

# Automatika

Journal for Control, Measurement, Electronics, Computing and Communications



ISSN: 0005-1144 (Print) 1848-3380 (Online) Journal homepage: <https://www.tandfonline.com/loi/taut20>

## Elementary operations: a novel concept for source-level timing estimation

Nikolina Frid, Danko Ivošević & Vlado Sruk

To cite this article: Nikolina Frid, Danko Ivošević & Vlado Sruk (2019) Elementary operations: a novel concept for source-level timing estimation, *Automatika*, 60:1, 91-104, DOI: 10.1080/00051144.2019.1581695

To link to this article: <https://doi.org/10.1080/00051144.2019.1581695>



© 2019 The Author(s). Published by Informa UK Limited, trading as Taylor & Francis Group



Published online: 20 Feb 2019.



Submit your article to this journal [↗](#)



Article views: 224




View related articles [↗](#)



View Crossmark data [↗](#)



# Elementary operations: a novel concept for source-level timing estimation

Nikolina Frid , Danko Ivošević and Vlado Struk

Faculty of Electrical Engineering and Computing, University of Zagreb, Zagreb, Croatia

## ABSTRACT

Early application timing estimation is essential in decision making during design space exploration of heterogeneous embedded systems in terms of hardware platform dimensioning and component selection. The decisions which have the impact on project duration and cost must be made before a platform prototype is available and software code is ready to be linked and thus timing estimation must be done using high-level models and simulators. Because of the ever increasing need to shorten the time to market, reducing the amount of time required to obtain the results is as important as achieving high estimation accuracy. In this paper, we propose a novel approach to source-level timing estimation with the aim to close the speed-accuracy gap by raising the level of abstraction and improving result reusability. We introduce a concept – *elementary operations* as distinct parts of source code which enable capturing platform behaviour without having the exact model of the processor pipeline, cache etc. We also present a timing estimation method which relies on elementary operations to craft hardware profiling benchmark and to build application and platform profiles. Experiments show an average estimation error of 5%, with maximum below 16%.

## ARTICLE HISTORY

Received 9 October 2018  
Accepted 5 February 2019

## KEYWORDS

Timing estimation; system level design; heterogeneous embedded systems; elementary operations

## 1. Introduction

Systems on Chip (SoC), which are used to run modern complex applications, must have the heterogeneous structure of processing, memory and communication elements to meet high performance, energy efficiency and low price goals. Due to the exponential growth of heterogeneous system complexity, it is estimated that designers productivity will have to increase up to ten times to successfully meet system requirements and constraints within the similar time and cost limits [1]. The key to success is making good decisions in early design stages, before assembly of the first prototype. Raising abstraction level in all design phases enables separation of computation from communication and using separated application and platform models. This leads to a more efficient approach to design space exploration (DSE) [2]. Early timing estimation is one of the most important phases in DSE. In recent years, the traditional approach using highly accurate Instruction Set Simulator (ISS) has been replaced by high-level timing estimation models which enable obtaining estimates in early design stages [3–10].

In this paper, we propose a source-level application execution time estimation method based on a concept named *elementary operations* which enables capturing architectural effects and compiler optimizations influence. The estimation method consists of two phases: *analysis* and *estimation*. In the analysis phase,

application and platform configurations considered for design are profiled. *Application profile* is obtained by transforming application source code into a list of elementary operations structured in loops, branches and sequences. It is independent from platform and compiler optimization level, and hence the same application profile can be used to estimate execution time on any platform. *Platform profile* is obtained by executing a benchmark entitled “*ELOPS benchmark*”<sup>1</sup> on every platform configuration and for each compiler optimization level separately. This benchmark was specially crafted as a part of this research to measure execution times of elementary operations on real platforms. Results of benchmark run on each platform configuration make the *platform profile* for that respective configuration. In the estimation phase, the proposed timing estimation algorithm combines application and target platform profiles to provide timing estimate.

Accuracy of our approach is evaluated using the JPEG image compression algorithm and Advanced Encryption Standard (AES) algorithm on several hardware configurations based on two RISC processors: ARM A9 and Microblaze, custom-built for Xilinx Zynq-based ZC706 platform. Achieved accuracy (i.e. error rate) is similar to the most accurate state of the art source-level timing estimation methods. The strong point is a significant reduction in time and effort required to obtain results due to reusability of

application and platform profiles. Also, the proposed method can be easily scaled for systems with hundred or more elements of the same type, in a similar fashion to the method demonstrated in [11].

The rest of this paper is organized as follows. Section 2 provides an overview of the current state of art in area of high-level timing estimation. The proposed method for source-level timing estimation is presented in Section 3. The flow of application timing estimation is described in Section 4. Test cases used for evaluating the proposed method are given in the first part of Section 5. The results of the conducted experiments are presented and discussed in the rest of the section.

## 2. Related work

Authors in [12] propose a source-level simulation infrastructure that provides a full range of performance, energy, reliability, power and thermal estimation. For timing simulation, they build upon their previous work [3] which uses simulation-based approach with back annotation on intermediate representation (IR) level. Simulating pipeline effects on basic block boundaries requires additional pair-wise block simulation for every possible block pair combination on a cycle-accurate reference. They consider a high-level cache model by reconstructing target memory traces solely based on IR and debugger information. Simulation of the entire application execution is done using System C and transaction-level modeling (TLM) [13] with estimation error below 10%. Simulating pipeline effects on basic block boundaries requires additional pair-wise block simulation for every possible block pair combination.

Other approaches use machine learning and mathematical models for early timing estimation. Authors in [8] use artificial neural networks (ANN). ANN gives timing estimate based on execution time and total number of each instruction type. Estimation error is around 17% but the method is much more flexible compared to simulation methods and provides a higher level of result reusability. After the initial training period, estimation results are obtained rapidly.

Methods presented in [9] and [10] are based on linear regression and ANNs with higher error rates – around 20%. Authors in [14] use model tree-based regression technique as a machine learning method of choice.

Authors in [11,15,16] propose hybrid methods: first simulation is used to obtain the execution time of each procedure on each type of processing element, then analytical methods are used to account for cache and communication effects.

In [17], authors use linear regression for calculating timings but they use a set of specially crafted training program to identify instruction costs of an

abstract machine. Authors try to capture effects of cache, pipeline and code optimization by crafting examples with longer instruction sequences and loops. However, since they rely on IR, they face challenges when introducing code optimizations because virtual instructions in the translation of the training program are not in close correspondence with the compiled version.

The concept of *elementary operations* has first been introduced in [18] in attempt to characterize platform behaviour without having the exact hardware model. This preliminary method for early timing estimation lacked compiler optimization support and ability to estimate input dependent application tasks. In this paper, we extend our previous work and present an improved method.

## 3. Elementary operations approach

Our approach is based on decomposing a piece of source code written in C programming language (standard C11) to *elementary operations* – distinct parts of source code which enable capturing platform behaviour without having the exact model of processor pipeline, cache etc. The set of elementary operations is finite with several subsets: integer, floating point logic and memory operations. These sets are co-related to parts of RISC-like architecture processor and memory datapath.

### 3.1. Classification of elementary operations

We propose a multi-level elementary operations classification scheme. The top level contains four operation classes: INTEGER, FLOATING POINT, LOGIC and MEMORY. Second level of classification is based on *origin* of operands (i.e. location in memory space): *local*, *global* or *procedure parameters*. This stems from the difference in locality due to the way compiler implements operands stored in different parts of memory space. It is expected that each group will show different timing behaviour: local variables, being heavily used, are almost always in cache, while global and parameter operands must be loaded from an arbitrary address and can cause a cache miss. Third level of classification is by operand type: (1) scalar *variables* and (2) *arrays* of one or more dimensions. Pointers are treated as scalar variables when the value of pointer is given using a single variable, or as arrays when the value of pointer is given using multiple variables.

