Exploiting Intellectual Properties in ASIP Designs for Embedded DSP Software

Hoon Choi, Ju Hwan Yi, Jong-Yeol Lee, In-Cheol Park, and Chong-Min Kyung
Department of Electrical Engineering,
Korea Advanced Institute of Science and Technology, Taejon, Korea
hchoi@ieee.org, icpark@ee.kaist.ac.kr

Abstract

The growing requirements on the correct design of a high-performance system in a short time force us to use IP's in many designs. In this paper, we propose a new approach to select the optimal set of IP's and interfaces to make the application program meet the performance constraints in ASIP designs. The proposed approach selects IP's with considering interfaces and supports concurrent execution of parts of task in kernel as software code with others in IP's, while the previous state-of-the-art approaches do not consider IP's and interfaces simultaneously and cannot support the concurrent execution. The experimental results on real applications show that the proposed approach is effective in making application programs meet the performance constraints using IP's.

1 Introduction

As time to market pressures and product complexities increase, the pressure to reuse complex building blocks (also known as Intellectual Property, or IP) increases significantly. Recently, a study says that extreme reuse of IP's will become crucial if chip-design cost is to be kept reasonable. It concludes that systematic and effective design reuse would reduce chip-development cost by 50 percent in three years and by more than 70 percent in six years, compared with the cost of developing chips without reuse [1].

There have been a number of works on the automatic synthesis of interfaces between IP's using different communication protocols [2-7]. However, there have not been many works for the efficient use of IP's as accelerators in the processor-core based designs which are common in industries, e.g., ARM-core based designs. In this case, the interface problem becomes relatively simple because of the fixed processor-core. However, we have to decide the parts of application program to be accelerated and IP's to be used for those. Since the use of IP requires additional area, the IP should be used as little as possible if the performance constraint is met. In addition, we have to select the best interface for the selected IP's from those supported. The selected interface should be as small as possible with meeting the performance constraint. Lastly, since the IP's can run in parallel with the processor-core, we have to consider the possible parallel execution of the processor-core and IP's. A pair of IP and interface that is minimal and satisfies the constraint has more favor. In addition to those problems, if the processor-core is an ASIP-core, there is one more problem: How to incorporate the added IP's and interfaces in the instruction set.

Though the work in [7] pointed out the importance of processor-core based designs and proposed a way to synthesis the interface, it did not consider other problems. The work in [8] handled the selection of hardware accelerators in an ASIP. However, the hardware accelerators were very simple ones such as a multiplier and a divider, thus it did not consider the interface problem and possible parallel execution. These lead us to consider the problem of efficient use of IP's in ASIP's. In this paper, we propose a new approach to select the optimal set of IP's and interfaces to make the application program meet the performance constraints with considering possible concurrent execution.

The rest of this paper is organized as follows. In section 2, we briefly give some backgrounds on the target architecture of the ASIP to be synthesized and our ASIP synthesis system. In section 3, we describe the interface methods we support, and the proposed approach is described in section 4. Experimental results are shown in section 5.

2 Backgrounds

The target ASIP consists of an ASIP-core (also called as kernel) and IP's selected to accelerate the application programs. The ASIP-core architecture is a pipelined DSP processor controlled by μ-programing. Like the most DSP processors, it has a separate address generation unit (AGU), and can access two data-memories (XDM and YDM) simultaneously to fetch operands. The μ-code is composed of eight fields to enable parallel execution of an arithmetic operation and a register move operation. Each operation in a field of the μ-code word is called an MOP (μ-operation).

The ASIP supports three classes of instructions: P, C and S classes. First, P-class contains instructions that are not only primitive but also essential in all applications, i.e., simple arithmetic instructions and control instructions such as branch and call. The P-class instructions are always supported in all of the generated ASIP's and executed in the kernel. Second, C-class is composed of application specific instructions that are more complex than P-class instructions. Though C-instructions are also executed in the kernel with the assistance of μ-codes, they are more powerful than P-instructions because they can control all of the units in the kernel and can reduce the code-memory size and the number of code-fetches [9]. Lastly, S-class is a set of instructions that are supported by the accelerators for high performance. In other words, these are the instructions used to incorporate the IP's into the instruction set.

The overview of our ASIP synthesis system, Partita, is as follows. The inputs are the application program written in C, typical input data for the application, and performance constraints such as maximum execution time allowed. The application program is transformed into a MOP list and sample-executed with the given

DAC 99, New Orleans, Louisiana

typical input data to obtain running frequency profile. The matching of MOP list to the P-instructions, generation of C-instructions, and that of S-instructions are performed sequentially to find a cost-effective solution meeting the performance constraint. The details of these can be found in [9]. After generating instructions we start to generate hardware modules required. If S-instructions are generated, the corresponding IP's are integrated with appropriate interfaces. Other necessary hardware modules such as the decoding unit and the fetch unit are also synthesized with considering the newly generated C-instructions and S-instructions. All newly generated instructions are encoded in the instruction space, and the μ-ROM is optimized with including the μ-codes for the C-instructions and S-instructions. In this paper, we mainly focus on the *S-instruction generation* and the *Interface selection*.

