Towards using of ASIPs for Approximate Computing

Daniel Moya Sánchez
Computer Engineering Academic Area
Instituto Tecnológico de Costa Rica
Email: danielmscr1994@gmail.com

***Abstract*—IT systems have been facing certain problems, among them the cost in area, power and execution time, which restrict the performance of a chip. Application-Specific Instruction Processors (ASIPs) are used to propose a solution towards the use of the approximate computing area, which is a design paradigm that proposes a reduction in the accuracy or precision of the computation to obtain opportunities for improvement in terms of area, power and execution time. This paper evaluates the design of Application-Specific Instruction Processors (ASIPs) for error-tolerant applications in three specific applications, through the use of ASIPMeister and Dlxsim tools, which allow the synthesis required for a processor's hardware with its Instruction Set Architecture (ISA). Speedups in the total cycles were achieved from 1.08X to 1.91X, while adding minimal area (1% at most) and power (maximun of 5 mW) requirements.**

I. Introduction

Information Technology (IT) systems seek to give a better quality of life to people. In this task, these systems have been facing certain challenges, among them the cost in area, power, and execution time, which restrict the performance of a chip. Ideally, an application should fit the real needs of the user and, in general, of the area of application, so that an optimal use of resources is achieved. Currently, processor design is not only focused on having more performance but also an appropriate resource management; however, some challenges in this field are due to physical limitations, for instance:

- electrical characteristics of the CMOS transistors, which restrict the energy consumption in embedded systems and which is an issue that designers should consider for specific purpose components in processors;
- memory wall, which corresponds to the difference between the growth of processing capacity against the speed of data gathering from memory;
- and utilization wall, which limits the maximum use of hardware simultaneously due to the heat dissipation capability of a system.

In order to face the problems mentioned above, a current research area corresponds to approximate computing. This is a novel design paradigm that proposes a reduction in the accuracy or precision of computations to obtain opportunities for improvement in area, power and execution time. To apply this paradigm, it is necessary to identify error-tolerant applications and determine, in a specific manner, which sections or functions within these can be replaced by approximate versions, so that a balance can be obtained between the output quality and the general computing resources required. For a given error-tolerant system, the framework on Figure 1 can be applied to include the approximate computing paradigm [1].

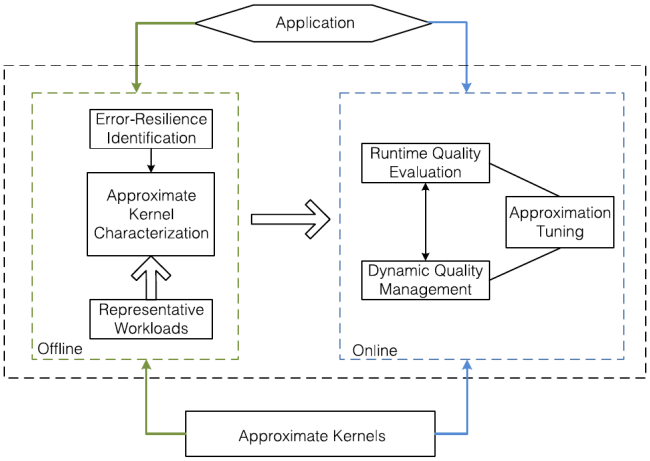

Figure 1: An existing approximate computing framework. Taken from [1]

The key elements of Figure 1 consist of approximate kernels, which are the implementation (techniques) of the approximate functions, these could be done at a hardware layer or at a software layer; the identification of the error-tolerant parts and its specific details (e.g. impact analysis); and the quality management which implies a continuos evaluation to determine if the application meets the desired requirements [1].

One way to implement approximate computing is through Application-Specific Instruction Processors (ASIPs). An ASIP is a processor that uses an application-specific instruction set, this means that, although it can execute a wide range of applications, it is optimized for a specific one, in which the ASIP can execute with improved performance (for instance, energy consumption or execution time would be lower) compared to a General Purpose Proccesor (GPP). With the

use of ASIPs, instructions and even functions where there is error tolerance can be implemented as special approximated instructions, that reduce the resource consumption while keeping the error from the approximation under an acceptable threshold. Although Application Specific Integrated Circuits (ASICs) show better performance results, ASIPs possess more flexibility. Optimizations for an ASIP can be seen in different forms, including [2]:

- Instruction extension: Customized instructions can be made to extend the base Instruction Set Architecture (ISA).
- Inclusion or exclusion of predefined blocks: Not only specific software can be added to extend an architecture but also customized hardware in the form of specialized blocks; also, regular blocks not used can be excluded.
- Parameterization: Certain variables, such as cache sizes or number of registers, can be customized to adjust for a specific application.