Operations which belong to INTEGER and FLOATING POINT classes are: addition (ADD),<sup>2</sup> multiplication (MUL) and division (DIV). LOGIC class contains logic operations (LOG): (i.e. *and*, *or*, *xor* and *not*) and shift operations (SHIFT): operations that perform bit-wise movement (e.g. rotation, shift, etc.). Operations in MEMORY class are: single memory assign

**Table 1.** Elementary operation classification.

		INTEGER	FLOATING POINT	LOGIC	MEMORY
local	variable	INT_loc_var ADD	FP_loc_var ADD	LOG_loc_var LOG	MEM_loc_var ASSIGN
		INT_loc_var MUL	FP_loc_var MUL	LOG_loc_var SHIFT	MEM_loc_var PROC
		INT_loc_var DIV	FP_loc_var DIV		
array		INT_loc_arr ADD	FP_loc_arr ADD	LOG_loc_arr LOG	MEM_loc_arr ASSIGN
		INT_loc_arr MUL	FP_loc_arr MUL	LOG_loc_arr SHIFT	MEM_loc_arr BLOCK
		INT_loc_arr DIV	FP_loc_arr DIV		MEM_loc_arr PROC
global	variable	INT_glob_var ADD	FP_glob_var ADD	LOG_glob_var LOG	MEM_glob_var ASSIGN
		INT_glob_var MUL	FP_glob_var MUL	LOG_glob_var SHIFT	
		INT_glob_var DIV	FP_glob_var DIV		
array		INT_glob_arr ADD	FP_glob_arr ADD	LOG_glob_arr LOG	MEM_glob_arr ASSIGN
		INT_glob_arr MUL	FP_glob_arr MUL	LOG_glob_arr SHIFT	MEM_glob_arr BLOCK
		INT_glob_arr DIV	FP_glob_arr DIV		
parameter	variable	INT_par_var ADD	FP_par_var ADD	LOG_par_var LOG	MEM_par_var ASSIGN
		INT_par_var MUL	FP_par_var MUL	LOG_par_var SHIFT	MEM_par_var PROC
		INT_par_var DIV	FP_par_var DIV		
array		INT_par_arr ADD	FP_par_arr ADD	LOG_par_arr LOG	MEM_par_arr ASSIGN
		INT_par_arr MUL	FP_par_arr MUL	LOG_par_arr SHIFT	MEM_par_arr BLOCK
		INT_par_arr DIV	FP_par_arr DIV		MEM_par_arr PROC

```

1  int g_var1 = 56;
2  float g_var2 = 5.6;
3
4  void function (int *a, int *d, float *f, float *g)
5  {
6      int x, y, z, b[100], c[100], i;
7      float e, h[100];
8
9      ...
10     z = y-x;           // INT_loc_var ADD
11     g_var1 += i;       // INT_glob_var ADD
12     b[i] = a[i] + d[i]; // INT_par_arr ADD
13
14     e = y/x;           // FP_loc_var DIV
15     g_var2 = g_var1 * y; // FP_glob_var MUL
16     g[i] = f[i] * i;   // FP_par_arr MUL
17
18
19     for(i=1;i<100;i++){
20         d[i] = b[i]; // MEM_par_arr ASSIGN
21         c[i] = c[i] & b[i]; // LOG_loc_arr LOG
22         d[i] = d[i-1]>>1; // LOG_par_arr SHIFT
23     }
24
25     ...
26 }

```

**Figure 1.** An example of elementary operations identification in source code.

(ASSIGN), block transaction (BLOCK) and procedure call (PROC). MEMORY BLOCK represents a transaction of a block of size 1000 and it can only have *array* operands. MEMORY PROC represents a function call with one argument and a return value. Arguments can be variables and arrays, declared locally or given as parameters of the caller function, but never global. All of these operations are listed in Table 1. Abbreviations indicated in the table are used further on when referring to a specific class. Sample source code given in Figure 1 illustrates how code statements can be correlated to elementary operations classification scheme. Each operation is denoted using abbreviations from Table 1.

Accuracy of timing estimation using the proposed classification scheme was analysed on two RISC processors: *ARM Cortex-A9* and *Microblaze*, implemented on *Xilinx Zynq-based ZC706* platform. First, the actual execution time of each elementary operation from Table 1 was measured for each processor. Each operation was

repeated in a *for*-loop a thousand times to compensate for timer setup and to create a context to capture the effects of compiler optimizations, cache and pipeline. Then, test cases were crafted in a way to contain constructs commonly found in real-world application code. For each test case, elementary operations were identified and, using previously obtained execution times, a timing estimate was calculated. Finally, each test case was executed on both target processors in order to obtain actual execution times and compare them to estimated ones.

### 3.1.1. Sequence of operations

A sample source code given in Figure 2 illustrates four examples of sequences of operations:

- (1) five *INT\_loc\_var* ADD operations in a single statement
- (2) *for*-loop with five statements in a sequence, each containing one *INT\_loc\_arr* ADD operation

```

1 void function ()
2 {
3     int a, b, c, d, e, f, g;
4     int x[1000], y[1000], z[1000], m[1000], n[1000];
5     float fa, fb, fc, fd, fe, ff, fg;
6     float f_x[1000], f_y[1000], f_z[1000], f_m[1000], f_n[1000];
7
8     c = a + b + d + e + f + g; // INT_loc_var ADD
9                                // -> 5 sequential operations in 1 statement
10
11    for (i=0; i<1000; i++) {
12        z[i] = x[i] + y[i]; // INT_loc_arr ADD
13        y[i] = x[i] + z[i]; // -> 5 sequential operations
14        m[i] = y[i] + z[i];
15        n[i] = m[i] + x[i];
16        x[i] = m[i] + n[i];
17    }
18
19    fc = fa * fb * fd * fe * ff * fg; // FP_loc_var MUL
20                                        // -> 5 sequential operations in 1 statement
21
22    for (i=0; i<1000; i++) {
23        f_z[i] = f_x[i] * f_y[i]; // FP_loc_arr MUL
24        f_y[i] = f_x[i] * f_z[i]; // -> 5 sequential operations
25        f_m[i] = f_y[i] * f_z[i];
26        f_n[i] = f_m[i] * f_x[i];
27        f_x[i] = f_m[i] * f_n[i];
28    }
29 }

```

**Figure 2.** Sequence of elementary operations.

**Table 2.** Execution times for operations in Figure 2.

		INT_loc_var ADD 5 operations	INT_loc_arr ADD 5 operations	FP_loc_var MUL 5 operations	FP_loc_arr MUL 5 operations
ARM	Estimated time [s]	7.13E-08	2.57E-04	1.13E-07	2.57E-04
	Actual time [s]	2.14E-08	2.16E-04	4.20E-08	2.15E-04
	Error [%]	233.18	18.98	169.05	19.53
Microblaze	Estimated time [s]	4.50E-07	9.89E-04	5.75E-07	1.06E-03
	Actual time [s]	1.70E-07	7.38E-04	2.55E-07	8.13E-04
	Error [%]	164.71	34.01	125.49	30.38

- (3) five *FP\_loc\_var* MUL operations in a single statement
- (4) *for*-loop with five statements in a sequence, each containing one *FP\_loc\_arr* MUL operation

It must be noted that in our approach *for*-loops are considered to be an implicit part of operations with array type of operands. This is because when the execution time of each elementary operation is measured, it is done in a loop and all overhead added by the loop is already included in the obtained timings.

According to the initial classification scheme proposal, each elementary operation is treated separately during code analysis and timing estimation. Estimated and actual execution times for the source code in Figure 2 are presented in Table 2. Estimation error is calculated using formula

$$\text{Error} = (\text{Estimated time} - \text{Actual time}) / \text{Actual time} * 100\%$$

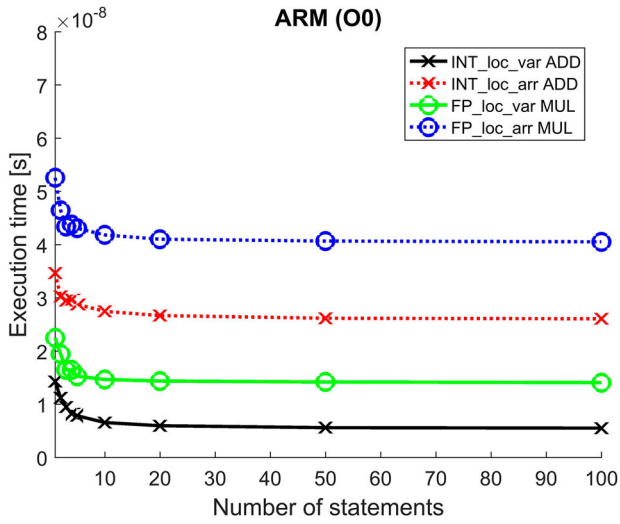
Due to high error, which in some cases goes above 200%, additional analysis was done to investigate how execution time per operation varies with increase in total number of operations in a sequence. Execution time of sequences of 2, 3, 4, 5, 10, 20, 50 and 100

operations for all operation classes have been measured on both ARM and Microblaze for optimization levels O0 – O2. Results for the *INT\_loc\_var* ADD, *INT\_loc\_arr* ADD, *FP\_loc\_var* MUL and *FP\_loc\_arr* MUL are shown in Figure 3.