3 Interface Methods

Selecting the best interface method for each IP is crucial to use the full power of IP. In this section, we explain the interface methods we support. The general interface method is shown in Fig. 1. It can have in/out-buffers if needed and the in/out-controller that controls the interface scheme. The protocol transformer transforms IP specific various protocols into our standard synchronous one. We have selected to use the synchronous protocol as our standard one due to the fact that many IP's for DSP applications operate in synchronous (and pipelined) mode. The techniques for the transform have been researched in many works [2-6] and we borrowed one from them, hence we will not go into the subject in detail in this work. The operands in memory are fetched by the in/out-controller and passed to the in-buffer. The data in the in-buffer is passed to the IP via protocol transformer. The results from the IP are stored in the out-buffer, and then passed to memory under the control of the in/out-controller.

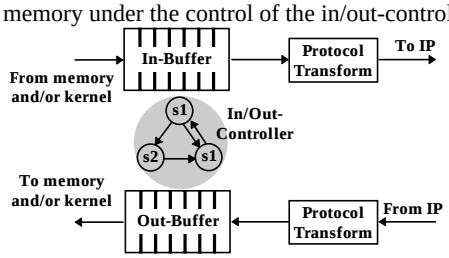

Fig. 1: General interface method

By changing the in/out-controller and/or by inserting/eliminating the buffers, various interface methods are possible. The factors we consider in deciding a specific interface method for an IP are as follows. First, input and output characteristics of IP are considered: the number of input (output) ports, the input (output) data rate, the number of input (output) data, the latency from input to output, and whether the IP is pipelined or not. For example, an IP having more than two in-ports requires the in-buffer because the kernel can transfer only two operands at a cycle to the IP. The input data rate determines whether we can use software interface method or not. Second, parallel execution is considered. Parallel execution enables additional reduction in execution time by overlapping the execution of kernel with that of IP's as illustrated in Fig. 2.

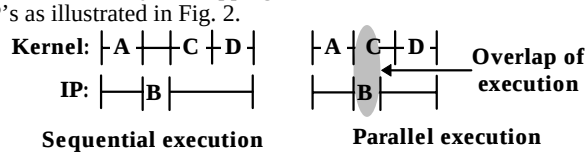

Fig. 2: Execution of four code segments, A-D, in kernel and IP

However, it generally requires input and output buffers to avoid memory contention between the kernel and IP's. Hence, the area penalty should be considered.

Now we address the characteristics of each type of interface we support. Specifically, we support four interface types shown in Fig. 3 to trade-off performance and cost.

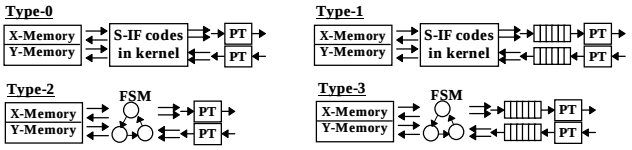

Fig. 3: Four types of interfaces

Type 0 (Software interface w/o buffer): This is the cheapest one but has the lowest performance. The in/out-controller runs in the kernel as software, and there are no in/out-buffers. Since the kernel runs the controller, no other codes can run in parallel with IP's. The maximum number of operands that can be passed to/from an IP in a cycle is limited to two, one from/to X-memory and the other from/to Y-memory. Hence IP's with more than two in/out-ports cannot be supported by this type.

We generate this software interface from the template shown in Fig. 4. It can support a pipelined IP with 4 clock-cycle data in/out-rate. It is programmed in μ-codes and works as a S-instruction. The template is composed of four parts. Initialization codes come first in line 1. In lines 2-5, the pipeline of the IP is filled up by the data from memory until the first result becomes available. Passing operands from memory to IP and passing results from IP to memory are performed in line 6-9. In lines 7 and 8, several operations are processed in a cycle, since the kernel has enough resources and the μ-codes can utilize them. In lines 10-13, the data under processing in pipeline are fully processed, and the results are passed to the memory.

As this interface is the cheapest one, we use this one as much as possible if we can meet the performance constraint. For IP's whose in/out-rate is more than four clock-cycles, we can modify the template by adding some NOPs. However, for IP's with in/out-rate less than four clock-cycles we have to slow down the clock signal connected to IP to use this type of interface. In this case, the performance degradation caused by the slow clocking is also considered in the generation of instructions.

