Sequential Composition of Execution Time Distributions by Convolution

Marcelo Santos
Björn Lisper
School of Innovation, Design and Eng.
Mälardalen University, Sweden.
email: mcsantos75@gmail.com, bjorn.lisper@mdh.se

Verônica Lima
Department of Statistics
Federal University of Bahia, Brazil
email: cadena@ufba.br

George Lima
Department of Computer Science
Federal University of Bahia, Brazil
email: gmlima@ufba.br

***Abstract*—Embedded real-time systems are increasingly being assembled from software components. This raises the issue how to find the timing properties for the resulting system. Ideally, these properties can be inferred from the properties of the components: this is the case if the underlying timing model is *compositional*. However, compositional timing models tend to provide a simplified view. An important question is then: when is a compositional model "accurate enough" to meet the requirements for an analysis that is based on the model?**

In this paper we consider a simple, statistical compositional model for execution time distributions of sequentially composed components, which assumes that the distributions of the underlying random variables are independent. This assumption is only approximately correct in general, as dependencies can appear due to both software and hardware effects. We have made an experimental investigation of how hardware features affect the validity of the timing model. The result is that for the most part, the effect of hardware features on the validity of the model is small. The hardware feature with the strongest influence in the experiment was the reorder buffer, followed by branch table associativity, L2 cache size, and out-of order execution.

I. Introduction

Many embedded systems have requirements on the timing properties. These can be requirements on the worst-case timing, but also on "softer" properties such as throughput, performance, and "soft" worst-case behaviour where the requirements may be expressed using probabilities for overruns.

Analysis of timing properties is often divided into *system-level* analysis and *code-level* analysis. An example of worst-case system-level analysis is response-time analysis (RTA). On code level, the archetypal worst-case analysis is Worst-Case Execution Time (WCET) analysis. However, also other analyses are concievable. For throughput and performance, it is of interest to characterize the timing behavior also for other than the worst case. A possibility is to consider the execution time to be a random variable, and characterize the timing behavior by its probability distribution. If we know the probability distributions on code level, for individual tasks, then the distributions for system-level timings can be estimated from these [1]. Soft real-time requirements that are expressed in probabilistic terms, like the probability of a violated deadline, can then be decided from these distributions.

Component-based software development [2] is a strong trend that is gaining ground also for embedded systems. Software is then built out of predefined components, which are combined to achieve the desired functionality. For such systems it is interesting to derive properties of the system from the properties of its parts. A model where properties can be derived in this fashion is called *compositional*. Let composition of components be expressed by the binary operator $\otimes$, consider the property $p(c)$ of components c, and let $\oplus$ be a binary operator composing properties. If $p(c_1 \otimes c_2) = p(c_1) \oplus p(c_2)$, then the model for properties $p(c)$ is compositional. If $\oplus$ is inexpensive to evaluate, then properties for a system can be quickly derived from the properties of its parts using the model.

However, compositional models are usually approximate: the simpler they are, the more approximate they tend to be. In a given situation it is important to know whether an estimate of a property, which is derived using a compositional model, is accurate enough. Thus, compositional models need to be investigated w.r.t. precision.

We consider code-level component models, where function block-like components are used to implement sequential tasks. Since the tasks are sequential, the codes for the components will be executed in some sequential order. Thus, we consider *sequential composition* of components $c_1; c_2$, expressing a composite component where c_1 and c_2 are executed in sequence. We assume a probabilistic timing model where the timing property for a component c is its execution time distribution $T(c)$. Under the assumption of an underlying additive timing model for sequential composition, and independence of execution times for different components, the distribution $T(c_1; c_2)$ can be calculated from $T(c_1)$ and $T(c_2)$ by *convolution* [3]. This provides a simple, compositional timing model for execution time distributions, which can be used to quickly estimate the execution time distributions for tasks from the distributions of its components.

However, this timing model rests on the assumption that the execution times for components are independent. This is in general not true: covariations in execution time can arise both on software level, for instance through shared, or dependent inputs to the components, and on hardware level through the influence of the hardware state on the execution time of instructions. Especially for modern high-end processors, with features such as caches, parallel functional units, and out-of-order pipelines, this influence can be strong. We therefore need to know how strong the covariations of execution times are that it can cause in practice.

We have made an experimental evaluation of how hardware architecture features affect the validity of the compositional model for execution time distributions above. We have measured the execution time distributions for code running in isolation, composed them, and compared with the distributions for the codes executing in sequence. We have used the Kolmogorov-Smirnov goodness-of-fit test [4] to test the hypothesis that the probability distributions are equal. Furthermore, we have investigated which architectural features have the largest influence on the validity of the model. For this purpose we have used another statistical method, *fractional factorial design* [5], which is a systematic method to find out the likely cause for an observed variation. In order to apply this method we needed to do experiments with different hardware configurations, and therefore we used the SimpleScalar tool chain [6] to simulate these configurations.

