Reliable Estimation of Execution Time of Embedded Software

Paolo Giusto
Cadence Design Systems, Inc.
2670 Seely Avenue
San Jose', CA 95134, U.S.A.
giusto@cadence.com

Grant Martin
Cadence Design Systems, Inc.
2670 Seely Avenue
San Jose', CA 95134, U.S.A.
gmartin@cadence.com

Ed Harcourt
Cadence Design Systems, Inc.
270 Billerica Rd , Chelmsford MA, 01824, U.S.A.
harcourt@cadence.com

Abstract

Estimates of execution time of embedded software play an important role in function-architecture co-design. This paper describes a technique based upon a statistical approach that improves existing estimation techniques. Our approach provides a degree of reliability in the error of the estimated execution time. We illustrate the technique using both control-oriented and computational-dominated benchmark programs.

1. Introduction

Embedded system designers are under increasing pressure to reduce the design turnaround time, often in the presence of continuously changing specifications. One of the major design validation problems is the evaluation of different HW/SW partitionings. Today's approaches are often carried out at the *co-verification* level - a *virtual* prototype of the system under validation is built [3] [4]. The embedded SW is compiled and run on an instruction set simulator (ISS) while the hardware implementation is run on a VHDL or Verilog simulator - the communication between the two partitions being simulated at the microprocessor pin level. The advantage of this approach is in the accuracy of the simulation results. The disadvantage is the simulation speed - complete system simulations are too slow for exploring the design space efficiently. Moreover, building such a detailed system model (including interfaces between HW and SW) is time-consuming.
In the VCC methodology [15], the *function/architecture* co-design paradigm [6] raises the level of abstraction, increases the simulation speed, and therefore reduces the turnaround time. VCC supports an *IP based* design flow where an architectural IP is represented in terms of its *performance* model. This is used to *back-annotate* the behavioral description of the design with timing information. To make sure that the HW-SW trade-offs are accurately explored, it is important to provide accurate estimates of the execution times of the behaviors that are mapped to SW implementations. The VCC SW estimation framework [12] [5] models both the target system (CPU instruction set, target compiler, etc.) and the structure of the software program at an abstraction level that makes the estimate of execution time reasonable without losing too much accuracy.
In this paper, we present a technique based upon a statistical approach that improves the current VCC SW estimation framework. Our main contribution is the provision of *reliable* estimates of the execution time of SW running onto a microprocessor. This is achieved in two steps. In the first step, a model of the target processor is derived in terms of a *Virtual Processor Instruction Set* [12] [5] by combining:

1. a front-end compiler optimizer, such as [1] which minimizes inaccuracies in the code to be estimated (dead code, loop invariant code, etc)

2. a set of domain specific (wireless, multimedia, automotive) benchmark programs

3. the VCC virtual compiler and estimator which determines the number of executed *Virtual Instructions* for the benchmark program

4. an ISS for the target environment, that is used to measure the actual execution time of the benchmark program

5. use of multiple linear regression [11] to determine a predictor equation for the estimated cycle count along with degree of reliability in the error of the estimation

In the second step, we use a 2-sample t-test to measure the similarity of new application code to be estimated to the existing set of benchmarks used to derive the processor model. If they are similar, then the application code is optimized with the very same front-end optimizer and then a simulation model with annotated execution time is produced by the VCC compile-code generator/annotator. If they are not, the conclusion is that the new application code is not drawn from the same population as the benchmark sample. Therefore, a new processor model derived by a population similar to the new application code should be used. This paper is organized as follows. Section 2 presents related work on estimation and motivates our proposal. Section 3 describes our technique. Section 4 shows some preliminary results. Section 5 concludes the paper.

2. Related Work and Motivation

SW performance estimation methods have been around for some time. The results to date present disadvantages that can limit their effectiveness. The methods can be described in terms of the following features:

- level of abstraction: source code based, object code based, ISS based, etc.
- constraints on the type of SW for which the technique is applicable (*e.g.,* control vs computation dominated code, static loop bounds, pointers vs static data structures, etc.).
- type of analysis: static vs dynamic
- estimation vs modeling, statistical, etc
- cost of constructing the model
- simulation speed vs. accuracy of the results
- granularity of the analysis

A source-based approach analyzes the original C source code. For VCC this entails compilation onto a virtual instruction set, and allows one to quickly obtain estimates without the need for a compiler for the target processor [12]. An object-based approach translates the assembler generated by the target compiler to assembly level-functionally equivalent C [12]. In both cases the code is annotated with timing and other execution related information (*e.g.,* estimated memory accesses) and is used as a precise, yet fast, software simulation model.