Software Interface Template for type 0

```
1  cnt_in_only = # of data in input only part;
   cnt_in_out = # of data in middle part;
   cnt_out_only = # of data in output only part;
2  in-data_x = DM_x[]; in-data_y = DM_y[];
3  IP_in_x = in-data_x; IP_in_y = in-data_y;
4  cnt_in_only = cnt_in_only - 1;
5  if(cnt_in_only != 0) goto 2;
6  in-data_x = DM_x[]; in-data_y = DM_y[];
7  IP_in_x = in-data_x; IP_in_y = in-data_y;
   out-data_x = IP_out_x; out-data_y = IP_out_y;
8  DM_x[] = out-data_x; DM_y = out-data_y;
   cnt_in_out = cnt_in_out - 1;
9  if(cnt_in_out != 0) goto 6;
10 out-data_x = IP_out_x; out-data_y = IP_out_y;
11 DM_x[] = out-data_x; DM_y = out-data_y;
12 cnt_out_only = cnt_out_only - 1;
13 if(cnt_out_only != 0) goto 10;
```

Fig. 4: Interface template for type 0

Type 1 (Software interface w/ buffer): This type is similar to type 0 except that it has in/out-buffers. The buffer enables to handle an IP having more than two in(out)-ports by assigning a buffer to each port and to transfer high-rate in/out-data. In fact, high-rate transfer occurs between the buffer and the IP, while low-rate transfer occurs between the buffer and kernel to fill the data re-

quired to start the IP into the buffer and to move results from the buffer to memory after the IP finishes its job. In addition, parallel execution without memory contention is possible because the IP accesses the buffers instead of memory.

The template for this type is shown in Fig. 5. In lines 2-5, the in-buffer is filled up, and then the IP is activated to run in line 6. The IP runs between line 6 and 7 with getting operands from the in-buffer and putting results to the out-buffer. The code to run in kernel while the IP is working, henceforth *parallel code* for brevity, comes between line 6 and 7. In lines 7-10, the results in the out-buffer are moved to memory.

Software Interface Template for type 1

```
1 cnt_in = # of input data;
  cnt_out = # of results;
2 in-data_x = DM_x[]; in-data_y = DM_y[];
3 buff_in[][] = in-data_x; buff_in[][] = in-data_y;
4 cnt_in = cnt_in - 1;
5 if(cnt_in != 0) goto 2;
6 IP_start = 1; /* activates IP */
                  …
  /* Codes that will run in kernel while */
  /*          IP runs come here          */
                  …
7 out-data_x = buff_out[][]; out-data_y = buff_out[][];
8 DM_x[] = out-data_x; DM_y[] = out-data_y;
9 cnt_out = cnt_out - 1;
10 if(cnt_out != 0) goto 7;
```

Fig. 5: Interface template for type 1

Type 2 (Hardware interface w/o buffer): This is also similar to type 0 except that the in/out-controller is implemented in a FSM. The maximum number of operands that can be passed to (from) IP in a single clock cycle and the maximum number of in/out-ports that can be handled are the same as those of type 0. Though the in/out-controller is not executed in kernel, this type may not be adequate for parallel execution because of the memory contention.

Hardware Interface Template for type 2

```
1 cnt_in_only = # of data in input only part;
  cnt_in_out = # of data in middle part;
  cnt_out_only = # of data in output only part;
  /* bus connection setting */
  IP_in_x = data_x_1; IP_in_y = data_y_1;
  data_x_2 = IP_out_x; data_y_2 = IP_out_y;
2 repeat(cnt_in_only -- != 0)
    addr_x_1 = …; addr_y_1 = …; rw_x_1 = r; rw_y_1 = r;
3 repeat(cnt_in_out -- != 0)
    addr_x_1 = …; addr_y_1 = …; rw_x_1 = r; rw_y_1 = r;
    addr_x_2 = …; addr_y_2 = …; rw_x_2 = w; rw_y_2 = w;
4 repeat(cnt_out_only -- != 0)
    addr_x_2 = …; addr_y_2 = …; rw_x_2 = w; rw_y_2 = w;
```

Fig. 6: Interface template for type 2

The template for this type is shown in Fig. 6. It assumes a dual-ported data memory, i.e., one read port and one write port that can be accessed simultaneously, and operates in a DMA mode to pass operands and results. In line 1, in/out-ports of the IP are connected to those of data memories. One of data ports of X-memory, $data_{x\_1}$, is connected to one in-port of the IP, IP_{in_x}, and one out-port of IP, IP_{out_x}, is connected to the remaining data port of X-memory, $data_{x\_2}$. A similar connection is applied to Y-memory and the remaining ports of the IP. These connections are for the DMA operation, and actually performed by MUXs and tri-state buffers. In step 2, data are passed to the IP in the DMA mode to fill the pipeline up. Line 3 is to pass operands to the IP and to save results to memory. In line 4, results remaining in the pipeline are moved to memory. Notice that each line requires only one clock cycle by the help of hardware, hence for each clock the maximum rate of input data passing is two and that of output data passing is also two.

