# Efficient Software Performance Estimation Methods for Hardware/Software Codesign

Kei Suzuki *† and Alberto Sangiovanni-Vincentelli ‡
† Central Research Laboratory, Hitachi Ltd., 1-280 Higashi-koigakubo, Kokubunji, Tokyo 185, JAPAN
‡ Department of Electrical Engineering and Computer Sciences, University of California Berkeley, Berkeley, CA 94720

## Abstract

*The performance estimation of a target system at a higher level of abstraction is very important in hardware/software codesign. In this paper, we focus on software performance estimation, including both the execution time and the code size. We present two estimation methods at different levels of abstraction for use in the POLIS hardware/software codesign system. The experimental results show that the accuracy of our methods is usually within* $\pm 20\%$.

## 1 Introduction

One of the most important purposes of hardware/software codesign is to find the optimum hardware/software partition of a system level specification under particular criteria. These criteria are usually related to the performance (speed, or the number of clock cycles) and cost (number of components, die size, or code size) of target systems. To measure the performance and cost of a target system at a low abstraction level, such as the gate level or assembly-language level, is easy and accurate, but it requires a long design iteration time to find the best partition or configuration for the target system. Performance and cost estimation at a higher level of abstraction is necessary to reduce the exploring time of the design space of the target systems. Furthermore, it can play an important role in the synthesis and optimization of hardware and software. Needless to say, such a performance and cost estimation must be quick and accurate.

The cost of a mixed hardware/software system based on a standard micro-processor (including a micro-controller and a digital signal processor) depends on the size of the hardware. The most effective way to reduce the hardware size is to implement a given functionality with a program on the microprocessor. However, the software implementation of the functionality often fails to meet the performance requirement. One possible approach that avoids this problem is to choose a critical portion in the program which does not satisfy the performance requirement, and then implement it through hardware components. In this approach, the performance estimation of software is the key to finding the critical portion in the software implementation.

In this paper, we present two methods for accurate and fast estimation of software performance in embedded real-time reactive systems designed with the *POLIS* system[1][2]. The *POLIS* system is based on a set of representations of a design at different levels of abstraction. The highest level of abstraction consists of a network of interacting finite-state machines of a particular kind; these are called codesign finite state machines (CFSMs). This representation does not discriminate between hardware and software implementation: it is a semantically unambiguous system level representation. In the design process supported by *POLIS*, once the system description is translated into CFSMs and verified for system level properties by either simulation or formal verification, a partitioning process takes place to identify the components of the design that are candidates for a software implementation. The CFSMs corresponding to this partition are then mapped into another representation called a software graph (s-graph). The s-graph is manipulated to optimize the trade-off between the performance and the code size of the final implementation in the target system. Intelligent partitioning of CFSMs and s-graph optimization must be based on some measure of quality in the final implementation. An estimation at the CFSM level will provide designers with preliminary timing information on the target system, and will provide a measure for hardware/software partitioning. The estimation at the s-graph level will be helpful for s-graph optimization and software module scheduling. Therefore, we developed an estimation algorithm for each level.

A parametric model of the target processor and its compiler as well as two estimation algorithms are presented here. Extensive experimentation has been carried out to evaluate their accuracy. Not surprisingly, estimation at the s-graph level is more accurate than the other two methods. Nevertheless, estimation at the CFSM level is sufficiently accurate for the optimization to be carried out in a meaningful way.

This paper is organized as follows: Section 2 describes related work in software estimation, and in Sec. 3 a formal description of the abstractions used in *POLIS* is given. In Sec. 4,

[1] * Kei Suzuki was a Visiting Industrial Fellow at University of California Berkeley.



33rd Design Automation Conference ®


DAC 96 - 06/96 Las Vegas, NV, USA 

the model for the target processor and the associated compiler is described, and the software estimation algorithms are explained. In Sec. 5, experimental results are shown, and our conclusions are presented in Sec. 6.

## 2 Related work

The performance of software in embedded hardware/software systems depends on the "structure" (instruction mix, data accesses, etc.) of the software program as well as on the components of the target system (i.e., the instruction set and CPU speed). An effective estimation procedure has to model both the target system and the software program at an abstraction level that makes the estimation time reasonable without losing too much accuracy. The structure of the software program becomes more and more difficult to take into account as the level of abstraction rises because the assembly code is further and further removed from the abstract representation.