ASICs represent a hardware solution to a problem which is very limited and have high costs and a high time-to-market, but achieve the greatest performance. Contrary, GPPs are seen as a software solution which are very flexible but they are the least efficient. ASIPs are in the middle of these two as they balance flexibility and performance to have a good trade-off between those variables.

Contribution: This work evaluates the design performance, in area, power and execution time, of ASIPs with special instructions for error-tolerant applications. For this purpose, three different error-tolerant applications were evaluated and then, for each one, an special instruction, that reflects a recurrent operation in the original code, was implemented. Finally, the performance of each optimization was evaluated against the original version to determine the impact of the designed ASIPs.

Since translating a whole application from the C programming language to assembly code was too costly, only a smaller version which contains the key processing was developed for testing purposes. The ASIPs developed do not use explicitly approximate hardware, this is considered the main future work for this paper.

II. Context and Background

The relationship between GPPs, ASIPs, and ASICs is shown in Figure 2. Approximate computing techniques can also be implemented on GPPs, due to their extremely flexible nature. However, because they are designed for any kind of computation, it falls on the programmers, or compilers, not on specialized hardware modules, to make the performance of software running on these systems as high as possible. On the other hand, since ASICs are a pure hardware solution, approximate hardware modules would be difficult to manage in terms of quality evaluation, this could cause high cost of development, since the hardware would not be

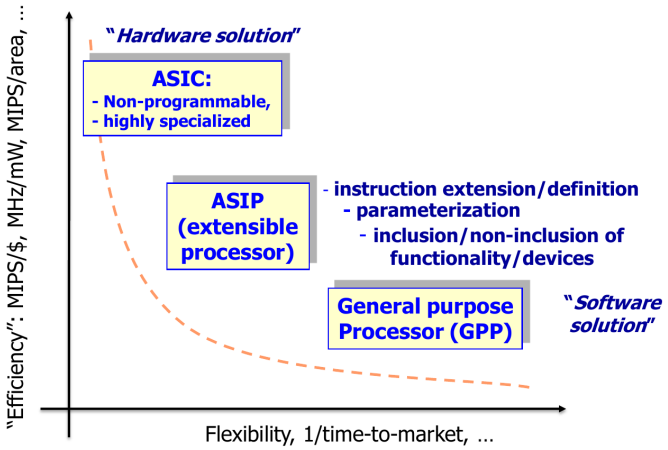

Figure 2: Comparison between GPPs, ASIPs, and ASICs. Taken from [3]

programmable. ASIPs can adjust to specific requirements of a given application (through extended instructions) so that a better balance of cost savings and amount of error is achieved. This project focuses on that goal; to design ASIPs for a set of error-tolerant applications.

Some of the research efforts in the approximate computing area focus on software solutions, for example, in [4] the loop perforation technique is explained, which uses two particular methods, criticality testing (filtering of critical loops) and perforation space exploration (determining the best performance for a specific accuracy loss bound). A higher level solution is mentioned in [5], where a scheduling framework for meeting the performance and thermal design power is proposed. Several versions of a task are used by the scheduler to provide tradeoffs between different levels of quality of service and performance. A neural network based solution is detailed in [6], where a learning model is trained to identify how an approximable region of code behaves for different approximate versions.

On the other hand, hardware solutions have also been proposed; from approximate adders and multipliers to the design of ASICs. For example, in [7] a methodology for the automation of creating approximate circuits is proposed. The general idea involves using the original circuit as an input and a quality function to adjust and verify an approximate circuit so the quality constraint is met for a given hardware configuration. A similar approach is taken in [8], where contrary to the previous proposal, the approximate circuits are sequential, which extends the same purpose for more possible approximate circuits. Finally, in [9], a more global approach is taken, where the approximate circuits are generated using behavioral-level descriptions (and not RTL like the previous proposals); from there, an Abstract Synthesis Tree (AST) is created, which is later analyzed and modified to include the approximate circuit logic, so that, as the other methods, the

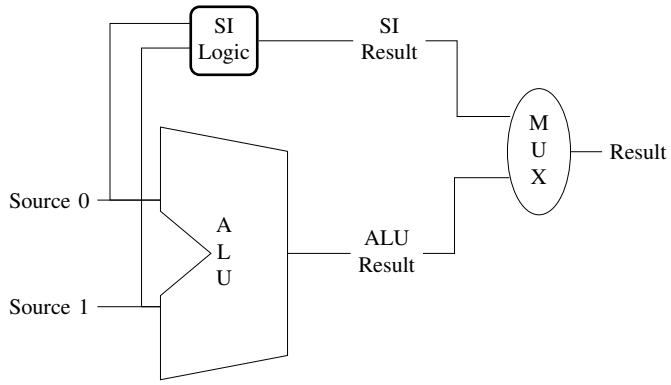

Figure 3: ASIP Hardware implentation.

circuit is tested and its performance evaluated.

