Comparing DSP Software Performance Prediction Models at Source Code Level — From Analytical to Statistical

Erh-Wen Hu[1], Weihua Liu[1], Bogong Su[1*], Jian Wang[2]

[1] Dept. of Computer Science, William Paterson University, Wayne, NJ, USA
[2] Mobile Broadband Software Design, Ericsson, Ottawa, ON, Canada

*Corresponding author: Tel: 1 973 720 2979; email: sub@wpunj.edu
Manuscript submitted September 25, 2018; accepted January 23, 2019.
doi: 10.17706/jsw.14.6. 247-256

Abstract: Efficient performance prediction at source code level is essential in reducing the turnaround time of software development, particularly when the source code is subject to changes due to modification of problem specification. In this paper, we investigate and compare five performance prediction models from practical standpoint to determine the usefulness of these models. To verify the effectiveness of these models, we select a set of functions from PHY DSP Benchmark and TIC64 DSP processor for experiment. Comparing the predicted performance to the actual measured execution time, we observed that the relative prediction error generated from two of the five models are low and can thus be used for practical purposes.

Key words: Performance prediction, source code level, analytical model, statistic model.

1. Introduction

Performance prediction is an essential phase of DSP software development. It can be carried out at either the source code or lower level. Prediction at source level is much faster because it does not require the code to be compiled and executed. Thus, performance prediction at source code level helps reduce the turnaround time of software development especially when the source code must be modified due to changes in problem specification [1]-[20].

Over the past few years, we have separately investigated and developed five different models to estimate the execution time of DSP software at source code level. The formulations of these models are presented in section 2. To determine the effectiveness of these models in practice, we need to compute the predicted execution time of practical DSP application software and compare the computed result with experimentally measured data.

To this end, we have chosen eight most frequently used functions from the Long Term Evolution or LTE Uplink Receiver, a major part of the PHY benchmark [10] as the testing set of samples. PHY is an open-source benchmark developed by Chalmers University of Technology of Sweden and Ericson. It provides a realistic implementation of the baseband processing for an LTE mobile base station. Its source code is written in C and can be freely downloaded from [19]. In choosing the set of samples for the testing set, we used *gprof*, a popular profiling tool in UNIX, to make sure that the selected set is indeed the most frequently executed kernel functions in the PHY benchmark.

In Section 2 below, we provide a summary description of our five prediction models. Section 3 discusses the experimental procedures including how experimental results are gathered and organized. Section 4 discusses how the experimental results are used as a basis to predict the performance of whole system, i.e., the entire PHY benchmark application. Sections 5 presents related work reported by others. Sections 6 discusses the factors that affect the effectiveness of the five models.

2. Prediction Models

In this section, we discuss our five source code level prediction models presented in this paper, which includes a simple analytical model, an enhanced version of the simple analytical model, a precise analytical model taking into consideration the hardware-specific features, a statistical model using linear regression, and a comprehensive model combining both analytical and statistical models.

All five models focus on predicting the execution time of loops because loop execution alone predominates the total execution time of the entire DSP application. In fact, as reported in [13], loop execution takes up 70% execution time of the selectable mode vocoder application.

2.1. Simple Analytical Model

The simple analytical model is based on our earlier work on DSP processor [12]. We use a simple formula to calculate the loop execution time in terms of the clock cycles. Considering a two-level nested loops, the predicted execution time or *PET* can be expressed as follows:

$$\mathrm{PET} = N_{outer} * (L_{outer} + N_{inner} * L_{inner}) \tag{1}$$

where N_{inner} and N_{outer} are the numbers of iterations of the inner and outer loops; L_{inner} and L_{outer} are the numbers of statements in the inner loop and the outer loop, respectively.

2.2. Enhanced Analytical Model

Compared to the measured values, we observed that the simple analytical model tends to under-predict the performance for most of the samples being investigated. The major reason is that the inner loops of those samples make calls to some small complex functions which incur extra execution time not fully accounted for in the simple analytic model. If the assembly code of a simple sample is known, we can find the ratio of its inner loop body length in assembly code to the length of the source code. This ratio can then be used to adjust the predicted time of those under-predicted testing samples. For example, in our experiment we select a simple loop from sample #31 *cholsolve_4xX_complex* and find the ratio equals 3. We then multiply L_{inner} by 3 in (1) to compute the PET or predicted execution time of those testing samples that under-predict the performance.