Most of the results reported in the literature, are from the object code level which is the lowest level of abstraction, and are concerned with software that has a limited structure since programs using constructs such as dynamic data structures, recursive procedures, and unbounded looping are virtually impossible to evaluate [3-6].

In [7], 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.

Software performance estimation is becoming more important as new approaches for the synthesis and verification of real-time embedded systems are developed [8-12]. In this context, several papers have proposed a number of approaches. 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 [9] and [10], statistical methods are proposed to model the performance of a target CPU so that several CPUs can be evaluated with respect to the code that must be run on them. A model proposed in [11] 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 [12], the estimation method used in the COSYMA system is presented. In this system the target system that runs the given software program is also synthesized. The approach consists of "running" the code on an RT-level model of the target system to extract timing characteristics from the simulation results.

## 3 Abstraction models in *POLIS*

In this section, we describe two abstraction models used in the software synthesis process: CFSMs and s-graphs. An example of a CFSM network is shown in Fig. 1. Each CFSM is a reactive finite state machine with a set of input and output events. The specifications for this network is as follows: If the driver's seat belt has not been fastened, five seconds after a car's ignition key is turned on, an alarm will beep for ten seconds or until the key is turned off. In this figure, for

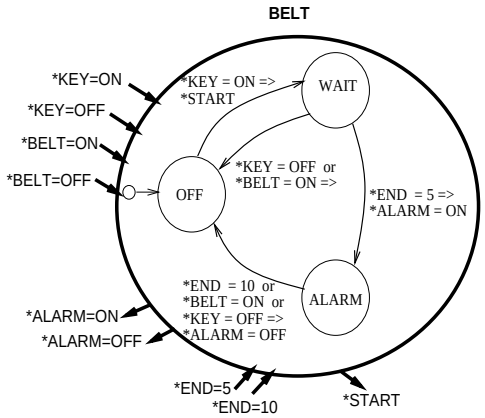


Figure 1: A simple example of a CFSM network

instance, a transition from the OFF state to the WAIT state "*KEY = ON $\Rightarrow$ *START" says that "if the present state is OFF and a *KEY=ON event occurs, then the next state will be WAIT and the *START event will be emitted". This behavior of CFSMs can be represented by a transition relation function, and an MDD (multi-valued decision diagram)[13] is used as its representation.

Each CFSM in a network is partitioned into hardware parts and software parts. For the hardware parts, a CFSM is mapped into an abstract hardware description format, and synthesized into a combinational circuit and a set of latches.

A CFSM implemented as software is translated into a data structure called an s-graph, which represents the control flow of a given behavior. The s-graph is a directed acyclic graph (DAG) with one source node and one sink node. Figure 2 shows an s-graph for the simple seat belt example. Here STATE represents a state in the CFSM. Each node contained in an s-graph can be one of four types: BEGIN, END, TEST, and ASSIGN. The source node is a BEGIN type and the sink node is an END type. BEGIN and ASSIGN nodes have one child, and a TEST node has two or more children. A TEST node is associated with a predicate $P(V)$, and an ASSIGN node is associated with a pair: a function $A(V)$ and an output variable $z$, where $V$ is a set of input variables (events) $V \equiv (v_1, v_2, \ldots, v_n)$. The behavior represented by an s-graph is defined by the structure of the graph and the predicates or functions associated with each node. The semantic of an s-graph is evaluated by the following procedure: (1) start with the BEGIN node, (2) traverse each node along its edge, until reaching the END node, (3) at a TEST node, select one corresponding child with the value of the associated predicate $P(V)$, and (4) at an ASSIGN node, assign the value of the associated function $A(V)$ to the output variable $z$.