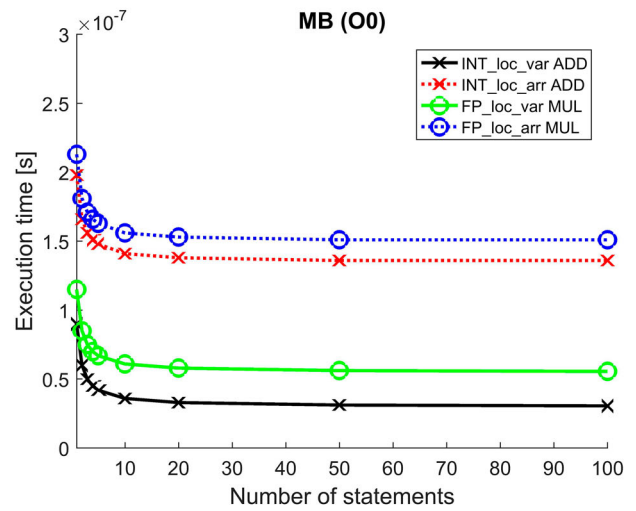
It can be observed that execution time per one elementary operation decreases exponentially with the increase in total number of operations in a sequence. The same behaviour is observed for all other types of elementary operations on both processors, but for the sake of brevity is not shown here. This leads to conclusion that due to pipelining and decreasing of loop overhead, sequence lengths plays an important role when estimating timings of sequences of operations which belong to the same class. Experiments also show that by measuring timings only for several lengths such as 2, 3, 4, 5, 10, 20 and 50 an approximation with less than 10% error can be done for any other sequence lengths. This makes profiling a much faster and more efficient process.

### 3.1.2. Operations with mixed types and origin of operands

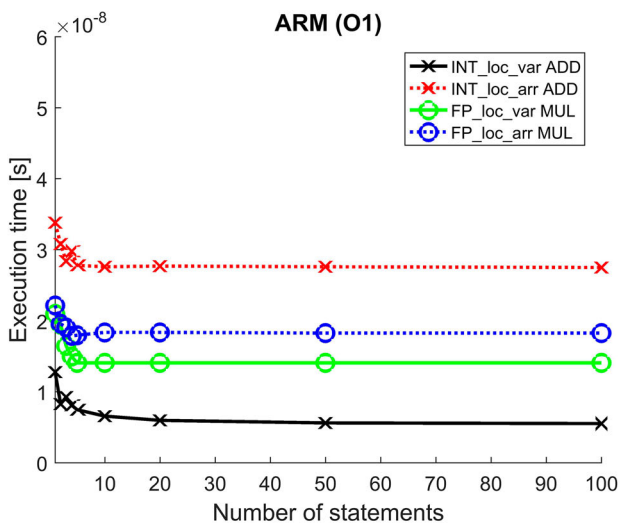
Source code in Figure 4 represents four cases of operations with mixed type and origin of operands. The initial elementary operations classification scheme



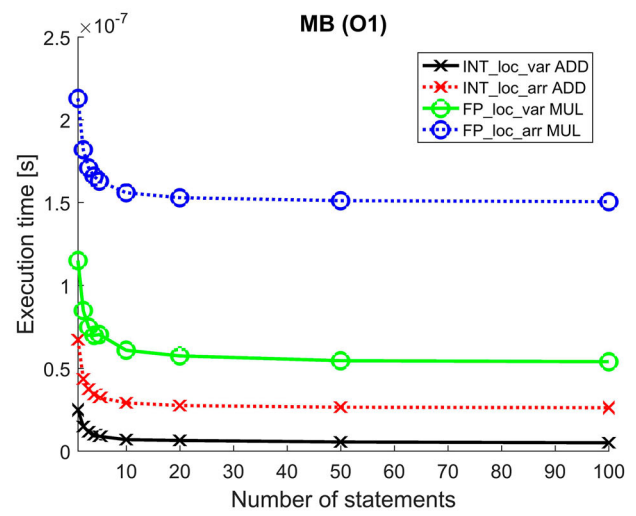
(a)



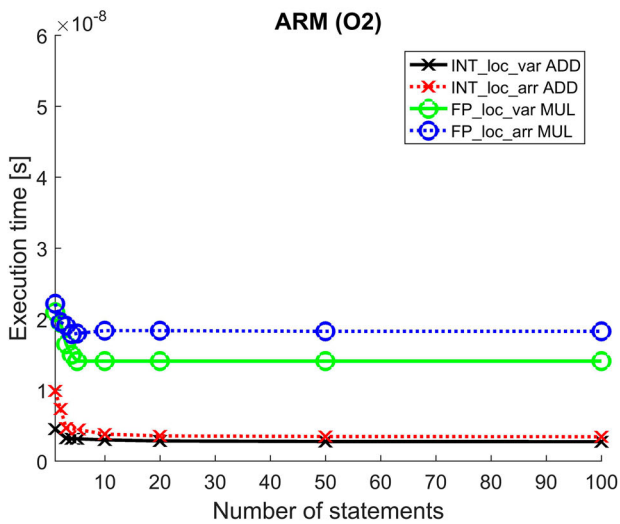
(b)



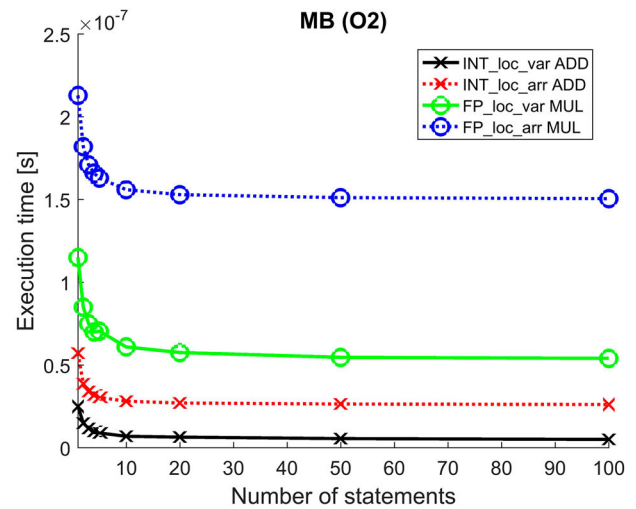
(c)



(d)



(e)



(f)

**Figure 3.** Execution time per statement. (a) ARM opt. level O0 (b) Microblaze opt. level O0 (c) ARM opt. level O1 (d) Microblaze opt. level O1 (e) ARM opt. level O2 and (f) Microblaze opt. level O2.

proposal does not give explicit specifications for determining elementary operation class in such cases. Thus, we additionally introduce *origin priority*. Priorities

are defined based on difference in locality due to the way the compiler implements each of the operand type (from highest to lowest) and the idea is to select

```

1  int g_a, g_c;
2  int gf[1000], gh[1000];
3  void function ()
4  {
5      int i;
6      int x[1000];
7
8      g_c=g_a+i;           // OP1: INT_glob_var ADD
9
10     for (i=0; i<1000; i++)
11         gf_h[i] = gf[i] + x[i]; // OP2: INT_glob_arr ADD
12
13     for (i=0; i<1000; i++)
14         gf_h[i] = gf[i] + i;    // OP3: INT_glob_arr ADD
15
16     for (i=0; i<1000; i++)
17         gf_h[i] = x[i] + i;    // OP4: INT_glob_arr ADD
18 }

```

**Figure 4.** Example of operations with mixed types and origin of operands.

**Table 3.** Execution times for operations in Figure 4.

		OP1	OP2	OP3	OP4
ARM	Estimated time [s]	2.89E-08	3.48E-05	3.48E-05	3.48E-05
	Actual time [s]	2.85E-08	4.73E-05	2.86E-05	3.53E-05
	Error [%]	1.33	-26.49	21.85	-1.36
Microblaze	Estimated time [s]	1.05E-07	1.83E-04	1.83E-04	1.83E-04
	Actual time [s]	1.00E-07	1.88E-04	1.53E-04	1.58E-04
	Error [%]	5.00	-2.66	19.61	15.82

operation class based on the operand with the highest priority:

- (1) parameter array
- (2) global array
- (3) local array
- (4) parameter variable
- (5) global variable
- (6) local variable

All operations in the given example are classified as operations with global operands, even though OP1, OP3 and OP4 contain local variables while OP2 and OP4 contain local array operands. The comparison of estimated and actual execution times for these test cases is presented in Table 3. It shows that in the case of OP2 and OP3, where local operands are present, estimation error goes over 20%. Error for OP4 is slightly lower, but this is probably because the proposed method gives underestimation in case of local arrays (OP2) and overestimation in case of local variables (OP3), so the numbers even-out.