The rest of this paper is organized as follows: in Section II, we introduce our statistical timing model and the different statistical tests and analyses that we have used. Section III describes the experimental setup: method, simulator, and benchmarks used. In Section IV we present the results of the experiments. Section V gives an account for related work, and in Section VI we wrap up and give ideas for future research.

II. Statistical Model and Analysis

A. Model

We model the execution time of a component (or a piece of code in general) by a random variable. Now consider two software components c_1 and c_2. Let their execution times be modeled by the random variables X_1 and X_2, respectively. For the sequentially composed component $c_1; c_2$ we model its execution time by the random variable $X_1 + X_2$, capturing the assumption that execution times of sequentially executing pieces of code simply add up.

B. Convolution of Distributions

For a random variable X, denote its probability density function by f_X. In general, if the random variables X and Y are independent then the probability density function for $X + Y$ is given by [3]:

$$f_{X+Y}(y) = \int f_X(y-x) f_Y(x) dx$$

The integral defines a binary operation on functions called *convolution*. A similar convolution operation is defined for discrete probability functions p_X, and a corresponding result holds for discrete independent random variables [7]:

$$p_{X+Y}(y) = \sum_{\forall x} p_X(y-x) p_Y(x)$$

This operation has some interesting and useful properties: it is associative and commutative, and can be generalized to any number of variables. For discrete random variables X and Y it can be computed in time $O(mn)$, where m and n are the number of elements in the probability spaces of X and Y, respectively.

Truth		H_0 is true	H_0 is false
Decision	reject H_0	type I error	right decision
	accept H_0	right decision	type II error

Fig. 1. Types of errors in hypothesis testing.

C. Hypothesis Testing

Hypothesis testing means to decide, from a number of samples (tests, or sets of observations), whether one should consider a property to be true or not. How can we find out? We may never know for sure, but a statistical test will give us guidance in making a decision. In statistics we can state this problem using two hypotheses: let H_0 denote the hypothesis that the property is true, and let H_1 denote the hypothesis that it is false.. The hypothesis H_0 is called *null hypothesis* and H_1 is called *alternative hypothesis*. We must decide whether to accept or reject the hypothesis H_0 based on a sample. In doing so, we might be making a mistake. For example, suppose the property really is true. If we decide to accept H_0, then we made the right decision. But if we reject it then we will make an error, called a *type I error*. On the other hand, if the real truth is that the property is false, and we reject H_0, then we would make the right decision. However, if we decide to accept H_0 in this situation, we would be doing a *type II error*. Figure 1 summarises the types of errors. The probability of a type I error is usually denoted by α and is commonly referred to as the *significance level* of a test. The probability of a type II error is usually denoted by β. The *power* of a test is defined as $1 - \beta$, i.e., the probability of correctly rejecting the null hypothesis.

The general aim in hypothesis testing is to use statistical tests that make α and β as small as possible. This goal requires compromise, since making α small involves rejecting the null hypothesis less often, whereas making β small involves accepting the null hypothesis less often. These actions are contradictory; that is, as α increases, β will decrease, while as α decreases, β will increase. The general strategy is to fix α at some specific level, and to use the test that minimises β. A common choice is $\alpha = 0.05$.

D. Kolmogorov-Smirnov Test

In order to test that the two samples come from the same distribution, we have used the two-sample Kolmogorov-Smirnov goodness-of-fit test (KS test) [4]. It is a nonparametric statistical method for comparing two sets of data, and is independent of the underlying distribution. Given two independent samples S and S', the test evaluates the following null hypothesis: *the two independent samples S and S' came from the same distribution* against the alternative hypothesis: *the two samples came from different distributions*. The test uses the *empirical cumulative distribution function* (ECDF) for each of the samples: let $S = \{x_1, ..., x_n\}$ be a random sample of size n. The ECDF $F_S(x)$ is defined by

$$F_S(x) = \frac{1}{n} \times [\text{Number of observations } \leq x]$$

The statistic of the test is given by

$$D_{S,S'} = \max_x |F_S(x) - F_{S'}(x)|$$

I.e., the test statistic is the maximum vertical distance between the ECDF's of S and S'. The null hypothesis is rejected at significance level α if

$$D_{S,S'} > K_\alpha \sqrt{\frac{n+n'}{nn'}}$$

where n and n' are the sizes of the two samples, respectively. The coefficient K_α is the critical value of the Kolmogorov distribution, and it depends on the significance level and on the sizes of the samples.

Equivalently, the acceptance of the null hypothesis can be based on the so-called "p-value", which is the probability of finding a distance bigger than (or equal to) $D_{S,S'}$ in the population assuming that the null hypothesis is true. The null hypothesis is accepted if the p-value is greater than the chosen significance level α. This is just another way of expressing the acceptance test.

E. Fractional Factorial Design

Fractional factorial design is a methodology for experimentally determining the influence of factors on some entity. Rather than testing with all combinations of possible values of the factors, a reduced set of combinations is tested. It is usually assumed that combined effects of different factors on the entity are low compared to the direct influence of each factor. Under this assumption, the number of tests can be significantly reduced.