Using static analysis, constructs such as dynamic data structures, recursive functions, and unbounded looping are impossible to evaluate [17] [16] [13]. In [18], a software synthesis system is proposed, where all the primitives for constructing a program are defined as a fixed sequence of instructions. The execution time and code size of these instructions are pre-calculated, hence, they can be used to yield accurate predictions of performance. In [14] a set of linear equations is used to implicitly describe the feasible program paths. This approach has the advantage of not requiring a simulation of the program, hence it can provide conservative worst-case execution times.

Software performance estimation has become more important as new approaches for the synthesis and verification of real-time embedded systems have been developed. A simple prediction method is presented in [8], where execution time is made proportional to the product of the number of executed instructions and the MIPS rating of the target system. In [21] [10], statistical methods are proposed to model the performance of a target CPU so that several CPUs can be evaluated. In [9] the author estimates software performance by the number of execution cycles needed for each instruction in the program, the number of memory read/writes, and the number of cycles per memory access. In COSYMA [22], the given software program runs on a synthesized RT-level target system and SW timing characteristics are extracted from the simulation results.

In [3] [4] filtered information is passed between a cycle-accurate ISS and a hardware simulator (*e.g.,* by suppressing instruction and data fetch-related activity in the hardware simulator). This approach is precise but slow and requires a detailed model of the hardware and software. Performance analysis can be done only after completing the design, when architectural choices are difficult to change. In [19] the control flow graph (CFG) of the compiled software description is annotated with information useful to derive a cycle-accurate performance model (*e.g.,* pipeline and cache). The analysis is performed on the code generated for each basic block and information about the optimization performed by an actual compiler (register allocation, instruction selection and scheduling). The object code-based approach described in [12] partially uses this scheme.

Instead of restricting the input one can alternatively require that a trace of the program's execution on some sample data be used to drive the analysis. In unmodified form, this method requires a very detailed model or an instance of a system similar to the one being designed. To avoid this requirement, a trace-driven approach can be used [5]. Performance analysis can occur when a statistically relevant set of benchmarks is applied [21].

In the computational-dominated code domain, [7] has introduced a novel methodology for estimating execution time of SW running on a DSP. The technique is based upon defin-

ing a set of kernel functions whose execution times are pre-characterized, for example, via profiling. The algorithm to be estimated is then built from a static composition of the kernels for DSP applications (synchronous dataflow). In [20] [6], the POLIS source-based SW estimation method is presented: the original C code is annotated with timing estimates predicting compiler optimizations. This does not require a complete design environment for the chosen processor(s), since the performance model is relatively simple (an estimated execution time on the chosen processor for each high-level language statement). The approach is targeted to control-dominated code, and it cannot consider compiler and complex architectural features (*e.g.*, pipeline stalls due to data dependencies).

Our approach can be classified as source-based , with no constraints on the SW (any arbitrary C code), with relatively low cost for the modeler, with relatively fast simulation speed, dynamic (simulations are needed) and statistical analysis-based approach. We are able to provide a degree of reliability in the estimates which was missing in the approaches described in [5] [12] [20]. We are also able to provide a semi-automatic way, via a predictor equation, to find and then assign a performance model to the kernel function therefore improving [7]. Also the affinity of a new benchmark program to the existing set can be inferred, and the error of the estimation vs the control/computation ratio of the SW are estimated. We are not claiming to be able to provide the accuracy of an object-based or ISS based approach. Our claim is that the error in the estimates has a degree of statistical confidence, and therefore the designer can make an assessment whether the estimates can be used to make trade-off decisions or a more expensive technique such as object or ISS-based should be used.