2.3. Precise Analytical Model

Based on our earlier work on source level loop optimization [11] we proposed a precise analytical model for performance prediction, which is machine-dependent and needs detailed hardware information such as the resource limitation of parallel function units and the latencies of instructions. With the understanding that the source code is written in C, the algorithm of the model is listed below:

1) Decomposing C statements in the innermost loop to DSP operations.
2) Building the data dependency graph DDG of DSP operations under the limitation of resources and instruction latencies.
3) Optimizing the inner loop body based on DDG with list scheduling or source level software pipelining [11].

4) Using the length of optimized loop body to replace L_{inner} in simple analytical model to calculate the predicted execution time.

Fig. 1 presents an example of TI C64 DSP processor. The latencies of its memory load, multiply, and branch instructions are 5, 2, and 6 clock cycles respectively; the rest of instructions executes in one clock cycle. Fig. 1(a) shows the source code of a simple loop in chest function of PHY, Fig. 1(b) and (c) show the DSP operations inside the loop body from the C statements and the result of list scheduling. Fig. 1(d) is the result after source level software pipelining in which L_{inner} = 1.

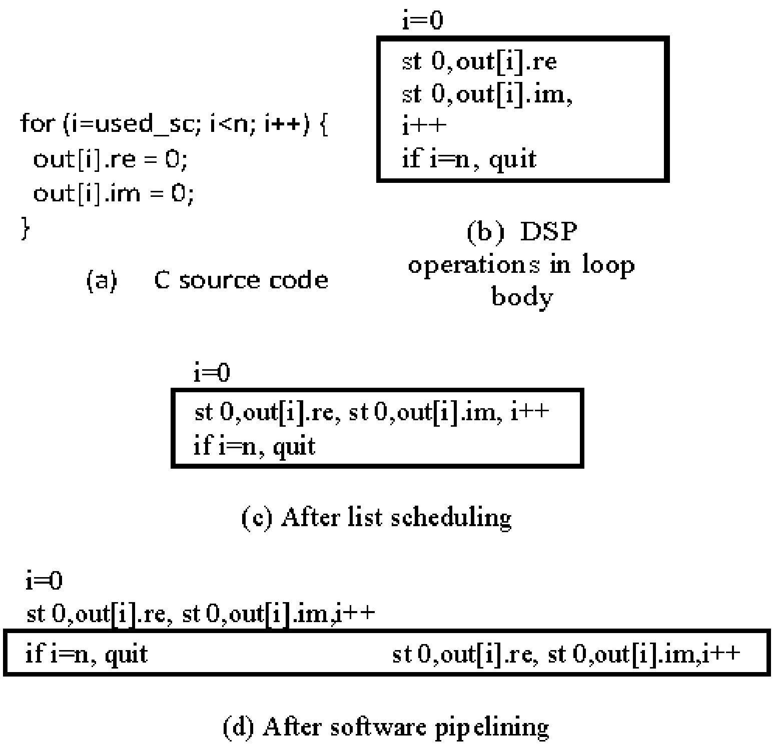


Fig. 1. Example of precise analytical model.

2.4. Statistic Model

Our statistical model uses the popular statistic tool IBM SPSS-23 to perform multiple linear regression [5]. SPSS-23 generates (2) below to predict the execution time for each testing sample.

$$PET = b_0 + \sum_{j=1}^{p} b_j X_j \tag{2}$$

where *PET*, the predicted execution time is the dependent variable; X_1, X_2, ... X_p are independent variables representing attributes of the corresponding testing sample; and b_0, b_1, b_2, ... b_p are coefficients generated from attributes of the training samples by SPSS. The attributes of all training samples are listed in Table 1.

We take all five functions with different input data from the tele-communication group of EEMBC [18] and all eight functions of SMV benchmarks [4] as the samples of training set. EEMBC benchmark is an industry standard for embedded processors and software developed by the Embedded Microprocessor Benchmark Consortium, a non-profit organization [18]. SMV benchmark is developed by us in 2006 which consists of eight kernels chosen from the Selectable Mode Vocoder (SMV) application program for 3G wireless communications. The testing set contains eight samples from PHY same as we use for analytical models.

We gather 17 static attributes from the source code of the training and testing sets as shown in Table 1.