The Plackett and Burman design [5] is an instance of fractional factorial design. It can accurately quantify the effects of single factors, and using *foldover* it can also quantify the combined effects of pairs of factors. For each factor a *high* and a *low* value is selected. The experiment is then run with different combinations of high and low values, for the different factors, in a systematic fashion. For N factors, the number of runs of the experiment in this design is the next integer K multiple of four greater than N ($2K$ for foldover). The high and low values need not be numeric: they can, for instance, be two different ways of doing branch prediction. In [8] the use of the methodology was proposed to provide a sound methodological basis for experimental computer architecture design.

Table I shows how to estimate the influence from different factors P_i on an entity R from the experiment. In the matrix, -1 stands for low value and 1 for high value. Each row represents one run, with the combination of high/low-values for the factors given by the matrix entries and the rightmost element the resulting value of R. For each factor P_i, its influence on R is now estimated as the inner product of column i and the column for R. The influences are shown in the last row of the table.

P_1	P_2	P_3	P_4	P_5	P_6	P_7	R
-1	-1	-1	-1	-1	-1	-1	1
$+1$	$+1$	$+1$	-1	$+1$	-1	-1	9
$+1$	$+1$	-1	$+1$	-1	-1	$+1$	2
$+1$	-1	$+1$	-1	-1	$+1$	$+1$	3
-1	$+1$	-1	-1	$+1$	$+1$	$+1$	7
$+1$	-1	-1	$+1$	$+1$	$+1$	-1	4
-1	-1	$+1$	$+1$	$+1$	-1	$+1$	3
-1	$+1$	$+1$	$+1$	-1	$+1$	-1	6
1	13	7	-5	11	5	-5	

TABLE I

EXAMPLE OF A CONFIGURATION OF RUNS FOR THE FRACTIONAL FACTORIAL DESIGN. WHEN USING FOLDOVER, THE LINES ARE REPEATED AND THE SIGNS FOR P_1 TO P_7 ARE INVERTED.

III. EXPERIMENTS

We represented code for components c_1, c_2 by compiled code for for C functions `c1()`, `c2()`. Call these codes `C1`, `C2`. We then emulated the effect of sequential composition $c_1; c_2$ of components by sequential execution `C1;C2` of the compiled codes. One can argue that execution of sequentially composed components also will include the transfer of values between ports and similar, which renders this simple model a bit crude. However, a realistic assumption is that this is effectuated by some "glue code" `g` executing in-between `C1` and `C2`. Then sequential composition of components can be reduced to sequential composition of code, albeit also involving code for `g`. We can simply consider `g` as code for a "virtual" component g, and then consider the sequential composition $c_1; g; c_2$. The execution time distribution for this composition can now be estimated by the convolutions of the three individual distributions, and the validity of this estimation can be tested in the same way as described below.

Given some selected C functions `c1()`, `c2()` representing component code, the experiment would proceed as follows:

- compile `c1()`, and `c2()` into `C1`, and `C2`, respectively,
- run `C1;C2` for a number of inputs to give an ECDF for its execution time,
- run `C1` and `C2` in separation for different inputs, to provide estimated distributions from which an ECDF for their convolution can be calculated;
- apply the Kolgomorov-Smirnoff test to test, for some selected significance level, the null hypothesis that the underlying distributions for the execution times of `C1;C2`, and the convolution of distributions of the execution times for `C1` and `C2`, are the same, and
- repeat the above for a number of different pairs of benchmarks.

However, to simplify the measurement process we measured the time for full compiled C programs rather than individual functions. Such programs always include a call to a `main()` function. In order to compensate for its execution time, we measured the execution time for the "empty" C program `main()`, and then subtracted this time from the measured execution times for `main(){c1();}`, `main(){c2();}`, and `main(){c1();c2();}`.

In the compilations of all these programs, the data and object code for `c1()` and `c2()` were included so that the executable code could have similar sizes for the different runs.

We selected C programs to represent components from the MiBench benchmark suite [9]. The suite emphasises diversity in order to reflect the needs of the wide range of applications in the embedded systems domain. It is composed of freely available standard C source code, where slight adaptations have been made in some of them to increase portability. The benchmarks are grouped into six categories:

- automotive and industrial control: as some embedded processors do not have dedicated hardware, this category provides basic math abilities, bit manipulation, simple data organisation and image processing,
- network: benchmarks for devices like switches and routers where the embedded processors need to do shortest path calculations, tree and table lookups,
- security: common algorithms for data encryption, decryption and hashing,
- consumer devices: this category focus on multimedia applications, with image, MP3 and MPEG processing and HTML typesetting,
- office automation: category with text manipulation algorithms and speech processing, and
- telecommunications: this category consists of voice encoding and decoding, frequency analysis and checksum algorithms.