After an s-graph is constructed, it is optimized according to criteria that reflect running time or code size. In the last stage

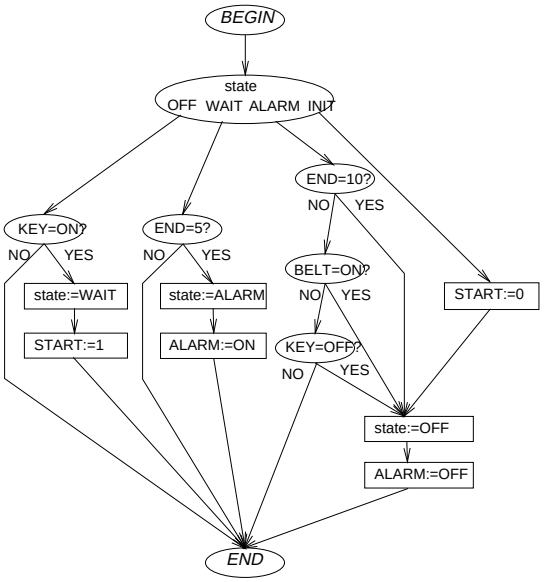


Figure 2: An s-graph for a simple seat-belt alarm system

of the software synthesis flow, the translation of an s-graph into a C program can be done in a straightforward way. The graph is traversed in a depth-first manner. Each node in an s-graph is directly translated into a few C statements depending on its node and associated variable type. A TEST node corresponds to an *if* or *switch* statement, and an ASSIGN node corresponds to an assignment statement. This translation process results in a C program that has the same structure as its original s-graph.

POLIS has another software synthesis path which generates the software module scheduler. The generated scheduler controls the execution of each CFSM implemented as software.

# 4 Performance estimation methods

In the *POLIS* system, partitioning, timing verification, and optimization at the CFSM level and at the s-graph level are guided by an estimation of the final implementation. This estimation is based on models of the target system and of the "structure" of the code.

## 4.1 Modeling the target system

This section describes a model for the target system. The C code generated by *POLIS* always has the same structure (Fig. 3).

Therefore execution time $T$ and code size $S$ are modeled as follows:

$$T = T_{pp} + kT_{init} + T_{struct}$$
$$S = S_{pp} + kS_{init} + S_{struct}$$

Where $T_{pp}$ is the execution time for entering and exiting (the *return* statement) the function ($(1) + (4)$ form Fig 3), $T_{init}$ is the average execution time for initializing local variables

function() $\ldots(1)$
{
  Initialization of local variables(assignment statements); $\ldots(2)$
  Structure of mixed *if* or *switch* statements
    and assignment statements; $\ldots(3)$
  *return*; $\ldots(4)$
}

Figure 3: Structure of the generated C code

($(2)$), $k$ is the number of local variables, and $T_{struct}$ is the execution time for the structure of mixed conditional statements generated from TEST nodes in the s-graph and assignment statements generated from ASSIGN nodes ($(3)$), which is the execution time along a path determined by the structure of $(3)$. Similarly, $S_{pp}$ is the code size for entering and exiting the function, $S_{init}$ is the average code size for initializing the local variables, and $S_{struct}$ is the total code size for $(3)$.

As mentioned in the previous section, an s-graph and the C code generated from the s-graph have the same structure, and the kind of C statement (i.e. *if*, *switch*, or assignment) depends on the corresponding node (TEST or ASSIGN) and the associated variable type in the s-graph. The execution time and code size of each C statement that appears in the s-graph depends on the kind of C statement, the code generated by the target compiler, and the performance of the target CPU. Therefore, $T_{struct}$ and $S_{struct}$ can be expressed as follows:

$$T_{struct} = \Sigma p_i C_t(node\_type\_of(i), variable\_type\_of(i)),$$

where $p_i$ takes value 1 if node $i$ is on a path, otherwise $p_i$ takes 0, and $C_t(n, v)$ is the execution time for node type $n$ and variable type $v$, and

$$S_{struct} = \Sigma C_s(node\_type\_of(i), variable\_type\_of(i)),$$

where $C_s(n, v)$ is the code size for node type $n$ and variable type $v$.

Here, the target system, including the target compiler, can be modeled by $C_t$ and $C_s$. Both $C_t$ and $C_s$ can be obtained as a set of cost parameters by using simple benchmark programs containing a mix of the C statement that appears in the generated C programs and analyzing the execution time and code size of the programs on the target compiler and the target CPU. Also, $T_{pp}$, $T_{init}$, $S_{pp}$, and $S_{init}$ can be determined beforehand, because they are constant and independent of the structure of $(3)$.

