Equivalence Checking of Loops before and after Pipelining by Applying Symbolic Simulation and Induction

Shanghua Gao[1], Takeshi Matsumoto[2], Hiroaki Yoshida[2,3], Masahiro Fujita[2,3]
[1] *Department of Electronics Engineering, University of Tokyo*
[2] *VLSI Design and Education Center, University of Tokyo*
2-11-16, Yayoi, Bunkyo-ku, Tokyo, Japan, 113-0032
[3] *Core Research for Evolutional Sciense and Technology, JST*
{*gao,matsumoto,hiroaki*}*@cad.t.u-tokyo.ac.jp, fujita@ee.t.u-tokyo.ac.jp*

Abstract— When applications contain large loops, high level synthesis often takes advantage of software pipelining technique in order to improve the performance. High level synthesis with pipelining utilization needs complicated algorithms. So it is desired to check its correctness. In this paper, we propose a novel approach for equivalence checking of loops before and after pipelining. The proposed approach applies a combination of symbolic simulation technique and induction method. We develop a prototype equivalence checker based on the approach. The experimental results show that our proposed method can verify the equivalence of loops before and after pipelining.

I. Introduction

High level synthesis (*HLS*) is an essential step in system-level design. It automatically generates register-transfer-level or gate-level designs from behavioral descriptions written in high-level design language such as C or C++. In many cases, the behavioral descriptions, for example, digital signal processing algorithms, contain large loops that are critical for the total execution time. In order to improve the performance of HLS, software pipelining technique, which can accelerate the execution of loops, is often utilized. In this paper, we call the high level synthesis which takes advantages of software pipelining technique as *pipeline synthesis*.

Since pipeline synthesis needs complicated algorithms, the synthesis result may contain bugs even if the synthesis is performed automatically. In addition, designers may manually optimize the result of pipeline synthesis. In both cases, it is desired to check the correctness of pipeline synthesis. However, it is not an easy task because of the following two reasons: (1) the existence of loops. Although unrolling the loops may enable the verification process, it takes long time for verification if the loop body is large and/or the loop has a large number of iterations. Furthermore, verification methods that unroll loops could not cope with the cases of infinite loops. (2) pipelined executions. In pipelined loops, different iterations of a loop are executed in parallel, which makes it easy to cause resource conflict and easy to violate the dependence across iterations.

To perform formal equivalence checking, symbolic simulation based methods are widely used[1, 2]. During symbolic simulation, they collect all equivalent (sub)expressions in both of the designs under verification and put them into an equivalence class. If a pair of outputs is put into the same equivalence class at the end of symbolic simulation, the equivalence is proved. In [5], symbolic simulation based equivalence checking between designs of HLS is proposed. It translates two designs under verification into FSMD (Finite State Machine with Data) and verifies the equivalence by extracting loop invariants. In general, finding invariants needs a lot of computational effort, hence, takes long time. In our work, the invariants are given from the synthesis result and proved using symbolic simulation and induction. This is possible because our target of the verification is only two loops whose input/output equivalence can be defined by designers with constant throughput and latency. Therefore, we do not have to consider invariants extraction. Recently, in [3, 4], equivalence checking of parallelization or synchronization design is proposed. However, it cannot verify the equivalence of pipelining since their target is the scheduling of behaviors that are assigned to different modules.

In this paper, we propose a novel formal verification method for equivalence checking of loops before and after pipelining. The proposed method uses a combination of symbolic simulation technique and induction method. It has the following four advantages: (a) it performs the verification without unrolling the loops, (b) it can also deal with infinite loops, (c) its computation complexity is much less than that of methods that unroll loops, and (d) at the same time, it checks whether the pipelining result obeys the across-iteration dependence or not. In addition, we put forward to check whether the pipelining result contains resource conflicts or not before performing symbolic simulation. This could reduce the verification time since if the synthesis result contains resource conflict, we already can judge that the loops before and after pipelining are not equivalent. At present we have developed a prototype equivalence checker based on the proposed method. In the experiment, in addition to the pipelining results obtained from automatic pipeline synthesizer, we also prepare some examples by hand where we intentionally insert some bugs that make the pipelined loop unequivalent to the original loop before pipelining. The experimental results show that the developed equivalence checker can detect such unequivalence.