3. The Reliable Execution Time Estimation Technique

A dynamic source-based SW estimation technique is based on the idea of abstracting the actual machine instructions which would be executed by a SW task running on a target processor into a set of *Virtual Instructions* [5] [12]. Each basic block in the SW task is compiled into a count of the number of virtual instructions which would cover the execution of the block; during simulation the SW task is executed natively on a host workstation, but the number of virtual instructions which would be executed on the actual target processor is accumulated [5] [12]. Modeling the combination CPU/Compiler at this level of abstraction has been proven to be a difficult task. Different compilers produce different code on the same source code. Therefore, we believe that any source-based approach is not a push button solution. The key idea is to provide a *correct interpretation* of the Virtual Machine instructions depending upon how well the VCC compiler/estimator models the pair CPU/Compiler - two different interpretations are available. In a *strict* interpretation, the VCC compiler models well the target environment; therefore each Virtual Instruction truly represents the corresponding class of assembly instruction(s) and it makes sense to consider the cycles provided by a Data Book as a good approximation of the real cycles and assign those cycles to the Virtual Instructions. In the *relaxed* one, the Virtual Instruction is a factor in predictor equation since the VCC compiler does not model *as well* the target environment. The usage of a front-end optimizer minimizes the sources of inaccuracies. However, the back-end optimizations are very specific and different from processor to processor, hence very difficult to capture in a general model. Therefore, a non-accurate number of Virtual Machine Instructions might be executed during the VCC simulation. To provide a good estimate of the executed time, the cycle counts of the Virtual Machine Instructions need to compensate the inaccuracies and therefore may assume, for example, negative values. The relaxed interpretation leads to a statistical approach with degree of confidence in the prediction. Anything in between strict and relaxed should be evaluated case by case.

In the current VCC incarnation, the Virtual Instructions are a simplified view of a RISC instruction set; there are 25 [12], including LD (load from memory), LI (load immediate), ST (store), OP(i,c,s,l,f,d) (basic ALU operator for integer, char, short, long, float, double), SUB(subroutine call), RET (return from subroutine), GOTO, and IF (branch).

Each Virtual instruction on the target processor is characterized via a number of methods as to the number of actual target processor cycles which are covered by the Virtual instruction. This leads to the predictor equation:

$$Cycles = K + \sum_i P_i * N_i$$

where N_i is equal to the number of Virtual Instructions of type i, P_i is equal to a parameter which translates the Virtual Instruction onto a cycle count, and K is the intercept. N_i is computed by compilation of the SW task in VCC and the generation of an annotated version of the task which accumulates each occurrence of a virtual instruction during host-based execution. P_i can be determined in several ways:

1. from the datasheet of the target processor (strict interpretation)
2. using a best fit least squares approach to a calibration suite
3. using a stepwise multiple linear regression approach over sets of tasks drawn from a similar domain (relaxed interpretation). Note that this approach can start from a solution derived from the above method and then be used for tuning the results.

Estimation Approach	Type	Creation Effort	Accuracy	Speed
Static Analysis	Source Code-Based	Easy w/ Code Restrictions	Low	No Simulation
Statistical VI (no constraints)	Relaxed Source Code-Based	Easy w/ SW Benchmarks	Medium	100+ Times
Data Book VI	Strict Source Code-Based	Easy w/ SW Benchmarks	Medium	100+ Times
Tuned VI (constraints)	Semi-Strict Tuned Source Code-Based	Easy w/ SW Benchmarks	Medium	100+ Times
Kernel Function	DSP Oriented	Moderate w/ Profiling	Good To Very Good	N/A
Compiled-Code ISS	Object-Based	Moderate	Very Good	50+ Times
ISS Integration	ISS in the loop	Very High	Excellent	1

Table 1. SW Estimation Techniques

In the latter case, constraints on the set of linear equations must be relaxed since we relax the interpretation in the meaning of Virtual Instruction Set. In table 1[1] we illustrate trade-offs involved with the different interpretations of the Virtual Instructions. Also, the positioning of the technique w.r.t. the object-based as well as ISS based techniques is shown. Using processor *A* as an example, we will now illustrate these techniques and discuss their advantages and disadvantages. The sample set we used in these studies consisted of 35 control-oriented (decision dominated) SW tasks running approximately 200 cycles per task. These were drawn from a real control-oriented automotive application.