The effect of a cache can be included in the parameters. The C-code structure mentioned above does not include loops, so the cache will never be hit unless the code has already been in the cache. Whether the code has been in the cache will depend on the scheduling algorithm for the CFSMs, the number of software implemented CFSMs, the average size of the running code of programs synthesized from CFSMs, and the size of the cache. The average cache hit rate can be extracted through the CFSM level simulation. The average number of pipeline hazards caused by branches can also be predicted from the CFSM level simulation.

Currently we use four attributes to characterize a system, seventeen cost parameters to model the execution time, and fifteen cost parameters to model the code size. The attributes for characterizing the system are:

- the name of the parameter set (such as "MC68HC11"),
- a name for a unit of execution time(e.g., *cycle* or $\mu$ *second*),
- a name for a unit of code size(e.g., *byte*), and
- the size of an integer variable.

The parameters for execution time or code size correspond to the kind of C statement generated from a node in the s-graph. These are:

- a TEST node with an event-type variable (which detects only the existence of an event),
- a TEST node with a multi-valued variable with a bit mask (which tests one bit of the variable),
- a TEST node with a multi-valued variable (which leads to a *switch* statement),
- an ASSIGN node with an event-type variable (which emits an event to other CFSM models),
- an ASSIGN node which assigns a constant to a variable, and
- an ASSIGN node which assigns one variable to another variable.

In the case of a TEST node which has two outgoing edges, the parameters for each edge (i.e. true case and false case) are in the set. For a TEST node which has more than three edges, the executing time for the $k$-th edge is represented as $T_{switch} = C_{base} + kC_{case}$, by using two parameters $C_{base}$ and $C_{case}$.

The other parameters for the execution time and code size are defined for:

- pre-processing and post-processing for a C function such as the *return* statement (together these correspond to $T_{pp}$ and $S_{pp}$),
- a branch operation (generated from a *goto* statement),
- initialization of a local variable (corresponds to $T_{init}$ and $S_{init}$),
- average execution time and size for pre-defined software library functions,
- the size of pointers, and
- the size of integer variables.

A set of benchmark programs to determine the above cost parameters consists of about 20 functions, each with 10 to 40 statements. The value of each parameter is determined by examining the execution time (or number of cycles) and the code size of each function. A profiler or an assembly level code analysis tool, if available, can be used for this extraction process.

Synthesized programs may contain pre-defined functions and user defined functions. We have twenty eight pre-defined functions in the *POLIS* library, including arithmetic functions, logical functions, and relation functions (such as equal or greater than). To improve the accuracy of the estimation for a program which contains these functions, the estimator accepts additional parameters for each pre-defined function or user defined function. For example, the additional parameters for a table-look-up function are defined with a quintuple of (name, maximum execution time, minimum execution time, code size, and bit width for output). These parameters are referred to when an input variable of a TEST node or an ASSIGN node is associated with the function.

## 4.2 S-graph level estimation

The s-graph has a set of properties that makes it possible to estimate the performance of software implemented in a target system.

**Property 4.1** *Each node in an s-graph has a one-to-one correspondence with only a few statements in the synthesized C code.*

**Property 4.2** *The form of each statement is determined by the type of corresponding node.*

**Property 4.3** *The s-graph is a DAG, hence it does not include loops in its structure.*

The reason for the third property is that each s-graph represents only input-output relations. Infinite behavior (looping) is dealt with at the module-scheduler (the operating system) level.

Each node in an s-graph is weighted according to precalculated cost parameters in the pre-process. Each edge in the s-graph is also weighted so that it represents a cost for a branch instruction. Since the s-graph is a DAG, we use a simple DFS to find both the maximum cost path and the minimum cost path, and to sum up the code size. A simple algorithm is shown in Fig. 4, where all the costs that represent the estimated performance are a triple *(max_time, min_time, and code_size)*. Here *max_time* represents the maximum execution time (in cycles), *min_time* represents the minimum execution time (in cycles), and *code_size* represents the size of the binary program code. At the top level, the procedure *SGtrace* begins at the BEGIN node as $sg_i$ and the s-graph is traversed recursively. This procedure returns a triple which contains both a maximum execution time and a minimum execution time from the END node to the $sg_i$. The computational complexity of this graph traversing algorithm is $O(E)$, where $E$ is the number of edges in the s-graph, because each edge in the s-graph is traversed only once. We must also take into account *goto* statements in the synthesized C code, however. The compiler is intelligent enough to not generate branch instructions for redundant statements, such as a *goto* statement to the next statement. Therefore, we adopt the same traversing order as used in the C-code generation process in *POLIS* and add the cost of a branch instruction to each edge only for the following two cases: (1) a node with one child (an edge) which is the unvisited END, and (2) all children nodes that have already been visited, unless the current node is the last node in the traversal.

