Synthesis of Application Specific Instructions for Embedded DSP Software

Hoon Choi, *Student Member*, *IEEE*, Jong-Sun Kim, Chi-Won Yoon, *Student Member*, *IEEE*, In-Cheol Park, *Member*, *IEEE*, Seung Ho Hwang, *Member*, *IEEE*, and Chong-Min Kyung, *Member*, *IEEE*

Abstract—Application specific instructions play an important role in reducing the required code size and increasing performance in embedded DSP systems. This paper describes a new approach to generate application specific instructions for DSP applications. The proposed approach is based on a modified subset-sum problem and supports multicycle complex instructions, as well as single-cycle instructions, while the previous state-of-the-art approaches generate only the single-cycle instructions or just select instructions from the fixed super-set of possible instructions. In addition, the proposed approach can also be applied to the case that instructions are predefined. Experimental results on real applications show that various given constraints can be met by the generated set of application specific instructions without attaching special hardware accelerators.

Index Terms—Application specific instruction-set processor, instruction synthesis, hardware/software co-design, digital signal processing, embedded system.

1 Introduction

Due to the advance of VLSI technology, a lot of ASICs (Application Specific Integrated Circuits) are being used in numerous systems. Compared to general purpose processors, an ASIC can satisfy various constraints, such as performance, area, and power, by finding the optimal architecture for an application. However, as the complexity of applications increases, more flexibility is required to accommodate design errors and specification changes which may happen at later design stages. Since an ASIC is specially designed for one behavior, it is difficult to accommodate any changes at a later design stage. In contrast, programmable processors can be easily adapted to different applications by changing only the programs. It is the reason that ASIPs (Application Specific Instruction set Processors) are widely accepted in numerous systems.

Generally, an ASIP has a programmable architecture tuned to an application area. Choosing an optimal instruction set for the specific application under the constraints, such as chip area and power consumption, is crucial in enhancing the performance of the ASIP. This leads to several works to develop the tools for analyzing the given application and determine the optimal instruction set which maximizes the performance.

There have been many works related to the ASIP synthesis [1], [2], [3], [4], [5], [6], [7], [8], [9], [10], [11], [12], which can be categorized into four classes. First, the design of area-efficient hardware blocks of an ASIP was handled in [1], [2], [3], [4]: An evolution programming approach for area-efficient design was presented in [1], [2], and the grouping problem that n control-data flow graphs are bundled into m $(< n)$ groups was considered to synthesize area-efficient multifunction accelerators [3], [4]. These are focused on the design of hardware blocks, but do not consider the relation between the synthesized hardware and the corresponding instruction. Second, the matching of a code sequence into a predefined instruction set was handled in [5], [6], [7], [8], [9]. A tree-based approach employing dynamic programming techniques was proposed in [5]. In [6], [7], the instruction selection and the register allocation were merged into a single tree covering phase and an optimal scheduling algorithm was proposed to minimize the number of memory spills. Integer linear programming (ILP) based approaches considering instruction-level parallelism were proposed in [8], [9]. However, these approaches did not take into account the generation of new instructions optimal for a given application. Third, how to select instructions and implement them was handled in [10], [11]. Instructions are selected from a fixed super-set of all the possible instructions based on the intermediate language of GNU compiler. Hence, it cannot generate new instructions for a specific application, but can select them from the predefined super-set. Last, the instruction generation problem was treated in [12] by formulating the problem as a modified scheduling problem of micro-operations (MOPs). In the approach, each MOP is represented as a node to be scheduled and a simulated annealing scheme was applied for solving the scheduling problem. This work is important in that it tried to generate application specific complex instructions. Complex instructions are more powerful than simple instructions because complex instructions can use the full power of execution engines supported in the processor, resulting in higher performance by exploiting more parallelism in MOPs. However, in general purpose DSP processors, complex instructions have not been widely used because their

• *The authors are with the Department of Electrical Engineering, Korea Advanced Institute of Science and Technology, Taejon, Korea.*
E-mail: {hchoi, jskim, grumpy}@snoopy.kaist.ac.kr.
{icpark, shwang, kyung}@ee.kaist.ac.kr.

For information on obtaining reprints of this article, please send e-mail to: tc@computer.org, and reference IEEECS Log Number 109702.

complex instructions are too general to be used efficiently and powerfully for a specific application. In contrast, complex instructions of an ASIP can be efficient if the complex instructions are tuned to the application which the ASIP aims at.

Although application specific complex instructions are usually executed in multiple cycles, only single-cycle complex instructions are considered in [12]. The multicycle instruction, however, has two noticeable advantages over the single-cycle instruction. First, it can reduce the program memory size, which might be crucial in embedded systems. Second, it can reduce the number of required code fetches, thus speeding up the execution, especially if the code is stored in external memory that is much slower than the ASIP. In addition, the fewer memory accesses lead to a reduction in power consumption [13] since fetching codes from external memory consumes large power.

In this paper, we propose a new approach based on the *subset-sum problem* [14] to generate an optimal instruction set including multicycle complex instructions as well as single-cycle complex instructions. The proposed method can also be applied to match a code sequence to the predefined instructions.

The rest of this paper is organized as follows. In Section 2, we give some background on the target architecture of the ASIP to be synthesized and an overview of our ASIP synthesis system. In Section 3, we describe the new approach to generate complex instructions. We also deal with how to apply the proposed method to large-sized problems. Section 4 shows how the proposed approach can be used to match a code sequence to the predefined instructions. Experimental results are shown in Section 5.

2 Background

In this section, we briefly address the target micro-architecture of the ASIP to be synthesized and the overview of our ASIP synthesis system called *Partita*.

The target architecture is a pipelined DSP processor controlled by the μ-program. Like most DSP processors, it has a separate address generation unit (AGU) and can access two data-memories (XDM and YDM) to fetch two memory-operands simultaneously. The μ-control word in the μ-ROM is composed of four fields: two fields for two simultaneous data-memory accesses, one field for arithmetic, multiply, and shift operations, and one field for register data transfer operations. Hence, an arithmetic operation (or a multiply operation or a shift operation) and a register move operation can be executed in parallel. Each operation in a field of the μ-control word is called a MOP (micro-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. P-class instructions are always supported in all the generated ASIPs and executed in the execution kernel. Actually, we support 38 P-instructions: 23 for computing operations (e.g., ALU and MPY), 11 for control operations (e.g., branch), and 4 for special operations. Second, C-class is composed of the instructions that are more complex than P-class instructions. Though it is also executed in the same execution kernel, it is more powerful than the P-class due to two reasons. First, a C-instruction can control all the units in the kernel at the same time, while a P-instruction can use a limited number of units because of the instruction encoding constraint. For example, a 16-bit instruction format of the P-instruction is not sufficient for specifying the functions of all the units in the kernel. In contrast, the C-instruction is transformed into a sequence of wide μ-control words that can control all the units simultaneously. In other words, the C-instruction can fully use the parallelism supported in the kernel. Second, C-instructions help reduce the code-memory size and the number of code-fetches because a C-instruction is comparable to several P-instructions. It is very important in the embedded systems (the main target of the ASIP) that usually have a small internal code-memory. Therefore, generating an appropriate C-instruction set is a very important task for the synthesis of the ASIP. Last, S-class is composed of instructions that are supported by special hardware units called S-HWs.