3.1 Datasheet approach

The datasheet approach draws the parameters Pi from a study of the published processor *A* datasheet and by analogy between the actual machine instruction set and the virtual instruction set. This has two main issues: first, some interpretation of the cycles reported per instruction is required - effects such as pipelining have an impact ; secondly, for instructions with variability in their number of cycles, based on processor state, a decision must be made as to whether to use worst, best or some nominal case.
For example, in processor *A*, the LD and ST instructions (load and store to memory) take a nominal 3 cycles. However, the processor has a three-stage pipeline, and depending on the compiler quality and the task, the processor may be able to perform other instructions while waiting on memory and not stall. In fact, intelligent compilation in most cases reduces the actual LD and ST effective cycle count to 1 or very near it. Using the original cycle count of 3 gives a very pessimistic estimator.
Another similar issue occurs with SUB and RET (subroutine calls and returns). Processor A will store to memory only the part of the register set actually in use, which can vary from 0 to 15 registers, on a call to a routine. Similarly on return, only the needed number of registers are recovered from memory. Use of the worst case in SUB and RET, assuming all 15 user registers must be saved, leads to parameters of 19 and 21 for SUB and RET, which are very pessimistic or conservative in most cases. However, the actual number of registers typically used varies from task to task in a dynamic way and we cannot find a nominal or typical value without a statistical study of some kind.
We used 2 parameter files (called *basis* files) derived from datasheet analysis of cycle counts over the set of 35 benchmarks. In the first, all Virtual instructions are estimated on the most conservative basis; in the second, the loads and stores are reduced from 3 to 1 cycle. Using the first estimator, the error % (comparing prediction to actual cycles) ranges from -8.5% to 44%, where a positive error indicates a pessimistic estimator. In general, the estimator is conservative and the spread of error is over 50%. Using the second basis file with more realistic load and store cycle counts, we get an error range of -28% to 18% - clearly less pessimistic (actually now a little optimistic) and with a total error range of 46%. However, we wish to improve the technique and reduce the expected error ranges. The datasheet method is insufficiently dynamic and not tuned to particular SW task domains and thus cannot be expected to give a very good estimator for particular kinds of tasks. Although one can *correct* the load/store cycle counts, there is no easy way to correct for subroutine call and return overheads; thus in general for large tasks with much hierarchical function call structure the technique will still be very conservative. This in fact has been our experience with VCC

3.2 Calibration approach

A different approach to deriving a processor basis file is to create a special calibration suite of programs with each test attempting to stress some part of the virtual instruction set; then, do a least squares fit of actual cycles for the set of tasks to the numbers of occurrences of the virtual instructions in each task. This is used to derive a basis file in which the parameters for each virtual instruction are based on experimental data from the calibration suite.
In this basis file, the parameter for loads and stores was 0.1, for multiplies ranging from 2.6 to 149.3, (depending on operand type), for IF 1.6, etc. In applying this to our set

[1] code restrictions are bounded loops, no recursion, etc.

of 35 benchmarks, we had an error range of -55 to -15% (all estimates optimistic, underestimating the number of cycles), for a total error range of 40%.
Several problems exist with the calibration approach:

- Choice of calibration test suite - in our case the experiments were done with a few standard programs (e.g. SPEC type programs such as 8 queens, sort, fft, and a set of highly synthetic programs created to stress particular virtual instructions). The relationship between this kind of suite and any particular embedded SW domain in terms of characteristics is marginal at best. In particular, it may over-emphasis mathematical processing (since many of the synthetic programs are to find values for MUL and DIV variations) at the expense of good control-oriented predictors.

- Over-determined analysis - with 25 virtual instructions and a calibration suite of about 18 programs, this is an over-determined[1] system in which the least squares fit will achieve perfection or near-perfection on the calibration suite but has little *a priori* basis on which one can apply it to other programs. In fact, with least squares fitting, and regression, if there are fewer benchmarks (equations) than independent variables, then it is always guaranteed that a perfect fit can be found. But the resulting predictor will not be robust. For example, suppose we have 2 benchmarks and 3 variables: $Cycles = A*LD+B*OP+C*MUL$ and 2 benchmarks with $Cycles = 800, LD = 22, OP = 600, MUL = 2$ and $Cycles = 50, LD = 1, OP = 3, MUL = 1$ - a solution is: $A = 35, B = 0, C = 15$. Note that this is a perfect predictor (although LD would attract 35 cycles, each MUL 15, and each OP 0 - a solution with no operative sense). However, there are several *perfect* fit solutions.