These results indicate that it is necessary to modify the existing solution by expanding the origin priority-based approach and give each operation additional attributes to denote different types and origin of operands. Attribute *mod* will be added in case when an operation has operands of mixed types and origin. Attribute value will indicate the following cases: (1) presence of variable operands in an operation with arrays, (2) presence of *global* variable in operations with *parameter* operands, (3) presence of *global* arrays in operations with *parameter* array operands, (4) presence of *local* variables in operations with *parameter* or *global*

operands, (5) presence of *local* arrays in operations with *parameter* or *global* array operands and (6) presence of constants. List of these values is given in Table 5 in column *Values* for operation modifier attribute *mod*.

### 3.1.3. Array operands index

Index of array operands can have more than one dimension and/or can be calculated using the values of more than one variable. The same applies to *struct* types in C source code. Sample source code in Figure 5 illustrates such examples. OP1 is an example of MEM\_par\_arr ASSIGN operation with a 3-dimensional array, and OP2 is a similar example with a *struct* containing 2-dimensional array. At this point, operations with *structs* are classified as operations with arrays. Operations OP3 to OP7 are examples of arrays with index calculated based on values of several variables.

Table 4 shows the results for code in Figure 5. In case of OP1 and OP2 estimation error is above 60%. Almost the same error is observed in case of 3-dimensional array and *struct* containing 2-dimensional array (making it also a 3-dimensional structure). In case of OP3 to OP7 it can be observed that the results are severely underestimated, but it can also be observed that the underestimation increases as the number of operations required for index calculation increases.

These results suggest that the initial classification scheme, which does not recognize multiple dimensions and index structure, should be extended even further. Several attributes will be added to operations with arrays to indicate specifics about index type. These attributes will indicate (1) *type* of array index, which can be *simple* – given using a single variable, *complex* – calculated based on two or more variables and a *constant*,

```

1  int gf_f[1000], gf_g[1000], gf_h[1000];
2  void function (struct_i *Y, struct_i *X, int *h, int *f)
3  {
4      ...
5
6      for (i=0; i<10; i++)
7          for (j=0; j<10; j++)
8              for (m=0; m<10; m++)
9                  h[i][j][m] = f[i][j][m];           // OP1: MEM_par_arr ASSIGN
10
11     for (i=0; i<10; i++)
12         for (j=0; j<10; j++)
13             for (m=0; m<10; m++)
14                 Y[i].field[j][m] = X[i].field[j][m]; // OP2: MEM_par_arr ASSIGN
15
16     for (i=0; i<1000; i++)
17         gf_f[i] = gf_f[(i*5)%1000] + gf_h[i];       // OP3: INT_glob_arr ADD
18     for (i=0; i<1000; i++)
19         gf_g[i] = gf_f[(i*7+2090)%1000] + gf_h[i]; // OP4: INT_glob_arr ADD
20     for (i=0; i<1000; i++)
21         gf_g[i] = gf_f[(i*7+x*3)%1000] + gf_h[i]; // OP5: INT_glob_arr ADD
22     for (i=0; i<1000; i++)
23         gf_g[i] = gf_f[(i*7+x*3+7)%1000] + gf_h[i]; // OP6: INT_glob_arr ADD
24     for (i=0; i<1000; i++)
25         gf_g[i] = gf_f[(i*7+x*3+y*5)%1000] + gf_h[i]; // OP7: INT_glob_arr ADD
26
27     ...
28 }

```

**Figure 5.** Array index examples.

**Table 4.** Execution times for operations in Figure 5.

		OP1	OP2	OP3	OP4	OP5	OP6	OP7
ARM	Estimated time [s]	2.55E-05	2.55E-05	3.48E-05	3.48E-05	3.48E-05	3.48E-05	3.48E-05
	Actual time [s]	6.94E-05	6.26E-05	6.25E-05	6.15E-05	6.80E-05	6.86E-05	7.88E-05
	Error [%]	-63.21	-59.30	-44.32	-43.44	-48.81	-49.28	-55.81
Microblaze	Estimated time [s]	1.60E-04	1.60E-04	1.83E-04	1.83E-04	1.83E-04	1.83E-04	1.83E-04
	Actual time [s]	2.42E-04	2.52E-04	2.58E-04	2.73E-04	2.93E-04	2.98E-04	3.28E-04
	Error [%]	-33.88	-36.51	-29.07	-32.97	-37.54	-38.59	-44.21

**Table 5.** Extended model: attributes overview.

Attribute group	Attribute name	Values
Sequence of operations	<i>seq</i>	positive integer
Operation modifier	<i>mod</i>	"var" – at least one variable operand of the same origin is present
		"glob_var" – at least one global variable operand is present
		"glob_arr" – at least one global array operand is present
		"loc_var" – at least one local variable operand is present
		"loc_arr" – at least one local array operand is present
Index modifier	<i>type</i>	"simple" – index is given as a single variable "complex" – index must be calculated using more than one variable "const" – index is a constant value
	<i>dim</i>	positive integer – dimension of array index
	<i>add_nr</i>	positive integer – number of addition operations used for index calculation
	<i>mul_nr</i>	positive integer – number of multiplication operations used for index calculation

(2) *dimension* of array index (3) *number of addition* operations in *complex* type array index, (4) *number of multiplication* operations in *complex* type array index. List of these attributes is given in Table 5 under *index modifier* attribute description. Finally, since arrays and *structs* show similar timings, they will continue to be treated equally.

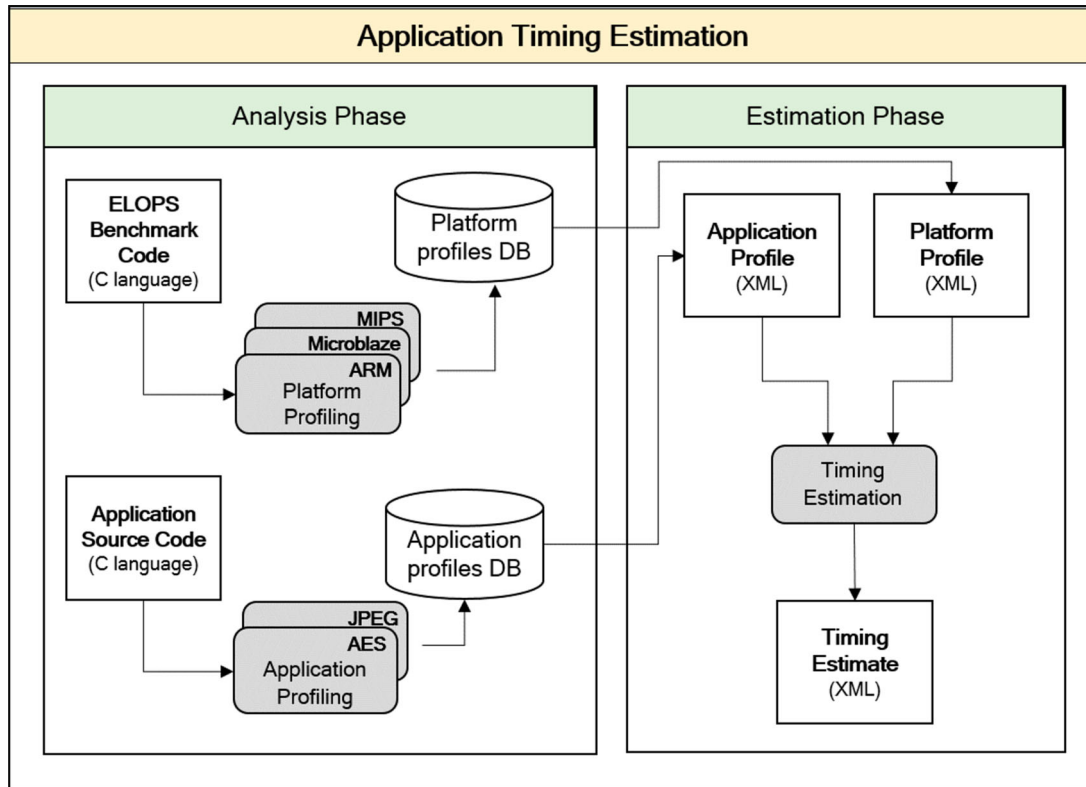
### 3.2. Classification scheme attributes overview

Based on previously discussed observations, the classification scheme is extended to incorporate the

proposed modifications. These extensions are included as attributes to each class of elementary operations. According to the three groups of possible cases which have effect on execution duration of elementary operations defined in Table 1, three groups of attributes are listed in Table 5 under column *Attribute group*.

Sequence of operations is denoted with attribute *seq*. *Operation modifier* group contains attribute *mod* which is present in operations with operands of mixed types and origin. Attribute value is a space separated list which can contain one or more of the following elements as listed in Table 5





**Figure 6.** Application timing estimation flow.

*Index modifiers* is a group of four attributes, all listed in Table 5, which are added to elementary operations with arrays.