Now, we briefly address our ASIP synthesis system, Partita, shown in Fig. 1. The inputs to Partita are the application program written in C, typical input data for the application, and the constraints, such as maximum execution time allowed. The input application is transformed into a MOP list while preserving almost all the concurrency in the source program. We sample-run the MOP list with the given typical input data to obtain the profile of the running frequency of each MOP. However, in the case that the timing requirement is hard (real-time system) and there is a loop whose loop-count depends on the input data, we cannot use sample-run with typical input data. Instead, we use a static timing analysis technique [16] based on abstract simulation that guarantees the maximum execution time. We first match the MOP list to the P-instructions to estimate the execution time when only the P-instructions are used. If it meets the performance constraint, we actually match the MOP list to the P-instructions. However, if not satisfactory, we start to generate C-instructions from the MOP list. If the generated C-instructions make the code sequence meet the timing constraint, we map the rest of the MOP list, not covered by the generated C-instructions, to the P-instructions. However, if the generated C-instructions are still not sufficient, we try to generate S-instructions. If the generated S-instructions fail to meet the timing constraint, we conclude that synthesizing an ASIP that meets the timing constraint is impossible. Otherwise, the rest of the MOP list is mapped to the P-instructions and C-instructions. This instruction generation phase is performed for all the paths in the application program, i.e., we check the timing constraints for all the paths.

After generating instructions that meet the given timing constraint, we generate hardware modules required to execute the instructions. If S-instructions are needed, the corresponding S-HWs are synthesized. Other necessary hardware modules, such as the decoding unit and the fetch unit, are also synthesized with consideration of the newly generated C-instructions and S-instructions. All newly generated instructions are encoded in the instruction space,

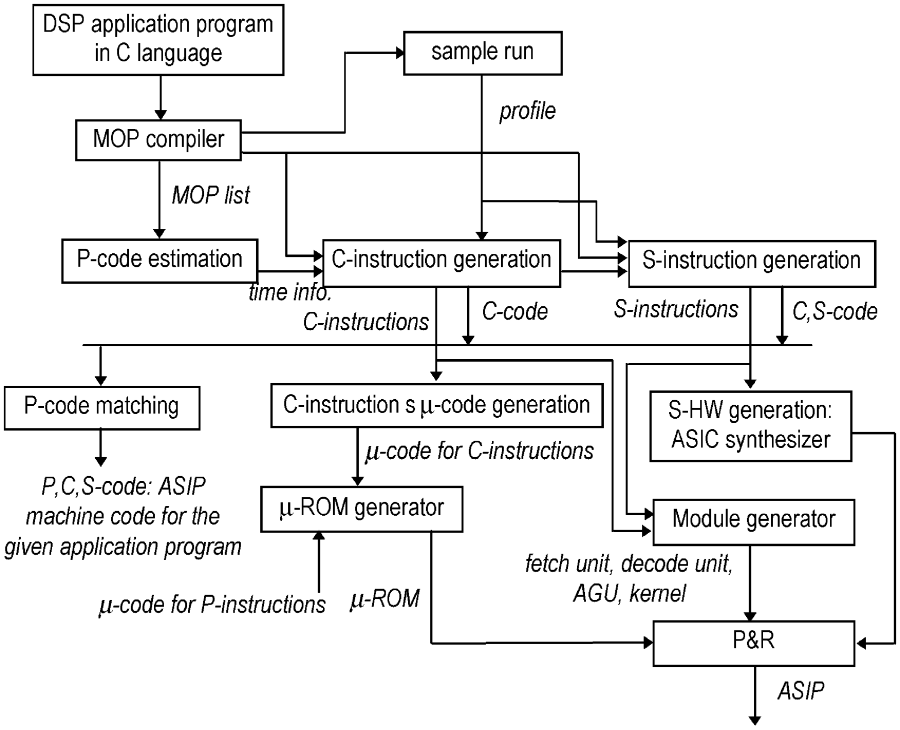

Fig. 1. Partita system overview.

and the μ-ROM is optimized including the μ-codes for the C-instructions generated. In this paper, we mainly focus on the *C-instruction generation* block shown in the middle of Fig. 1.

3 Generation of C-Class Instructions

We first describe the generation of single-cycle C-instructions (SCC-instructions) and then extend it for multicycle C-instructions (MCC-instructions). The C-instruction generation problem is solved by transforming it into a subset-sum problem. Notice that this kind of separation between SCC-instructions and MCC-instructions is just for the convenience of explanation. We actually generate both of them simultaneously in a single framework.

3.1 Generation of Single-Cycle C-instruction

This subsection explains how SCC-instructions are generated from a MOP list. The difference between the estimated execution time using only the P-instructions (T_p) and the given constraint on the maximum execution time (T_c) is represented as T_d (i.e., $T_d = T_p - T_c$). For an SCC-instruction, SC_i, generated by merging several MOPs, there is generally a speed gain g_i. The problem of generating SCC-instructions can be formally stated as follows:

Problem 1. *Given a MOP list, generate an SCC-instruction set such that 1) the total gain should be no less than T_d, 2) the generated instructions should be used as many times as possible in the application, and 3) the number of SCC-instructions generated should be as small as possible.*

The rationale behind the requirements is to generate a small set of SCC-instructions which can be frequently used in the application. This prevents a code sequence whose pattern is rarely used in the application from being generated as an SCC-instruction. Such a rarely used code sequence can become an SCC-instruction if and only if it is indispensable for meeting the timing constraint. The number of C-instructions should be as small as possible since each C-instruction requires additional space in the μ-ROM and makes the instruction decoder complex.

We can solve this problem optimally by formulating it as the *subset-sum problem* [14].

Subset-sum problem. *Given S and t, where S is a set $\{x_1, x_2, \ldots, x_n\}$ of positive integers and t is a positive integer, find a subset of S whose sum is as large as possible but not larger than t.*

For the sake of formulation, we need to define some terms. A MOP in the given MOP list, denoted as m_i, may or may not have dependencies on other m_js. A compatible MOP group, C_k, is the set of m_is that can be performed in the same cycle and specified in a single μ-control word, i.e., m_is that have no dependencies one another and can be packed together in a single μ-control word. This means that a compatible MOP group is a candidate SCC-instruction. As an example, given a MOP list $\{m_1, m_2, m_3, m_4, m_5, m_6, m_7, m_8\}$, assume that m_1, m_2, and m_3 can be performed in a single-cycle, and the same is true for m_6 and m_7. The possible five C_ks are shown in Fig. 2. Note that we take account of all the possible C_ks for m_1, m_2, and m_3 (i.e., not only C_4 but also C_1-C_3).

Since an m_i may be included in more than one C_k, there is a constraint in selecting C_ks such that the selected C_ks have no common m_is. For example, we have to select only one among C_1-C_4. The *compatible MOP group selection*

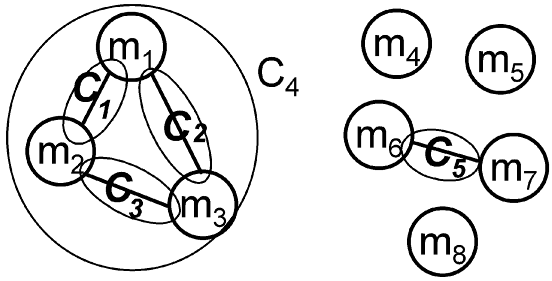

Fig. 2. Possible C_ks.