Table 1. Attributes of Training and Testing Samples

	No.	Function name	Benchmark	Attriburtes																	Execution Time
				No. of statements	No. of iterations in outer	No.of iterations in inner	No. of statements in	No. of statements in	No. of loops	levels of nested loops	total iterations	No. of Complex Var	CC*	function calls	branches	No. of inputs	No. of outputs	Sizes of input data (byte)	Sizes of output data (byte)	$N_{inner} * L_{inner}$	
Training set	1	Autocorrelation - Data1	EEMBC	5	8	12	3	2	1	2	100	0	3	0	0	2	1	16	8	200	509
	2	Autocorrelation - Data2		5	16	1017	3	2	1	2	16272	0	3	0	0	2	1	1024	16	32544	49193
	3	Autocorrelation - Data3		5	32	485	3	2	1	2	15520	0	3	0	0	2	1	500	32	31040	47297
	4	Convolutional Encoder - Dat		13	1024	5	5	3	2	3	5120	0	5	0	1	5	1	512	512	15360	42014
	5	Convolutional Encoder - Dat		13	1024	4	5	3	2	3	4096	0	5	0	1	5	1	512	512	12288	41501
	6	Convolutional Encoder - Dat		13	1024	3	5	3	2	3	3072	0	5	0	1	5	1	512	512	9216	40297
	7	Fixed-point Bit Alloc. - Data2		17	80	256	4	11	1	2	20480	0	7	0	3	6	1	256	256	225280	159448
	8	Fixed-point Bit Alloc. - Data3		17	70	20	4	11	1	2	1400	0	7	0	3	6	1	20	20	15400	20968
	9	Fixed Point Bit Alloc. - Data6		17	130	100	4	11	1	2	13000	0	7	0	3	6	1	100	100	143000	114856
	10	FFT/IFFT - Data1		22	8	128	5	14	3	3	1024	0	5	0	0	2	2	2048	5500	14336	45218
	11	FFT/IFFT - Data2		22	8	128	5	14	3	3	1024	0	5	0	0	2	2	2525	5635	14336	45218
	12	FFT/IFFT - Data3		22	8	128	5	14	3	3	1024	0	5	0	0	2	2	2717	3549	14336	45218
	13	Viterbi Decoder - Data1		167	42	128	9	48	2	2	5376	0	17	8	3	1	1	1196	1196	258048	233467
	14	Viterbi Decoder - Data2		167	42	128	9	48	2	2	5376	0	17	8	3	1	1	2109	2109	258048	233478
	15	Viterbi Decoder - Data3		167	42	128	9	48	2	2	5376	0	17	8	3	1	1	2034	2034	258048	233483
	16	Viterbi Decoder - Data4		167	42	128	9	48	2	2	5376	0	17	8	3	1	1	2395	2395	258048	233483
	17	FLT_filterAP_fx	SMV	11	170	9	8	3	1	2	1530	0	3	0	0	4	2	724	4	4590	3197
	18	LPC_Chebps_fx		16	1	3	8	7	1	1	3	0	1	0	0	3	1	16	2	21	56
	19	LPC_autocorrelation_fx		9	17	240	3	7	1	2	4080	0	4	2	0	5	1	622	34	28560	7965
	20	FCS_Excit_Enhance_fx		31	80	20	1	6	3	2	1600	0	12	1	0	8	1	398	340	9600	12857
	21	LSF_Q_New_ML_search_fx		30	1152	10	14	4	5	3	11520	0	32	0	3	11	3	2900	380	46080	45729
	22	c_fft_fx		34	6	128	4	16	3	3	768	0	15	0	1	3	1	512	256	12288	10565
	23	FCB_add_sub_contrib_phi		19	1	8	9	10	1	1	8	0	6	0	2	9	1	3458	4	80	243
	24	PIT_LT_Corr_Rmax_fx		45	68	80	31	2	3	3	5440	0	10	0	0	7	3	848	8	10880	21079
	31	cholsolve_4xX_complex	PHY	34	2	3	5	23	2	3	6	3	11	8	0	3	1	128	64	138	702
Testing set	25	mf		10	1	1200	0	9	2	1	1200	2	3	1	0	5	1	9600	4800	10800	70431
	26	soft_demap		18	43200	6	1	14	1	2	259200	2	8	2	1	5	1	4800	4800	3628800	10374917
	27	ifft		40	10	700	17	17	1	3	7000	2	5	0	0	3	1	2400	2400	119000	110922
	28	chest		10	3	300	0	6	2	1	900	10	4	0	0	5	1	4800	4800	5400	3423
	29	fft		38	10	700	17	17	1	3	7000	4	5	0	0	3	1	2400	2400	119000	109975
	30	ant_comb		29	1	1200	0	29	1	1	1200	1	8	7	0	4	1	4800	4800	34800	479246
	32	matrix_a_a_herminte_plus_I		60	3	3	1	50	3	1	9	8	21	11	0	5	1	128	64	450	3143
	33	matrix_mult_4xX_complex		27	3	3	1	26	1	2	9	3	8	7	0	4	1	128	64	234	1569

Below is the list and description of those attributes.

