IPSJ Transactions on System LSI Design Methodology Vol. 3 69–80 (Feb. 2010)

Regular Paper

Performance Estimation with Automatic False-Path Detection for System-Level Designs

Takeshi Matsumoto,[†1] Tasuku Nishihara[†2]
and Masahiro Fujita[†1]

When designing today's highly complicated systems consisting of several hardware and software modules, it is essential to estimate the performance such as worst-case or best-case execution time in early design stages. Such estimation is essential to explore architecture and hardware/software partitioning in system-level design. A maximum execution time estimated topologically without considering false-paths is longer than the real. In this paper, we propose an static estimation method of maximum execution time in system-level designs, considering false-paths. Also, we adopt an approximation approach in order to avoid the path explosion problem. The experimental results show that our method can provide much smaller estimated maximum execution time than the method without considering false-paths. At the same time, the results show us that the maximum execution time can be estimated to a very small range, by applying both simulation-based method and our static method.

1. Introduction

To efficiently design System-on-a-Chips (SoCs), which consists of both software and hardware, system-level design methodology has been introduced to improve the productivity. In system-level design, designers start from abstracted design descriptions of systems considering both hardware and software parts, which results in that designers can take several advantages such that faster simulation, more flexible hardware/software partitioning, and less amount of design descriptions, compared to designing from RTL (Register Transfer Level). At the same time, system-level design enables us to verify/estimate designs from very early design stages, which can reduce the risk of reworking in preceding design stages.

†1 VLSI Design and Education Center, The University of Tokyo
†2 Department of Electronics Engineering, The University of Tokyo

One of the most important issues in system-level design is to estimate the performance (worst-case or best-case execution time) of a given design so that designers can judge whether the design satisfies the required performance or not. If designers can recognize in system-level that the performance does not satisfy the requirement, it should be solved in system-level by refining architectures, hardware/software partitioning, or algorithms. When the performance problem is found in the later than RTL, designers have to refine the architectures or the circuits in RTL or even back in system-level. However, the estimated execution time in system-level can be different from that of synthesized circuits. Even in such cases, the execution time estimation in system-level is still important since we can know relative execution time and explore better architectures and hardware/software partitioning. Therefore, the method we propose in this paper can contribute to supporting architecture exploration in system-level.

For logic gate circuits, maximum (or minimum) execution time analysis is mainly performed statically by the topological traverse of circuits. Throughout this paper, we refer to all estimation methods that do not need input patterns as static method, while some static methods that estimate the maximum execution time only by topological traverse of circuits or design descriptions as topological method. This static timing analysis (STA) techniques have been established to estimate the timing of a given circuit without feeding any input patterns. However, the estimated maximum execution time by STA is usually longer than the real when false-paths in the circuit are not avoided. It means that the accuracy of the static methods depends on how many false-paths are detected and avoided from the estimation. Currently, there are several estimation methods for system-level designs. However, in those methods, designers have to specify false-paths manually, which is not a practical way for today's large designs having a great number of false-paths. Therefore, static execution time estimation methods which can automatically avoid false-paths are required.

Based on the discussion above, in this paper, we propose a maximum execution time estimation method for system-level designs. The target designs can have parallel behaviors and synchronizations among them, which need to be taken into account in system-level design. Also, they are assumed to be written in C-based language such as SpecC [1] or SystemC [2], which are usually used to describe

designs in system level. In this work, we assume SpecC is used to describe the behaviors, but our proposed method is applicable even if designs are described in other C-based languages. When a given design under estimation is very large, the estimation takes too long time since it must keep the path conditions for all execution paths in order to detect false paths. To overcome this problem, our proposed method approximate the intermediate estimation results when a number of execution paths under estimation exceeds to some specified threshold. This approximation can reduce the runtime of the performance estimation by limiting the number of paths under estimation up to the threshold. Thus, the smaller the threshold is, the faster we can get the estimation result. However, when the approximation is carried out, the estimation results may less accurate since the intermediate estimation results are abandoned, which results in that some false-path cannot be detected. Through the experiments, we show that our proposed method is applicable to large designs and its results are much better than ones without considering false-paths. In addition, applying both random simulation based estimation and our static estimation method, we can get a small range of possible maximum execution time of the designs.

The rest of this paper is organized as follows. In Section 2, we introduce some related works. In Section 3, some preliminaries on system-level design descriptions are provided. Then, in Section 4, our proposed method is described. Experimental results are shown in Section 5. Finally, we give concluding remarks in Section 6.

2. Related Works

In this section, existing performance estimation methods are introduced.

Originally, timing analysis technique is used to estimate the signal propagation time in hardware circuits to identify the minimum and maximum delays. Since a timing violation results in an incorrect output from the circuit, it is essential to ensure the 100% satisfaction of timing requirements. Therefore, static timing analysis (STA), which does not depend on input patterns and satisfies 100% coverage, has been developed and used widely [4]. Moreover, since the runtime of static timing analysis based on topological traversal of the circuit (i.e., without considering false-paths) increases only linearly in the size of the circuit, it is widely utilized in industry. Static timing analysis is basically carried out on a timing graph consisting of nodes and directed edges with the value of the delay between two nodes. Also, an STA method which can handle user-specified false-paths is proposed by Belkhale and Suess [5]. In the method, a false-path is represented as a tag in a timing graph.