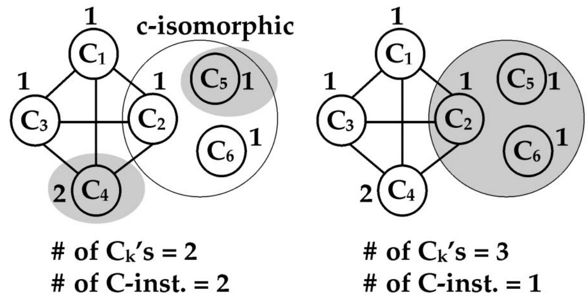

Fig. 4. Example for the third requirement.

constraint (CGSC) can be represented by a conflict graph where each node represents a C_k and an edge between two nodes represents that they have at least one common MOP, hence, only one of them can be selected as a solution. The conflict graph for Fig. 2 is shown in Fig. 3.

Each C_k is associated with a gain g_k that is the speed gain achievable by the introduction of the corresponding SCC-instruction. Now, Problem 1 can be restated as follows:

Problem 2. *Given a MOP list, select C_ks satisfying the CGSC, such that the following three requirements are met: 1) the sum of g_ks of selected C_ks should be no less than T_d, 2) the generated instructions should be used as many times as possible in the application, and 3) the number of different SCC-instructions corresponding to the selected C_ks should be as small as possible.*

The third requirement cannot be replaced by "*the number of selected C_ks should be as small as possible.*" The number of selected C_ks is not always equal to that of SCC-instructions to be generated because a number of C_ks can be implemented by one C-instruction. For example, given a MOP list $\{m_1, m_2, m_3, m_4, m_5, m_6, m_7, m_8\}$, assume that T_d is 3 and the possible C_ks are $C_1 = \{m_1, m_2\}$, $C_2 = \{m_1, m_3\}$, $C_3 = \{m_2, m_3\}$, $C_4 = \{m_1, m_2, m_3\}$, $C_5 = \{m_4, m_5\}$, and $C_6 = \{m_7, m_8\}$. The associated g_ks are computed as 1, 1, 1, 2, 1, and 1, respectively. Let us assume that C_2, C_5, and C_6 can be supported by one C-instruction. (Henceforth, such C_ks are called *c-isomorphic*. The exact meaning of and how to decide the c-isomorphism will be described later.) If we try to minimize *the number of selected* C_ks, the solution is to select C_4 and C_5. In this case, the number of SCC-instructions is equal to that of the selected C_ks. However, if we select C_2, C_5, and C_6, the number of SCC-instructions is *one* (not three because they are mapped into the *same* C-instruction), while the number of the selected C_ks is three. This is illustrated in Fig. 4.

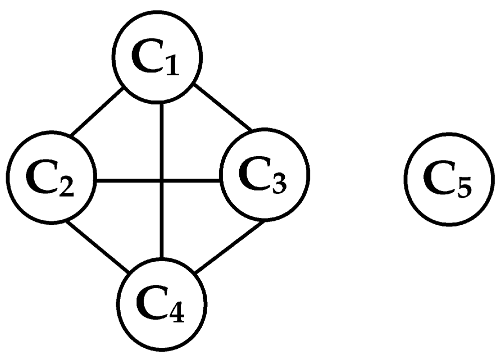

Fig. 3. Conflict graph for Fig. 2.

Now, consider the second requirement. We may try to meet the second requirement (i.e., try to find frequently used C-instructions) by representing all the C_ks that are c-isomorphic as a new single $\tilde{C}_k$ whose gain is set to the sum of all the gains of the C_ks and then selecting $\tilde{C}_k$s based on the gain. It is based on the assumption that if the single $\tilde{C}_k$ is selected, all the c-isomorphic C_ks are selected and implemented by one C-instruction. Though the scheme may be a good method to meet the second requirement, it does not allow the case in which only part of the c-isomorphic C_ks are selected and the others are not selected.

As an illustration, consider the above example again. Let us assume that c-isomorphic C_2, C_5, and C_6 are represented as $\tilde{C}_2$. Then, the gain of C_1, $\tilde{C}_2$, C_3, and C_4 are 1, 3, 1, and 2, respectively (notice that $\tilde{C}_2$'s gain is the sum of the gains of C_2, C_5, and C_6). If we use the above scheme, $\tilde{C}_2$ having the largest gain is selected, and the c-isomorphic C_ks (i.e., C_2, C_5, and C_6) are automatically selected. Therefore, in that scheme, the total gain is limited to 3 (note that only one among C_1-C_4 can be selected). If the given T_d were 4, we could not find a solution. However, the solution for $T_d = 4$ can be obtained by selecting C_4, C_5, and C_6 (two C-instructions with gain 4). As shown in Fig. 5, only two among the three c-isomorphic C_ks are selected in the solution. This example claims that we should take account of the possibility that not all the c-isomorphic C_ks are mapped into a C-instruction; some of them may be 1) included in more larger C-instructions, 2) divided for different C-instructions, 3) mapped into P-instructions later, etc.

We now present the way to solve Problem 2 using a modified subset-sum problem. Problem 2 can be reformulated as follows:

Problem 3. *Given S and T_d, where S is a set of gain g_k corresponding to C_k, $\{g_1, g_2, \ldots, g_n\}$, find a subset of S whose sum is no less than T_d with satisfying following two requirements: 1) the generated instructions should be used as many times as possible in the application, and 2) the number of SCC-instructions should be as small as possible.*

We can see that Problem 3 is an extension of the subset-sum problem. Therefore, Problem 3 can be solved by an extended subset-sum problem solver [14]. Fig. 6 shows the

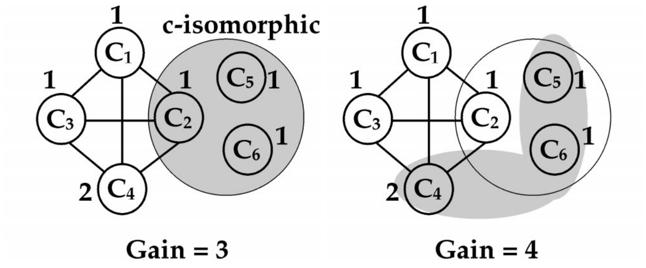

Fig. 5. Example for the second requirement.

pseudocode of the proposed SCC-instruction generation algorithm using the subset-sum problem.

For a given MOP list, a dependency graph (G_d) among MOPs is first built. The data dependency, output dependency, and data anti-dependency among MOPs are checked and represented in the G_d. Each node in the graph represents a MOP and each edge represents the dependency between two nodes. Based on G_d, all the possible C_ks are generated and their gains are computed. Be aware that we do not consider the c-isomorphic C_ks in this gain computation phase. They are considered in the subset-sum problem solver. Then, a conflict graph (G_c) to represent the *CGSC* is built. A subset-sum problem solver extended for the generation of SCC-instructions is employed to find an optimal solution for the given C_ks, S, and G_c. Then, the MOP list is modified based on the SCC-instructions found by the subset-sum problem solver.