The rest of this paper is organized as follows. Section 2 formulates the verification problem. Section 3 discusses the proposed solution method. Section 4 shows the experimental result and Section 5 gives the conclusion.

II. Problem Formulation

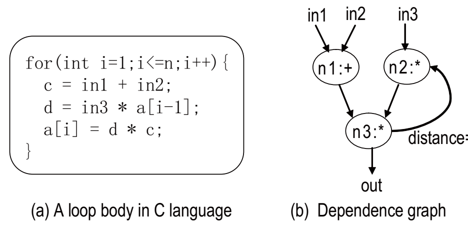
We need to check the equivalence between loops written in high-level design description such as C-based languages (in this work, we assume the original loops are written in C language) and its pipelined one. In this section, we formulate the equivalence of two loops before/after pipelining.

A. Loop before Pipelining

In this paper, we assume that a body of a given loop (i.e. a set of statements inside the loop) consists of assignments and arithmetic/logic computations for variables and arrays. The given loop is transformed into a *dependence graph* representation. In this work, we use low level virtual machine (LLVM) compiler infrastructure for the transformation.

The dependence graph of an innermost loop consists of a set of four elements ($DG = \{V, E, d, \delta\}$). $V = \{v_i | i = 0, ..., n_v\}$ is a set of nodes of the graph, where each node $v_i \in V$ corresponds to an arithmetic or logic operation in the loop. $E = \{e_j | j = 0, ..., n_e\}$ is a set of edges, where each edge $e_j = (v_{i1}, v_{i2})$ represents a dependence from v_{i1} to v_{i2}. $d(v_i)$ is the computation delay of an operation v_i. $\delta(v_{i1}, v_{i2})$ represents a distance function that assigns a non-negative integer to each edge $(v_{i1}, v_{i2}) \in E$. This value indicates that an operation v_{i2} of I^{th} iteration depends on an operation v_{i1} of $(I - \delta(v_{i1}, v_{i2}))^{th}$ iteration. So, if the dependence from v_{i1} to v_{i2} is within the same iteration, $\delta(v_{i1}, v_{i2})$ equals to zero. If the dependence from v_{i1} to v_{i2} is across from an iteration to another, $\delta(v_{i1}, v_{i2})$ is an integer larger than zero.

The dependence graph is nothing but a graph-based representation of the original loops in C language. Each operation of loops in C language has one-to-one correspondence to the node of the dependence graph. And so does the relationship between the operations of loops and the edges of dependence graph. As an example, Figure 1(a) gives a loop in C source code and Figure 1(b) gives its corresponding dependence graph.



```
for(int i=1;i<=n;i++){
  c = in1 + in2;
  d = in3 * a[i-1];
  a[i] = d * c;
}
```

Fig. 1. A dependence graph of a loop

B. Loop after Pipelining

The pipelining result of loops depends on the target architecture on which the original loops are mapped. Here, we describe the architecture with a set of functional units (FUs) $FU = \{fu_i | i = 1, 2, ..., n_{fu}\}$, a set of registers $Reg = \{r_j | j = 1, 2, ..., n_r\}$, and a set of wires $W = \{w_l | l = 1, 2, ..., n_w\}$ that connect them.

The pipeline synthesis is to map the original loops onto the target array-based architecture. It contains the following sub-tasks: (1) Scheduling, which assigns for each operation a time slot at which the operation is scheduled. (2) FU binding, which assigns for each operation a FU on which the operation is executed. (3) Placement, which determines the location of each FU. (4) Routing, which determines by which wire(s) a data is transferred from its source FU/register to its destination FU/register. (5) Register binding, which assigns for each variable (a) register(s) at which its data is stored.

Therefore, the output of pipeline synthesis of loops contains the results of scheduling, binding, placement and routing. In the pipelined loop, each iteration has the same schedule, and is initiated a fixed time intervals later than the previous iteration. This fixed time intervals is termed initiation interval (II).

The results of pipelining consist of the following information.