On the other hand, worst-case execution time analysis of software programs is rather harder than that of hardware circuits because it is undecidable in general and equivalent to solving the halting problem. Puschner et al suggested the restrictions to make this problem decidable: absence of dynamic data structures such as pointers, recursion, and unbounded loops [6]. Since these restrictions force programmers to specify the actual loop bounds, several approaches to specify the loop bounds and false-paths have been proposed. In particular, Park expresses the set of all possible path sequences in a regular expression [7]. In the work, a language named Information Description Language (IDL) is provided for the users to specify loop bounds and false-paths.

Malik and Li propose a method which does not explicitly enumerate program paths by converting the problem of determining the bounds into an integer linear programming (ILP) problem, in order to deal with the exponential increase of the number of paths in the program [8),9]. It is possible to formulate restrictions on the program flows such as loop bounds or excluding path dependencies. Their method is realized as a tool Cinderella [9].

To apply execution time estimation techniques in system-level designs, parallel behaviors and communications/synchronizations among them should be taken into account. In[10),11)], Siebenborn, et al. propose a communication analysis method to detect the points of synchronization in SystemC. The communication analysis is carried out on a communication dependency graph (CDG), then a timing constraint on each synchronization point is produced. After solving synchronization problems, performance analysis is carried out in the form of ILP.

The techniques above can handle only user-defined false-paths. Then, we propose a method to automatically detect false-paths and avoid them from the estimation results. In our method, to decide whether a path-condition is satisfiable, we have to solve the satisfiability of logic equations including arithmetics. For this purpose, we use SMT (Satisfiability Modulo of Theory) solver in our work.

```
behavior B1(void){
  void main(void){
    ...
  }
};
behavior B2(void){
  void main(void){
    ...
  }
};
```

```
behavior Main(void){
  B1 b1;
  B2 b2;
  void main(void){
    par{
      b1.main();
      b2.main();
    }
  }
};
```

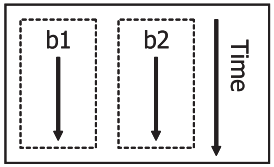

Fig. 1 Parallel execution.

3. Preliminaries

In this paper, the proposed method is described based on the syntax and semantics of SpecC language. Note that the proposed method is applicable other system-level design description languages that have the ability to describe parallel behaviors and synchronization. SpecC is an extension of ANSI-C language in order to describe hardware functionalities such as hierarchical structure, parallel behavior, synchronization, execution time, communication, and state transition. In the remaining of this section, we explain how parallel behaviors and synchronizations are described in SpecC.

3.1 Parallel Behavior

Parallel behaviors in SpecC are described using `par` statements as shown in **Fig. 1**. Only `main` methods of behaviors can be put under `par` statements, and then these behaviors are executed in parallel. In Fig. 1, the behavior `b1` and `b2` are executed in parallel. In other words, the executions of `b1` and `b2` start and end at the same time.

3.2 Synchronization

While parallel behaviors are communicating with each other, synchronization is required to guarantee the execution order of statements which access shared variables. Synchronization in SpecC is described by `wait/notify` statements and `event` variables as shown in **Fig. 2**. `event` variables are used as arguments of `wait/notify` statements. A `wait` statement suspends the execution of the behavior until the `event` variable of its argument is triggered by a `notify` statement. In Fig. 2, the statement `x = 0` in the behavior `B1` is guaranteed to be executed before the statement `x++` in the behavior `B2` by synchronization.

```
behavior B1(int x,
            event e){
  void main(void){
    x = 0;
    notify(e);
  }
};
```

```
behavior B2(int x,
            event e){
  void main(void){
    wait(e);
    x++;
  }
};
```

Fig. 2 Synchronization.

```
void main(void){
  int a, b;
  a = 0;
  waitfor(10);
  b = a + 5;
  waitfor(20);
}
```

Fig. 3 Description of execution time.

3.3 Execution Time

In order to deal with timing information in system-level, execution time can be described by `waitfor` statements. The two `waitfor` statements in **Fig. 3** represent that 10 unit time passes after the execution of `a = 0` and 20 unit time passes after the execution of `b = a + 5`, respectively. In SpecC, the execution of statements except for `waitfor` statements does not take any time, hence, does not affect the execution time estimation. In this work, the estimation is carried out by collecting and counting `waitfor` statements.

As described above, our proposed method is applied to the design descriptions where `waitfor` statements are already included. Therefore, an execution time of each assignment (or each basic block) must be figured out in advance. Those `waitfor` statements are usually inserted in the design flow in the following way.

- For hardware parts, the execution time is derived from the results of behavior synthesis that is carried out to know rough performance/area/power of the hardware
- For software parts, the execution time is derived for each assignment (or each basic block) based on the number of operations in the assignment (or the block)

- When there are any timed communications with modules outside, the timing constraints can be described in the design descriptions using `waitfor`

4. Proposed Method

In this section, we introduce the method to statically estimate the maximum execution time of a given design in SpecC.