The details of the extended subset-sum problem solver is as follows. L_i is a list of possible solution candidates. Each element of L_i (i.e., each solution candidate) is in the form of (*TG*, *XG*, *CS*, *ISS*) which represents the total gain, extra gain, the list of selected C_ks, and the size of corresponding C-instruction set (with considering c-isomorphic C_ks), respectively. Operation $L_{i-1} \oplus i$ denotes a new list derived from L_{i-1}; for each element in the L_{i-1}, the gain of C_i (i.e., g_i) is added to the *TG*, 1 is added to *XG* if *CS* already contains some C_js that are c-isomorphic with C_i, C_i is added to *CS*, and *ISS* is updated to the number of different C-instructions in *CS*. Before deriving the new list by $\oplus$ operation, the *CGSC* is checked between the *CS* of every element in L_{i-1} and the C_i to be added. In Step 5, elements that have almost no possibility to become a solution are eliminated from the list. The details of this pruning is addressed in Section 3.3. In Steps 6-8, the best element of L_i that is most frequently used in the code with satisfying the minimum C-instruction-set size is searched among the elements that meet T_d. Notice that *XG*, extra gain, gives a favor to the C-instructions which are used frequently in the code. The (*TG′*, *XG′*, *CS′*, *ISS′*) keeps the best solution found.

As an illustration of the proposed algorithm, the step-by-step change of L_i is shown in Fig. 7 for the example in Fig. 4; among six C_ks with gains 1, 1, 1, 2, 1, and 1, respectively, C_2, C_5, and C_6 are c-isomorphic. We can see in L_6 that $(3, 2, [C_2, C_5, C_6], 1)$ is found as the solution for $T_d = 3$ and $(4, 1, [C_4, C_5, C_6], 2)$ for $T_d = 4$, which are the optimal solutions as explained before. Therefore, one SCC-instruction is generated for $T_d = 3$ and two SCC-instructions (one for C_4 and the other for C_5 and C_6) for $T_d = 4$.

SCC generation

```
1  Gd ← dependency graph(MOP list)
2  C ← generate all the possible Ck's(Gd)
3  S ← compute gain(C)
4  Gc ← conflict graph(C)
5  SCC's ← Subset-sum solver for SCC(C, S, Gc)
6  C-code ← change code(MOP list, SCC's)
```

Subset-sum solver for SCC

```
1  n ← |S|
2  L0 ← [(0, 0, ∅, 0)]   /* (TG, XG, CS, ISS) */
3  for i ← 1 to n
4     Li ← Li-1 ∪ (Li-1 ⊕ i)
5     pruning(Li)
6     for every element in Li whose TG is larger than Td
7       if XG > XG' or (XG == XG' and ISS < ISS') then
8          TG' ← TG, XG' ← XG, CS' ← CS, ISS' ← ISS
9  return (TG', XG', CS', ISS')
```

Fig. 6. SCC-instruction generation algorithm.

3.2 Generation of Multicycle C-Instruction

We extend the proposed method for the generation of MCC-instructions. The problem of generating MCC-instruction is almost the same as that of SCC-instruction generation.

Problem 4. *Given an MOP list, generate an MCC-instruction-set satisfying the following three requirements: 1) the total gain should be no less than T_d, 2) the generated MCC-instructions should be used as many times as possible in the application, and 3) the number of generated MCC-instructions should be as small as possible.*

A major difference between the SCC-instruction generation and the MCC-instruction generation is in the generation of C_ks. Only the m_is that can be executed together in a cycle are considered in the SCC-instruction generation. However, in the MCC-instruction generation, a C_k can include m_is that can be executed in sequel as well as in parallel. This enlarges the solution space and, as a result, more powerful instructions can be found. However, the enlarged solution space causes the explosion of possible C_ks for a large-sized code. To overcome the situation, we use the following three techniques.

1. We limit the maximum length of C_k to a certain value based on the following rationale. A long C_k (i.e., C_k includes a large number of MOPs) has little chance to be selected as an MCC-instruction because the extra gain (*XG*) and C-instruction set size (*ISS*) favor MCC-instructions applicable multiple times in the code. Thus, we can prune such a long C_k from the solution space at the expense of little degradation of solution quality.
2. Not all the sequences of MOPs become the C_ks. The MOPs in a C_k should have some relations, such as data-dependency, among them. The rationale behind this is that unrelated MOPs have no reason to be packed into an instruction.

L_0 = [(0, 0, ∅, 0)]
L_1 = [(0, 0, ∅, 0), (1, 0, C_1, 1)]
L_2 = [(0, 0, ∅, 0), (1, 0, C_1, 1), (1, 0, C_2, 1)]
L_3 = [(0, 0, ∅, 0), (1, 0, C_1, 1), (1, 0, C_2, 1), (1, 0, C_3, 1)]
L_4 = [(0, 0, ∅, 0), (1, 0, C_1, 1), (1, 0, C_2, 1), (1, 0, C_3, 1), (2, 0, C_4, 1)]
L_4 = [(1, 0, C_1, 1), (1, 0, C_2, 1), (1, 0, C_3, 1), (2, 0, C_4, 1)] : after step 5
L_5 = [(1, 0, C_1, 1), (1, 0, C_2, 1), (1, 0, C_3, 1), (2, 0, C_4, 1),
(2, 0, [C_1,C_5], 2), (2, 1, [C_2,C_5], 1), (2, 0, [C_3,C_5], 2), (3, 0, [C_4,C_5], 2)]
L_5 = [(2, 0, C_4, 1), (2, 0, [C_1,C_5] , 2), (2, 1, [C_2,C_5], 1),
(2, 0, [C_3,C_5], 2), (3, 0, [C_4,C_5], 2)] : after step 5
L_6 = [(2, 0, C_4, 1), (2, 0, [C_1,C_5] , 2), (2, 1, [C_2,C_5] , 1), (2, 0, [C_3,C_5] , 2),
(3, 0, [C_4,C_5] , 2), (3, 0, [C_4,C_6] , 2), (3, 1, [C_1,C_5,C_6] , 2), (3, 2, [C_2,C_5,C_6] , 1),
(3, 1, [C_3,C_5,C_6] , 2), (4, 1, [C_4,C_5,C_6]) , 2)]
L_6 = [(3, 0, [C_4,C_5] , 2), (3, 0, [C_4,C_6] , 2), (3, 1, [C_1,C_5,C_6] , 2),
(3, 2, [C_2,C_5,C_6] , 1), (3, 1, [C_3,C_5,C_6] , 2), (4, 1, [C_4,C_5,C_6]) , 2)] : after step 5

Fig. 7. Applying the proposed algorithm to an example.

3. A C_k whose weighted gain (i.e., gain multiplied by its occurrence frequency) is much less than that of other C_ks can be eliminated from the C_k list because such a C_k has little chance to be selected as a C-instruction, but increases the complexity.

These techniques are very effective in reducing the number of C_ks to be considered and, thus, lead to reducing the computation time. If we fail to find a solution under the length limit n, we increase the limit and then retry to find a solution. Since the solution space to be searched increases as the limit increases, the chance of finding a solution increases also. However, we cannot increase the limit beyond some bound because it may significantly degrade programmability. If the bound is reached, we have to generate S-class instructions which are assisted by special hardwares.