Type 3 (Hardware interface w/ buffer): This is the most expensive and powerful interface we support. By adding buffers to type 2, type 3 can handle IP's having more than two in(out)-ports at a high-rate of in(out)-data transfer. In addition, parallel execution is possible. The template for this type is shown in Fig. 7.

Hardware Interface Template for type 3

```
1 cnt_in = # of input data;
  cnt_out = # of results;
  /* bus connection setting */
  buff_in[][] = data_x; buff_in[][] = data_y;
  data_x = buff_out[][]; data_y = buff_out[][];
2 repeat(cnt_in -- != 0)
    addr_x = …; addr_y = …; rw_x = r; rw_y = r;
                  …
  /*  Codes that will run while IP runs  */
  /*          come here                  */
                  …
3 repeat(cnt_out -- != 0)
    addr_x = …; addr_y = …; rw_x = w; rw_y = w;
```

Fig. 7: Interface template for type 3

Different input and output data rates

Hitherto, interface templates are described with assuming that input data rate and output data rate are the same. However, in some IP's such as an interpolation filter, the rates can be different. Such an IP can be handled in type 2 by dividing the in/out-controller into an in-controller and an out-controller that run separately at their data rate. Type 1 and type 3 interfaces are also able to handle such an IP by running the in-buffer controller and the out-buffer controller separately. However, we have to change the software template for type 0 to meet the different in/out data rates, which is very hard and not always possible. Therefore, we support only type 1, type 2 and type 3 for such an IP.

Performance gain and implementation cost

Given a pipelined IP, if type 0 or type 2 interface is employed, passing data to/from the IP occurs in parallel with the operation of the IP. Hence the execution time can be expressed as $MAX(T_{IP}, T_{IF})$ where T_{IP} is the total execution time of the IP, and T_{IF} is that of interface. Given that T_{SW} is the execution time of software for the same task, the performance gain can be defined as $T_{SW} - MAX(T_{IP}, T_{IF})$.

In type 1 and type 3, buffer filling is first performed before an IP runs. After the IP finishes its operation, results in buffers are moved to memory. The execution time is $T_{IF\_IN} + MAX(T_{IP}, T_B) + T_{IF\_OUT}$, where T_{IF_IN} is the time to fill the in-buffer, T_B is the time to pass data between the buffer and the IP by a buffer controller, and T_{IF_OUT} is to move results from the out-buffer to memory. If a parallel code is available, the execution time is effectively reduced by $MIN(T_{IP}, T_C)$ where T_C is the execution time of the parallel code. Thus the overall performance gain is $T_{SW} - (T_{IF\_IN} + MAX(T_{IP}, T_B) + T_{IF\_OUT} - MIN(T_{IP}, T_C))$. This equation clearly shows that in terms of overall performance a slower IP with a parallel code can be better than a faster one with no parallel code.

The cost of implementation area can be expressed as $A_{IP} + A_{CNT} + A_B$, where A_{IP} is the area for the IP, A_{CNT} for the in/out-controller, and A_B for buffers (applied only to type 1 and type 3). For type 0 and type 1, A_{CNT} is the code-memory area needed for storing interface codes, while for type 2 and type 3, it is the area for a FSM.

4 Optimal Selection of IP's and Interfaces

The optimal selection of IP's and the interfaces is the same as that of S-instructions and their implementation methods, called as *optimal S-instruction generation problem*. In this section, we deal with this problem. Specifically, we select S-instructions from S-instruction candidates with considering their implementation methods.

Definition 1: A function call in C-source code can be an *S-instruction candidate* if the function can be implemented using an IP. For brevity, we will use the term *s-call*.

Definition 2: An *S-IP* is an IP that can perform only a single function, and an *M-IP* is an IP that can perform multi-functions.

Definition of Parallel Code

Definition 3: Given a CDFG (Control Data Flow Graph) representation of MOP list, where each node represents a MOP and a directed edge between two nodes represents the data/control dependency, a node that has no transitive closure edges with a s-call$_i$ is regarded as an *independent code to the s-call$_i$ (IC$_i$)*.

Definition 4: An *Independent code segment to a s-call$_i$ (ICS$_i$)* is a set of IC$_i$'s that are in the same execution branch, e.g., in the same conditional branch, and can be listed in a sequence.

Definition 5: *Parallel code to a s-call$_i$ (PC$_i$)* is the largest *ICS$_i$* in execution time that is in the same execution branch with s-call$_i$ and can be arranged right after s-call$_i$.