4.1 Problem Definition

Given the following things as inputs, our proposed method generates an estimated maximum execution time of the given design.

- A system-level design description in SpecC with `waitfor` statements
- The threshold number of paths. This number will be used to apply the approximation of the estimation to avoid the path explosion problem.
- The constraints to the input signals of the design if any

As explained in the previous section, in this work, we assume that `waitfor` statements are inserted to SpecC designs to describe the execution time of statements, in advance to the application of the proposed method. Then, our method tries to detect the maximum execution time in a given SpecC design. In other words, the method identifies the true path where the sum of the execution times in `waitfor` is largest.

In addition, we assume that the following restrictions are satisfied in a given design description.

- Pointers, dynamic data structures, dynamic memory allocations, and recursive function calls are removed from the design descriptions by appropriate transformation
- There is no deadlock. The proposed method does not generate the estimated execution time in the presence of deadlocks, since the execution may not finish if any deadlock exists.
- When the design has loops, the number of the iteration must be specified for each loop by users.

To remove pointers and dynamic memory allocations, pointer analysis methods need to be applied to identify the variables that are pointed by pointer variables [15)–17)]. To remove recursive function calls, we unroll them up to some sufficient number of times. This number of times for unrolling is assumed to be given by users. If the number is smaller than the real execution, the estimated execution time by the proposed method may shorter than the real. False-paths in design descriptions before the removal of pointers, dynamic memory allocation, and recursive function calls are still kept in design descriptions after the removal of them. On the other hand, removal of pointers may introduce additional false-paths in design descriptions, since the variable pointed by a pointer is generally path-sensitive and not decided before the execution, which can make our performance estimation described in Section 4.3 worse.

To satisfy the second restriction, deadlocks must be detected and removed before the execution time is estimated. This can be done by usual verification process applying simulation and formal verification methods such as one proposed by Sakunkonchak [13)].

4.2 False Path Detection and Removal

One of the main contributions of this work is to detect and remove false-paths from the analysis automatically. To calculate the false-path aware maximum execution time, a pair of execution time (from the beginning to the current point) and the corresponding path condition is maintained for each path. To realize this, the proposed method keeps a set of $< path\ ID, delay, path\ condition >$ for each path. Then, it traverses the control flow of the given design with the set updated in the following way.

- At the beginning, only one element $< path_0, 0, true >$ belongs to the set.
- For a `waitfor(t)` statement, *delay* of the corresponding paths will be incremented by t.
- For a conditional branch, the element of the corresponding path $< path_n, t, cond >$ is duplicated into two elements $< path_n, t, cond >$ and $< path_{n+1}, t, cond >$. Then, for the path going to then-side of the branch, the path condition is updated by adding the condition of the branch. On the other hand, for the path going to else-side, the path condition is updated by adding the negation of the condition.
- For an assignment statement, the post-condition of the assignment (i.e., the assigned variable in the left-hand side of the assignment must be equal to the right-hand side expression) is added to the path condition of the element.
- If the path condition of an element becomes $false$, the element is removed

```
void main(int input){
  int a;
  if(input > 0){
    a = 1;
    waitfor(3);
  } else {
    a = 0;
    waitfor(4);
  }
  if(a == 0){
    ...
    waitfor(2);
  } else {
    ...
    waitfor(5);
  }
  ...
}
```

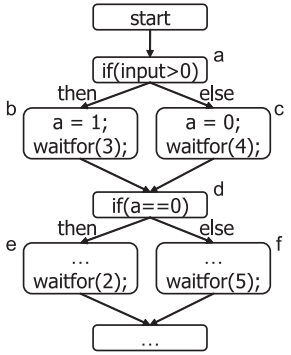

Fig. 4 An example of false-path reduction.

from the set.

- For a `par` statement under which n behaviors are running in parallel, a new $n-1$ elements that have the same delay and path condition are created. Then, each element of the total n elements is assigned to one of the parallel behaviors.
- For a `wait` statement, if the execution delay of the corresponding `notify` statement, T_{notify}, is larger than the delay of the element, T_{curr}, the delay of the element is updated to T_{notify}.
- For a `notify` statement, there is nothing to be done.

At the points where multiple control flows are merged, the traversal suspends until all paths along the merged control flows are traversed. The end of `par` is also considered to be a merge point of control flow. If there are multiple elements having the same path condition at the end of `par`, the delays of them are set to be the maximum among them.

We introduce the method with an example shown in **Fig. 4**. At the beginning of the execution, the initial execution time and path condition are set to 0 and *true*, respectively. Next, a conditional branch `if(input > 0)` comes. The times and conditions of both then-side path ($a \rightarrow b$) and else-side path ($a \rightarrow c$) are calculated as follows. The new path conditions are calculated by taking the conjunctions of the current condition, the branch condition for the taken path (then or else), and the assignment relations of variables. Also, the new execution times are calculated by summing up the arguments of `waitfor` statements in the corresponding branch.

- $Path(a \rightarrow b)$
  $Exe.time : 3$, $Condition : (input > 0)\&\&(a = 1)$
- $Path(a \rightarrow c)$
  $Exe.time : 4$, $Condition : (input \leq 0)\&\&(a = 0)$

Next, the execution times and the path conditions of both then-side path ($d \rightarrow e$) and else-side path ($d \rightarrow f$) are calculated as follows.