3.3 Pruning Search Space

The pruning of Fig. 6 is very important in reducing the computation time of the algorithm and making the algorithm handle practical-sized problems. Here are the pruning techniques:

1. The element whose C-instruction size, *ISS*, is larger than the instruction space allocated for C-instructions is eliminated.
2. The element whose total gain, *TG*, has no possibility to meet T_d is eliminated. This is checked by computing the maximal gain obtainable from the remaining C_ks. In the computation, we consider the *CGSC* by taking account of MWIS (Maximal Weighted Independent Set) in the remainders.
3. We stop the algorithm as soon as we find a solution meeting the given constraints without further searching all the remaining solution space. This is the most effective way to reduce the computation time of the proposed algorithm. However, not to degrade solution quality much, our algorithm has to be changed to the BFS (Best First Search) style. This is achieved by sorting the C_ks in descending order of their weighted gain before calling the subset-sum problem solver. If we assume that a solution is found after processing C_i, what we can get at best by processing C_js sorted after C_i is a solution that has a larger gain but the same number of C-instructions. Since the larger gain has no meaning in our objective once the required gain is met, we can stop the searching as soon as we find a solution.
4. The element whose total gain is much less than those of other elements is eliminated. This is also possible due to the sorted C_ks. Such an element has little chance to beat other elements and become a solution because the remaining C_ks have less weighted gains and may increase the C-instruction set size.

The overall complexity of the algorithm without pruning is $O(2^n)$, where n is the number of C_ks. The complexity of each of the pruning technique is as follows: The first, the third, and the fourth pruning techniques are performed simultaneously by just scanning each element of L_i for each i. Thus, the complexity of them is $O(|L_i|)$. For the second technique, we have to compute the MWIS of each element of L_i, hence the complexity is $O(|L_i| \bullet (n+e))$. Here, n is the number of C_ks and e is the number of edges in the C_k conflict graph (G_c). So, the total complexity of pruning techniques is $O(|L_i| \bullet (n+e+1)) \cong O(|L_i| \bullet (n+e))$.

Now, we think about the effect of pruning. Assume that, on the average, the first, the second, and the fourth techniques prune away $(1-p)$ portion of L_i $(0 \leq p \leq 1)$, i.e., p is the average portion of L_i that is not pruned away. And, m is the number of C_ks processed until we find a solution. Then, since we stop the algorithm as soon as we find a solution (the third pruning technique), overall complexity of the algorithm with pruning becomes $O(2^m p(n+e))$. Though it depends on the characteristics of the application program and T_d, m is generally much less than n, thus the pruning can reduce much of the computation time in many cases.

3.4 C-Isomorphism in Generating C-instructions

In this part, we present the definition of and the way to consider the *c-isomorphism* in generating C-instructions.

C_1=ADD R1, R2, R3 MOV R4, R1	**C_2=ADD R1, R2, R3 MOV R4, R1**
C_3=ADD R5, R2, R3 MOV R4, R5	**C_4=ADD R5, R6, R3 MOV R4, R5**

Fig. 8. Example for c-isomorphism.

Given a MOP list, C_ks are said to be *identical* if they have the same operation sequence and the same operands. Clearly, identical C_ks can be implemented by a C-instruction. Since identical C_ks rarely exist in real codes, we use *c-isomorphic* C_ks instead of the identical C_ks in generating C-instructions.

Two isomorphic C_ks are said to be *c-isomorphic* if a single C-instruction can specify both of them by encoding some information in the instruction format. Because of the limit on the operand encodings allowed in the instruction format, not all the isomorphic C_ks can be c-isomorphic. As an illustration, consider Fig. 8 showing four C_ks.

We can see that C_1 and C_2 are identical, hence they can be implemented by a C-instruction. For C_1 and C_3, we can see that they are not identical but isomorphic in topological point of view; R1 in C_1 is replaced by R5 in C_3. To map C_1 and C_3 into a C-instruction, we have to provide some information to the instruction regarding which register should be used for the first operand. If such an encoding is allowed in the instruction format, C_1 and C_3 can be mapped into a C-instruction, but if not, we cannot merge them. Similarly, in order to unite C_1 and C_4 as a single C-instruction, we need to encode two operands; one for R1 in C_1 and R5 in C_4, and the other for R2 in C_1 and R6 in C_4.

This c-isomorphism is considered when we compute *XG* in the subset-sum solver, i.e., in step 4 of Fig. 6. We first check whether *CS* already contains any C_j that is isomorphic with C_i to be added. If such C_js exist, we compute the required encoding information to make C_i and C_js c-isomorphic, and check whether the encoding obeys the encoding constraints. If it is true, *XG* is increased by one.

Here, we need to address the use of temporary registers to reduce the required encoding information. Let us assume that the ASIP has three temporary registers accessible in the μ-codes. We can use these temporary registers to reduce the required encoding information by replacing the general registers of C_ks with them. As an illustration, assume that R1 in C_1 and R5 in C_3 of Fig. 8 are not used any more after the MOV instruction, i.e., they are not live variables after the MOV instruction. By replacing them with a temporary register, C_1 and C_3 become identical with no encoding information at all. We analyze the variable's life time to find such registers.

The encoding constraint has a significant effect on the resulting code. If we increase the number of registers to be encoded in the instruction format, we can find C-instructions used more frequently in the code. However, due to the additional encoding information, the size of the instruction format (i.e., the number of bits) required for the C-instruction increases. The size of μ-ROM may decrease because of the C-instructions that cover many c-isomorphic C_ks. On the contrary, if we decrease the allowed encodings, the size of the instruction format decreases and the size of μ-ROM may increase.

3.5 Instruction Generation Considering Other Basic Blocks

Hitherto, we have addressed how to generate C-instructions for a basic block (a sequence of consecutive codes in which flow of control enters at the beginning and leaves at the end without halt or possibility of branching except the end). In this part, we present a way to consider other basic blocks simultaneously in the generation of C-instructions. It is very important in that it can enable us to find C-instructions more frequently applicable in a global point of view. Notice that we assume the basic blocks are on a path of the application program and the required gain, T_d, for the path is given. The details of computing the gain and the way to generate C-instructions for multipaths are addressed in the next section.

The method is as follows: We scan all the basic blocks to generate the possible C_ks and the corresponding conflict graphs. Since C_ks in different basic blocks have no common MOPs, no additional edges are necessary between the conflict graphs of different basic blocks. We run the subset-sum problem solver with the gathered C_ks. The subset-sum problem solver then finds the optimal C-instruction set for all the basic blocks, not for a single basic block.

However, for a long path, the method may become inefficient because of the large number of C_ks. In this case, we first group the basic blocks and then apply the above method to each group separately. To minimize the possible degradation of solution quality caused by the grouping, we have to make the basic blocks in the same group have isomorphic C_ks as many as possible, and those in different groups as few as possible. After generating C-instructions for a group, the generated C-instructions are used to guide the generation of C-instructions for other groups. In other words, C_ks that are c-isomorphic with the already generated C-instructions have more possibility to be selected in the solution.

The grouping is performed as follows: First, the connectivity between two basic blocks are computed, which represents the number of isomorphic C_ks increased if those two basic blocks are included in the same group. Second, we build a graph where a vertex is a basic block and an edge represents the connectivity between two basic blocks. We divide the graph by using the min-cut partitioning algorithm [17] until the number of C_ks in each partition falls into a size that can be handled efficiently.

Now, we address how to distribute the required gain, T_d, to each group in a path. We sort the C_ks gathered from all the basic blocks in descending order of their weighted gains. Starting from the first C_k in the list, we compute the expected gain of each basic block when the C_k becomes a C-instruction. In this step, we consider the possibility that some C-instructions cannot be applied simultaneously in a block if they have common MOPs. This process is performed until the total expected gain is no less than T_d. We compute the expected gain of each group by summing up those of the basic blocks contained in the group. T_d is distributed to each group in proportion to the expected gain of the group.