Informally, PC$_i$ is the longest code segment in execution time that can start to run in kernel right after the s-call$_i$, and can run concurrently with the IP corresponding to the s-call$_i$. Note that actually not all of the codes in PC$_i$ run in parallel with the IP. Only a part of PC$_i$ whose execution time is no more than that of IP actually does. Given that there are multiple execution paths after a s-call$_i$, we compute PC$_i$'s for all the execution paths and use the shortest one as PC$_i$ to guarantee the minimum performance gain for all execution paths.

4.1 ILP formulation

We first tackle a restricted version of problem, Problem 1, and then show how to extend the formulation for a more general problem, Problem 2.

Problem 1: This is the optimal S-instruction generation problem under the following restrictions.

- PC$_i$ cannot contain other s-calls.
- Multiple s-calls to the same function are always implemented in the same way.

As an illustration, in Fig. 8 showing four execution paths after fir(), the parallel code of the fir() is the code segment between the fir() and dct() of path P4. And the two fir()'s are always implemented in the same way.

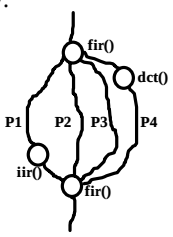

Fig. 8: Example for problem 1

- s-calls to different functions can be implemented in a single M-IP.

All s-calls in Fig. 8, i.e., two fir()'s, dct() and iir(), can be mapped into a single M-IP. By mapping several s-calls to a single M-IP, we can reduce area cost. However, such a scheme can be bad in performance point of view because an M-IP, in general, is not as good in performance as an S-IP optimized for a single function.

- To meet the performance constraint, some of s-calls may be mapped into S-IP's instead of M-IP's that can reduce the area.

In Fig. 8, for instance, given that the single M-IP for all s-calls cannot meet the performance constraint in P4, we may use S-IP for dct() to speed up.

Problem 1 can be formulated in Integer Linear Programming (ILP) as follows. Terms that will be used in the formulation are described below.

- SC$_i$: s-call$_i$.
- SCS: Set of all SC$_i$'s.
- IMP$_i$: All possible implementation methods for SC$_i$. Each implementation method contains interface method, IP, parallel code, area, power and performance gain.
- IMP$_i$ = $\cup_j$ IMP$_{ij}$, where IMP$_{ij}$ is the j'th possible implementation method for SC$_i$ using IP's.
- s_{ijk}: 1 if IMP$_{ij}$ uses k'th IP, otherwise 0.

The data base of IMP$_i$ is built up and the s_{ijk} is computed using the MOP list and IP library.

- T$_i$: Required performance gain for path P$_i$ to meet the performance constraint.

Decision variables are:

- x_{ij}: 1 if IMP$_{ij}$ is used to implement SC$_i$, otherwise 0.

We have to solve the problem under the following constraints.

1) For each SC$_i$, at most one implementation method can be selected.

$$\sum_j x_{ij} \le 1 \tag{1}$$

Note that if Eq. 1 is equal to 0, SC$_i$ will be implemented fully in software without IP's. Thus only the SC$_i$ whose Eq. 1 is equal to 1 is implemented using IP's and becomes an S-instruction.

2) For each path, required performance gain should be satisfied.

$$\forall P_k, \sum_{SC_i \text{ in } P_k} \left(\sum_j x_{ij} g_{ij} \right) \ge T_k \tag{2}$$

where g$_{ij}$ is the performance gain corresponding to IMP$_{ij}$.

The objective is to minimize the total area which is the sum of areas for IP's and interfaces. The area for IP's is

$$\sum_k z_k a_k$$

where $z_k = 1$ if $\sum_{i,j} s_{ijk} x_{ij} > 0$ and 0 otherwise (i.e., z_k is 1 if and only if the k'th IP is used at least once in the overall code), and a_k is the area of the k'th IP. Notice that even if IP$_k$ is used more than once for the implementation of several SC$_i$'s, it is actually included only once in a chip. Thus the area of IP$_k$ should be counted only one time.

The area for interface is

$$\sum_{i,j} x_{ij} c_{ij}$$

where c$_{ij}$ is the area for the interface method of IMP$_{ij}$.

Thus, the total area to minimize is

$$\sum_k z_k a_k + \sum_{i,j} x_{ij} c_{ij} \tag{3}$$

z_k is linearized as follows by using a technique for *fixed charge problem* [10].

$$\sum_{i,j} s_{ijk} x_{ij} \le M z_k \ (z_k = 0 \text{ or } 1,\ M \text{ is a value} \ge \sum_{i,j} x_{ij})$$

If the left-hand side of equation is larger than 0, z_k becomes 1. On the other hand, if the left size is 0, z_k can be 1 or 0. However, the objective function forces z_k to be 0.