- Possible lack of robustness - the calibration suite and over-determined analysis means that for programs from a different domain, the calibration suite basis file may give very inaccurate results. Indeed, this was seen in our set of 35 control programs where it was both extremely optimistic and gave worse results on aggregate than the data book approach.

- High correlations between the Virtual instructions - the assumption is that these instructions are all statistically uncorrelated - that they are all independent variables with the cycle count being the only dependent variable. However, in actual programs the relative frequency of one virtual instruction often has very high correlation with others (for example, loads with basic ALU operators, or loads with IFs). This implies that simpler, more robust estimators may be possible in which the number of independent variables is reduced to a minimum.

- Difficulties in user interpretation - Users expect the calibration approach to give parameters which *make sense*. That is, all virtual instruction parameters must be 1 or greater (since no real instruction takes less than a cycle to execute), preferably integral, and scaling logically. However, the best fit approach is just looking for parameters in an equation. On taking this approach, the parameters no longer have any real relationship to *cycle count* for a virtual instruction. They are just *multiplicative factors* derived from curve fitting and used in a predictor equation. This is a hard point to make to users who are unfamiliar with this kind of approach.

However, the calibration approach does point the way towards a more solidly-grounded statistical approach, but one based on 3 premises:

- analysis based on actual SW programs drawn from specific domains - automotive, communications, control-oriented, computation-dominated

- an attempt to reduce the number of Virtual instructions used in the predictor to a minimal number of independent variables and thus give a more robust and meaningful estimator

- by exposing positive and negative correlations between various virtual instructions and cycle count, to move away from the idea that parameters or factors have a *cycle count* meaning.

3.3 The Statistical Estimator approach

In this 3rd approach, we abandon the idea of creating a single predictor for all SW tasks. Instead we use a statistical technique to derive a predictor for specific SW task domain, and then study the applicability of the predictor to other domains. The approach used is a stepwise multiple linear regression approach, along with basic multiple linear regression, correlation analysis, and *art*(user's intuition). The SW task domain is the set of 35 control oriented automotive benchmarks used earlier. As a control set, we have a set of 6 Esterel benchmarks on which to try the results derived from the 35 tasks. As we will see, the estimators derived from the set of 35 give poor results on the control set of 6. We then conduct a simple 2-sample t-test on the 2 sets of benchmarks to give us a basis for concluding that the control sample is not drawn from the same population as the benchmark sample.
The VCC annotator generates only 10 virtual instructions (LD, LI, ST, OPi, MULi, DIVi, IF, GOTO, SUB and RET) from the 35 benchmarks. We start with the assumption that

[1] in a statistical sense

these are all Independent variables, and use total cycle count for the task (Cycles) as the dependent variable[2]. On applying the stepwise multiple regression, we get some interesting results. First, the numbers of MULi and DIVi in the benchmark set are a constant, and therefore must be thrown out - with no variance, an assumed independent variable will have no correlation with the dependent variable. Secondly, only one independent variable, LD, is added to the equation, giving an equation of $Cycles = 145 + 4 * LD(1)$ and a $R^2 = .363$.
The R^2 measure is a key one in regression. Essentially, it measures how much of the total variance in the dependent variable (in this case, $Cycles$) can be explained by the variance in the independent variables which are being regressed upon. So 36% of the variance in cycle count is explained by the variance of the LD virtual instruction. In addition, note that regression in general will give equations with intercepts(ie a constant factor). These can be interpreted in several ways, one of which is the amount of setup required to run and stop a software task. However, another view is that it is just a constant which makes the regression fit better. It can be misleading to assume that regression parameters measure anything other than correlation. So the 4 for the LD parameter can be interpreted that each LD on average attracts 4 cycles of various instructions including itself; or it can be interpreted as an essentially meaningless, statistical phenomenon that can be used to predict cycles but has no inherent *meaning*. A figure (.36) is not very good. However, it is interesting to note that a fairly low R^2 can still give a *respectable* predictor. When we back-apply this equation to the set of 35 samples, we get an error range of -13 to +17% - a range of 30%. Note that this is better centered and a smaller error than from any of the databook or calibration suites. Of course, this is to be expected from a statistical approach. The applicability of this equation to other samples will be discussed later.
If we perform ordinary multiple linear regression, using all variables except MULi and DIVi, we get several more results:

- RET is zeroed out since it is directly related to SUB in the sample (thus the two are 100% correlated)
- we get an equation of $Cycles = 354 + 1.5 * LD + 31.9 * LI - 30.4 * ST - 7.1 * OPi + 13.6 * IF - 5.1 * GOTO - 51.5 * SUB(2)$ with a $R^2 = .4965$. This equation explains 49.65% of the variability of the cycle count. When back applied we get an error range of -10.3 to +19.4%

Note that:

- the intercept (354) is greater than the number of cycles (200-250) of most of the sample set
- some of the coefficients and parameters are negative and large (31.9 for LI) (-51.5 for SUB)
- the package complained that multicollinearity [3] is a severe problem (i.e. that several of the supposed independent variables of LD, LI, ST, OPi, IF, GOTO and SUB are actually correlated highly).

In other words, this equation is a pure statistical fit of the cycle count to the input variables without any regard for the parameters having an *operative* meaning related to cycles per instruction.
To reduce the multicollinearity problem we generated a correlation matrix for the independent variables and got correlations of LD-OPi of .92, OPi-IF of .99, and LD-IF of .88. This implies we can throw out 2 of these 3 variables since they are all highly correlated. We re-ran regression with just 5 independent variables: LD, LI, ST, GOTO and SUB and got an equation of $Cycles = 273 - 0.9 * LD + 23.3 * LI - 18.9 * ST + .06 * GOTO - 38.7 * SUB(3)$ with a $R^2 = .47$ and an error range of -10 to +22.5%. Note again that there is no operative or implied meaning to the intercept and coefficients. The package reported that multicollinearity is a mild problem (we could throw out other variables - eventually we will end up back at the result reported by Stepwise regression with just LD in the equation).

4. Experimental Results

We applied 2 of the regression equations (1) and (3) above to a sample set of 6 from some Esterel benchmarks. These performed poorly, overestimating the cycle counts by (for (1)) 23% to 60%, and for (3), 87% to 184%. In this sample set, we also had virtual instruction OPc appear and we used the parameter for OPi where applicable (in (3)).
What can account for the poor results? Essentially the applicability of a statistically derived predictor based on sample A, to a new sample B, must rest on an argument of similarity - that sample B has similar characteristics to A. In another perspective, one can argue that Samples A and B could have been drawn from the same underlying population. One way of testing this hypothesis is a 2-sample t-test. This tests the assumption that the 2 samples are drawn from the same underlying normal distribution with equal means and variances.
We need some characteristic of the SW tasks to compare, and one that is independent, for example of cycle count. One idea is that the ratio of the number of virtual IF instructions to the total cycle count is a measure of the *control-dominance* of a SW task. i.e. control-dominated tasks will have a higher ratio than algorithmic or mathematically dominated ones. This is perhaps a tenuous argument, but it is

[2] this is the assumption made in regression

[3] it exposes the redundancy of variables and the need to remove variables from the analysis

difficult to come up with an unambiguous measure of control dominance for tasks.
Using this ratio, we get the following statistics: for the 35-sample automotive control batch, $meanratio = .1077, std.deviation = .016$; for the 6-sample Esterel batch, $meanratio = .0300, stddeviation = .0168$.
The 2-sample t-test rejected the hypothesis that these 2 samples could have been drawn from the same underlying normal distribution; in fact, normality itself was rejected for the second sample (the variable does not have a normal distribution, *i.e.,* in the bell curve).
Using this 2-sample t-test idea, we can apply this kind of discriminating function to new batches of tasks to determine whether it is reasonable to apply a predictor equation drawn from another sample to the new one. Thus this may allow us to discriminate between domains of applicability of predictors. To test this further, we went back to the first batch of 35 automotive control examples and randomly selected 18 of them, and re-ran regression on the 5 variables LD, LI, ST, GOTO and SUB: this gave us the equation $Cycles = 219 + 1.3*LD + 10.9*LI - 10.2*ST - 5.2*GOTO - 21.3*SUB$ with a $R^2 = .568$. We applied this equation to the remaining half of the first batch (17 samples) and got an error range of the predictor of -12% to +5%. Applying the 2-sample t-test to these batches of 18 and 17 tasks, using the characteristic of ratio of IFs to total cycles, we accept the null hypothesis - ie there is a high probability that the 2 samples could be drawn from the same population (which they are). This demonstrates that a predictor drawn from a particular domain sample can with justice be applied to further samples from that domain, and that a simple discriminator can be used to check if samples of SW tasks could indeed be drawn from the same population (and thus the applicability of the discriminator).