The sizes of the benchmarks vary from small, with a few lines of source code (like quicksort), to big ones, like the GhostScript interpreter. We used the following benchmarks:

- `susan`: an image processing benchmark. It has algorithms for smoothing, and corner and edge recognition. Object code size: corner 117872 bytes, smoothing 117128 bytes, edge 118300 bytes;
- `fft`: fast Fourier transform. Object code size: 15140 bytes;
- `mm`: matrix multiplication. Object code size: 4396 bytes;
- `primes`: test an array of integer, if they are prime or not. Object code size: 2844 bytes;
- `adpcm`: adaptive differential pulse code modulation Object code size; 26580 bytes;
- `compress`: compression of ASCII text. Object code size: 11896 bytes;
- `qsort`: sort an array of strings. Object code size: 1504 bytes;
- `pbmsrch`: Pratt-Boyer-Moore string search. Object code size: 3760 bytes;
- `dijkstra`: shortest path in a weighted graph - the graph is the same in the simulations, with each run finding the path between two different nodes. Object code size: 5596 bytes;
- `sha`: a hash algorithm. Object code size: 7564 bytes;
- `rijndael`: encryption of a message. Object code size: 85836 bytes;
- `gauss`: triangulation of sparse matrix with Gaussian elimination algorithm. Object code size: 8488 bytes.

We used the SimpleScalar tool chain to run our experiments [6], [10]. It can provide detailed simulations of modern out-of-order microprocessors, allowing a wide range of hardware configurations. The tool chain has four simulators: `sim-outorder` provides the most cycle-accurate execution time, and is the one we use here. It implements the RUU (Register Update Unit) structure in order to deal with out-of-order execution.

The suite includes a gcc cross compiler (GNU GCC v2.7.2) that generates executable code from C source code. The simulator accepts as input binary code and configuration parameters, executes the code, and outputs statistics about the execution including the execution time measured in cycles. For the detailed model, specific parameters can be used to configure the processor core, the memory hierarchy and the branch predictor. The architecture is derived from the MIPS-IV ISA, with a few additions. The instruction length is 64 bits, and the instruction set is called PISA (Portable Instructions Set Architecture).

The benchmarks were compiled into object code with the SimpleScalar compiler, `sslittle-na-sstrix-gcc`, generating code for the PISA architecture using optimisation level zero, and were adapted to perform IO of data from/to global data structures. They were later linked to be executed in isolation and together in sequential composition. After the linking, the sizes of the executable files varied from 8194 to 206172 bytes.

IV. Results

We first tested the hypothesis that the convolution of execution time distributions accurately approximates the distribution of sequentially composed code. We ran two sets of compositions of two and three components, testing 100 compositions of pairs (and triples, respectively) of benchmarks drawn from the set described in Section III. For each composition we made 100 runs each for each individual component and 100 runs of their composition to estimate their execution time distributions. We used the R statistics package [11] to calculate the greatest vertical distance D between convolution and measured cumulative distribution, and the p-value. The p-value was then used to evaluate the KS test. We used the commonly used significance level $\alpha = 0.05$.

SimpleScalar was configured to simulate a processor with multiple functional units, cache memories, and branch prediction according to the following: 128kb instruction L1 cache with LRU (associativity 2), 256 kb direct-mapped L1 data cache, 32 kb instruction L2 cache with LRU (associativity 8). Latency 1 cycle to L1 caches, 5 cycles to L2 cache, 50 cycles to main memory, memory access bus width 4, two-level branch prediction, four integer ALU's, four FP ALU's, in-order issue.

The result was that with $\alpha = 0.05$, the null hypothesis was rejected by the KS test only for a few compositions: two for compositions of two components, and one for compositions of

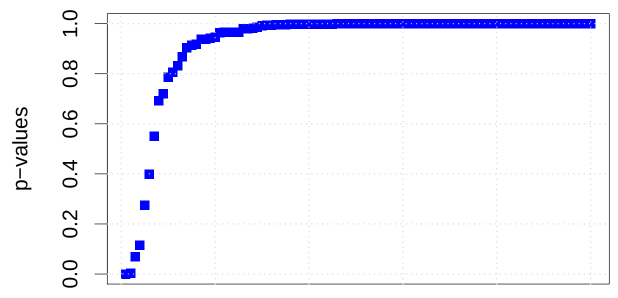

Fig. 2. The sorted p-values for 100 random sample compositions of two components.

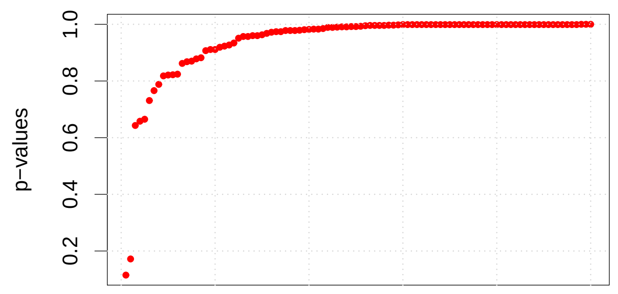

Fig. 3. The sorted p-values for 100 random sample compositions of three components.

three components. The sorted p-values are plotted in Figures 2 and 3, respectively. As can be seen, most p-values are very close to one meaning that there is very little evidence against the null hypothesis.