Removing restrictions of Problem 1

In Problem 1, the case that an s-call can be implemented in software as a parallel code of another s-call is not considered due

to the following two restrictions: 1) Multiple s-calls to the same function are always implemented in the same way, and 2) PC_i cannot contain s-calls. In this part, we remove those restrictions. Followings are two examples motivating the removing.

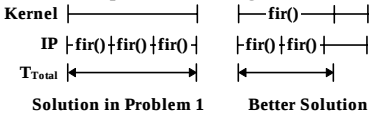

Fig. 9: Motivating example for eliminating restrictions

Fig. 9 shows two different executions of three fir()'s, given that a fir() is independent of others and the software implementation of all three fir()'s cannot meet the performance constraint. The best solution in Problem 1 is to map all fir()'s into an IP. In the solution, the kernel has nothing to do, hence the total execution time(T_{Total}) is the same as that of IP. However, the better solution is to run one fir() in the kernel and other twos in the IP. To find this solution, we have to remove the restrictions of Problem 1.

Fig. 10 shows two execution paths, P1 and P2(shaded one), have a common s-call, fir(). Assume that P1 has performance margin large enough to allow one of three fir()'s to be implemented in software. In addition, assume that for P2 to meet the performance constraint, fir() has to be the parallel code of dct(). In this case, the only solution is to implement the common fir() in software and other fir()'s of P1 using IP. However, such a solution is not allowed in Problem 1.

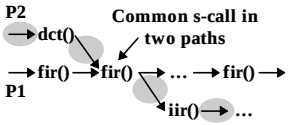

Fig. 10: Motivating example for eliminating restrictions

By removing the restrictions, the followings are possible.

- s-calls to the same function can be implemented in *different* ways.
- PC_i can contain software implementations of other s-calls.

We call this problem as **Problem 2**. In Problem 2, software implementation of fir() can be PC of dct() in Fig. 10. In addition, the sequence of software implementations of fir() and iir() can be the PC of dct(). However, selecting fir() as a parallel code of dct() prevents fir() from being implemented in the IP. Similarly, selecting fir() and iir() as parallel codes of dct() hinders IP-based implementations of fir() and iir(). This can be formally stated as follows:

Let IMP-A and IMP-B are any two of $\cup_i IMP_i$, i.e., all the IMPs for all SC_i's.

- IMP-A and IMP-B are said to have *SC conflict* if they are for the same SC_i.
- IMP-A and IMP-B are said to have *SC-PC conflict* if IMP-A is for SC_i and IMP-B uses software implementation of the SC_i as a parallel code, and vice versa.

Selection rule: We cannot select both of IMP-A and IMP-B as the solution if they have SC conflict and/or SC-PC conflict, i.e., only one of them can be selected.

Now we explain how to extend the ILP formulation for Problem 1 for Problem 2. First, new possible IMPs having software implementations of s-calls as parallel code are added to the IMP data base. Second, to keep the selection rule, we add the following equation for any two IMP's, say IMP_{ij} and IMP_{kl}, having a SC-PC conflict.

$$x_{ij} + x_{kl} \le 1 \cdot$$

Note that for two IMPs having a SC conflict, we do not have to add any equations to keep the selection rule because Eq. 1 that is already in the formulation prevents such a solution.

Handling Hierarchy

Heretofore, we have not considered the hierarchy in the application. Fig. 11 shows an example code for image processing that has hierarchy.

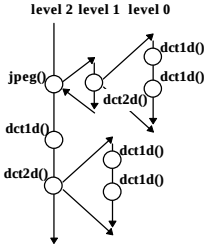

Fig. 11: An application having hierarchy

Main routine calls jpeg(), jpeg() calls dct2d(), and so on. Given that IP's for dct1d(), dct2d() and jpeg() are available, we use the technique *IMP flatten* to handle the hierarchy in the ILP formulation. In computing IMPs of upper-level s-call, all possible IMPs of lower-level s-calls are considered. For example, IMPs of dct1d() at level 0 are considered in computing those of dct2d() at level 1, and those are considered for computing the IMPs of jpeg(). In this way, all the IMPs of lower levels s-calls are included in those of top level s-calls to which ILP formulation is applied.

5 Experimental Results

The proposed method has been implemented in C language on a SPARC-20 workstation with 128 Mbyte main memory. We tested the proposed method on two real DSP applications: GSM(TDMA) system and JPEG system. For a given program, we first transform it into a MOP list, and then mapped it into the P-instructions. Then we employed the proposed method to reduce the execution cycle.

5.1 GSM(TDMA) System

For the encoder part having 18 s-calls, a set of 23 IP's including several filters, correlators and quantizers were prepared. For some s-calls, there were two or three different IP's available. IMP's were generated for each s-call, and the total number of IMP's was 42. Among them, one was generated with considering the hierarchy between s-calls, and three of them exploited the parallel code. And among the three, one used the software implementation of other s-call as its parallel code.