- On which FU and at which time step an operation v_i in the original loops is mapped.
- By which register and which wires and at which time step a data dependence e_j in the original loops is realized.

Below, the correspondence between the original loop and its pipelined one is defined.

1. Each operation v_i of k^{th} iteration of the original loop $v_i[k]$ (where $k = 1, 2, ..., n$) is corresponding to the pipelining results $(fu(v_i[1]), t(v_i[1]) + II \times (k - 1))$, which means v_i of k^{th} iteration is executed on $fu(v_i[1])$ at time step $t(v_i[1]) + II \times (k - 1)$.

2. Each edge $e_j = (v_{i1}, v_{i2})$ of k^{th} iteration $e_j[k]$ is corresponding to the pipeline results $\{(r_{j1}, t_{p1} + II \times (k-$

1)), $(r_{j2}, t_{p2}+II\times(k-1)), ..., (r_{jm}, t_{pm}+II\times(k-1))\}$ and $\{(w_{l1}, t_{s1} + II \times (k-1)), (w_{l2}, t_{s2} + II \times (k-1)), ..., (w_{ln}, t_{sn} + II \times (k-1))\}$. Here, $\{(r_{j1}, t_{p1} + II\times(k-1)), (r_{j2}, t_{p2}+II\times(k-1)), ..., (r_{jm}, t_{pm}+II\times(k-1))\}$ represents the register binding result which means that the output of operation v_{i1} is stored in register r_{j1} at time step $t_{p1} + II \times (k-1)$, stored in register r_{j2} at time step $t_{p2} + II \times (k-1)$, and so on. $\{(w_{l1}, t_{s1} + II \times (k-1)), (w_{l2}, t_{s2} + II \times (k-1)), ..., (w_{ln}, t_{sn}+II\times(k-1))\}$ represents the routing result which means that the output of operation v_{i1} is transferred by wire w_{l1} at time step $t_{s1}+II\times(k-1)$, transferred by wire w_{l2} at time step $t_{s2}+II\times(k-1)$, and so on.

Here, we illustrate the results of the pipelined loops. Assume that the original loop which contains three operations (as in Figure 1) is going to be mapped on an architecture which contains two functional units (as in Figure 2(a)). The pipelining result is given in the following forms:

1. Scheduling and placement result for operations can be expressed as: (1) $n1[k] : (1 + 2 \times (k-1), FU1)$

   (2) $n2[k] : (2 + 2 \times (k-1), FU2)$

   (3) $n3[k] : (3 + 2 \times (k-1), FU2)$.

2. There are six data transfers, and their register binding and routing results can be expressed as:

   (1) $e(in1, n1)[k] : \{(0 + 2 \times (k-1), r1)\}$

   (2) $e(in2, n1)[k] : \{(0 + 2 \times (k-1), r2)\}]$

   (3) $e(in3, n2)[k] : \{(0 + 2 \times (k-1), r3), (1 + 2 \times (k-1), r3)\}$

   (4) $e(n1, n3)[k] : \{(1 + 2 \times (k-1), r1), (2 + 2 \times (k-1), w1), (2 + 2 \times (k-1), r4)\}$

   (5) $e(n2, n3)[k] : \{(2 + 2 \times (k-1), r5)\}$

   (6) $e(n3, out)[k] : \{(3 + 2 \times (k-1), r4)\}$

   Please notice that only data transfers, whose source FU and destination FU are not in the same location, need to use wires.

For easier understanding, we also give the mapping result on the architecture as in Figure 2(b), and the pipelining result as in Figure 2(c).

C. Equivalence between Loops before/after Pipelining

From the correspondence between loops before/after pipelining described in the previous subsection, we define the following equivalences.