Fig. 4 shows an example where the KS test accepted the null hypothesis. In this example, D is 0.047. The p-value is 0.977, and as it is greater than $\alpha = 0.05$, we accept the null hypothesis.

Next, we systematically investigated which hardware features would most likely be responsible for a rejection of the null hypothesis. We used the Plackett and Burman design for this purpose. From the more than 40 hardware configuration parameters available in the SimpleScalar simulation tool, $N = 37$ parameters were chosen. These parameters were all used in [8], and their high and low values were set to the same values as there. The parameter K was set to 40 (next multiple of four), resulting in 80 different hardware configurations (with

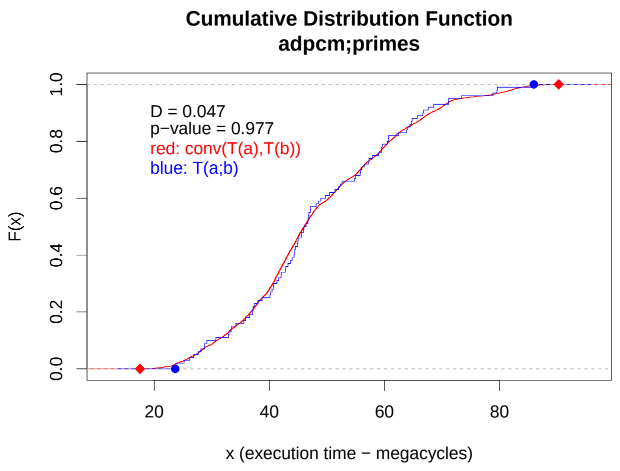

Fig. 4. Composition of two components: we accept the hypothesis that the convolution has the same distribution as the execution time of the composition.

Hardware feature	N
I-tlb size ENTRIES	1
mem latency first next	1
D-tlb size ENTRIES	1
I-tlb assoc	1
lsq entries	1
L1 i-cache block size	1
memory ports	2
int alus	2
branch misprediction penalty	2
ras entries	2
speculative branch update	2
L1 d-cache latency	2
D-tlb assoc	2
L1 i-cache repl policy	2
L1 d-cache repl policy	2
btb entries	2
L1 i-cache latency	2
int mult div units	3
L1 i-cache assoc	3
L1 d-cache size KB	3
I-tlb latency	3
fetch queue entries	3
L2 cache latency	3
execution order	4
L2 cache size KB	4
btb assoc	4
ROB entries	5

TABLE II
NUMBER OF TIMES A HARDWARE FEATURE WAS THE MAIN VARIABLE AFFECTING THE DEVIATION BETWEEN THE REAL EXECUTION TIME DISTRIBUTION AND THE CONVOLUTION.

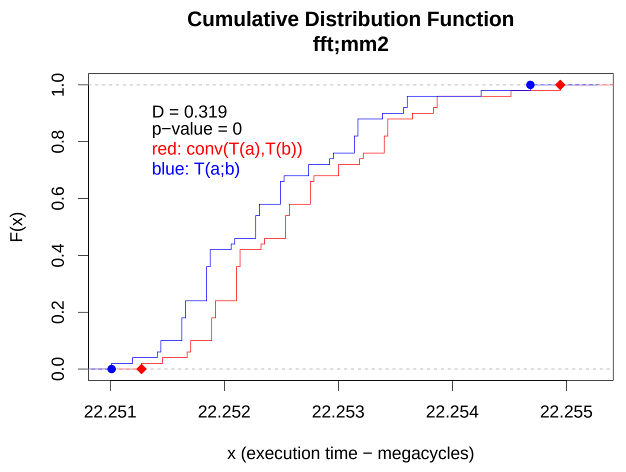

Fig. 5. For this composition, using perfect branch prediction in the simulation, the hypothesis is rejected.

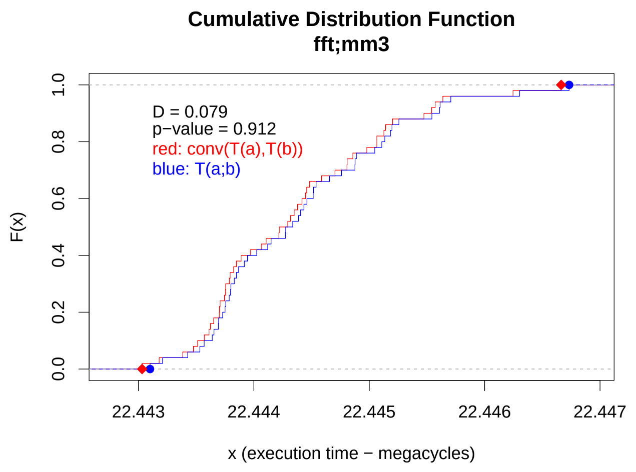

Fig. 6. The hypothesis for the same composition as in Figure 5 is accepted when we use the 2-level adaptive branch prediction algorithm in the hardware simulator.

foldover). Each configuration was then tested for 63 different compositions of two components drawn from the selected set of benchmarks, estimating the cumulative distribution functions in the same way as before. For each composition, we recorded which hardware feature had the biggest estimated influence on the deviation between convolution and measured execution time distribution. The result is given in Table II. For this particular sample, the number of entries in the reorder buffer (ROB) was the most frequent parameter having the strongest influence.