- $Path(d \rightarrow e)$
  $Exe.time : 2$, $Condition : (a = 0)$
- $Path(d \rightarrow f)$
  $Exe.time : 5$, $Condition : (a \neq 0)$

Then, these pairs are merged with the current set by taking a direct product as follows.

- $Path(a \rightarrow b) \wedge Path(d \rightarrow e)$
  $Exe.time : 5$, $Condition : (input > 0)\&\&(a = 1)\&\&(a = 0) \rightarrow infeasible$
- $Path(a \rightarrow b) \wedge Path(d \rightarrow f)$
  $Exe.time : 8$, $Condition : (input > 0)\&\&(a = 1)\&\&(a \neq 0) \rightarrow feasible$
- $Path(a \rightarrow c) \wedge Path(d \rightarrow e)$
  $Exe.time : 6$, $Condition : (input \leq 0)\&\&(a = 0)\&\&(a = 0) \rightarrow feasible$
- $Path(a \rightarrow c) \wedge Path(d \rightarrow f)$
  $Exe.time : 9$, $Condition : (input \leq 0)\&\&(a = 0)\&\&(a \neq 0) \rightarrow infeasible$

Since the conditions of the two path conditions $(a \rightarrow b) \wedge (d \rightarrow e)$ and $(a \rightarrow c) \wedge (d \rightarrow f)$ are false, the elements of these paths are removed from the current set. Therefore, the pairs of the other two feasible paths are maintained afterward.

To decide the validity of the path conditions, we use some SMT solver in this work. It can decide the validity/satisfiability of a logic including arithmetics. However, it cannot decide the validity/satisfiability when a path condition includes very complicated non-linear arithmetics. In such cases, the element of the path cannot be removed from the set.

4.3 Approximation of Estimation

Although the false-path aware method proposed in the previous section can

reduce the number of paths, the set of pairs of the execution time and its path condition of all feasible paths are maintained during the analysis. Since the number of feasible paths is usually exponential to the size of the design, the proposed method may not work. Therefore, we proposed to approximate the intermediate estimation results in order to reduce the number of paths if the number of paths exceeds a certain user-defined threshold.

The algorithm of the proposed method including this approximation method is shown in **Fig. 5**. When the approximation is applied, the elements maintaining the execution time and the path condition at that point are destroyed and the estimation process continues with a new set having only one element where the execution time is the maximum of the old set and the path condition is $true$. The approximation is defined to create a new element $< path_{new}, t_{new}, true >$ and remove all elements corresponding to the paths at a merge point, where t_{new} is the maximum execution time in the elements that will be removed. This approximation occurs only at a merge point of control flow if the number of the execution paths that is merged at that point exceeds the specified threshold number.

The approximation makes the analysis results less accurate since the false-paths cannot be considered any more even if they can be detected only checking the conditions both before and after the point where the approximation is applied. Therefore, the results may be larger than the real maximum execution time. However, this method enables to work the proposed method for large designs which have huge number of paths. If the conditions of most false-paths can be decided to be false locally within the areas where no approximation is carried out, then the estimation results will close to the real execution time. Also, it is well known that, in most of false-paths in large programs, infeasibility of the path conditions is caused by infeasibility of some branch conditions located in very small portions of the source codes. Because of this locality, the proposed method can detect and avoid many false-paths even when the approximation is performed, as long as the threshold number is not set to be too small. If the threshold number is too small, the approximation is carried out very frequently, which results in a large number of false-paths are missed. The discussion in this paragraph will be confirmed through the experimental results.

```
Given: CFG (control flow graph) of a design description and a threshold number
EstimateMain {
  start := (the start node of CFG of a given design description)
  INIT_SET := { <0, 0, TRUE> };
  SET := EstimateWithApproximation(start, INIT_SET);
  return GetMaxExecutionTime(SET); //Find the maximum time in SET
}

EstimateWithApproximation(curr_node: CFG node,
                          SET: a set of <path_id, time, condition> ) {
  node := curr_node;
  while(node is not "end" nor "merge point of control flows") {
    if (node is "conditional branch") {
      //Do estimation along THEN path until the merge point
      AddBranchCondition(SET); //Add the branch condition to all elements in SET
      then_node := GetNextNodeInThenBranch(node);
      SET_then := EstimateWithApproximation(then_node, SET);

      //Do estimation along ELSE path until the merge point
      AddNegatedBranchCondition(SET); //Add the negated branch condition to all elements in SET
      else_node := GetNextNodeInElseBranch(node);
      SET_else := EstimateWithApproximation(else_node, SET);

      SET := Merge(SET_then, SET_else); //Just merge two sets into one set
      node := GetMergeNode(node); //Get the merge point of the branch

      //If the next node is not a merge point, and # of elements in SET is greater than the threshold,
      //then do approximation.
      //Otherwise, SET will be approximated, if needed, at the next merge point
      node := GetNextNode(node); //Get the next node of the merge point
      if(node is NOT "merge point of control flows" and |SET| > threshold) {
        SET := Approximate(SET);
      }
    }
    else {
      SET := Estimate(node, SET); //Apply the method in Section 4.2
      node := GetNextNode(node); //Get the next node in CFG
    }
  }
  return SET;
}

Approximate(SET: a set of <path_id, time, condition>) {
  max_time := GetMaxExecutionTime(SET); //Find the maximum time in SET
  new_id := GetNewID(); //Get a new path ID
  //Return a set consisting of one element having the max time in the previous set and a condition TRUE
  return { <new_id, max_time, TRUE> };
}
```

Fig. 5 Algorithm of the performance estimation with approximation.