III. Design of the ASIPs

As explained before, an ASIP is able to execute a wide range of operations but also some specialized instructions for an specific application. Following this scheme, a general processor with common instructions (for example add, substract, shift, among others) was used within the ASIPMeister tool, and one specific instruction was added for each different selected application. This is showed in Figure 3, where, for certain operations (controlled by the processor signals) the ALU is ignored and instead the result from the special additional hardware is used (Special Instruction, SI).

The three aproximate applications found were K-Means, K-Nearest Neighbors (KNN) and the Sobel filter. All the special instructions correspond to new assembly instructions of arithmetic tpye, which modify the Instruction Set Architecture (ISA) using the Dlxsim tool. For the K-Means application, the special instruction *eucl* was implemented, which executes part of the euclidean distance operation, as follows (considering *rd* as the destination register, *rs0* and *rs1* as the registers with the input values):

$$rd = (rs0 - rs1)^2 \tag{1}$$

At a hardware level, this instruction uses a combinatorial block (the ALU is not used), consisting in an adder (which executes a substraction when the second input is negated) and a multiplier. At a software level, the *eucl* instruction is used in the inner loop as the main operation in the calculation of the array index of the nearest cluster center, as showed in the Algorithm 1.

For the KNN application, the special instruction *absv* was implemented, which executes a subtract operation with an absolute result, which is used frequently in the KNN algorithm (the euclidean distance remains the main operation in this algorithm too). At at high abstraction level, this operation executes:

input : Clusters, Number of clusters and coordinates
output: Array index of nearest cluster center

$min\_dist \leftarrow$
 eucl_distance$(num_dims, coord1[0], coord2[0])$;
for $j \leftarrow 1$ **to** *number of clusters* **do**
  for $i \leftarrow 1$ **to** *of number of coordinates* **do**
    $dist \mathrel{+}= eucl(coord1[i], coord2[i])$;
  end
  if $dist < min\_dist$ **then**
    $min\_dist \leftarrow dist$
    $index \leftarrow i$
  end
end
Return i

Algorithm 1: K-Means algorithm extract

$$rd = rs0 > rs1\,?\,rs0 - rs1 : rs0 - rs1 \tag{2}$$

At a hardware level, a simple adder (which executes a substraction) and a mux are used, in an additional combinatorial block. At a software level, the *absv* instruction is used to compare the resulting estimate vs the actual prediction given by the user, where if the given prediction is within the 20% of the resulting estimate, the prediction is considered correct, otherwise wrong, as showed in the Algorithm 2.

input : Prediction attribute and k value
output: Number of correct predictions

for $j \leftarrow 1$ **to** *number of samples* **do**
  $N =$ *number of validated samples*;
  $actual \leftarrow sample[N].Attributes[Predict\_attr]$;
  $result \leftarrow$ KNN(N, k_value);
  if $absv(actual, result)/actual < 0.2$ **then**
    $correct\_count++$
  end
end
Return $correct\_count$

Algorithm 2: KNN algorithm extract

For the last application, the Sobel filter, the special instruction *sob* was developed, which allows computing the following operation in a single cycle:

$$rd = rs0^2 + rs1^2 \tag{3}$$

The operation described in (3) allows the execution of two multiplications and an addition operation, which is used frequently in the Sobel algorithm. Other operations in this algorithm could not be turned into special instructions (for example, a matrix multiplication) because they require more than two paremeters.

At a hardware level, this instruction requires two multipliers and one adder (the ALU is not used). At a software level, the *sob* instruction is used to weight the values from the convolution in both x with y dimensions, the result is used if it is not above a maximum threshold (otherwise the maximum value is used), as showed in Algorithm 3.

input : Image array of MxN dimensions and kernel array of IxJ dimensions
output: New s value of a pixel

```
// Depends on filter direction;
for k ← 1 to M or N do
    sx ← convolve(Sub − image, Kernel_X);
    sy ← convolve(Sub − image, Kernel_Y);
    s = sqrt(sob(sx, sy));
    if s >= (256/sqrt(256 * 256 + 256 * 256)) then
        s = 255/sqrt(256 * 256 + 256 * 256)
    end
end
Return s
```

Algorithm 3: Sobel algorithm extract

To test these special instructions, small assembly codes were developed which consist in the execution of the selected operation through an array of 100 elements. For the three approximate applications found, two assembly codes were developed, one with where the special instruction is used and another equivalent with common assembly instruction to execute the same as the special instruction.