#### 4. Application timing estimation

The proposed application execution time estimation method based on elementary operations consists of *analysis* and *estimation* phase as indicated in Figure 6. The first step of *analysis* phase is platform profiling. In this step a specially crafted *ELOPS* benchmark, described later, is compiled and run on every platform configuration and for each optimization level separately. The platform profile is created based on the results of benchmark runs and contains timings of elementary operations. The second step is application profiling. Application profile is a transformation of original C source code into a list of elementary operations structured in loops, branches and sequences. Application profiling is done only once on the original C source code. For this purpose common compiling constructs such as *abstract syntax tree* (AST) and *control and data flow graph* (CDFG) are used. Application and platform profiles created during the analysis phase are permanently stored in database.

In estimation phase, first platform and application profiles are retrieved from database. Then a timing estimation algorithm, described later in this section, combines application and platform profiles to provide timing estimate.

##### 4.1. Platform profiling

Platform profiling starts with the execution of *ELOPS* benchmark<sup>3</sup> on every platform configuration considered for final design. A platform configuration is a pair of a specific processor and a memory connected to it, used to store instructions and data.

*ELOPS* benchmark is designed based on elementary operations classification scheme to measure execution time of each operation from Table 1 and timing effect of every possible attribute listed in Table 5. For each operation sub-class listed in Table 1 (e.g. INTEGER ADD, LOGIC SHIFT etc.), three main groups of benchmark entries are defined: *local*, *global* and *parameters*. Each group contains two sub-groups: *variable* and *array*. *Array* sub-group branches further by two criteria, *array index type* and *dimension*. This means that for each operation from Table 1 and for each origin operand group, there are five *base* benchmark entries. All benchmark entries are systematized in Table 6.

Each *base* benchmark entry has sub-variants in which the different lengths of sequences of operations are measured. The distinction is made between multiple occurrences of the same operation in one statement – named *sequential operations*, and sequence of statements belonging to same elementary operations class – named *sequential statements*. In our current implementation, the benchmark contains entries for the following sequence lengths: 2,3,5 and 10.

Timing effects of attributes listed in Table 5 are also measured using this benchmark by introducing

**Table 6.** Platform benchmark systematization.

Origin	Base		Modifiers	Sequence lengths measured
Local	variable		const	single sequential operations (length: 2,3,5,10) <sup>a</sup> sequential statements (length: 2,3,5,10)
	array	<i>type</i> = "simple", <i>dim</i> = 1 <i>type</i> = "simple", <i>dim</i> = 2 <i>type</i> = "simple", <i>dim</i> = 3 <i>type</i> = "complex", <i>dim</i> = 1	var const	single sequential operations (length: 2,3,5,10) <sup>a</sup> sequential statements (length: 2,3,5,10)
			add_nr mul_nr loc_var const	single sequential operations (length: 2,3,5,10) <sup>a</sup> sequential statements (length: 2,3,5,10)
			const Var loc_var loc_arr add_nr mul_nr	single sequential operations (length: 2,3,5,10) <sup>a</sup> sequential statements (length: 2,3,5,10)
Global	variable		glob_var loc_var const	single sequential operations (length: 2,3,5,10) <sup>a</sup> sequential statements (length: 2,3,5,10)
	array	<i>type</i> = "simple", <i>dim</i> = 1 <i>type</i> = "simple", <i>dim</i> = 2 <i>type</i> = "simple", <i>dim</i> = 3  <i>type</i> = "complex", <i>dim</i> = 1	const Var loc_var loc_arr add_nr mul_nr	single sequential operations (length: 2,3,5,10) <sup>a</sup> sequential statements (length: 2,3,5,10)
			glob_var loc_var const	single sequential operations (length: 2,3,5,10) <sup>a</sup> sequential statements (length: 2,3,5,10)
			const var loc_var loc_var glob_arr glob_arr add_nr mul_nr	single sequential operations (length: 2,3,5,10) <sup>a</sup> sequential statements (length: 2,3,5,10)
Parameters	variable		glob_var loc_var const	single sequential operations (length: 2,3,5,10) <sup>a</sup> sequential statements (length: 2,3,5,10)
	array	<i>type</i> = "simple", <i>dim</i> = 1 <i>type</i> = "simple", <i>dim</i> = 2 <i>type</i> = "simple", <i>dim</i> = 3  <i>type</i> = "complex", <i>dim</i> = 1	const const var loc_var loc_var glob_arr glob_arr add_nr mul_nr	single sequential operations (length: 2,3,5,10) <sup>a</sup> sequential statements (length: 2,3,5,10)
			glob_var loc_var const	single sequential operations (length: 2,3,5,10) <sup>a</sup> sequential statements (length: 2,3,5,10)
			const var loc_var loc_var glob_arr glob_arr add_nr mul_nr	single sequential operations (length: 2,3,5,10) <sup>a</sup> sequential statements (length: 2,3,5,10)

<sup>a</sup> Measured only for *base* benchmarks.

modifiers to each *base* benchmark entry. For each modifier listed in Table 6 under column *Modifiers*, separate benchmark entry is created. Also, for each modifier, the following lengths of sequential statements are measured: 2,3,5 and 10.

All measurements are done by executing an operation in a loop for a thousand times to compensate for timer setup effects and to create a context which will capture effects of optimizations and hardware features such as cache and pipeline better. The special case are two elementary operation sub-classes: MEMORY BLOCK and MEMORY PROC. The MEMORY BLOCK class is measured as a single transaction of a block of size 1000 (using *memcpy* function) and it can have only *array* operands. MEMORY PROC class is measured as a function call with one argument and a return value.

In our implementation benchmarks do not contain entries for arrays with an index of dimension higher than 3 because at this point, there was no code in our test applications which contained structures of higher dimensions.

In order to enable accurate estimations for code containing compiler optimizations, the benchmark has to be compiled and run separately for all optimization levels. This way the same optimizations which will occur in e.g. looped or sequential execution in the source code, will also be present in the benchmark code. The measurements are then combined into a platform element profile in an XML document.

#### 4.2. Application profiling

Application profiling uses code analysis and profiling method described in [19] which is slightly adapted to be compatible with the elementary operations classification scheme. Application source code processing starts with the generation of call-tree statistics to produce profiling information at procedure call-graph abstraction level. The compiler transformation flow starts with parsing the source code to the *abstract syntax tree* (AST). During recursive traversal of the tree, information about data structures, types of variables and procedure arguments is used to identify elementary operations according to the proposed classification scheme. AST is further transformed to a *control and data flow graph* (CDFG) representation by using recursive traversal of the tree with the introduction of temporary variables that form the three-address code statement notation. During that process, the key is recognizing points where uniform instruction flow is broken by condition testing in branch or loop jump conditions. The final application profile is obtained by unifying procedures calls statistics and profiles obtained using AST and CDFG for each procedure separately.

The output of the entire process is application profile as an abstract model written in an XML structure. In it, the original application source code is transformed into a multi-level structure of elementary operations organized in loops, branches and sequences.

**Table 7.** Application profile elements.

Element	Possible sub-elements	Attributes Name	Description
application	procedure	name	application name
procedure	loop, branch, operation	name	procedure name
loop	loop, branch, operation	count	number of time the loop is executed
branch	true-body, false-body	cond	condition of branching
true-body	loop, branch, operation	t_count	number of times the true-body element is executed
false-body	loop, branch, operation	f_count	number of times the false-body element is executed
operation	–	class	elementary operation class (INT, FLOAT, MEM, LOG)
		type	elementary operation subclass (e.g. add, mul)
		mod	operation modifier (e.g. var, glob_var)
		index_type	type of index: <i>simple</i> , <i>complex</i> or <i>const</i> <sup>a</sup>
		dim	index dimension <sup>a</sup>
		add_nr	number of addition operations used for index calculation <sup>b</sup>
		mul_nr	number of multiplication operations used for index calculation <sup>b</sup>

<sup>a</sup> Applicable only to operations with arrays. <sup>b</sup> Applicable only to operations with arrays with *complex* index type.

Each application is composed of one or more *procedures* which directly correspond to procedures (functions) in original C source code. *Procedures* can contain any number of *loops*, *branches* or *operations*. *Loop* represents a *for* or a *while* loop, and *branch* represents an *if-else* or *switch-case* conditional constructs. *Loops* and *branches* can have any number of *loops*, *branches* and *operations* as sub-elements. *Operation* represents a single statement or a sequence of operations that has been assigned an elementary operation class. Operations have attributes which cover the extension to classification scheme as discussed in Section 3.1.1. All possible profile elements and attributes are listed in Table 7.