Table 1 shows the results - selected s-calls and their implementation methods to meet the required performance gain (**RG**). The actual gain (**G**) and the relative area cost (**A**) of the implementations are also listed in the table. Each implementation method shows IP, interface type, gain and area cost for the s-call to be implemented using the IP. For example, "SC_{13}: IP12, IF0, 115037, 3" means that s-call SC_{13} has to be implemented using IP12 and type 0 interface, and the gain and area cost are 115037 and 3, respectively. The column **S** shows the number of S-instructions. This is always no more than that of the selected s-calls (shown in column **O**) because s-calls to be implemented in the same way, i.e., the same IP and the same interface method, can be merged and implemented in a single S-instruction. We can see the followings from the result. First, in many cases type-0 interface, the cheapest one, was used to reduce area cost. Second, SCs that can be implemented using the same IP are selected as many as possible to reduce the area cost by sharing the IP. This also reduces the number of S-instructions. Third, as the required gain increases, more powerful IP's and interface types are employed. For example, IP13 becomes to be used as RG becomes 238702, and when RG becomes 381923 its interface changes from type-1

to type-3 to obtain more gain by including parallel execution gain. Be aware that such a solution was not possible in the previous approach because it neither supported the parallel execution nor considered the interface method with IP's.

Table 2 shows the results for the decoder part. We supported 10 IP's for 11 s-calls, and the total number of IMP's was 27. In this case, software interface was employed for all IMP's of the solution except the one for SC_{10} when RG is 211286. We can see that interface method for SC_{10} changed from type 0 to type 2 to get more gain when RG is 211286.

5.2 JPEG System

The JPEG encoder has 2D-DCT as its main function. 2D-DCT consists of two 1D-DCTs, and 1D-DCT calls FFT. In FFT, a number of complex number multiplications are performed. We supported five IP's: one for 2D-DCT, one for 1D-DCT, one for FFT, one for complex multiplication, and one for zig_zag function. Seven IMP's were generated for 2D-DCT with considering the hierarchy and two IMP's were generated for zig_zag.

Table 3 shows the results. It clearly shows the change of IP and interface method as RG increases. When RG is 12157384, only the complex multiplication was implemented using an IP. However, the IP for 2D-DCT with type-3 interface was used in the last row to meet the required gain. Similar results were obtained for the decoder part.

6 Conclusions

In this paper, we proposed a new approach to select the optimal set of IP's and interfaces to make the application program meet the performance constraints in processor-core based designs. We selected IP's with considering the interfaces and supported concurrent execution of codes in the kernel and IP's. We first presented an ILP formulation for a restricted problem, and then described how to extend it for a generalized problem and how to handle hierarchy. The experimental results indicate that the proposed approach is so effective that we can make the application program meet the performance constraints using IP's.

References

[1] M. Keating, "A Financial Model for Design Reuse," *http://www.synopsys.com/roi/*, Sept. 1998.

[2] R. Passerone, J. A. Rowson and A. Sangiovanni-Vincentelli, "Automatic Synthesis of Interfaces between Incompatible Protocols," *35th Design Automation Conference*, pp. 8-13, 1998.

[3] J. Smith and G. De Micheli, "Automated Composition of Hardware Components," *35th Design Automation Conference*, pp. 14-19, 1998.

[4] K. S. Chung, R. K. Gupta and C. L. Liu, "An Algorithm for Synthesis of System-Level Interface Circuits," *International Conference on Computer-Aided Design*, pp. 442-447, 1996.

[5] S. Narayan and D. Gajski, "Interfacing Incompatible Protocols using Interface Process Generation," *32nd Design Automation Conference*, pp. 468-473, 1995.

[6] R. B. Ortega, L. Lavagno and G. Borriello, "Models and Methods for HW/SW Intellectual Property Interfacing," *NATO ASI Proceedings on System Synthesis*, 1998.

[7] P. Chou, R. B. Ortega, G. Borriello, "Synthesis of the Hardware/Software Interface in Microcontroller-Based Systems," *International Conference on Computer-Aided Design*, pp. 488-495, 1992.

[8] A. Alomary, T. Nakata, Y. Honma, M. Imai and N. Hikichi, "An ASIP Instruction Set Optimization Algorithm with Functional Module Sharing Constraint," *International Conference on Computer-Aided Design*, pp. 526-532, 1993.

[9] H. Choi, I.-C. Park, S. H. Hwang and C.-M. Kyung, "Synthesis of Application Specific Instructions for Embedded DSP Software," *International Conference on Computer-Aided Design*, pp. 665-671, 1998.