IV. Analysis of Results

Each assembly code for the approximate applications (both the version using the special instruction and the version without it) were simulated in order to obtain the total number of cycles and integer operations (for example, multiplications, additions, etc.) for comparison. At a hardware level, the architecture with the additional specialized hardware was simulated in order to obtain data to compare between the optimized version and the original processor, in terms of area (slices and LUTs) and power.

The main contribution of the developed ASIPs is the speedup in terms of the total number of cycles and integer operations, as showed in Figure 4. For the *eucl* instruction, the total cycles were reduced from 5130 to 4730 in the optimized version, and the integer operations from 1528 to 1128. For the *absv* instruction, the total cycles changed from 2534 to 1330 and the integer operations from 2133 to 1128 with the optimized version. Finally, the *sob* instruction achieved a reduction in total cycles from 8930 to 4730 and integer operations from 1928 to 1128 in the optimized version.

The *eucl* instruction got the smaller reduction because its transformation only combined two separated assembly

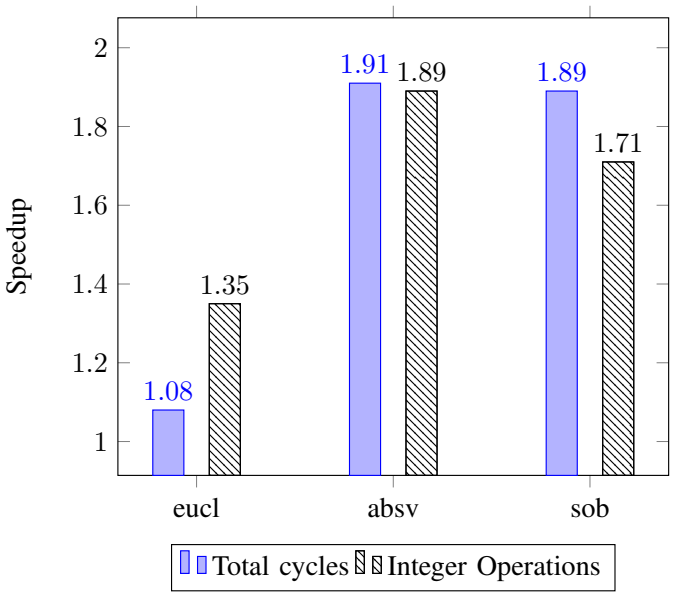

Figure 4: Cycles comparison.

Table I: Area comparison

Metric	*eucl* instr.	*absv* instr.	*sob* instr.	No special instr.
# Slices	3998	4058	4223	3990
% Slices	5%	5%	6%	5%
# LUTs	6384	6465	6079	6199
% LUTs	9%	9%	8%	8%

instructions (substract and then multiply) into one operation. The *sob* instruction, although somewhat similar to the *eucl*, got a bigger reduction in number of cycles because it transformed tree single assembly operations (two multiplications and one addition) into only one special instruction. Finally, the *absv* got the most significant reduction in total cycles since it performs a very specific operation, while the normal assembly instructions' version used branch logic to test for a negative value in the substraction and apply the proper operation.

The additional area caused by the ASIP hardware had a minimal impact compared to the general processor, as showed in Table I. Every optimized version of the processor for each special instruction had a difference of area of 1% at most compared to the processor without the ASIP, which for practical use is not significant.

As same as the area, the impact on the power was very low, as showed in Figures 5 and 6. The maximum difference on power was 5 mW in the dynamic power, as showed in Figure 5. The quiescent power stayed the same for all processor's versions, at 1163 mW, which added to the dynamic power makes a total power between 1493 to 1498 mW; as showed in Figure 6 this difference is not noticeable.

The minimum impact on area and power was mainly due to the main processor having already a lot of hardware to

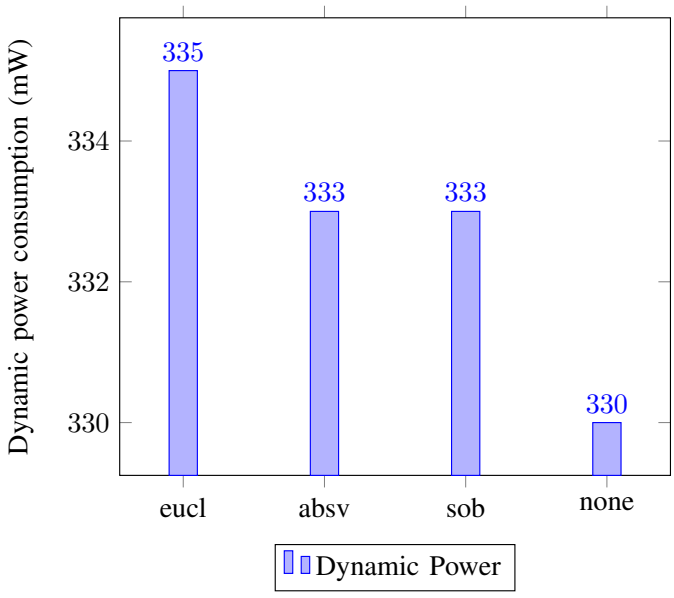

Figure 5: Cycles comparison.