For applications which have data-dependent behaviour, the precision of profiling can be highly dependent on input data in run-time. In such cases, loops iteration count or branch condition evaluation result cannot be resolved without simulation and analysis of variable data values. Since these facts define the number of existing running paths through the application source code both during analysis of hierarchical task graph and formation of control and data flow graph, estimation must rely on one or more simulation runs to determine either the exact number or upper and lower boundaries and statistical probabilities for these values. In our research so far, we have employed a commonly accepted approach of running instrumented code on a host PC (i.e. *host-compiled*) for determining data-dependent behaviour, [3,4,7].

### 4.3. Timing estimation

After obtaining both application and platform profiles, the final step is to combine the two to estimate application execution time. The algorithm is described in short using pseudo-code in Figure 7.

Finally, it is important to accent the reusability of the proposed method. Each application needs to be profiled only once and the obtained profile can be used in the future as is for any platform configuration at hand. In the same manner, each type of platform element needs to be profiled only once and the obtained data can be

reused for timing estimation of any other application. The reusability of profiling results also helps achieve better scalability when building platforms with multiple elements of the same type.

## 5. Experimental setup and results

Verification of elementary operation approach has been done on the commonly used real-world applications:

- (1) *The Advanced Encryption Standard (AES)* [20] – used in two different implementation versions.
  - (a) *AES\_G* – the first version of the *AES* where data is accessed via global variables,
  - (b) *AES\_P* – the second version of the *AES* where data is accessed via procedure parameters.
- (2) *JPEG image compression algorithm* – using implementation as described in [21].

This particular set of applications encompasses all types of elementary operations and represents well the key features of applications for which heterogeneous embedded systems are used most often: multimedia, compression and encryption.

Xilinx Zynq ZC706 reconfigurable evaluation board has been chosen as target platform. Three configurations, each composed of one processor and one memory element have been used:

- (1) *MB1* – MicroBlaze, a 32-bit RISC Harvard architecture soft processor core in the following configuration: 5-stage pipeline with hardware multiplier, barrel shifter and floating-point unit operating at 200 MHz. The processor is connected to 128 KB FPGA-based BRAM memory, operating also at 200 MHz, via local memory bus (LMB). This memory is used for storing both instructions and data.
- (2) *ARM1* – A single core of ARM Cortex-A9 processor is used in the following configuration: operating frequency at 667 MHz, 32 KB L1 cache and 512 KB L2 cache with both instructions and data

**Algorithm 1** Timing Estimation

---

```

for all Procedures in ApplicationProfile do
  for all Elements in Procedure do
    execution_time_est+ = ESTIMATE(Element)
  end for
end for
function ESTIMATE(Element)
  timing_estimate = 0
  if Element is Operation then
    timing_estimate = CALCULATE_OP_DURATION(Element, seq_length)
  else if Element is Branch then
    for all Elements in True-body do
      timing_estimate+ = t_count*ESTIMATE(Element)
    end for
    for all Elements in False-Body do
      timing_estimate+ = f_count*ESTIMATE(Element)
    end for
  else if Element is Loop then
    for all Elements in Loop do
      timing_estimate+ = ESTIMATE(Element)
    end for
    timing_estimate* = count
  end if
  return timing_estimate
end function
function CALCULATE_OP_DURATION(Element, seq_length)
  opclass = decode operation class from Element class and type attributes
  duration = seq_length* PLATFORM_PROFILE.EXEC.TIME(Element, seq_length)
  duration* = CALCULATE_ATTRIBUTE_CONTRIBUTION(Element, opclass)
  return duration
end function

```

▷ loop count attribute

---

**Figure 7.** Timing estimation algorithm.

stored in DDR3 SDRAM memory operating at 533 MHz.

- (3) **ARM2** – A single core of ARM Cortex-A9 processor is used in the following configuration: operating frequency at 667 MHz, 32 KB L1 cache and 512 KB L2 cache but the instructions were stored in DDR3 SDRAM operating at 533 MHz and data is stored in 128 KB FPGA-based BRAM memory operating at 200 MHz.

These configurations are a popular general purpose choice for low-power or thermally constrained, cost-sensitive devices (e.g. smart-phones, digital TV, and both consumer and enterprise applications enabling the Internet of Things).

### 5.1. Test cases

AES\_G and AES\_P have been tested for an example input of 32 bytes of data and JPEG has been tested on *Lenna* image. Each test case and each platform configuration have been profiled and timing estimation has been calculated based on these profiles using method described in Section 4. Then, each test case has been executed on each platform configuration to obtain actual timings. Total execution time for AES\_G, AES\_P and JPEG was measured as the time taken for the entire application to run. Parts of AES\_G and AES\_P (AddRoundKey, ShiftRows, etc.) were measured in 1000x loops because of very small time scale,

to negate timer setup effects. Parts of JPEG applications were not measured in a loop because time scale is orders of magnitude larger than the timer setup overhead. Tests were performed for optimization level O0 – O2. Optimization level O3 has not been considered since level O2 is still the recommended option by most embedded systems manufacturers in order to avoid potentially incorrect execution if the source code is not written exactly following C standard, [6].

Table 8 contains results of timing estimation for AES\_G implementation on all three configurations: *MB1*, *ARM1* and *ARM2*.

Estimation results are calculated for a single run of each individual procedure in AES encryption algorithm: *KeyExpansion*, *AddRoundKey*, *SubBytes*, *ShiftRows* and *MixColumns*. Each table column contains the results for one procedure: (1) *Est.* – estimated execution time, (2) *Act.* – actual execution time, and (3) *Err.* – error between estimated and actual execution time. In the last column of the table are the results for a complete run of the AES algorithm. Timing estimation has been done for three optimization levels: O0, O1 and O2. Average error is around 4.5% with minimum below 1% and maximum below 16%.

Results for second AES implementation – *AES\_P*, for all three platform configurations are presented in Table 9. The achieved average error is around 6% with minimum below 1% and maximum below 16%.

Table 10 shows results for *JPEG* test case. Timing estimation has been done for all three configurations:

**Table 8.** Timing estimation comparison for *AES\_G*.

			KeyExpansion	AddRoundKey	SubBytes	ShiftRows	MixColumns	Total
MB-00	Est.	[s]	5.81E-05	4.71E-06	5.95E-06	5.20E-07	5.60E-06	4.33E-04
	Act.	[s]	5.88E-05	4.98E-06	5.47E-06	5.05E-07	5.57E-06	4.08E-04
	Err.	[%]	-1.19	-5.40	8.76	2.97	0.54	6.13
MB-01	Est.	[s]	1.77E-05	1.38E-06	1.46E-06	2.80E-07	1.62E-06	1.22E-04
	Act.	[s]	1.72E-05	1.35E-06	1.41E-06	2.55E-07	1.48E-06	1.14E-04
	Err.	[%]	2.91	2.22	3.55	9.80	9.46	7.02
MB-02	Est.	[s]	1.83E-05	1.38E-06	1.38E-06	2.00E-07	1.47E-06	1.17E-04
	Act.	[s]	1.76E-05	1.42E-06	1.39E-06	1.90E-07	1.47E-06	1.15E-04
	Err.	[%]	3.98	-2.82	-0.72	5.26	<b>0.00</b>	1.74
ARM1-00	Est.	[s]	1.42E-05	1.01E-06	1.23E-06	8.28E-07	1.81E-06	1.03E-04
	Act.	[s]	1.38E-05	1.11E-06	1.18E-06	8.10E-07	1.65E-06	9.89E-05
	Err.	[%]	2.90	-9.01	4.24	2.22	9.70	4.15
ARM1-01	Est.	[s]	2.33E-06	1.67E-07	1.73E-07	3.93E-08	2.81E-07	1.68E-05
	Act.	[s]	2.29E-06	1.71E-07	1.58E-07	3.75E-08	2.82E-07	1.78E-05
	Err.	[%]	1.75	-2.34	9.49	4.80	-0.36	-5.62
ARM1-02	Est.	[s]	1.93E-06	1.35E-07	1.73E-07	4.60E-08	2.21E-07	1.46E-05
	Act.	[s]	2.02E-06	1.35E-07	1.58E-07	4.95E-08	2.11E-07	1.50E-05
	Err.	[%]	-4.46	<b>0.00</b>	9.49	-7.07	4.74	-2.67
ARM2-00	Est.	[s]	3.19E-04	2.37E-05	3.01E-05	3.93E-06	3.22E-05	2.28E-03
	Act.	[s]	3.17E-04	2.65E-05	2.83E-05	3.79E-06	3.07E-05	2.22E-03
	Err.	[%]	0.63	-10.57	6.36	3.69	4.89	2.70
ARM2-01	Est.	[s]	9.79E-05	5.18E-06	5.14E-06	2.57E-06	5.66E-06	4.99E-04
	Act.	[s]	9.96E-05	5.36E-06	5.32E-06	2.46E-06	5.21E-06	5.00E-04
	Err.	[%]	-1.71	-3.36	-3.38	4.47	8.64	-0.20
ARM2-02	Est.	[s]	9.79E-05	5.18E-06	5.14E-06	2.57E-06	5.66E-06	4.99E-04
	Act.	[s]	9.51E-05	5.24E-06	5.31E-06	3.04E-06	5.28E-06	5.05E-04
	Err.	[%]	2.94	-1.15	-3.20	<b>-15.46</b>	7.20	-1.19