1) Number of statements in a sample.
2) Number of statements in the inner loops.
3) Number of statements in the outer loop, which excludes the statements in the inner loops.
4) Number of iterations of the inner loops.
5) Number of iterations of the outer loop.
6) Number of loops, which includes all loops in the sample.
7) Number of levels of nested loop.
8) Number of total iterations, which equals the number of iterations of the outer loop * the number of iterations of inner loop.
9) Number of complex variables of inner loop.
10) CC*, a new metric we defined in 2006 [4] to measure code complexity based on Cyclomatic Complexity CC. A program can be graphically depicted by a control-flow graph. In a control-flow graph, CC = e-nn+np+1, where e, nn and np denote the number of edges, nodes, and connected components, respectively. We extended the definition of CC by taking into consideration of instruction level parallelism of the complex nested levels of loops and conditional branches. The

modified cyclomatic complexity CC* equals the sum of numbers of loops and branches in the program, plus the number of nested levels of loops and branches.

11) Total number of function calls inside a sample.
12) Total number of branches inside a sample.
13) Total number of input variables of a sample.
14) Total number of output variables of a sample.
15) Total data size of all input variables measured in bytes.
16) Total data size of all output variables measured in bytes.
17) $N_{inner} * L_{inner}$, the major parameter used to predict performance in analytical models.

2.5. Comprehensive Prediction Model

We proposed an approach [5], which is referred to as the *comprehensive model* in this paper. The model combines the statistical model with the analytical model described in sections 2.4 and 2.1, respectively, and it uses some heuristics, as described below:

1) Adding to the model a new independent variable *Repeating Times RT* as an additional attribute. Table 1 shows that the execution time of some samples, such as samples 1, 18, and 23, in the training set is very small relative to that of other samples. By repeating *RT* times, the wide range of variation among samples' execution time can be reduced. An iterative algorithm has been designed to determine the best value of *RT* for the training set [5]. With this new attribute, SPSS generates (3) which can be used to predict the execution time of testing samples.

$$APET_k=(b_0+\sum_{j=1}^{p} b_j X_{jk})+b_{RT}RT_k \quad (3)$$

where $APET_k$ is the adjusted predict execution time of testing sample_k generated by SPSS, X_{jk} is the value of jth-attribute in that sample. The value of $N_{inner} * L_{inner}$ from analytical model are used as the values for RT_k to determine the execution time of the k^{th} sample of the testing set by using (4) below.

$$PET_k = APET_k/RT_k \quad (4)$$

From the source code and assembly code of PHY kernel functions we notice that their kernel functions are quite different from the typical DSP functions in EEMBC and SMV, e.g. there is no complex variables in EEMBC and SMV functions. For this reason, we select a typical PHY kernel function, sample #31 with three complex variables, as a new member of the training set.

2) From the analytical model, we realize that some function calls and some optimization technique such as software pipelining can have large impact on execution time. To compensate these factors, we have designed and employed two heuristics to adjust RT_k.

3. Experiments

We use TI C64 DSP processor as the hardware platform for our experiments. All samples in the training and testing sets are compiled by TIC64 compiler and run on TIC64 simulator. The execution time as the performance metric is gathered from that simulator in terms of number of CPU clock cycles. We use relative errors *RE* and average absolute relative error *ARE* to describe the prediction accuracy. RE and ARE are defined as follows:

Table 2. Experiment Results of Five Models

Testing Sample		Prediction Modles				
Name	No.	simple analytic	enhanced analytic	precise analytic	statistic	compre-hensive
mf	25	-71.0%	-37.0%	-18.2%	-37.8%	-34.5%
soft_demap	26	-63.8%	6.2%	15.8%	-75.9%	-9.4%
ifft	27	7.3%	7.3%	23.9%	-51.3%	5.5%
chest	28	57.8%	57.8%	-3.0%	689%	-22.4%
fft	29	8.2%	8.2%	25.0%	-52.8%	7.9%
ant_comb	30	-92.7%	-78.2%	-47.7%	-99.8%	0.0%
matrix_a_a_herminte_	32	-83.4%	-50.2%	-34.8%	-1617%	-28.3%
matrix_mult_4xX_com	33	-79.9%	-40.1%	-7.1%	-1897%	-25.5%
ARE		58.0%	35.6%	21.9%	565%	16.7%

$$RE = [(Predicted - Actual)/Actual] \times 100\% \tag{5}$$

$$ARE = (\textstyle\sum_{i=1}^{M} |Relative\ error_i|)/M \tag{6}$$

In (5) and (6), the actual values are measured in experiments and M is the total number of testing samples, which equals 8. Table 2 presents the experimental results in terms of RE and ARE of all five models.

4. Predict the Performance of Whole System with Gprof

We use *gprof*, a popular UNIX profiling tool, for our experiments. In order to use gprof, we need to enable profiling while compile our programs, and then execute the programs on the host computer to produce the profiling data. Finally run gprof on the profiling data file to produce the analysis information, which includes function call graph and an overview of the execution time for all the functions. We use the *called times* of the major functions generated by gprof and the predicted clock cycles for these functions to compute the predicted performance of the whole system. Equation (7) shows how the predicted execution time of all major functions is computed. Comparing it with the actual measured system execution time, we can calculate the relative prediction errors for all five models. The results for both the predicted execution time of all major functions and *the* relative prediction errors for all five models are presented in Table 3.

The predicted execution time of all major functions

$$= \quad \textstyle\sum_{i=1}^{n} predicted\ execution\ time_i * called\ times_i \tag{7}$$

where called $times_i$ is the number of time the i^{th} function gets called.

5. Related Work

Reference [6] uses analytical modeling to predict the execution times of parallel programs. Reference [21] builds DDGs of basic blocks of DSP assembly code generated from C statements, uses list scheduling to determine their clock cycles, and then multiplies the number of execution time of those C statements collected by a host computer to predict the performance of a DSP processor. Based on programs ran on many different machines, [16] gathers program characteristics such as instruction mix, distribution of operands, and basic block size, then makes use of the squared Euclidean distance to identify the similarities among programs and uses the result to predict the performance of these programs on different hardware. Reference [3] collects certain static and dynamic microarchitecture-independent characteristics including instruction mix and cache miss rate from SPEC 2000 benchmark programs and uses their similarity to predict program performance on different hardware using three different analytical models.

Many articles use various statistical approaches to predict the performance of software on target computers. For example, [20] uses various types of instructions as attributes for multiple linear regression, [17] uses a regression-tree-based modeling, and [1] uses a nonlinear regression model. References [10] and [8] use various machine training approaches to predict software performance on multi-core processors. Reference [7] uses both static attributes such as numbers of different types of instructions and dynamic attributes such as number of cache misses for regression. Reference [2] proposes cross-architecture performance prediction. It is a machine training based technique using both static and dynamic attributes from many programs from some/different benchmarks. They use R package to implement the regression and predict the execution time on a GPU from a single thread CPU with an average error of 26.9%. Using a constrained locally sparse linear regression algorithm, [15] proposes a training-based analytical cross-platform performance prediction.

Table 3. Performance Prediction of Whole System

Function	Actual measured values			Simple anlytic model		Adjusted analytic model		Precise analytic model		Comprehensive model	
	Actual execution time	called times from gprof	Total execution time[1]	predicted execution time	Total execution time in whole system	predicted execution time	Total execution time in whole system	predicted execution time	Total execution time in whole system	predicted execution time	Total execution time in whole system
mf	70,431	24	1,690,344	20,400	489,600	44,400	1,065,600	57,620	1,382,880	46,122	1,106,917
soft_demap	10,374,917	1	10,374,917	3,758,484	3,758,484	11,016,252	11,016,252	12,009,870	12,009,870	9,399,476	9,399,476
ifft	110,922	60	6,655,320	119,000	7,140,000	119,000	7,140,000	137,486	8,249,160	117,073	7,024,391
chest	3,423	24	82,152	5,400	129,600	5,400	129,600	3,322	79,728	2,656	63,749
fft	109,975	24	2,639,400	119,000	2,856,000	119,000	2,856,000	137,480	3,299,520	118,700	2,848,802
ant_comb	479,246	36	17,252,856	34,803	1,252,908	104,403	3,758,508	250,806	9,029,016	479,124	17,248,473
matrix_a_a_hermint	3,143	2,400	7,543,200	522	1,252,800	1,566	3,758,400	2,048	4,915,200	2,253	5,406,724
matrix_mult_4xX_c	1,569	2,400	3,765,600	316	758,400	940	2,256,000	1,458	3,499,200	1,169	2,806,475
total			50,003,789		17,637,792		31,980,360		42,464,574		45,905,008
Relative prediction error comparing with total 8 functions					-64.73%		-36.04%		-15.08%		-8.20%
Relative prediction error comparing with whole PHY system					-67.50%		-41.70%		-21.75%		-15.41%