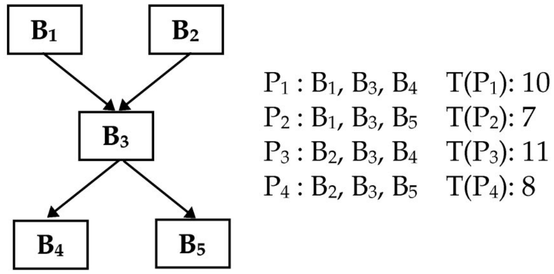

Fig. 9. Multipath case.

3.6 Instruction Generation Considering Multipaths

In the previous sections, we focused on generating C-instructions in one path. Since given application programs usually have more than one execution path, we describe in this section the details of computing the required gain for each path and the way to generate C-instructions for multipaths.

In real-time DSP applications, the time constraint to be met is hard. Stated in another way, all the execution paths should meet the given timing constraint. We first match the MOP list to the P-instructions and check whether the execution time of every possible execution path meets the given constraint. If it meets the constraint, the required gain for the path is zero. However, if not, the difference between the execution time and the timing constraint becomes the required gain for the path. As an illustration, consider the case shown in Fig. 9 where five basic blocks make four execution paths, P_1-P_4. The execution time for each path when mapped to P-instructions is also shown in the figure. Given that the timing constraint is 9, P_1 and P_3 cannot meet

TABLE 1
C-Instructions for the Code Size Reduction

Benchmark programs	P-size	T_d	SCC G	SCC C	SCC+MCC G	C	O	2	3	4	5	Time
convolution	48	4	4	2	4	1	2	0	1	0	0	1.1
		9	9	7	9	3	4	0	2	1	0	1.5
		14	-	-	15	4	5	0	1	1	2	2.2
complex_ update	51	5	5	3	5	1	5	1	0	0	0	1.3
		10	10	5	10	3	6	1	1	0	1	2.1
		15	-	-	15	4	7	1	0	1	2	2.6
for01	51	5	5	3	6	1	2	0	0	1	0	0.4
		10	10	7	10	2	4	0	1	1	0	0.8
		15	-	-	15	4	5	0	1	1	2	1.0
nested_for2	63	6	6	3	6	1	2	0	0	1	0	1.1
		12	-	-	12	2	5	0	1	1	0	1.6
		18	-	-	18	4	7	0	2	0	2	2.2
fir	67	6	6	3	7	3	3	0	2	1	0	2.2
		13	-	-	13	4	4	0	1	1	2	3.1
		20	-	-	20	6	7	0	3	0	3	4.4
complex_ multiply	68	6	6	2	8	1	2	0	0	0	1	20.1
		13	13	4	13	2	4	0	0	1	1	53.8
		20	-	-	20	3	6	0	0	2	1	101.4
parallel022	81	8	8	3	12	1	4	0	0	1	0	3.5
		16	16	8	18	3	7	1	0	1	1	45.3
		24	-	-	24	5	9	1	1	1	2	162.1
biquad_4_ sections	99	9	9	3	9	2	3	1	0	0	1	3.5
		19	19	10	19	3	10	1	1	0	1	8.4
		29	-	-	30	6	10	0	2	1	3	33.2
lms	125	12	12	2	12	1	4	0	0	1	0	59.9
		25	25	6	25	5	10	1	0	1	3	159.3
		36	-	-	36	10	13	2	2	0	6	86.5^{+}
fft	140	14	14	3	15	2	14	1	1	0	0	37.4
		28	-	-	28	6	13	1	1	2	2	36.7^{+}
		42	-	-	45	11	21	2	1	3	5	68.2^{+}
fir2dim	150	15	15	3	16	1	8	0	1	0	0	40.0
		30	-	-	31	2	14	0	1	1	0	70.1
		45	-	-	46	9	17	0	3	2	4	116.3^{+}
adpcm	195	19	19	6	19	1	19	1	0	0	0	140.5
		39	-	-	39	7	25	2	2	2	1	131.2^{+}
		58	-	-	60	14	33	4	2	4	4	175.4^{+}
edge	204	20	20	4	20	1	20	1	0	0	0	239.1
		40	-	-	40	8	28	3	2	1	2	157.5^{+}
		61	-	-	61	14	36	4	4	2	4	207.7^{+}

TABLE 2
C-Instructions for the Execution Time Reduction

Benchmark programs	P-cycle	T_d	SCC		SCC + MCC							
			G	C	G	C	O	2	3	4	5	Time
complex_ update	51	5	5	3	6	1	3	1	0	0	0	1.1
		10	10	5	10	1	5	1	0	0	0	1.3
		15	-	-	16	2	6	1	1	0	0	2.0
complex_ multiply	68	6	6	2	6	1	1	0	0	0	1	9.9
		13	13	4	14	2	3	0	0	1	1	67.2
		20	-	-	24	2	4	0	0	1	1	69.8
for01	87	8	8	5	20	1	10	0	1	0	0	0.4
		17	-	-	20	1	10	0	1	0	0	0.4
		26	-	-	26	2	11	0	1	0	1	0.5
parallel022	145	14	14	4	14	2	3	0	0	1	1	5.4
		29	-	-	30	3	16	1	1	1	0	19.6
		43	-	-	44	5	23	2	2	1	0	44.4
convolution	228	22	22	5	48	1	16	0	1	0	0	1.1
		45	-	-	48	1	16	0	1	0	0	1.1
		68	-	-	80	2	32	0	2	0	0	1.2
fir	232	23	-	-	32	1	16	0	1	0	0	1.7
		46	-	-	64	1	32	0	1	0	0	1.9
		69	-	-	80	2	48	1	1	0	0	3.3
biquad_4_ sections	277	27	27	3	28	1	28	1	0	0	0	5.2
		55	-	-	56	1	28	0	1	0	0	3.7
		83	-	-	84	2	52	1	1	0	0	6.3
lms	410	41	41	6	64	1	32	0	1	0	0	37.7
		82	-	-	96	1	48	0	1	0	0	40.2
		123	-	-	128	2	48	0	1	1	0	47.0
fft	1360	136	-	-	192	2	32	0	0	0	5	20.0
		272	-	-	353	5	61	0	0	2	3	27.1^{+}
		408	-	-	513	7	107	0	0	2	5	37.3^{+}
fir2dim	1509	150	150	8	178	1	89	0	1	0	0	34.8
		301	-	-	338	1	169	0	1	0	0	37.8
		452	-	-	466	1	233	0	1	0	0	40.6
adpcm	5382	538	-	-	561	1	561	1	0	0	0	82.1
		1076	-	-	1122	4	722	2	0	2	0	92.4^{+}
		1614	-	-	1640	10	1600	3	2	3	2	180.4^{+}
notch02	7643	764	-	-	800	1	400	0	1	0	0	5.3
		1528	-	-	1600	1	400	0	0	1	0	2.8
		2292	-	-	2400	2	800	0	0	2	0	5.1

the timing constraint, and the required gains for P_1 and P_3 are 1 and 2, respectively.

After computing the required gains for all the paths, we compute the gain required for each basic block in a path as explained in Section 3.5. For a basic block belonging to more than one path, several different gains may be required according to the paths. In that case, the largest one is chosen as the gain required for the basic block. Assume that the required gains for B_3 are 1 and 2 for P_1 and P_3, respectively. Then, 2 is chosen as the required gain for B_3.