**Table 9.** Timing estimation comparison for *AES\_P*.

			KeyExpansion	AddRoundKey	SubBytes	ShiftRows	MixColumns	Total
MB-00	Est.	[s]	6.69E-05	5.73E-06	6.51E-06	8.24E-07	6.14E-06	4.74E-04
	Act.	[s]	7.25E-05	5.64E-06	5.88E-06	8.80E-07	6.58E-06	4.72E-04
	Err.	[%]	-7.72	1.6	10.71	-6.4	-6.69	<b>0.42</b>
MB-01	Est.	[s]	1.81E-05	1.38E-06	1.47E-06	2.70E-07	1.48E-06	1.03E-04
	Act.	[s]	1.78E-05	1.37E-06	1.44E-06	2.50E-07	1.46E-06	1.15E-04
	Err.	[%]	1.68	0.73	2.08	8	1.37	-10.44
MB-02	Est.	[s]	1.74E-05	1.10E-06	1.22E-06	1.79E-07	1.29E-06	1.00E-04
	Act.	[s]	1.56E-05	1.11E-06	1.08E-06	1.80E-07	1.26E-06	9.23E-05
	Err.	[%]	11.54	-0.9	12.96	-0.56	2.38	8.34
ARM1-00	Est.	[s]	2.10E-05	1.30E-06	1.46E-06	3.09E-07	2.35E-06	1.34E-04
	Act.	[s]	2.23E-05	1.50E-06	1.54E-06	3.02E-07	2.53E-06	1.47E-04
	Err.	[%]	-5.83	-13.33	-5.2	2.32	-7.12	-8.84
ARM1-01	Est.	[s]	3.80E-06	2.43E-07	2.73E-07	5.89E-08	4.97E-07	2.59E-05
	Act.	[s]	3.78E-06	2.46E-07	2.51E-07	6.05E-08	4.56E-07	2.84E-05
	Err.	[%]	0.53	-1.22	8.77	-2.65	8.99	-8.8
ARM1-02	Est.	[s]	3.11E-06	2.22E-07	2.81E-07	6.81E-08	3.60E-07	2.29E-05
	Act.	[s]	3.16E-06	2.26E-07	2.62E-07	7.29E-08	3.48E-07	2.40E-05
	Err.	[%]	-1.58	-1.77	7.25	-6.58	3.45	-4.58
ARM2-00	Est.	[s]	3.41E-04	2.41E-05	2.16E-05	6.27E-06	3.51E-05	2.21E-03
	Act.	[s]	3.56E-04	2.69E-05	2.39E-05	5.41E-06	3.32E-05	2.26E-03
	Err.	[%]	-4.21	-10.41	-9.62	15.89	5.72	-2.21
ARM2-01	Est.	[s]	9.68E-05	5.12E-06	2.94E-06	2.05E-06	5.37E-06	4.37E-04
	Act.	[s]	8.86E-05	4.44E-06	2.96E-06	2.09E-06	4.73E-06	4.01E-04
	Err.	[%]	9.26	15.32	-0.68	-1.90	13.50	8.24
ARM2-02	Est.	[s]	9.63E-05	5.12E-06	2.94E-06	2.21E-06	4.89E-06	4.28E-04
	Act.	[s]	8.76E-05	4.44E-06	2.96E-06	2.59E-06	4.82E-06	3.99E-04
	Err.	[%]	9.93	<b>15.32</b>	-0.68	-14.70	1.45	7.27

*MB1*, *ARM1* and *ARM2*. Timing estimation was calculated for a single run of each individual procedure in JPEG compression algorithm: *CreateBlocks*, *Shift*, *DCT*, *ZigZag*, *HuffEncode* and *CreateImage*. Last column of the table contains results for the complete run of the JPEG algorithm. Application profiling of *HuffEncode* procedure required manual instrumentation and execution on a host PC using the exact same input as was used for test configurations because it is data-dependent and encoding result varies greatly in length between two different inputs. Overall, average

estimation error for JPEG is around 5% with the minimum below 1% and maximum below 17%.

To summarize, for all three test applications and for all three target platform configurations estimation accuracy remains approximately at the same level, with the average error around 5% and the maximum error below 17%. Estimation accuracy shows no significant degradation for any level of compiler optimization. Even for *ARM2* configuration, there is no deviation in error rate compared with results on the other two configurations. This particular configuration is more

**Table 10.** Timing estimation comparison for *JPEG*.

			CreateBlocks	Shift	DCT	ZigZag	HuffEncode	CreateImage	Total
MB-00	Est.	[s]	1.71E-03	6.96E-04	1.56E-03	7.59E-04	1.00E-02	4.35E-04	3.16E-02
	Act.	[s]	1.70E-03	7.17E-04	1.41E-03	7.41E-04	9.37E-03	4.61E-04	2.94E-02
	Err.	[%]	0.59	-2.93	10.64	2.43	6.72	-5.64	7.48
MB-01	Est.	[s]	2.96E-04	1.39E-04	4.28E-04	1.77E-04	2.75E-03	1.29E-04	1.31E-02
	Act.	[s]	2.80E-04	1.42E-04	4.40E-04	1.73E-04	2.83E-03	1.38E-04	1.25E-02
	Err.	[%]	5.71	-2.11	-2.73	2.31	-2.83	-6.52	4.8
MB-02	Est.	[s]	2.17E-04	1.12E-04	2.69E-04	1.50E-04	2.66E-03	1.13E-04	1.22E-02
	Act.	[s]	2.06E-04	1.14E-04	2.88E-04	1.45E-04	2.46E-03	1.22E-04	1.15E-02
	Err.	[%]	5.34	-1.75	-6.6	3.45	8.1	-7.38	6.09
ARM1-00	Est.	[s]	3.38E-04	1.12E-04	2.77E-04	9.09E-05	2.16E-03	7.68E-05	5.26E-03
	Act.	[s]	3.43E-04	1.09E-04	2.66E-04	9.41E-05	2.27E-03	7.13E-05	5.22E-03
	Err.	[%]	-1.46	2.75	4.14	-3.4	-4.85	7.71	0.77
ARM1-01	Est.	[s]	4.08E-05	1.97E-05	7.95E-05	2.01E-05	6.76E-04	1.75E-05	1.40E-03
	Act.	[s]	4.42E-05	2.08E-05	7.94E-05	2.11E-05	7.50E-04	1.67E-05	1.44E-03
	Err.	[%]	-7.69	-5.29	0.13	-4.74	-9.87	4.8	-2.78
ARM1-02	Est.	[s]	3.94E-05	1.97E-05	7.69E-05	1.61E-05	6.57E-04	1.48E-05	1.33E-03
	Act.	[s]	4.45E-05	2.06E-05	7.40E-05	1.75E-05	7.28E-04	1.51E-05	1.39E-03
	Err.	[%]	-11.46	-4.34	3.92	-8.0	-9.75	-1.99	-4.32
ARM2-00	Est.	[s]	8.97E-03	3.04E-03	1.22E-02	4.22E-03	5.98E-02	2.79E-03	1.30E-01
	Act.	[s]	8.36E-03	3.05E-03	1.24E-02	3.62E-03	5.88E-02	2.90E-03	1.27E-01
	Err.	[%]	7.3	-0.33	-1.67	<b>16.58</b>	1.7	-3.79	2.36
ARM2-01	Est.	[s]	1.34E-03	4.49E-04	2.81E-03	4.67E-04	4.71E-03	6.18E-04	1.78E-02
	Act.	[s]	1.42E-03	4.49E-04	2.70E-03	4.77E-04	4.23E-03	6.15E-04	1.71E-02
	Err.	[%]	-5.63	<b>0.00</b>	4.07	-2.1	11.35	0.49	4.09
ARM2-02	Est.	[s]	1.34E-03	4.53E-04	2.82E-03	4.70E-04	4.64E-03	6.18E-04	1.78E-02
	Act.	[s]	1.42E-03	4.56E-04	3.00E-03	4.97E-04	4.17E-03	6.13E-04	1.84E-02
	Err.	[%]	-5.63	-0.66	-6.0	-5.43	10.13	0.82	-3.26

sensitive to cache effects because processor communicates with a very slow memory. In case of inability to accurately capture cache hits – method would give overestimation, or in case of cache miss - underestimation. However, it must be noted that for all three test cases there was much larger chance of having a cache hit than a cache miss, because memory footprint of each of these applications remains within range of 50 KB – 250 KB. This means they fit well to cache size typical for embedded processors like ARM and likelihood for cache hits is much larger. On the other hand, all three test applications represent well, in both size and structure, common tasks for which embedded systems are used for: signal processing, vector and matrix operations, numeric calculations, search and sorting [22].

