedded and Intelligent Systems Laboratory, University of Essex, Colchester, U.K. He is also the Chief Scientist of UltraSoC Technologies Ltd., the CEO of Metrarc Ltd., and a Visiting Professor with the University of Kent. His current research interests include embedded systems and system-on-chip design, security, development support and technology, parallel and energy-efficient architectures, computer vision, data analytics, and the application of soft computing and image processing techniques for real-world problems. He is a member of the VDE and a Fellow of the IET.

• • •