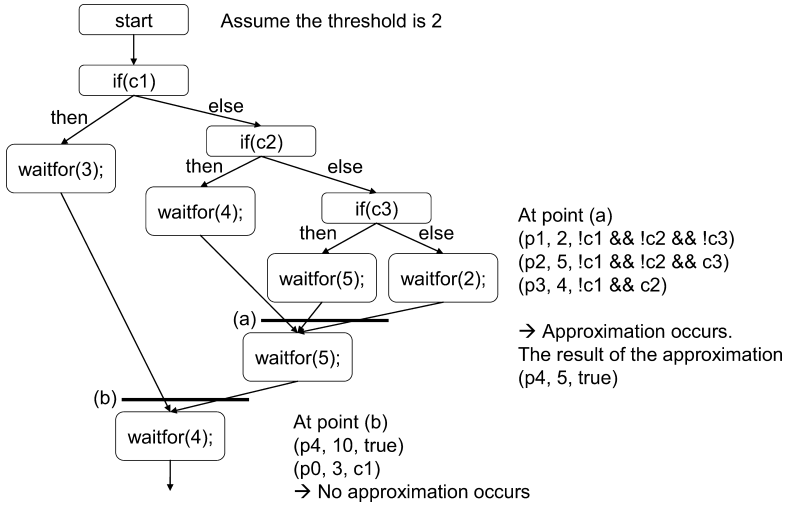

Fig. 6 An example of dividing estimation.

The threshold number of execution paths under estimation can be seen as a trade-off between the accuracy and the speed of the proposed performance estimation method. If we specify a larger number as a threshold, more false-paths can be detected and avoided from the estimation. It results in that the number of paths under estimation is reduced, hence, the runtime of the estimation will be reduced. However, large design descriptions typically have exponentially larger true-paths than false-path. Therefore, even when more false-paths are avoided from the estimation by using a larger threshold, the increase of the number of the kept paths is still exponential in most cases. Considering the discussion in this section, smaller threshold generates less accurate estimation results, but performs the estimation faster.

An intuitive illustration of this approximation is shown in **Fig. 6**. Here, we assume that the threshold number is specified to 2. At the point (a) in the figure, since there are three true paths from $p1$ to $p3$ in the set, the method approximates the estimation. As a result, the new element $p4$ is generated and the estimation is continued only for this element. On the other hand, at the point (b), we have only two paths $p0$ and $p4$, hence, no approximation occurs.

```
Approximate_N(SET: a set of <path_id, time, condition>
              N: the number of paths kept at approximation) {
  SET_N := GetTopN-LargestPaths(SET); //Get the top N largest paths in SET
  SET := SET \ SET_N; //Remove the top N largest paths from SET
  time := GetMaxExecutionTime(SET); //Find the maximum time in SET
  new_id := GetNewID(); //Get a new path ID
  //return the top N largest paths in the original SET and a new path
  return { SET_N, <new_id, time, TRUE> };
}
```

Fig. 7 Algorithm of the approximation remaining N paths.

In addition, we propose a method that keeps the top N largest execution time paths at every approximation, where N is a user-defined value, to improve the accuracy of the estimation. When this is applied, the top N largest execution time paths and their conditions are kept, while the other paths are removed. The algorithm of this approximation keeping N paths are shown in **Fig. 7**. In the algorithm, when an approximation occurs, the top N largest paths are selected from all paths at that point. Then, those selected N paths and a new path, which has the execution time of the $(N+1)$-th largest paths and the condition *true*, are kept after the approximation. If it is applied, the procedure `Approximate` in Fig. 5 is replaced by the procedure `Approximate_N` in Fig. 7. Since only N paths can be selected at an approximation, we sometimes need to select paths that have the same execution time so far. In such cases, the remaining paths are selected randomly. If this is applied, we can consider the estimation of the paths that have large execution time and across the point where the approximation is applied, which may result in more accurate false-path detection and estimation results. However, if we keep many such paths, the required computation for the estimation will increase with N.

5. Experimental Results

In this section, we show the experimental results to show the effectiveness of our method. We applied the proposed method to the following three example system-level designs.

- **Elevator controller.** An elevator controller installed in a building having six floors. The behavior of the controller is similar to a finite state machine. In the experiments, the target of the performance estimation is set to one

Table 1 The characteristics of the examples.

Example	Lines of code	# of branches	# of paths
Elevator controller	5705	245	2.9×10^{47}
ADPCM decoder	4765	913	2.2×10^{48}
ADPCM encoder	4813	921	7.8×10^{48}

state transition.