Comparing to results achieved by analytical methods, which have an average error in the range from 17% [8] to 20% [9,10], our results are better. They are, however, slightly worse than those obtained using simulation methods which achieve estimation error below 10% in worst cases, [3–7]. But compared to simulation methods, the strong point of our method is reusability of profiling results because both application and platform profiles can be reused in future. In that way, our method enables obtaining accurate source-level estimation in a shortened amount of time and helps close the gap between accuracy and speed.

## 6. Conclusion and future work

In this paper, we have proposed a method for source-level application execution time estimation in a heterogeneous computing environment based on a concept

named *elementary operations*. The method features a classification scheme used for identifying elementary operations in the source code. It enables profiling applications and platforms in a way which successfully handles compiler optimizations, pipeline and cache effects. This enables providing accurate application timing estimation while keeping the required effort input within reasonable limits. Based on the classification scheme, *ELOPS* benchmark is designed to measure execution time of each elementary operation type, within context like loops and sequences of operations, on real platforms.

Experimental results show an estimation error to be around 5% with maximum below 17%, which is comparable to best state-of-art simulation methods. The strong point of this method is that platform profiling needs to be done only once for each hardware configuration and these results are reused again later for any other application which is executed on the same hardware. The same applies to the application profiling: each application has to be profiled only once and the obtained profile can be used as is for any platform configuration at hand. Reusability of profiling results helps achieve better scalability when building platforms with multiple elements of the same type.

In the future, emphasis will be put on the full integration of the method into design space exploration process for heterogeneous multi-processor and multi-memory environments to eliminate the need to re-link and recompile source code using different development environments. Finally, application analysis will be improved by automating the instrumentation process in data-dependent parts of code.

## Notes

1. Available at: <https://gitlab.com/Frid/ELOPS.git>
2. both addition and subtraction operations are classified as ADD operation
3. Available at: <https://gitlab.com/Frid/ELOPS.git>

## Disclosure statement

No potential conflict of interest was reported by the authors.

## ORCID

Nikolina Frid  <http://orcid.org/0000-0002-2592-8206>

## References

- [1] Gajski DD, Vahid F. Specification and design of embedded hardware-software systems. *IEEE Design Test Comput.* 1995 Spring;12(1):53–67.
- [2] Keutzer K, Newton A, Rabaey J, et al. System-level design: orthogonalization of concerns and platform-based design. *IEEE Trans Computer-Aided Design Integrated Circuits Syst.* 2000 Dec;19(12):1523–1543. Available from: <http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=898830>
- [3] Chakravarty S, Zhao Z, Gerstlauer A. Automated, retargetable back-annotation for host compiled performance and power modeling. In: 2013 International Conference on Hardware/Software Codesign and System Synthesis, CODES+ISSS 2013 Sep. IEEE; 2013. p. 1–10. Available from: <http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=6659023>
- [4] Lin KL, Lo CK, Tsay RS. Source-level timing annotation for fast and accurate TLM computation model generation. In: Proceedings of the Asia and South Pacific Design Automation Conference, ASP-DAC; Jan. IEEE; 2010. p. 235–240. Available from: <http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=5419890>
- [5] Wang Z, Henkel J. Accurate source-level simulation of embedded software with respect to compiler optimizations. *Design, Automation & Test in Europe Conference & Exhibition.* 2012. p. 382–387.
- [6] Cheung E, Hsieh H, Balarin F. Fast and accurate performance simulation of embedded software for MPSoC. In: Proceedings of the Asia and South Pacific Design Automation Conference, ASP-DAC; Jan. IEEE; 2009. p. 552–557. Available from: <http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=4796538>
- [7] Gerin P, Hamayun MM, Pétrot F. Native MPSoC co-simulation environment for software performance estimation. In: Proceedings of the 7th IEEE/ACM International Conference on Hardware/Software Codesign and System Synthesis – CODES+ISSS '09; New York, New York, USA: ACM Press; 2009. p. 403. Available from: <http://dl.acm.org/citation.cfm?id=1629435.1629490>
- [8] Oyamada M, Wagner FR, Bonaciu M, et al. Software performance estimation in MPSoC design. In: 2007 Asia and South Pacific Design Automation Conference; Jan. IEEE; 2007. p. 38–43. Available from: <http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=4195993>
- [9] Palermo G, Silvano C, Zaccaria V. ReSPIR: a response surface-based pareto iterative refinement for application-specific design space exploration. *IEEE Trans Computer-Aided Design Integrated Circuits Syst.* 2009 Dec;28(12):1816–1829. Available from: <http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=5324029>
- [10] Cilaro A, Gallo L, Mazzocca N. Design space exploration for high-level synthesis of multi-threaded applications. *J Syst Archit.* 2013 Nov;59(10):1171–1183. Available from: <http://linkinghub.elsevier.com/retrieve/pii/S1383762113001537>
- [11] Roloff S, Hannig F, Teich J. Fast architecture evaluation of heterogeneous MPSoCs by host-compiled simulation. In: Proceedings of the 15th International Workshop on Software and Compilers for Embedded Systems – SCOPES '12; New York, New York, USA. ACM Press; 2012. p. 52–61. Available from: <http://dl.acm.org/citation.cfm?doid=2236576.2236582>
- [12] Zhao Z, Gerstlauer A, John LK. Source-level performance, energy, reliability, power and thermal (perpt) simulation. *IEEE Trans Computer-Aided Design Integrated Circuits Syst.* 2017 Feb;36(2):299–312.
- [13] Abdi S, Schirner G, Hwang Y, et al. Automatic TLM generation for early validation of multicore systems. *IEEE Design Test Comput.* 2011 May;28(3):10–19. Available from: <http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=5620889>
- [14] Guo Q, Chen T, Chen Y, et al. Microarchitectural design space exploration made fast. *Microprocess Microsyst.* 2013 Feb;37(1):41–51. Available from: <http://linkinghub.elsevier.com/retrieve/pii/S0141933112001433>
- [15] Javaid H, Ignjatovic A, Parameswaran S. Performance estimation of pipelined multiprocessor system-on-chips (MPSoCs). *IEEE Trans Parallel Distrib Syst.* 2014 Aug;25(8):2159–2168. Available from: <http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=6636892>
- [16] Flaskamp M, Sievers G, Ax J, et al. Performance estimation of streaming applications for hierarchical MPSoCs. In: Proceedings of the 2016 Workshop on Rapid Simulation and Performance Evaluation Methods and Tools – RAPIDO '16; New York, NY. ACM Press; 2016. p. 1–6. Available from: <http://dl.acm.org/citation.cfm?doid=2852339.2852342>
- [17] Altenbernd P, Gustafsson J, Lisper B, et al. Early execution time-estimation through automatically generated timing models. *Real-Time Syst.* 2016 Nov;52(6):731–760. Available from: <http://link.springer.com/10.1007/s11241-016-9250-7>
- [18] Frid N, Ivošević D, Sruk V. Performance estimation in heterogeneous MPSoC based on elementary operation cost. In: 2016 39th International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO); May. IEEE; 2016. p. 1202–1205. Available from: <http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=7522322>
- [19] Ivošević D, Sruk V. Unified flow of custom processor design and fpga implementation. In: Eurocon 2013 July. 2013. p. 1721–1727.
- [20] NIST. Advanced encryption standard (aes) – fips 197, 2001. Cited 2016-12-10. Available from: <http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf>
- [21] Wallace GK. The JPEG still picture compression standard. *Commun ACM.* 1991 Apr;34(4):30–44. Available from: <http://portal.acm.org/citation.cfm?doid=103085.103089>
- [22] Bui BD, Caccamo M, Sha L, et al. Impact of cache partitioning on multi-tasking real time embedded systems. In: 2008 14th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications; Aug; 2008. p. 101–110.