Definition 1 (Equivalence of Input Signal) *For every k, an input signal in_p of k^{th} iteration of the original loop (i.e. $in_p[k]$) is equivalent to an input signal in_q at time step $C_1 + II \times (k-1)$ (i.e. $in_q[C_1 + II \times (k-1)]$), where C_1 is a constant non-negative integer.*

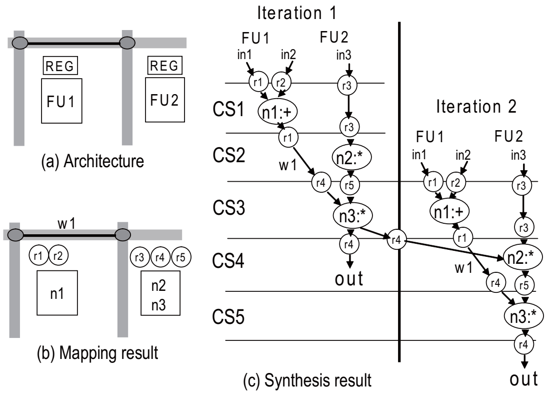


Fig. 2. A synthesis result

Definition 2 (Equivalence of Output Signal) *For every k, an output signal out_p of k^{th} iteration of the original loop (i.e. $out_p[k]$) is equivalent to an output signal out_q at time step $C_2 + II \times (k-1)$ (i.e. $x_q[C_2+II\times(k-1)]$), where C_2 is a constant non-negative integer.*

We can extend such kind of equivalence definition for a set of signals. If input/output signals in $\mathbf{S_p} = \{s_{p1}, ..., s_{pn}\}$ are equivalent to the corresponding signals in $\mathbf{S_q} = \{s_{q1}, ..., s_{qn}\}$ for $C = C_1, ..., C_n$, we denote it as $\mathbf{S_p} \equiv \mathbf{S_q}[\mathbf{C}]$ in the later sections.

D. Problem Definition

The equivalence checking problem of loop before and after pipelining is defined as below:
Given an equivalence of input signals between the original loop and its pipelined one $\mathbf{IN_p} \equiv \mathbf{IN_q}[\mathbf{C_1}]$ and an equivalence of output signals $\mathbf{OUT_p} \equiv \mathbf{OUT_q}[\mathbf{C_2}]$,
prove $(\mathbf{IN_p} \equiv \mathbf{IN_q}[\mathbf{C_1}] \Rightarrow \mathbf{OUT_p} \equiv \mathbf{OUT_q}[\mathbf{C_2}])$.
If it is proved, then the result of the verification is **equivalent**, otherwise the result is **not equivalent**.

The equivalences of the input and output signals are either generated from the results of pipelining automatically, or manually specified by designers. In both cases, the result will be false if the given equivalences are not correct. Note that our proposed method checks the equivalence specified by designers, and does not generate the equivalence to be proved automatically. However, in practical, the equivalence to be proved is strongly desired to be generated automatically, although it is not included in this paper.

III. Proposed Method

In this section, we present our proposed method for the equivalence checking problem in detail. Basically speaking, the proposed method is based on symbolic simulation and induction. Having an observation that if there exists resource conflict in the pipelining results, the pipelined execution of loops is mistaken, and the loops after pipelining is not equivalent to the original loops before pipelining. Thus, to reduce verification time, before performing equivalence checking, we would like to check whether the pipelining results contain resource conflict or not.

A. Checking of resource conflict

In the pipelined execution, the operations scheduled at time steps $T = \{k * II + p | k = 0, 1, 2, ..., n, II = 1, 2, ..., m, p = 0, 1, 2, ..., II - 1\}$ are executed concurrently. Let us name p, where $p \in [0, II - 1]$, a *rolled time step*. Then, the resource constraint requires that for each resource, at every rolled time step p, the resource is not used more than one time.

As stated in Section II., the results of pipelined loop are given from the point of view of operations and edges of original loop. If we can rearrange the results from the point of view of resources (FU/Register/Wire), then we can check whether there exists resource conflict or not easily. The rearranged results of pipelined loops contain the following information:

- For each FU, which operations are placed to this FU and at which time step.
- For each register, which variables are bound to this register and at which time step.
- For each wire, which variables are mapped to this wire and at which time step.

For the example in Figure 2, the results from the point of view of resources can be rewritten as follows:

1. $FU1 : \{(n1[k], 1 + 2 \times (k - 1))\}$ for FU1, which means that operation $n1[k]$ of k^{th} iteration is to be executed on FU1 at cycle $1 + 2 \times (k - 1)$.