- **ADPCM decoder.** The target of the estimation is the procedure to decode consecutive eight 4-bit input data.
- **ADPCM encoder.** The target of the estimation is the procedure to encode consecutive eight 16-bit input data.

The characteristics of those designs are shown in **Table 1**. Note that the numbers of paths shown in the table are corresponding to the whole execution paths including false-paths.

We define normalized remaining range e as:

$$e = \frac{T_{prop} - T_{sim}}{T_{topol} - T_{sim}}$$

where T_{prop}, T_{topol}, and T_{sim} denote the result of the proposed method, the result of the topological estimation (i.e., not considering false-paths), and the result of the random simulation, respectively. The normalized remaining range e shows how the range of the possible maximum execution time is limited by the proposed method, compared to the conventional topological estimation that does not consider false-paths. If e is 100%, it means that the result by the proposed method is equal to that by the topological estimation. Actually, we do not know the real maximum execution time of the experimented designs, we refer to T_{sim} as lower bound of the maximum execution time. T_{sim} of each example is obtained by simulation with 2 billion random patterns, which takes several hours for each designs. Also, we refer to T_{topol} as upper bound of the maximum execution time, since it is obtained without considering false-paths. Therefore, the real maximum execution time T_{real} exists between T_{topol} and T_{sim}. T_{prop} obtained by the proposed method is always not less than T_{real} since some false-paths may not be avoided. If we can get less T_{prop}, it means that we can identify the smaller range where T_{real} potentially exists. In that sense, $T_{prop} - Tsim$ can be seen as a remaining range to identify T_{real}. In addition, by divided by $T_{topol} - T_{sim}$, e can work as a metric to evaluate how accurate the estimation result by the proposed method is. For any results, $0 \leq e \leq 1$ is always satisfied. When e is smaller, which means T_{prop} is smaller, we can say that the estimation result by the proposed method is more accurate.

Table 2 Experimental results for different thresholds on an elevator controller design. $T_{sim} = 148, T_{topol} = 406$.

	Threshold for approx.	# of approx.	# of CVC3 calling	Processing time [s]	Estimated exe. time	Remaining range [%]
By the topological method						
EL1	-	-	-	0.3	406(T_{topol})	100
By the proposed method without approximation						
EL2	∞	-	-	> 50000	N/A	N/A
By the proposed method with approximation and various thresholds						
EL3	10000	2	786772	13240	157	3.5
EL4	5000	2	125964	829	152	1.6
EL5	2000	2	121443	783	152	1.6
EL6	1000	3	115256	765	159	4.3
EL7	500	3	134969	1013	160	4.7
EL8	200	5	39231	250	165	6.6
EL9	100	7	12195	44	201	20.5
EL10	50	10	6215	20	214	25.6
EL11	20	14	3630	12	228	31.0
EL12	10	28	1958	5.2	295	57.0
EL13	5	50	1086	2.5	338	73.6
EL14	2	118	483	0.8	393	95.0
EL15	1	245	245	0.5	406	100

The experimental results are shown in **Tables 2**, **3**, **4** and **5**. The experiments are carried out on a Linux computer with Xeon 3.0 GHz CPU and 5 GB memory. The proposed method is implemented upon FLEC framework [14]. In FLEC, system-level designs are represented in ExSDG, a kind of dependence graph representation. By accessing ExSDG data of the designs, we can traverse control flow and evaluate statements in the designs. As an SMT solver, we use CVC3 [3),12)].

In Tables 2, 3 and 4, the results of variousthresholds for three example designs are shown. The first columns show the identical name of experiments. The first result of each design, EL1, DEC1, and ENC1, is the result by the topological method where false-paths are not considered. Therefore, the results of them are

Table 3 Experimental results for different thresholds on an ADPCM decoder design. $T_{sim} = 788, T_{topol} = 839$.

	Threshold for approx.	# of approx.	# of CVC3 calling	Processing time [s]	Estimated exe. time	Remaining range [%]
By the topological method						
DEC1	-	-	-	7.4	839(T_{topol})	100
By the proposed method without approximation						
DEC2	∞	-	-	> 50000	N/A	N/A
By the proposed method with approximation and various thresholds						
DEC3	500	8	163774	45152	807	37.3
DEC4	200	8	163774	44981	807	37.3
DEC5	100	14	46155	7607	814	51.0
DEC6	50	24	20170	1861	815	52.9
DEC7	20	48	8792	532	808	39.2
DEC8	10	80	5264	294	807	37.3
DEC9	5	175	2791	118	815	52.9
DEC10	2	432	1495	45	815	52.9
DEC11	1	913	913	9.4	839	100

Table 4 Experimental results for different thresholds on an ADPCM encoder design. $T_{sim} = 835, T_{topol} = 901$.