Based on the computed gain for each basic block, C-instructions are generated by considering one path after another path. In this phase, we give priority to the path that requires large gain. After generating C-instructions for a path, the C-instructions are used to guide the generation of C-instructions for other ones to reduce the possible degradation due to the path-by-path C-instruction generation.

4 Matching to P-Class Instructions

This section describes how the matching of a MOP list to the P-instructions is performed in the proposed framework. A P-instruction is a predefined one-cycle instruction that can perform a limited set of MOPs in a cycle. Compared to the SCC-instruction, the difference is that the possible set of MOPs that can be performed in parallel is limited for P-instructions: A predefined, limited set is allowed due to the instruction encoding constraint. So, we can regard the P-instruction as a special subset of the SCC-instruction. Thus, if we allow the C_ks to include only MOPs that can be executed in parallel in a single P-instruction, we can use the algorithm described in Fig. 6 for the P-instruction matching.

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 the DSPStone benchmarks [15] and some well-known DSP applications. For

TABLE 3
Maximum Gains under Various Constraints on the Number of C-Instructions

Benchmark programs	P-size	Gain in code size				P-cycle	Gain in execution cycle			
		1	2	3	4		1	2	3	4
convolution	48	3	7	11	16	228	64	128	144	148
complex_up	51	5	9	13	17	51	10	16	20	28
for01	51	6	10	13	15	87	20	26	31	36
notch02	62	4	8	12	16	7643	2000	2800	3600	4000
nested_for2	63	6	12	14	18	6161	2096	4016	4022	4025
fir	67	3	5	9	13	232	64	80	86	90
complex_mul	68	9	13	19	22	68	6	12	18	24
parallel022	81	12	16	18	19	145	16	19	29	39
biquad_4_s	99	8	13	19	21	277	64	92	108	116
lms	125	12	16	20	24	410	96	160	208	216

each benchmark program, we first mapped it into P-instructions. Then, we employed the proposed C-instruction generation algorithm to reduce the code size and the execution time. For all the experiments, the maximum number of encodings and the maximum length of the C-instruction are set to 6 and 5, respectively.

5.1 Code Size Reduction

Table 1 shows the statistics of C-instructions generated for the reduction of code size. The **P-size** shows the code size (i.e., the number of instructions) of the P-instruction code. The required gain is shown in $\mathbf{T_d}$; for each benchmark, we tried three gains—10 percent, 20 percent, and 30 percent reduction of code size. **SCC** shows the results obtained by generating only SCC-instructions, i.e., turning off the ability to generate MCC-instructions. **G** represents the gain (code size reduction) and **C** represents the number of generated SCC-instructions. The columns under **SCC+MCC** show the results obtained by generating MCC-instructions, as well as SCC-instructions. **O** shows how often the generated C-instructions are used in the code. The following four columns, **2-5**, show the number of 2-cycle C-instructions, 3-cycle C-instructions, etc., that were generated. The last column, **Time**, shows the CPU time in seconds taken for the generation of the C-instructions ($^+$ means that the basic block grouping explained in section 3.5 was used for that case).

In many cases, we could not meet $\mathbf{T_d}$ by generating only SCC-instructions. We could meet $\mathbf{T_d}$ for all the cases by generating MCC-instructions. Note that though we allowed SCC-instructions to be generated, as well as MCC-instructions, SCC-instructions were not generated at all under **SCC+MCC**; only the MCC-instructions were generated. We can also see that the number of C-instructions under **SCC+MCC** is much smaller than that under **SCC**. These clearly show the importance of MCC-instructions. The number of MCC-instructions and that of their occurrences indicate that the proposed method generates the valuable MCC-instructions which are used frequently in the code; for example, a C-instruction that is used 19 times was found for the benchmark *adpcm*. The CPU time is no more than three minutes, which is reasonable for the optimization of embedded software.

5.2 Execution Time Reduction

Table 2 shows the statistics of generated C-instructions for the reduction of execution time. The **P-cycle** shows the execution time (in cycles) of the P-instruction code obtained by the profiler. The required gain is set to 10 percent, 20 percent, and 30 percent reduction of the execution cycle. The column **O** shows how often the generated C-instructions are executed.

We could meet $\mathbf{T_d}$ for all the cases by generating MCC-instructions. In contrast, we could not meet $\mathbf{T_d}$ under **SCC** in many cases, and the number of C-instructions generated under **SCC** was larger than that under **SCC+MCC**. In addition, we can see that the proposed method generates C-instructions, which are executed frequently in the code with reasonable CPU time. We found a C-instruction that is executed 561 times for the benchmark *adpcm*.

5.3 Maximum Gain under Various Number of C-Instructions

Table 3 shows the maximum gain obtainable by limiting the number of C-instructions allowed. **Gain in code size** and **Gain in execution cycle** show the maximum gain in code size and execution cycle, respectively, that we can achieve under the given number of C-instructions. We can see that large gain (especially in execution cycle) is obtained by generating only a few C-instructions. This clearly shows that we can achieve large gain by merely generating proper

MVFRS R0, SP
MVTOS FP, R0
(a)

LDI R0, ***enc1***
ADD ***enc2*****, FP, R0**
(b)

MVTOS ***enc1*****, R0**
MOV R1, AR1
LDI ***enc2*****,** ***enc3***
(c)

CMP ***enc1*****,** ***enc2***
B.LE ***enc3***
(d)

ADD ***enc1*****,** ***enc1*****, 1**
CMP ***enc1*****,** ***enc2***
B.LE ***enc4***
(e)

LDI R0, ***enc1***
SHL ***enc2*****,** ***enc3*****, R0**
(f)

LDI ***enc1*****,** ***enc2***
PUSH ***enc1***
(g)

LDI R0, ***enc1***
ASR ***ir1*****,** ***enc2*****, R0**
ADD ***enc3*****,** ***enc3*****,** ***ir1***
(h)

Fig. 10. Some of the C-instructions generated.

C-instructions without using additional special hardware accelerator.

5.4 C-Instructions Generated

Some examples of the C-instructions generated in the above experiments are shown in Fig. 10. Figs. 10a, b, and c are C-instructions commonly used at the beginning of functions (subroutines). Notice that the execution cycle of Fig. 10c is two, not three, because MOV and LDI can be executed in one cycle. The codes in Figs. 10d and e are frequently used at the end of loops (e.g., for, while, etc.). Figs. 10f and g are the C-instructions that are equal to the *shift by immediate value* and the *push immediate value*, which are commonly used but are not supported in the P-instruction set. Fig. 10h is equal to $enc3 \mathrel{+}= enc2/2^{\wedge}enc1$ which is actively used in the *adpcm* benchmark. Notice that these C-instructions are MCC-instructions each of which is comparable to two or three P-instructions. This shows the importance of the generation of MCC-instructions.

6 Conclusions

In this paper, we presented a new approach to generate application specific instructions from the given DSP applications. We transformed the instruction generation problem to the extended subset-sum problem, and used the subset-sum problem solver to synthesize application specific complex instructions. Along with many things to be considered, such as c-isomorphism and encoding constraints, we showed the way to apply the proposed framework for the generation of single-cycle C-instructions and multicycle C-instructions. In addition, we described how to apply the proposed method to the practical problems that are large-sized and have multipaths. The experimental results indicate that the proposed approach is effective in reducing the code size, as well as increasing the performance. For numerous benchmarks including DSP applications, the proposed method can find multicycle C-instructions that are enough to meet the required timing constraints.