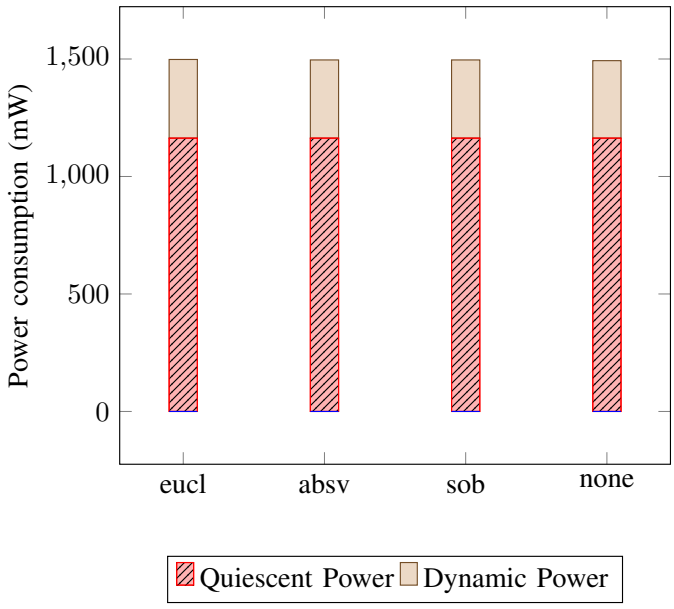

Figure 6: Power comparison.

support common assembly instructions, and while almost the whole processor remained, for all the three special instructions, the same for flexibility purposes, a considerable reduction in execution time was achieved.

V. Conclusion

The developed ASIPs showed speedups in total of cycles of almost 2X while increasing the overall hardware area and power at roughly 1% and 5mW respectively compared to the unmodified version. Future work involves implementing hardware-level approximation for the SI logic developed, which will increase the execution time at a cost of output quality, which will further give the ASIPs the freedom to adjust to more type of needs in different scenarios.

References

[1] Qiang Xu, Todd Mytkowicz, and Nam Sung Kim. Approximate computing: A survey. *IEEE Design & Test*, 2018.

[2] Jörg Henkel. Closing the soc design gap. *Computer*, 36(9):119–121, 2003.

[3] Jörg Henkel. Design and architectures for embedded systems (esii). 2006.

[4] Stelios Sidiroglou-Douskos, Sasa Misailovic, Henry Hoffmann, and Martin Rinard. Managing performance vs. accuracy trade-offs with loop perforation. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, pages 124–134. ACM, 2011.

[5] Cheng Tan, Thannirmalai Somu Muthukaruppan, Tulika Mitra, and Lei Ju. Approximation-aware scheduling on heterogeneous multi-core architectures. In *Design Automation Conference (ASP-DAC), 2015 20th Asia and South Pacific*, pages 618–623. IEEE, 2015.

[6] Hadi Esmaeilzadeh, Adrian Sampson, Luis Ceze, and Doug Burger. Neural acceleration for general-purpose approximate programs. *IEEE Micro*, 33(3):16–27, 2013.

[7] Swagath Venkataramani, Amit Sabne, Vivek Kozhikkottu, Kaushik Roy, and Anand Raghunathan. Salsa: systematic logic synthesis of approximate circuits. In *Proceedings of the 49th Annual Design Automation Conference*, pages 796–801. ACM, 2012.

[8] Ashish Ranjan, Arnab Raha, Swagath Venkataramani, Kaushik Roy, and Anand Raghunathan. Aslan: Synthesis of approximate sequential circuits. In *Design, Automation and Test in Europe Conference and Exhibition (DATE), 2014*, pages 1–6. IEEE, 2014.

[9] Kumud Nepal, Yueting Li, R Iris Bahar, and Sherief Reda. Abacus: A technique for automated behavioral synthesis of approximate computing circuits. In *Design, Automation and Test in Europe Conference and Exhibition (DATE), 2014*, pages 1–6. IEEE, 2014.