   $FU2 : \{(n2[k], 2 + 2 \times (k - 1)), (n3[k], 3 + 2 \times (k - 1))\}$ for FU2.
2. $w1 : \{(n1[k], 2 + 2 \times (k - 1))\}$ for wire $w1$, which means that the output of operation $n1[k]$ is going to be transferred by wire $w1$ at time step $2 + 2 \times (k - 1)$.
3. (1) $r1 : \{in1[k], 0 + 2 \times (k - 1)), (n1[k], 1 + 2 \times (k - 1))\}$ for register $r1$. It means that the input signal $in1[k]$ is stored to register $r1$ at time step $0 + 2 \times (k - 1)$, and that the output of operation $n1[k]$ is stored to register $r1$ at time step $1 + 2 \times (k - 1)$.

   (2) $r2 : \{in2[k], 0 + 2 \times (k - 1))\}$ for register $r2$.

   (3) $r3 : \{(in3[k], 0 + 2 \times (k - 1)), (in3[k], 1 + 2 \times (k - 1))\}$ for register $r3$.

   (4) $r4 : \{(n1[k], 2 + 2 \times (k - 1)), (n3[k], 3 + 2 \times (k - 1))\}$ for register $r4$.

   (5) $r5 : \{(n2[k], 2 + 2 \times (k - 1))\}$ for register $r5$.

After rearranging the pipelining result like this, we can check the resource conflict problem very easily. For each resource, at any rolled time step p, where $p \in [0, II - 1]$, check whether it is used more than one time or not. If for a certain resource, it is used more than one time, then we can judge that the scheduling result contains resource conflict. As for this example, the II equals two and there is no resource conflict in the result.

B. Application of Symbolic Simulation and Induction for Verification of Loops before and after Pipelining

In this section, we propose a method to check loops before and after pipelining for the given equivalence of input and output signals. We apply a combination of symbolic simulation technique and induction method.

We perform the equivalence checking without unrolling the loops. The pipelined loop has following characteristics: (1) Every iteration has the same schedule, and (2) Every iteration starts some fixed time intervals (namely, II) later than the previous iteration, in other words, the II is a constant. These characteristics enable us to apply induction method. Assume the distances of dependence that are included in the dependence graph of the original loops are $\mathbf{D} = \{d_1, d_2, ..., d_r\}$. Here, we denote the maximum distance as d_{max}. Then, assuming the given equivalence of the input signals, the proposed method try to prove the given equivalence by induction as follows.

- *Prove the base case* Assume the equivalence of the input signals of all 1st,...,d_{max}th iterations, prove the equivalence of the output signals and the recurrent output signals of all 1st,...,$(d_{max})^{th}$ iterations.
- *Prove the induction step* Assume the equivalence of the input signals of k^{th} iteration and the equivalence of the recurrent output signals of each $(k - d)^{th}$ iterations, where $d \in \mathbf{D}$, prove the equivalence of the output signals and recurrent output signals of k^{th} iteration.

If the above two conditions are proved by symbolic simulation based equivalence checking method, then we can say that the equivalence of the output signals is satisfied for every iteration of the loop execution.

To prove the equivalence, we perform the equivalence checking by using symbolic simulation technique, where each value of a variable and each operation among variables are treated as a symbol. At present, we have developed a prototype symbolic simulator for the equivalence checking of loops before and after pipelining. In the symbolic simulation based equivalence checking method, all equivalent (sub)expressions are collected in the same EqvClass. If two different EqvClasses are proved to be equivalent by substitution, they are merged into a single EqvClass. Below, we explain how to apply symbolic simulation based method to prove the induction step with the loop examples shown in Figures 1 and 2.