Acknowledgments

The authors would like to thank the anonymous reviewers for their invaluable comments. This work was performed as a part of the ASIC Development Project supported by the Ministry of Trade, Industry, & Energy, Ministry of Information and Communication, and Ministry of Science and Technology of Korea.

References

[1] W. Zhao and C.A. Papachristou, "An Evolution Programming Approach on Multiple Behaviors for the Design of Application Specific Programmable Processors," *Proc. European Design & Test Conference,* pp. 144-150, 1996.

[2] W. Zhao and C.A. Papachristou, "Synthesis of Reusable DSP Cores Based on Multiple Behaviors," *Proc. Int'l Conf. Computer-Aided Design,* pp. 103-108, 1996.

[3] K. Kim, R. Karri, and M. Potkonjak, "Synthesis of Application Specific Programmable Processors," *Proc. 34th Design Automation Conf.,* pp. 353-358, 1997.

[4] J.H. Yi, H. Choi, I.C. Park, S.H. Hwang, and C.M. Kyung, "Multiple Behavior Module Synthesis Based on Selective Groupings," *Proc. Design, Automation, and Test in Europe,* pp. 384-388, 1998.

[5] C. Liem, T. May, and P. Paulin, "Instruction-Set Matching and Selection for DSP and ASIP Code Generation," *Proc. European Design and Test Conf.,* pp. 31-37, 1994.

[6] G. Araujo and S. Malik, "Optimal Code Generation for Embedded Memory Non-Homogeneous Register Architectures," *Proc. Int'l Symp. System Synthesis,* pp. 36-41, 1995.

[7] G. Araujo, S. Malik, and M.T.-C. Lee, "Using Register-Transfer Paths in Code Generation for Heterogeneous Memory-Register Architectures," *Proc. 33rd Design Automation Conf.,* pp. 591-596, 1996.

[8] R. Leupers and P. Marwedel, "Instruction Selection for Embedded DSPs with Complex Instructions," *Proc. European Design Automation Conf.,* 1996.

[9] R. Leupers and P. Marwedel, "Time-Constrained Code Compaction for DSP's," *IEEE Trans. VLSI Systems,* vol. 5, no. 1, pp. 112-122, Mar. 1997.

[10] M. Imai, A. Alomary, J. Sato, and N. Hikichi, "An Integer Programming Approach to Instruction Implementation Method Selection Problem," *Proc. European Design Automation Conf.,* pp. 106-111, 1992.

[11] A. Alomary, T. Nakata, Y. Honma, M. Imai, and N. Hikichi, "An ASIP Instruction Set Optimization Algorithm with Functional Module Sharing Constraint," *Proc. Int'l Conf. Computer-Aided Design,* pp. 526-532, 1993.

[12] I.J. Huang and A.M. Despain, "Synthesis of Instruction Sets for Pipelined Microprocessos," *Proc. 31st Design Automation Conf.,* pp. 5-11, 1994.

[13] M.T.-C. Lee, V. Tiwary, S. Malik, and M. Fujita, "Power Analysis and Minimization Techniques for Embedded DSP Software," *IEEE Trans. VLSI Systems,* vol. 5, no. 1, pp. 123-135, Mar. 1997.

[14] T.H. Cormen, C.E. Leiserson, and R.L. Rivest, *Introduction to Algorithms,* chapter 37, pp. 978-983. The MIT Press, McGraw-Hill Book Company, 1992.

[15] V. Zivojnovic, "DSPStone: A DSP-Oriented Benchmarking Methodology," *Proc. Int'l Conf. Signal Processing Applications & Technology,* pp. 715-722, 1994.

[16] S. Malik, M. Martonosi, and Y.-T.S. Li, "Static Timing Analysis of Embedded Software," *Proc. 34th Design Automation Conf.,* pp. 147-152, 1997.

[17] C.M. Fiduccia and R.M. Mattheyses, "A Linear-Time Heuristic for Improving Network Partitions," *Proc. 19th Design Automation Conf.,* pp. 175-181, 1982.

Hoon Choi received the BS degree in electrical engineering from Yonsei University, Seoul, Korea, and the MS degree in electrical engineering from the Korea Advanced Institute of Science and Technology (KAIST), Taejon, Korea, in 1993 and 1995, respectively. He is currently pursuing the PhD degree in electrical engineering from KAIST. His research interests include hardware/software co-design, timing analysis, and power optimization. He is a student member of the IEEE.

Jong-Sun Kim received the BS degree and MS degree in electrical engineering from the Korea Advanced Institute of Science and Technology (KAIST), Korea, in 1995 and 1997, respectively. He is currently a PhD candidate in the same department at KAIST. His current research topic is hardware/software co-design with emphasis on the generation and synthesis of interface.

Chi-Won Yoon received the BS degree in electrical engineering from KAIST, Taejon, in 1997. Currently, he is working toward the MS degree in electrical engineering at KAIST. His research interests includes the code generation, hardware/software co-design, and system performance analysis. He is a student member of the IEEE.

In-Cheol Park received the BS degree in electrical engineering from Seoul National University in 1986, the MS and PhD degrees in electrical engineering from KAIST in 1988 and 1992, respectively. From May 1995 to May 1996, he worked at the IBM T.J. Watson Research Center, Yorktown Heights, New York, as a postdoctoral member of the technical staff in the area of circuit design. He joined KAIST in June 1996 as an assistant professor in the Department of Electrical Engineering. His current research interests include CAD algorithms for high-level synthesis and VLSI architectures for general-purpose microprocessors. He is a member of the IEEE.

Seung Ho Hwang received the BS degree in electronics engineering from Seoul National University in 1979, the MS degree in electrical engineering from KAIST in 1981, and the PhD degree in electrical engineering and computer science from the University of California, Berkeley. From May 1985 to May 1989, he worked at the University of California, Berkeley as a postgraduate researcher and at Schlumberger Technologies, Inc., as a senior software engineer from June 1989 to September, 1990. He joined KAIST in September 1990 as an associate professor in the Department of Electrical Engineering and, during his sabbatical year, he worked at Silicon Image, Inc. His current research interests include CAD algorithms for high-level synthesis, power estimation, hardware/software co-design, and VLSI design. He is a member of the IEEE.

Chong-Min Kyung received the BS degree in electronics engineering from Seoul National University in 1975, the MS and PhD degrees in electrical engineering from KAIST in 1977 and 1981, respectively. From April 1981 to January 1983, he worked at Bell Telephone Laboratories, Murray Hill, New Jersey, as a postdoctoral member of the technical staff in the area of semiconductor device and processor modeling. He joined KAIST in February 1983 as an assistant professor in the Department of Electrical Engineering, where he is now a professor. He was a visiting professor at the Institute für Theoretische Electrotechnik und Mebetatechnik at the University of Karlsruhe, Germany, from February 1989 to November 1989 as an Alexander von Humboldt fellow, working on VLSI layout algorithms, and as a visiting professor at the University of Tokyo from January 1985 to February 1985. His current research interests include CAD algorithms for all aspects of VLSI design, 3D computer graphics algorithms and hardware acceleration for still image rendering and animation, VLSI architectures for general-purpose microprocessors. He was the Asian Representative for ICCAD (International Conference on Computer-Aided Design) in 1993 and 1994. He is a member of the IEEE.