	Threshold for approx.	# of approx.	# of CVC3 calling	Processing time [s]	Estimated exe. time	Remaining range [%]
By the topological method						
ENC1	-	-	-	7.4	901(T_{topol})	100
By the proposed method without approximation						
ENC2	∞	-	-	> 50000	N/A	N/A
By the proposed method with approximation and various thresholds						
ENC3	500	11	150824	43010	884	74.2
ENC4	200	15	58466	12146	884	74.2
ENC5	100	16	44495	6213	885	75.8
ENC6	50	27	24827	2961	887	78.8
ENC7	20	55	9416	577	891	84.8
ENC8	10	88	5066	231	884	74.2
ENC9	5	183	290	113	892	86.4
ENC10	2	448	1490	44	892	86.4
ENC11	1	921	921	9.5	901	100

equal to T_{topol}. The threshold number for approximation is set to be infinitely large in EL2, DEC2, ENC2, which means that no approximation is applied and all paths are tried to be estimated in those experiments. In those experiments, we could not get the estimation result within 50000 seconds. In the tables, except for EL1, DEC1, and ENC1, the sixth columns show estimated execution times

Table 5 Experimental results for the elevator controller design with changing remaining paths ratio at approximation.

Threshold of approx.	# of remaining paths at approx.	Processing time [s]	Estimated exe. time (T_F)	Refinement rate [%]
500	400	1897	153	95.6
500	300	1491	153	95.6
500	200	1149	156	97.5
500	100	841	160	100
500	0	1013	160	100
200	160	702	160	97.0
200	120	567	161	97.5
200	80	489	163	98.7
200	40	416	161	97.5
200	0	250	165	100
100	80	265	183	91.0
100	60	191	187	93.0
100	40	136	196	97.5
100	20	92	197	98.0
100	0	44	201	100
50	40	109	188	87.9
50	30	83	191	89.3
50	20	71	203	94.9
50	10	36	208	97.2
50	0	20	214	100

estimated by our proposed method (i.e., T_{prop}). The estimation results are more accurate when T_{prop} is closer to T_{sim}.

Table 5 shows the results of the experiments for various ratios of keeping paths having largest execution times. This experiment is done for the elevator controller design. The last column shows how much the estimated execution time is reduced from the case all paths are abandoned at approximation (i.e., # of remaining paths is zero). The number is normalized, for each threshold, by the result of the method without keeping paths in approximation. The ratios of the keeping paths in approximation are set to 0%, 20%, 40%, 60%, 80% of the threshold for approximation.

From the experimental results, the following things are found out.

- In the case where false-paths are taken into account and approximation is not applied, the estimation method did not finish within 50000 seconds because of the large number of paths.
- In the case where our proposed approximation is applied, performance esti-

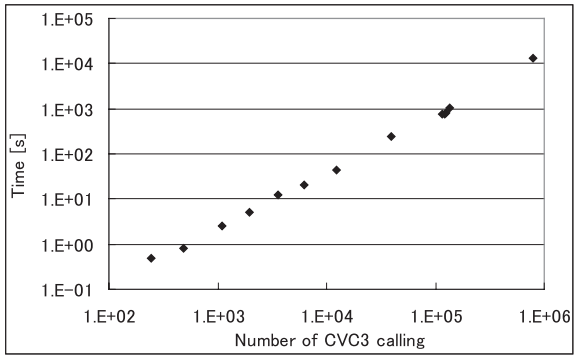

Fig. 8 Process time versus CVC3 calls for the elevator controller design.

mation results vary depend on the threshold number of paths for the approximation. We can see the tendency that the estimated result is smaller when a larger threshold is specified. This is because more false-paths are removed if a larger threshold is specified, in general.

- Processing time is strongly dependent on the number of CVC3 calling. The relation between processing time and the number of CVC3 calling for the elevator controller example is shown in **Fig. 8**. From this fact, we can say that the run time will reduce if the method is refined to have fewer calls of SMT solvers to check the validity of the path conditions.
- When more paths that have largest execution times are kept at every approximation, the estimation results become better. This implies that some infeasible paths are removed by keeping such execution paths across multiple approximation points. However, the processing time is increased as the number of paths remained increase.

As a conclusion of the experiments, the results of the proposed method can effectively reduce the range of possible maximum execution time of the designs. However, how much the range can be reduced is deeply dependent on the designs. Also, the approximation of estimation is essential to finish the estimation. In our experiments, the estimation cannot finish for any example designs if the approximation is not allowed.

The sufficient threshold number for approximation to obtain reasonably accurate estimation results is dependent on designs under estimation. However, from the experimental results, the following two points are found to appropriately select the threshold number for approximation. The first point is that the runtime of the estimation method rapidly increases with the increase of the threshold number. Therefore, it may take too long time if the specified threshold is large. On the other hand, the runtime is very short for small thresholds. The other point is that the estimation results saturate when the threshold becomes larger. Based on those observations, to obtain reasonably accurate estimation results, users should start with a small threshold, where the estimation can be done very quickly, then increase the threshold until the runtime of the estimation becomes too long or the estimation results saturate.

6. Conclusion

In this paper, we proposed the method to estimate the maximum execution time of system-level designs with detecting and removing false-paths automatically. The proposed method can solve the problem in the conventional method that users have to specify false-paths manually. Furthermore, we adopted an approximation method to avoid a path explosion problem. This approximation is applied when the number of paths under estimation exceeds the user-specified threshold. Although the approximation may lead over-estimation because false-paths which lie across multiple approximation points cannot be detected, we can obtain much better estimation results, compared to the method without considering false-paths, in practical run time. Through the experimental results, we confirmed that the proposed method can estimate large designs in several hours and the results is much better than those by the method not considering false-paths. We also show that we can limit the possible maximum execution time into a very small range by applying both simulation-based and our static methods. We believe that the estimated maximum execution time is useful when exploring architecture and hardware/software partitioning in system-level design.