1. First, by applying symbolic simulation of k^{th} iteration of the original loop written in high level design language, EqvClasses are generated. We generate an equivalence class for each statement included in the loop. Figure 3 illustrates this EqvClass generation. For the original loop shown in Figure 3(a), total three EqvClasses are generated for k^{th} iteration as shown in Figure 3(b).

```
for(int i=1;i<=n;i++){
   c = in1 + in2;
   d = in3 * a[i-1];
   a[i] = d * c;
}
```

(a) a loop in C language

```
Equivalence class for iteration k:
E1[k]={ADD(in1[k],in2[k]),c[k]};
E2[k]={MUL(in3[k],a[k-1]),d[k]};
E3[k]={MUL(d[k],c[k]),a[k]};
```

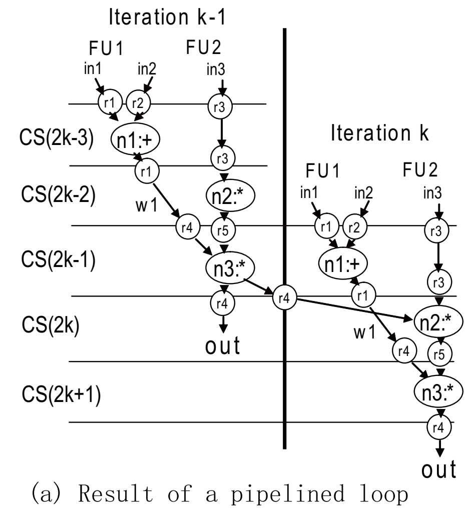
(b) EqvClasses for iteration k

Fig. 3. Generated EqvClasses for the original loop of k^{th} iteration

2. Then, in the same way, EqvClasses are generated for k^{th} iteration of the pipelined loop. Different from the original loop written in high-level design language, the synthesized loop is already mapped to hardware resources. Therefore, all such resources including FUs, registers, and wires are symbolically simulated for each time step from the start of k^{th} iteration to the end. In the example of Figure 4(a), total nine EqvClasses of the time steps from $2k-2$ to $2k+2$ are generated.

3. Finally, EqvClasses coming from the assumptions are added. They consist of two parts: (1) the equivalence of the input signals of k^{th} iteration, and (2) the equivalence of the recurrent output signals of each $(k-d)^{th}$ iteration, where $d \in \mathbf{D}$. For the example, EqvClasses corresponding to the assumptions are shown as Figure 5(a), and the target equivalence that will be proved is shown as Figure 5(b).

Then, with the EqvClasses from the symbolic simulation of both the original loops and the pipelined loop of k^{th} iteration, EqvClass merge is performed until we cannot find any more EqvClass that can be merged. As a result, if the equivalence of the output signals of k^{th} iteration is proved, we can say that the induction step is successfully proved.



(a) Result of a pipelined loop

```
For k-th iteration:
EC1[k]={in1[2k-2],r1[2k-2]}
EC2[k]={in2[2k-2],r2[2k-2]}
EC3[k]={in3[2k-2],r3[2k-2]}
EC4[k]={r3[2k-2],r3[2k-1]}
EC5[k]={ADD(r1[2k-2],r2[2k-2]),r1[2k-1]}
EC6[k]={r1[2k-1],w1[2k],r4[2k]}
EC7[k]={MUL(r3[2k-1],r4[2k-1]),r5[2k]}
EC8[k]={MUL(r4[2k],r5[2k]),r4[2k+1]]}
EC9[k]={r4[2k+1],out[cs(2k+2)]}
```

(b) Equivalence class for iteration k

Fig. 4. Generated EqvClasses for the pipelined loop of k^{th} iteration

IV. Experimental Result

A prototype equivalence checker for loops before and after pipelining has been implemented in c++/linux environment. The implementation of this equivalence checker is completely separate from the automatic pipeline synthesizer. The results of pipeline synthesis of loops are recorded in a file. The equivalence checker reads this file, and another file that records original loop in C description. and generates equivalence classes for both loops under verification by performing symbolic simulation. The experiment is carried out on a PC with Pentium 4 3.2E GHz processor and 2GB memory.

The examples used in this experiment consist of four digital signal algorithms. In particular, *iir* is an infinite impulse response filter. *wavelet* is a part of kernel loops in the wavelet transform. *idct* is a two dimensional inverse discrete cosine transform, and *jfdctfst* is a forward DCT used in MPEG4.