4.1 Further experiments - interpreting a predictor equation

We were able to add additional samples to our sample of 35 and with a total sample of 45 relatively control dominated samples, use of stepwise and multiple regression analysis, and some *art* eventually produced a very interesting predictor equation for processor A: (the art consisted in the selection of variables in order to reduce the independent set to a small yet interesting core) $Cycles = 75 + 1 * (OPi + OPc) + 3.4 * IF + 20 * SUB(4)$. This had an error range on back-substitution into the 45 samples of -30 to +20%. Looking at the samples, the IF to Cycle count ratios varied quite considerably, making us question whether these sample sets could really legitimately be combined. More interesting, perhaps, is to consider if there is a hidden meaning behind the parameters (this interpretation stretches beyond the statistics into the art). If we do what we cannot do based on the technique, and ascribe *operative* meaning to the parameters we can suggest that

- 75 is equal to the number of cycles on processor A to set up and close a task run
- each basic integer or character operator (OPi and OPc) attract 1 real machine cycle
- each IF (which then may lead to a branch) attracts 3.4 cycles, and
- each subroutine call and associated return attracts 20 cycles. On processor A, this implies something like 4-5 user registers are active on a call and need to be restored on a return, which seems reasonable.

However this kind of overloaded interpretation needs to be done quite carefully. Applying this randomly to more control oriented samples for processor A would be interesting.

4.2 Further experiments - Mathematical (fft) tasks

To further study the techniques, we took a set of Virtual instruction and cycle counts for 18 FFT tasks. Here we used the predictor (4) and found that it was very poor - over 100% error. We generated a predictor using regression for the FFT and got $Cycles = 286,387 + 2153 * MULd$. In fact the intercept of 286,387 could be thrown out and we could use $Cycles = 2153 * MULd$ - these benchmarks ran for a huge number of cycles (eg. 564,038,767, or over 1 billion cycles for others), thus the intercept of 286 thousand is trivial in comparison. In these benchmarks, the error using this predictor is +/-1%. What we have discovered is actually a *kernel* function [7] - an *internal* kernel function, rather than an *external* one. A kernel function is a predictor for a heavily mathematical SW task in which the cycle count is dominated by statically-predictable mathematical operations rather than dynamic control dominated branching. Due to the static nature of the computations (for example, loops with *a priori* known iteration sizes rather than based on dynamic iteration counts or convergence tests) kernel functions can be highly predictable and with very low error. The FFT examples clearly demonstrate this phenomenon. In this case, the kernel function is expressed in terms of an *internal* characteristic (the number of MULd's is equal to double multiplies) in the task, rather than an *external* characteristic (sample size, etc.). Either kind of kernel function is possible.
To reinforce our analysis of populations and thus the applicability of a predictor equation derived from one sample set in one domain being used on another, we ran another 2-sample t-test on the 45 control-sample and the 18-FFT sample. We again used the ratio of Virtual IFs to total cycles

as the discriminator. For the 18-FFT example, the average ratio was around .00035 with a very low standard deviation (ie. .035%, as opposed to about 10% for the 45-sample average). The 2-sample t-test rejected very soundly the hypothesis that these 2 samples could have been drawn from the same population. Thus a predictor drawn from one batch would have little relevance if used on the other, as our results indicate.

5. Conclusions

This study has discussed a method to derive, using regression analysis, statistically-based predictor equations for SW estimation, based on task samples from particular domains. We have also begun to study discriminator functions and tests to allow us to make conclusions on the applicability of an estimator derived in one domain to be used on subsequent samples. This will give us better confidence in the transportability of such estimators. Further, the use of these techniques on heavily mathematical examples allows the identification of internally-based kernel functions.
However, the work is far from complete. More benchmarks drawn from a wider set of domains is of interest. We are, at the moment, trying these techniques on the embedded benchmarks developed by the EDN Embedded Microprocessor Benchmark Consortium EEMBC [2]. These are drawn from several interesting domains such as automotive engine control, industrial control, wireless and wired communications, and multimedia. Deriving predictors from such widely disparate domains, and studying their accuracy in other domains, and the use of discriminating functions, using in this case an industry standard set of SW tasks, should prove very interesting. In addition, we are planning to work with specific design groups using VCC in applying these techniques to their specific design domains, choice of processors, compilers and optimizations. We believe that the more particular and bounded is the application space, the greater accuracy will be possible.