References

1) SpecC: http://www.cecs.uci.edu/˜specc/
2) SystemC: http://www.systemc.org/

3) CVC3: http://www.cs.nyu.edu/acsys/cvc3/
4) Hitchcock, R.B.: Timing Verification and the Timing Analysis program, *Proc. 19th Design Automation Conference*, pp.594–604 (June 1982).
5) Belkhale, K.P. and Suess, A.J.: Timing analysis with known false sub graphs, *Proc. 1995 IEEE/ACM International Conference on Computer-Aided Design*, pp.736–740 (Nov. 1995).
6) Puschner, P. and Koza, C.: Calculating the maximum execution time of real-time programs, *Real-Time Systems*, 1, 2, pp.159–176 (Sep. 1989).
7) Park, C.Y.: Predicting program execution times by analyzing static and dynamic program paths, *Real-Time Systems*, 5, 1, pp.31–62 (Mar. 1993).
8) Malik, S., Martonosi, M. and Li, Y.-T.S.: Static Timing Analysis of Embedded Software, *Proc. 34th Design Automation Conference*, pp.147–152 (June 1997).
9) Li, Y.-T.S. and Malik, S.: Performance Analysis of Embedded Software Using Implicit Path Enumeration, *IEEE Trans. Computer-Aided Design of Integrated Circuits and Systems*, 16, 12, pp.1477–1487 (Dec. 1997).
10) Siebenborn, A., Bringmann, O. and Rosenstiel, W.: Worst-Case Performance Analysis of Parallel, Communicating Software Processes, *Proc. 10th International Symposium on Hardware/software Codesign* (*CODES 02*), pp.37–42 (May 2002).
11) Siebenborn, A., Bringmann, O. and Rosenstiel, W.: Communication Analysis for System-On-Chip Design, *Proc. Conference on Design, Automation and Test in Europe* Vol.1, pp.648–653 (Feb. 2004).
12) Stump, A. Barret, C. and Dill, D.: CVC: A Cooperating Validity Checker, *Proc. 14th International Conference on Computer-Aided Verification*, pp.500–504 (2002).
13) Sakunkonchak, T. Komatsu, S. and Fujita, M.: Synchronization Verification in System-Level Design with ILP Solvers, *Proc. IEICE Trans. Fundamentals of Electronics, Communications and Comuputer Sciencces*, Vol.E89-A, No.12, pp.3387–3396 (Dec. 2006).
14) Kojima, Y., Nishihara, T., Matsumoto, T. and Fujita, M.: FLEC: A Framework for System-Level Debugging Support, Formal Verification and Static Analysis, *Proc. 15th Workshop on Synthesis and System Integration of Mixed Information Technologies*, pp.341–346 (Mar. 2009).
15) Zhu, J. and Calman, S.: Context Sensitive Symbolic Pointer Analysis, *IEEE Trans. Computer-Aided Design of Integrated Circuits and Systems*, Vol.24, No.4, pp.516–531 (Apr. 2005).
16) Semeria, L. and De Micheli, G.: Resolution, Optimization, and Encoding of Pointer Variables for the Behavioral Synthesis from C, *IEEE Trans. Computer-Aided Design of Integrated Circuits and Systems*, Vol.20, No.2, pp.213–233 (Feb. 2001).
17) Clarke, E., Kroening, D. and Yorav, K.: Behavioral Consistency of C and Verilog Programs Using Bounded Model Checking, *Proc. Design Automation Conference*, pp.368–371 (June 2003).

(Received June 1, 2009)
(Revised September 4, 2009)
(Accepted October 31, 2009)
(Released February 15, 2010)

(Recommended by Associate Editor: *Kiyoharu Hamaguchi*)

Takeshi Matsumoto received the B.S., M.S., and Ph.D. degrees in electronic engineering from the University of Tokyo, Tokyo, Japan, in 2003, 2005, and 2008, respectively. He has been a member of VLSI Design and Education Center in the University of Tokyo since 2008. His research interests include computer-aided design and formal verification, especially for high-level designs of digital systems.

Tasuku Nishihara received the B.S. and M.S. degrees in electronic engineering from the University of Tokyo, Tokyo, Japan, in 2005 and 2007, respectively. He is currently a Ph.D. student in the Department of Electronics Engineering, the University of Tokyo. His research interests include formal verification and design analysis for high-level designs of digital systems. He is a student member of IPSJ and IEICE.

Masahiro Fujita received the Ph.D. degree in engineering from the University of Tokyo, Tokyo, Japan, in 1985. He then joined Fujitsu Laboratories Ltd., Atsugi, Japan. From 1993 to 2000, he was with Fujitsu's U.S. research office and directed the CAD Research Group. In March 2000, he joined the Department of Electronic Engineering, the University of Tokyo, as a Professor. He is currently a Professor with the VLSI Design and Education Center, the University of Tokyo. He has been involved in many research projects on various aspects of formal verification.