note:1 Total measured CK of 8 kernel funcitons = 50,003,789. Actual execution time of whole PHY = 54,266,642.

6. Discussion

1) Table 2 summarizes the experimental results of all five models investigated in this paper. From it we observe that the simple analytical model has large under-prediction errors, which is caused by large overhead in assembler code especially when those samples involve many function calls. However, we also observe that some samples have over-prediction problem. This is due to the shortened inner loop body of assembly code generated by the loop optimization of compiler. The enhanced analytical model can reduce the under-prediction errors if we can get the length ratio between L_{inner} of assembly code and source code.
2) To further improve the accuracy of performance prediction, even the precise analytical model needs detailed information of hardware. Nevertheless, its predicted errors, as shown in Tables 2 and 3, is much improved to reach the acceptable level.

3) Table 2 shows that the prediction errors of the statistical model are very large because the number of training samples used in the paper is small. Also, the values of attributes among different samples vary over a wide range.
4) The comprehensive model combines statistical and analytical approaches; it also adds some sample from PHY to the training set. As shown in Table 3, its prediction error becomes reasonably small to be in practice to predict the performance of the entire PHY system; this is shown in Table 3. It is the best among the five models investigated in this paper for predicting the DSP performance at source level.
5) Both the precise analytical and the comprehensive models can be combined with other works [14] to predict the performance of source code running on different hardware.
6) Currently we are working on a project using machine learning techniques to predict DSP performance at source code level.

Acknowledgment

Su would like to thank the ART awards of William Paterson University. We would like to thank CS major students Adam Herzog, Eduardo Avila, and Beata Zaluska of William Paterson University for their testing data.

References

[1] Garland, J., & Bradley, E. (2013). On the importance of nonlinear modeling in computer performance prediction, *Advances in Intelligent Data Analysis XII, Lecture Notes in Computer Science*, Springer.

[2] Ardalani, N., Lestourgeon, C., Sankaralingam, K., & Zhu, X. (2015). Cross-architecture performance prediction (XAPP) using CPU code to predict GPU performance. *Proceedings of the MICRO-48*.

[3] Hoste, K., *et al*. (2006). Performance prediction based on inherent program similarity. *Proceedings of the* PACT'06.

[4] Hu, E., Ku, C., Russo, A., Su, B., & Wang, J. (2006). New DSP benchmark based on selectable mode vocoder (SMV). *Proceedings of the 2006 International Conference on Computer Design* (pp. 175-181).

[5] Hu, E., Su, B., & Wang, J. (2017). Software performance prediction at source level. *Proceedings of the SERA2017*.

[6] Kuhnemann, M., Rauber, T., & Runger, G. (2004). A source code analyzer for performance prediction. *Proceedings of the 18th Parallel and Distributed Processing Symposium*.

[7] Lee, S., & Wu, C. (2017). Performance characterization, prediction a and optimization for heterogeneous systems with multi-level memory interference. *Proceedings of the HSWC*.

[8] Namin, A., Sridharan, M., & Tomar, P. (2010). Predicting multi-core performance: A case study using solaris containers. *Proceedings of the IWMSE'10*.

[9] Rai, J., Negi, A., Wankar, R., & Nayak, K. (2010). Performance prediction on multi-core processors. *Proceedings of the Int. Conf. on Computational Intelligence and Communication Networks*.

[10] Sjalander, M., McKee, S., Brauer, P., Engdal, D., & Vajda, A. (2012). An LTE uplink receiver PHY benchmark and subframe-based power management. *Proceedings of the 2012 IEEE International Symposium on Performance* Analysis *of Systems and Software*.