Table I shows the experimental result. In this table, the first column refers to the examples used in the exper-

```
Assumptions:
E1[k]={in1[k],in1[2k-2]};
E2[k]={in2[k],in2[2k-2]};
E3[k]={in3[k],in3[2k-2]};
E4[k]={a[k-1],r4[2k-1]};
```

```
To prove:
(a[k],r4[2k1]);
(a[k], out[2k+2]);
```

Fig. 5. Initial condition and equivalence condition for iteration k

iment. The second column refers to the checking result, namely, the pipelined loop is equivalent to the original loop or not. The third column refers to the run time for the equivalence checking in unit of second, and the last column gives some necessary remarks.

In order to test our developed equivalence checker, besides the pipelined result obtained from pipeline synthesizer automatically, we also make some examples by hand, labeled with a (or two) asterisk(s). As for examples with an asterisk, we intentionally change the pipelining results so that some resource conflicts, such as FU/register/bus conflicts, occur in the pipelined loops. As for examples with two asterisks, we intentionally change the pipelining results so that some bugs occur, for example, a variable fails to be saved to a register at some time step. In the table, the remark *Reg binding* means that there are bugs in register binding, in other words, a variable is not saved at some time step. The remark *Routing* means that there are bugs in the routing result, namely, some data is not transferred successfully to its destination FU. The experimental results show that our developed equivalence checker can detect the unequivalence of examples we make by hand. For the pipelined loops obtained from the pipeline synthesizer, they are proved to be equivalent to the original loops before pipelining. As for the runtime, the equivalence checking of all the examples is finished in less than 1 second.

V. Conclusion

We have proposed an approach to check the equivalence of loops before and after pipelining. The approach is based on induction method and symbolic simulation technique. It performs the equivalence checking without unrolling loops, which enables us to verify loops having large number of iterations in short time, since the computation effort to check the equivalence will significantly increase if loops are unrolled. At the same time, we check the resource conflict in the pipelined and synthesized loops. This is useful to reduce verification time since if there

TABLE I
Experimental results of formal verification

Example	Op	Result	Time (s)	Remark
iir	24	Proved	0.13	—
iir*	24	Not Proved	0.1	Reg_conflict
iir**	24	Not Proved	0.19	Reg_binding
wavelet	30	Proved	0.2	—
wavelet*	30	Not Proved	0.19	FU_conflict
wavelet**	30	Not Proved	0.24	Reg_binding
idct	53	Proved	0.13	—
idct*	53	Not Proved	0.12	Wire_conflict
idct**	53	Not Proved	0.2	Routing
jfdctfst	56	Proved	0.22	—
jfdctfst*	56	Not Proved	0.14	Wire_conflict
jfdctfst**	56	Not Proved	0.18	Reg_binding
jfdctfst_2	111	Proved	0.23	—
jfdctfst_3	166	Proved	0.28	—
jfdctfst_4	221	Proved	0.31	—

exists resource conflict, we need not perform the symbolic simulation any more. In the experiment, we verified several loop examples before and after pipelining including ones that bugs are intentionally inserted. The results showed that our developed prototype equivalence checker detected such unequivalence in very short runtime.

References

[1] G. Ritter, "Formal Sequential Equivalence Checking of Digital Systems by Symbolic Simulation," PhD thesis, Darmastadt University of Technology and Universite Joseph Fourier, 2000.

[2] H. Saito, T. Ogawa, T. Sakunkonchak, M. Fujita, and T. Nanya, "An Equivalence Checking Methodology for Hardware Oriented C-based Specification," *Proc. of International Workshop on High Level Design Validation and Test*, pp.139–144, 2002.

[3] S. Abdi and D. Gajski, "A Formalism for Functionally Preserving System Level Transformations," *Proc. of Asia South Pacific Design Automation Conference 2005*, pp.139–144, January 2005.

[4] S. Abdi and D. Gajski, "Functional Validation of System Level Static Scheduling," *Proc. of Design, Automation and Test in Europe*, pp.542–547, March 2005.

[5] P. Ashar, A. Raghunathan, A. Gupta, and S. Bhattacharya, "Verification of Scheduling in the Presence of Loops Using Uninterpreted Symbolic Simulation," *Proc. of 1999 International Conference on Computer Design*, pp.458–466, 1999.