References

[1] Ace home page. Technical report, ACE, http://www.ace.nl/cont.htm.

[2] Eembc home page. Technical report, EEMBC, http://www.eembc.org/.

[3] Mentor graphics seamless. Technical report, Mentor Graphics, http://www.mentorg.com/seamless/.

[4] Synopsys eagle. Technical report, Synopsys, http://www.synopsys.com/products/hwsw/eagle_ds.html.

[5] W. Baker, M. Hartoog, and G. Martin. Scalable techniques for the performance estimation of codesigned hardware/software systems. *Proceedings of the Cadence Technical Conference*, May 1997.

[6] F. Balarin, M. Chiodo, P. Giusto, H. Hsieh, A. Jurecska, L. Lavagno, C. Passerone, A. Sangiovanni-Vincentelli, E. Sentovich, K. Suzuki, and B. Tabbara. *Hardware-Software Co-Design of Embedded Systems: The Polis Approach*. Kluwer Academic Press, June 1997.

[7] S. Chakravarty, S. Krolikoski, and G. Martin. Dsp software estimation using characterised kernel functions. *Proceedings of DSP Deutschland '99*, Sept. 1999.

[8] J. D'Ambrosio and X. Hu. Configuration-level hardware/software partitioning for real-time embedded systems. *Proc. of Int. Workshop on Hardware/Software Codesign*, Sept 1994.

[9] R. K. Gupta and G. D. Micheli. Constrained software generation for hardware-software systems. *Proc. of Int. Workshop on Hardware/Software Codesign*, Sep 1994.

[10] W. Hardt and R. Camposano. Trade-offs in hw/sw codesign. *Proc. of Int. Workshop on Hardware/Software Codesign*, Oct 1993.

[11] D. J. L. Hintze. Ncss 2000: Statistical system for windows, user guide. Technical report, Number Cruncher Statistical Systems, Kaysville Utah, URL: http://www.ncss.com/.

[12] M. Lazarescu, M. Lajolo, J. Bammi, E. Harcourt, and L. Lavagno. Compilation-based software performance estimation for system level design. *Proc. of Int. Workshop on Hardware/Software Codesign*, May 2000.

[13] Y. Li and S. Malik. Performance analysis of embedded software using implicit path enumeration. *Proceedings of DAC95*, 1995.

[14] S. Malik, M. Martonosi, and Y. Li. Static timing analysis of embedded software. *Proc. Design Automation Conf*, June 1997.

[15] G. Martin and B. Salefski. Methodology and technology for design of communications and multimedia products via system-level ip integration. *Proceedings of Design Automation and Test Conference - Designer Track*, 1998.

[16] C. Y. Park. Predicting program execution times by analyzing static and dynamic program paths. *The Journal of Real-Time Systems, Vol. 5*, 1993.

[17] P. Puschner and C. Koza. Calculating the maximum execution time of real-time programs. *The Journal of Real-Time Systems, Vol. 1*, 1989.

[18] T. Smith and D. Setliff. Towards an automatic synthesis system for real-time software. *Proceedings of Real-Time Systems Symposium*, 1991.

[19] F. Stappert. Predicting pipelining and caching behaviour of hard real-time programs. Technical report, Motorola, C-LAB internal document, Furstenalle 11, D-333102 Paderborn, Germany.

[20] K. Suzuki and A. Sangiovanni-Vincentelli. Efficient software performance estimation methods for hardware/software codesign. *Proc. Design Automation Conf.*, June 1996.

[21] W. Wolf and J. Martinez. C program performance estimation for embedded systems architecture sizing. *Proceedings of the Workshop on Hardware/Software Codesign*, September 1994.

[22] W. Ye, R. Ernst, T. Benner, and J. Henkel. Fast timing analysis for hardware/ software c-synthesis. *Proceedings of the ICCD93*, October 1993.