For the 63 different compositions, we had two cases where the null hypothesis was rejected and one case very close to rejection. We examined these a bit closer to see how strong the estimated influence of the 37 different hardware parameters was. Table III shows the estimated influence of each parameter on the p-value, and on the distance D, for the three cases. Interestingly, in all three cases the branch predictor was deemed to have the largest influence and not the ROB.

One of these cases is shown in Fig. 5. The plot shows the result with branch predictor set to "high", which means perfect branch prediction. Interestingly, if we change the configuration to the "low" value, 2-level adaptive branch prediction, then the null hypothesis is accepted. This case is shown in Fig. 6. So for this example, the branch predictor really has a significant influence on the validity of the convolution as composition operator for execution time distributions.

V. Related Work

Timing analysis is of fundamental importance to the successful design and execution of real-time systems. The execution time is needed, for example, in scheduling and schedulability analysis [12]. Often, the analysis aims at determining (or bounding) the worst-case timing. On code level, one key timing measure is the *worst-case execution time* (WCET) of a program, which is the largest possible execution time of a program executing on a given hardware while not being interrupted. It is often estimated through measurements; however, this is not reliable in general [13] as it is hard to find the input parameters that cause the worst execution time.

Thus, WCET analysis methods usually attempt to estimate the WCET from WCETs for small pieces of the program, typically basic blocks, which are combined using program flow information to form a WCET estimate. This is a bit similar to sequential component composition in that estimation of the total WCET typically is done by adding the local WCETs for the identified longest path. However, WCET analysis differs in that basic blocks typically are small, and their code is assumed known, whereas components can be "black boxes" and may contain larger code.

In *static* WCET analysis, a microarchitectural analysis attempts to bound the WCET for basic blocks using their code, and knowledge of the hardware timing model [14]–[17]. This is a white-box approach. *Hybrid methods* are more related to our approach: there, basic blocks are treated as black boxes and their WCETs are estimated from measurements [18], [19]. This work is closely related to ours in that timing profiles are generated for the basic blocks, and combined using a convolution model with possible dependencies to create a probabilistic model for the total execution time from which the WCET can be estimated. In [20], copulas were used to model the dependencies.

The basic blocks considered in hybrid WCET analysis are typically much smaller than the components that we consider. Thus dependencies between their execution times can be expected to have stronger correlations, which necessitates a more elaborate treatment.

Instead of estimating a value for the WCET, the work in [21] focusses on deriving a execution time probability distribution for a component in isolation, using measurements of response time, and does not tackle the problem of composition. A special monitor component is responsible for the measurements. Dynamic simulation and statistical analysis is also used in performance analysis [22], and can be used to find code

	Effect on p-values			Effect on distances		
HW feature	A	B	C	A	B	C
L2 cache latency	1.69	0.13	0.11	0.28	0.12	0
speculative branch update	1.07	0.14	0.11	0.2	0.04	0
ras entries	0.44	0.18	0.14	0.12	0.08	0.08
int mult div units	2.17	0.23	0.12	0.2	0.2	0.08
L1 i-cache latency	1.53	0.26	0.12	0.2	0.04	0.08
mem latency first next	0.59	0.57	0.79	0.32	0.16	0.64
D-tlb page size KB	0.2	0.6	0	0.08	0.08	0
memory ports	1.69	0.71	0.11	0.12	0.16	0
L1 d-cache block size	1.69	0.71	0.01	0.12	0.16	0.04
ROB entries	0.61	0.78	0.12	0.04	0.04	0.04
int alus	1.3	0.82	0.13	0.2	0.24	0.08
L1 i-cache assoc	1.71	0.91	0	0.28	0.2	0
L2 cache assoc	0.51	0.93	0	0	0.12	0
I-tlb assoc	1.07	1.05	0.03	0.16	0.12	0.07
L2 cache size KB	1.23	1.11	0.04	0.25	0.24	0.08
lsq entries	0.66	1.19	0.37	0.2	0.24	0.16
L2 cache block size BYTES	0.51	1.48	0.01	0.07	0.16	0.04
L2 cache repl policy	1.53	1.49	0	0.08	0.2	0
L1 d-cache latency	0.59	1.56	0.12	0.12	0.28	0.04
D-tlb assoc	1.23	1.62	0.01	0.24	0.24	0
execution order	1.23	1.64	0.13	0.2	0.16	0.08
L1 d-cache repl policy	0.59	1.68	0.03	0.16	0.28	0.08
mem bandwidth	0.51	1.71	0.12	0.09	0.25	0.09
L1 d-cache assoc	0.44	1.74	0.01	0.12	0.16	0.04
btb entries	1.69	1.94	0.04	0.4	0.25	0.09
btb assoc	0.44	2.26	0.05	0.07	0.24	0.12
I-tlb latency	0.77	2.42	0.11	0.12	0.32	0.04
L1 i-cache repl policy	0.61	2.63	0	0.12	0.37	0
I-tlb page size KB	2.33	2.65	0	0.24	0.4	0.01
fetch queue entries	0.2	2.78	0.11	0.08	0.4	0.05
branch misprediction penalty	0.59	3.22	0.72	0.32	0.52	0.53
I-tlb size ENTRIES	0.13	3.46	0.01	0	0.44	0
D-tlb size ENTRIES	0.2	3.91	0.02	0.12	0.44	0.04
L1 d-cache size KB	1.23	4.21	0.01	0.12	0.6	0.04
L1 i-cache size KB	2.81	5.34	0	0.24	0.88	0
L1 i-cache block size	0.66	6.4	0.05	0.08	0.84	0.11
branch predictor	9.29	10.87	38.76	33.65	1.08	7.1