In the proposed algorithm, a maximum cost path or a minimum cost path can be found by saving a child node which gives the largest cost or the smallest cost while traversing the graph.

**Algorithm:** *SGtrace($sg_i$)*
if ($sg_i$ == **NULL**) return (*C(0,∞,0)*);
if ($sg_i$ has been visited)
return ( pre-calculated $C_i$*(*,*,0)* associated with $sg_i$ );
$C_i$ = initialize (max_time = 0; min_time = ∞, code_size = 0);
for each child $sg_j$ of $sg_i$ {
$C_{ij}$ = *SGtrace*($sg_j$) + edge cost for edge $e_{ij}$;
if ($C_{ij}$.*max_time* > $C_i$.*max_time*)
$C_i$.*max_time* = $C_{ij}$.*max_time*;
if ($C_{ij}$.*min_time* < $C_i$.*min_time*)
$C_i$.*min_time* = $C_{ij}$.*min_time*;
$C_i$.*code_size* += $C_{ij}$.*code_size*;
}
$C_i$ += node cost for node $sg_i$;
return ($C_i$);

Figure 4: An algorithm for traversing an s-graph

The average execution time can be found by using a modification of the above algorithm. Here the average execution time is represented by:

$$C_{ave} = \Sigma p_{ij}(C_t(node\_type\_of(i), variable\_type\_of(i)) + C_e(i,j)),$$

where $p_{ij}$ is the possibility of executing node $i$ and going to node $j$, and $C_e(i,j)$ is the edge cost for edge $e_{ij}$. This formula can be calculated in the same way that the code size is calculated. We need to get each $p_{ij}$ for a previously given s-graph through an s-graph-level simulation or calculation using the probability of the input values.

### 4.3 CFSM-level estimation

While the estimation at the s-graph level is intrinsically accurate, estimating software models at the CFSM level is much more difficult since a CFSM model does not closely reflect the code "structure".

The proposed estimation method for the CFSM is also based on a graph-traversing algorithm. It is, in fact, almost the same as the s-graph-level traversing method described in the previous section. As mentioned, we use MDDs to represent the transition relation function of a CFSM. Each node in an MDD is associated with an input variable or an output variable (events). Both the s-graph and the MDD are decision diagrams. The MDD is a DAG whose structure is similar to that of an s-graph. The major differences are the ordering of variables and the representation of multi-valued variables. A multi-valued variable is associated with a node in the MDD, but might not be in the s-graph, since multi-valued variables can be represented by a set of nodes which test one bit of the variable. The relationship between an s-graph $S$ and the corresponding MDD $M$ can be given:

**Definition 4.1** *Let $P(z_1, \ldots, z_k)$ be a path on a decision diagram that contains nodes $z_1, \ldots, z_k$. Let $V(v_1, \ldots, v_l)$ be a set of variables associated with all the nodes in $P$. Then $V$ is said to appear on $P$.*

**Property 4.4** *Let $V(v_1, \ldots, v_n)$ be a set of input/output variables that appear on a path on $S$. Then $V$ appears on a path in $M$.*

Table 1: Experimental results from the s-graph-level method

| model name | | estimated | measured | % difference |
|---|---|---|---|---|
| FRC | min | 158 | 141 | 12.06 |
| | max | 469 | 496 | -5.44 |
| | size | 654 | 690 | 5.22 |
| TIMER | min | 223 | 191 | 16.75 |
| | max | 938 | 912 | 2.85 |
| | size | 1573 | 1436 | 9.54 |
| ODOMETER | min | 145 | 131 | 10.69 |
| | max | 361 | 363 | -0.55 |
| | size | 454 | 457 | -0.66 |
| SPEEDOMETER | min | 314 | 335 | -6.27 |
| | max | 880 | 969 | -9.18 |
| | size | 764 | 838 | -8.83 |
| BELT | min | 119 | 111 | 7.21 |
| | max | 322 | 323 | -0.31 |
| | size | 511 | 520 | -1.73 |
| FUEL | min | 197 | 171 | 15.20 |
| | max | 533 | 586 | -9.04 |
| | size | 637 | 647 | -1.55 |
| CROSS_DISPLAY | min | 262 | 221 | 18.55 |
| | max | 16289 | 16979 | -4.06 |
| | size | 32592 | 38618 | -15.60 |