[11] Su, B., Wang, J., & Esguerra, A. (1999). Source-level loop optimization for DSP code generation. *Proceedings of the ICASSP 99*.

[12] Su, B., *et al*. (2003). A new source-level benchmarking for DSP processors. *Proceedings of the ISPC2003*.

[13] Su, B., *et al*. (2005). Analysis of loop behavior of selectable mode Vocoder (SMV) and its impact of instruction level parallelism. *Proceedings of the GSPx 2005.*

[14] Ould-Ahmed-Vall, E., Woodlee, J., Yount, C., & Doshi, K. (20070. On the comparison of regression algorithms for computer architecture performance analysis of software applications.

[15] Zheng, X., Ravikumar, P., John, L., & Gerstlauer, A. (2015). Leaning-based analytical cross-platform performance prediction. *Proceedings of the 2015 SAMOS*.

[16] Saavedra, R., & Smith, A. (1996). Analysis of benchmark characteristics and benchmark performance prediction. *ACM Tran. on Computer System*, *14(4)*.

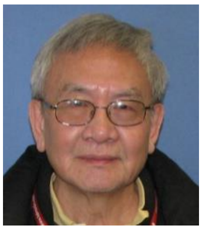
[17] Li, B., Rng, L., & Ramadass, B. (2009). Accurate and efficient processor performance prediction via regression tree based modeling. *Journal of System Architecture*, *55*, 457-467.

[18] Tele bench, an eembc bench. Retrieved from: http://www.eembc.org/benchmark/telecom_sl.php

[19] LTE uplink receiver PHY benchmark. (2011). http://sourceforge.net/projects/lte-benchmark

[20] Martin, G. (2006). Statistically based estimate of embedded software execution time.

[21] Pegatoquel, A., *et al*. (2003). Assembly code performance evaluation apparatus and method. US Patent, US 6598221 B1.



Erh-Wen Hu received his bachelor of science degree in electrical engineering from Cheng Kung University, Taiwan in 1967; the master's degree in materials science from SUNY US in 1972 and the master's degree in EE from University of Cincinnati US in 1973; and the Ph.D. degree from Polytechnic Institute of NY US in 1976.

From 1976 to 1978 he worked at Polytechnic Institute of NY as a research assistant professor. In 1978 he joined William Paterson University of New Jersey where he was one of the founders of a new Department of Computer Science in the early 1980s and served two terms as the Chairperson of the Department. He is currently a professor in Computer Science Department of William Paterson University of New Jersey US. His current research interests include instruction level parallelism, hardware and software optimization and benchmarking, software performance evaluation and prediction on DSP processors. He has authored or coauthored 50 research publications.

Weihua Liu earned her Ph.D. in computer science from University of Kentucky in 2016. Her research interests include data analysis and machine learning applications, cryptography and security, information security. She joined William Paterson University in New Jersey in 2016. She is currently an assistant professor in computer science Department of William Paterson University of New Jersey US.

Bogong Su got his bachelor degree in computer technology from Tsinghua University China in 1959 and Ph. D equivalency certification by Spantran Services, US in 1991.

He had worked with Tsinghua University China from 1959 to 1990, and City University of New York, NY US from 1991 to 1994. In 1994, he joined William Paterson University of New Jersey US, and he is currently a professor in computer science Department of William Paterson University of New Jersey. His research interests include instruction level parallelism, compiler optimization, software pipelining and de-pipelining, distributed artificial intelligence, DSP and embedded systems, benchmarking, system performance evaluation and prediction, and machine learning techniques. He has authored or coauthored more than 100 research publications.

Dr. Bogong Su is a life senior member of IEEE.

Jian Wang received his bachelor, master and Ph. D degrees of computer science from Tsinghua University China in 1986, 1989 and 1991, respectively.

He had done research and development work in INRIA France, from 1992 to 1993; Technology University of Vienna Austria, from 1993 to 1995; McGill University Canada, from 1995 to 1997; and BNR/Nortel Canada from 1997 to 2009. Since 2010, he has been working for Ericsson Canada. His research interests include software optimization, performance evaluation and prediction, benchmarking, instruction level parallelism and DSP/embedded systems. He has authored or coauthored more than 50 research publications.

Dr. Jian Wang was elected to IEEE senior member in 2004.