[10] H. A. Taha, Operations Research, Prentice Hall, 1997, Chapter 9, pp. 367-373

RG	Implementation Method	G	A	S	O
47740	SC_{13}: IP12,IF0,115037,3	115037	3	1	1
95480	SC_{13}: IP12,IF0,115037,3	115037	3	1	1
143221	SC_7:IP12,IF0,12531,3, SC_9:IP12,IF0,13489, 3, SC_{11}:IP12,IF0,12531,3, SC_{13}:IP12,IF0, 115037,3	153588	3	1	4
190961	SC_2:IP3,IF1,41670,14, SC_7:IP12,IF0, 12531,2, SC_9:IP12,IF0,13489,3, SC_{11}: IP12, IF0,12531,3, SC_{13}:IP12,IF0,115037,3	195258	17	2	5
238702	SC_7:IP12,IF0,12531,3, SC_9:IP12,IF0, 13489,3, SC_{11}:IP12,IF0,12531,3,SC_{13}:IP12,IF0,115037, 3, SC_{14}:IP13,IF1,162612,15	316200	18	2	5
286442	SC_7: IP12,IF0,12531,3, SC_9:IP12,IF0, 13489,3, SC_{11}:IP12,IF0,12531,3, SC_{13}:IP12, IF0,115037,3, SC_{14}:IP13,IF1,162612,15	316200	18	2	5
334182	SC_7:IP12,IF0,12531,3, SC_9:IP12,IF0,13489, 3, SC_{11}:IP12,IF0,12531,3, SC_{13}:IP12,IF0, 115037,3, SC_{15}:IP16,IF2,8200,3, SC_{14}: IP13, IF1,162612,15, SC_{16}:IP17,IF0,11576,3	335976	24	4	7
381923	SC_2:IP3,IF1,41670,14, SC_6:IP10,IF0,978,2, SC_7:IP12,IF0,12531,3, SC_9:IP12,IF0,13489, 3, SC_{10}:IP10,IF0,978,2, SC_{11}:IP12,IF0, 12531,3, SC_{12}:IP10,IF0,978,2, SC_{13}:IP12,IF0,115037,3, SC_{14}:IP13,IF3,164532,15.5,SC_{15}:IP16,IF2,8200,3.5, SC_{16}:IP17,IF0,11576,3	382500	41	6	11

Table 1: Experimental results for GSM encoder

RG	Implementation Method	G	A	S	O
22240	SC_4:IP5,IF0,14787,4, SC_6:IP5,IF0,13737,4	28524	4	1	2
44481	SC_8:IP5,IF0,126087,4	126087	4	1	1
111203	SC_8:IP5,IF0,126087,4	126087	4	1	1
133444	SC_6:IP5,IF0,13737,4, SC_8:IP5,IF0,126087,4	139824	4	1	2
155684	SC_2:IP5,IF0,13737,4, SC_4:IP5,IF0,14787,4, SC_6:IP5,IF0,13737,4, SC_8:IP5,IF0,126087,4	168348	4	1	4
177925	SC_2:IP5,IF0,13737,4, SC_4:IP5,IF0,14787,4, SC_6:IP5,IF0,13737,4, SC_8:IP5,IF0,126087,4, SC_{10}:IP6,IF0,14544,3	182892	7	2	5
200166	SC_2:IP5,IF0,13737,4, SC_6:IP5,IF0,13737,4, SC_9:IP8,IF0,8568,5, SC_{11}:IP10,IF0,9028,3, SC_4:IP5,IF0,14787,4, SC_8:IP5,IF0,126087,4, SC_{10}:IP6,IF0,14544,3	200488	15	4	7
211286	SC_1:IP2,IF0,978,2, SC_2:IP4,IF0,14235,32, SC_3:IP2,IF0,978,2, SC_4:IP4,IF0,15327,32, SC_5:IP2,IF0,978,2, SC_6:IP4,IF0,14235,32, SC_7:IP2,IF0,978,2, SC_8:IP4,IF0,131079,32, SC_9:IP8,IF0,8568,5, SC_{11}:IP10,IF0,9028,3, SC_{10}:IP6,IF2,15048,3	211432	45	5	11

Table 2: Experimental results for GSM decoder

RG	Implementation Method	G	A	S	O
12157384	SC_1:IP4,IF0,15040512,4	15040512	4	1	1
20262307	SC_1:IP2,IF1,37081088,11	37081088	11	1	1
37195000	SC_1:IP2,IF1,37081088,11, SC_2:IP5,IF2,113984,5.5	37195072	16.5	2	2
37282645	SC_1:IP1,IF1,37717440,27	37717440	27	1	1
37843700	SC_1:IP1,IF3,37729728,27.5, SC_2:IP5,IF2,113984,5.5	37843712	33	2	2

Table 3: Experimental results for JPEG encoder (IP1: 2D-DCT, IP2: 1D_DCT, IP3: FFT, IP4: C-MUL, IP5: ZIG_ZAG)