**Property 4.5** *In the above property, the corresponding paths on $S$ and $M$ contain the same set of input/output variables.*

The estimation algorithm of the MDD is based on the assumption that the maximum (minimum) cost path in an MDD is usually the maximum (minimum) cost path in the s-graph that is generated from the MDD. The traversing algorithm for the MDD is also based on a recursive DFS traversing, and it is very similar to the s-graph level algorithm shown in Fig. 4. It is clear that there is no relation between the code size and the number of the MDD nodes, because the number of nodes depends on the order of variables in the MDD. This algorithm's complexity is also $O(E)$, where $E$ is the number of edges in the MDD.

## 5 Experimental Results

The experimental results from our evaluation of the proposed methods are shown in Tables 1 and 2. In these experiments, the Motorola's M68HC11 micro-controller[14] and the *introl* C compiler, respectively, were used as the target CPU and compiler. To measure their accuracy, we developed an assembly-code static-analysis tool. This tool analyzes the output lists from the *introl* C compiler and calculates both the maximum execution time and the minimum execution time in terms of the number of clock cycles, and the code size in bytes. The cost parameters used in these experiments were also extracted with this tool. We used additional cost parameters for pre-defined library functions and user defined functions.

In Tables 1 and 2, the difference $D$ is defined as

$$D = \frac{cost_{estimated} - cost_{measured}}{cost_{measured}}$$

Table 1 shows the results when the s-graph-level method was used. The differences in the maximum number of execution cycles are within $\pm 10\%$, the differences in the minimum number of execution cycles are within $\pm 20\%$, and the differences in code size are also within $\pm 20\%$. We found several reasons for these differences when we analyzed the generated assembly code. One of the main reasons is the compiler optimiz-

Table 2: Experimental results from the CFSM-level method

| model name | | estimated | measured | % difference |
|---|---|---|---|---|
| FRC | min | 158 | 141 | 12.06 |
| | max | 460 | 496 | -7.26 |
| TIMER | min | 223 | 191 | 16.75 |
| | max | 917 | 912 | 0.55 |
| ODOMETER | min | 145 | 131 | 10.69 |
| | max | 358 | 363 | 1.38 |
| SPEEDOMETER | min | 314 | 335 | -6.27 |
| | max | 877 | 969 | -9.49 |
| BELT | min | 119 | 111 | 7.21 |
| | max | 402 | 323 | 24.46 |
| FUEL | min | 197 | 171 | 15.20 |
| | max | 620 | 586 | 5.80 |
| CROSS_DISPLAY | min | 262 | 221 | 18.55 |
| | max | 16283 | 16979 | -4.10 |

ation. Synthesized C statements that have the same structure are not always compiled into the same instruction sequence. This is because the optimization process in a compiler considers not only the kind of statement given but also the possible values of the variables in the statement. A second reason is related to the variety in data width, data values, and types of variables. In the HC11, the number of execution cycles depends on the bit width of variables, the assigned values (zero or not zero), and the types of variables (local/global, auto/static). We consider only the bit width of each variable in the cost parameters. A third important reason is related to long-distance branches. In the HC11, the number of execution cycles for conditional branches differs according to the length of the branch. Some of these problems can be solved by using additional cost parameters, and we expect to reduce the difference to about $\pm 5\%$.

The results of the MDD traversing method are shown in Table 2. The minimum number of execution cycles was about the same as with the s-graph level estimation. However, the difference between the estimated and the measured maximum number of execution cycles was larger than when the s-graph level method was used, but the difference was still within $-10\%$ to $+25\%$. The major reasons for these larger differences are: (1) the s-graph is optimized by the software synthesis process, and (2) the MDD traversing method does not take into consideration the costs of the *goto* statements.