TABLE III
ESTIMATED INFLUENCE OF EACH HARDWARE PARAMETER ON THE P-VALUES, AND DISTANCES D, USING THE FRACTIONAL FACTORIAL DESIGN FOR THE COMPOSITIONS A=FFT;RIJNDAEL, B=FFT;MM AND C=GAUSS;RIJNDAEL.

bottlenecks, parts of the code where the program spend most of the time, and optimisations can be focused in these specific parts. In [8] statistical analysis is used to analyse computer architecture performance, and the fractional factorial design is used to analyse how changes in the architecture affect the execution time.

The two-sample t-test is used in [23] as a hypothesis test to measure performance similarities between benchmarks. This test requires Student's t distribution for the samples, while our use of the Kolmogorov-Smirnov test has the advantage that it can be used for any distribution.

VI. CONCLUSION

In this work we investigated the execution time of sequentially composed components using statistical methods: we propose the use of convolution as a composition mechanism for the execution time distributions of components. We used the Kolmogorov-Smirnov goodness-of-fit test to evaluate how well the convolution seems to approximate the real distribution of the composed components when hardware influences are taken into account. Our results indicate that for the most part, the convolution provides a good approximation of the actual execution time distribution.

We then investigated which hardware features seem to influence the validity of the convolution approximation the most. We found that several such features could have the most significant influence, with high rankings for reorder buffer, branch table associativity, L2 cache size, and out-of order execution. However, for the the cases where the convolution did not provide a very accurate model, the branch predictor was found to have the largest influence on this discrepancy. This indicates that the branch predictor is an important factor affecting the validity of convolution as composition operator, and thus should be taken into account when selecting hardware for applications where timing-predictability is required.

The convolution model relies on an assumption that sequential timing models are additive, such that estimated execution times for sequentially executing activities can simply be added up. This assumption is used not only for sequential component composition, but also in other parts of the real-time area like schedulability analysis. Thus, our results can have significance also in those areas.

A topic for future research is to make a more detailed investigation how features of the composed codes interact with hardware features to strengthen or weaken their influence on the validity of simple compositional timing models like the convolution model. Obviously, features such as control structure (frequency and predictability of branches), memory access patterns, etc. will interact with hardware features such as branch predictors, pipelines, and memory systems to make the influence on timing model validity stronger or weaker. Is it possible to come up with simple code characteristics that can predict which hardware features will be influential for composition of certain codes?

We have investigated sequential composition on single-core architectures. For components running in parallel on a multicore architecture, simple compositional timing models are desirable as well as long as they are reasonably accurate. A possible example is $T(c_1 \,|\, c_2) = \max(T(c_1), T(c_2))$, where "$|$" is a parallel composition operator. As in the sequential case such simple timing models are not exact, and an interesting topic for future research is to investigate the influence of different hardware features on the validity of parallel timing models.

Another important topic for further work is to investigate the influence of software data/control dependencies on the validity of the convolution model. Such dependencies can cause covariations in certain execution paths in the components, which in turn may lead to covariations in execution time. It is important to find out how likely this is to appear in practice. Besides a statistical analysis, static program analysis could also be used to find potential dependencies.

VII. Acknowledgements

This work was supported by SSF – the Swedish Foundation for Strategic Research – through the PROGRESS Research Centre for Predictable Embedded Software Systems. Travel support for the first author to visit the LASID lab (Federal University of Bahia, Brazil) was provided by CUGS, the Swedish National Graduate School in Computer Science. Special thanks to Prof. Andreas Ermedahl and Prof. Raimundo Macêdo for their helpful comments.