On the basis of the experimental results, we can draw the following conclusions:

- The s-graph-level method provides an accurate estimation for all the aspects of analysis: the minimum execution time, the maximum execution time, and the code size. It is a useful technique for optimization in software synthesis because of its accuracy.
- The CFSM-level traversing method is less accurate in accuracy than the s-graph-level estimation, but it is still accurate enough when estimating the minimum and maximum execution time. CFSM-level estimation is important for automatic partitioning of CFSMs into hardware and software parts, and also for scheduler generation.

The CPU time of these estimation programs is usually under five seconds on a DEC system 5900/260. In other words, they make it possible to estimate the minimum and maximum execution times and the code size much faster than is possible when using only the software synthesis process in the *POLIS*.

# 6 Conclusion

We have proposed two software performance estimation methods for use with the *POLIS* hardware/software codesign system. These two methods were implemented and evaluated, and the experimental results showed that:

- The s-graph-level method has the highest accuracy of the two methods, being within $\pm 20\%$ when compared to an assembly-level analysis.
- The CFSM-level MDD traversing method has an accuracy ranging between $-10\%$ and $+25\%$ for the maximum time estimation.

The accuracy of both proposed methods is high enough for use in the processes contained in the POLIS system.

# Acknowledgment

The authors would like to thank Dr. L. Lavagno, P. Giusto, and Dr. E. Sentovich for valuable discussions and comments.

# References

[1] M. Chiodo, P. Giusto, A. Jurecska, H. Hsieh, A. Sangiovanni-Vincentelli and L. Lavagno, "Hardware- Software Codesign of Embedded Systems", IEEE Micro, Vol. 14, No. 4, pp.26-36, Aug. 1994.

[2] M. Chiodo, P. Giusto, A. Jurecska, L. Lavagno, K. Suzuki, E. Sentovich, H. Hsieh and A. Sangiovanni- Vincentelli, "Synthesis of Software Programs for Embedded Control Applications", In Proc. of DAC 95, pp.587-592, June 1995.

[3] E. Kligerman and A.D. Stoyenko, "Real-Time Euclid: A Language for Reliable Real-Time Systems", IEEE Trans. on Soft. Eng., Vol.SE-12, No.9, pp.941-949, Sep. 1986.

[4] P. Puschner and Ch. Koza, "Calculating the Maximum Execution Time of Real-Time Programs", The Journal of Real-Time Systems, Vol. 1, pp.159-176, 1989.

[5] C. Y. Park, "Predicting Program Execution Times by Analyzing Static and Dynamic Program Paths", Real-Time Systems, Vol.5, pp.31-62, 1993.

[6] YT. S. Li and S. Malik, "Performance Analysis of Embedded Software Using Implicit Path Enumeration", In Proc. of 32nd DAC, pp.456-461, June 1995.

[7] T. E. Smith and D. E. Setliff, "Towards an Automatic Synthesis System for Real-Time Software", In Proc. of 12th Real-Time Systems Symposium, pp.34-42, 1991.

[8] J.G. D'Ambrosio and X. Hu, "Configuration-Level Hardware/Software Partitioning for Real-Time Embedded Systems", In Proc. of Int. Workshop on Hardware/Software Codesign, pp.34-41, Sep. 1994.

[9] W. Wolf and J. C. Martinez, "C Program Performance Estimation for Embedded Systems Architecture Sizing", In Proc. of Int. Workshop Hardware/Software Codesign, Oct.1993.

[10] W. Hardt and R. Camposano, "Trade-Offs in HW/SW Codesign", In Proc. of Int. Workshop on Hardware/Software Codesign, Oct. 1993.

[11] R. K. Gupta and G. De Micheli, "Constrained Software Generation for Hardware-Software Systems", In Proc. of Int. Workshop on Hardware/Software Codesign, pp.56-63, Sep. 1994.

[12] W. Ye, R. Ernst, Th. Benner and J. Henkel, "Fast Timing Analysis for Hardware/Software Co-synthesis", In Proc. of ICCD, pp.452-457, Oct. 1993.

[13] T. Kam and R. Brayton, "Multi-valued Decision Diagram", Technical Report UCB/ERL M90/125, U.C.Berkeley, 1990.

[14] M68HC11 Reference Manual, Motorola, 1989.