References

[1] A. Burns, G. Bernat, and I. Broster, "A probabilistic framework for schedulability analysis," in *Proc. Third International Conference on Embedded Software (EMSOFT 2003*, 2003, pp. 1–15.

[2] K.-K. Lau and S. Wang, "A survey of software component models," University of Manchester, Tech. Rep. CSPP-30, 2006.

[3] T. T. Soong, *Fundamentals of Probability and Statistics for Engineers*. Wiley-Interscience, 2004.

[4] D. J. Sheskin, *Handbook of Parametric and Nonparametric Statistical Procedures, Third Edition*. Chapman & Hall/CRC, 2004.

[5] R. L. Plackett and J. P. Burman, "The design of optimum multifactorial experiments," in *Biometrika*, vol. 34, 1946, pp. 255–272.

[6] T. Austin, E. Larson, and D. Ernst, "Simplescalar: An infrastructure for computer system modeling," *Computer*, vol. 35, no. 2, pp. 59–67, 2002.

[7] C. M. Grinstead and J. L. Snell, *Introduction to Probability*. American Mathematical Society, 1997.

[8] J. J. Yi, D. J. Lilja, and D. M. Hawkins, "Improving computer architecture simulation methodology by adding statistical rigor," *IEEE Trans. Comput.*, vol. 54, no. 11, pp. 1360–1373, 2005.

[9] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown, "MiBench: A free, commercially representative embedded benchmark suite," in *Proc. Fourth Annual IEEE International Workshop on Workload Characterization (WWC-4)*. Washington, DC, USA: IEEE Computer Society, 2001, pp. 3–14.

[10] D. Burger and T. M. Austin, "The SimpleScalar tool set, version 2.0," *SIGARCH Comput. Archit. News*, vol. 25, no. 3, 1997.

[11] R Development Core Team, *R: A Language and Environment for Statistical Computing*, R Foundation for Statistical Computing, Vienna, Austria, 2008, ISBN 3-900051-07-0. [Online]. Available: http://www.R-project.org

[12] J. Ganssle, "Really real-time systems," in *Proc. Embedded Systems Conference, Silicon Valley 2006 (ESCSV 2006)*, Apr. 2006.

[13] R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, T. Mitra, F. Mueller, I. Puaut, P. Puschner, J. Staschulat, and P. Stenström, "The worst-case execution-time problem—overview of methods and survey of tools," *Trans. on Embedded Computing Sys.*, vol. 7, no. 3, pp. 1–53, 2008.

[14] J. Engblom, "Analysis of the execution time unpredictability caused by dynamic branch prediction," in *Proc. 8th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS'03)*. Washington, DC, USA: IEEE Computer Society, May 2003.

[15] J. Engblom and B. Jonsson, "Processor pipelines and their properties for static WCET analysis," in *Proc. 2nd International Workshop on Embedded Systems, (EMSOFT2002)*, ser. Lecture Notes in Computer Science, A. L. Sangiovanni-Vincentelli and J. Sifakis, Eds., vol. 2491. Grenoble: Springer, Oct. 2002, pp. 334–348.

[16] C. Ferdinand and R. Wilhelm, "Efficient and precise cache behavior prediction for real-time systems," *Real-Time Systems*, vol. 17, pp. 131–181, 1999.

[17] S. Thesing, "Safe and precise WCET determination by abstract interpretation of pipeline models," Ph.D. dissertation, Saarland University, 2004.

[18] G. Bernat, A. Colin, and S. M. Petters, "WCET analysis of probabilistic hard real-time systems," in *Proc. 23rd IEEE Real-Time Systems Symposium (RTSS'02)*. Austin, TX: IEEE Computer Society, Dec. 2002, pp. 279–288.

[19] G. Bernat, A. Colin, and S. Petters, "pWCET: a tool for probabilistic worst-case execution time analysis of real-time systems," University of York, Tech. Rep. YCS-2003-353, 2003.

[20] G. Bernat, A. Burns, and M. Newby, "Probabilistic timing analysis: An approach using copulas," *J. Embedded Comput.*, vol. 1, pp. 179–194, Apr. 2005.

[21] R. Perrone, R. Macedo, G. Lima, and V. Lima, "An approach for estimating execution time probability distributions of component-based real-time systems," *Journal of Universal Computer Science*, vol. 15, no. 11, pp. 2142–2165, jun 2009, http://www.jucs.org/jucs_15_11/an_approach_for_estimating.

[22] R. Jain, *The Art of Computer Systems Performance Analysis: Techniques for Experimental Design, Measurement, Simulation, and Modeling*. Wiley-Interscience, 1991.

[23] P. Giusto, G. Martin, and E. Harcourt, "Reliable estimation of execution time of embedded software," in *DATE '01: Proceedings of the conference on Design, automation and test in Europe*. Piscataway, NJ, USA: IEEE Press, 2001, pp. 580–589.