ELSEVIER

Microprocessing and Microprogramming 41 (1995) 37–52

Microprocessing and Microprogramming

# A new approach to schedule operations across nested-ifs and nested-loops†

Shih-Hsu Huang[a], Cheng-Tsung Hwang[b], Yu-Chin Hsu[b], Yen-Jen Oyang[a,*]

[a] *Department of Computer Science and Information Engineering, National Taiwan University, Taipei, Taiwan.*

[b] *Department of Computer Science, University of California, Riverside, CA 92521, USA*

Received 23 July 1993; revised 14 April 1994; accepted 30 June 1994

**Abstract**

This paper presents a new global scheduling algorithm for automatic synthesis of the control blocks of special-purpose microprocessors. The main distinction of the proposed algorithm is that it exploits the inheritances of structured programs. The optimization goal is to maximize the speedup of the processor and minimize the size of the control block. If compared with existing global scheduling algorithms such as Trace scheduling, Tree compaction, and Percolation scheduling, the proposed algorithm consistently achieves better results in terms of the speedup of the processor and the size of the control block.

*Keywords:* Fine-grain parallelism; Global scheduling; High-level synthesis; Global mobility; Control minimization

## 1. Introduction

VLSI technologies have pushed chip density to over one million transistors in the early 1990s and continue to improve. Systems of such complexity are very difficult to design by handcrafting each transistor or by defining each signal in the boolean logic form. As a result, there exists an emerging demand to develop new design methodologies which allow the designer to deal with the design at more abstract levels where functionality and tradeoff are easier to understand. As logic and register transfer levels synthesis tools have gained a stable foothold in real-world applications, automatic synthesis of digital systems starting from behavior-level descriptions [1] is the next step on the ladder of design automation hierarchy.

One of the most important applications of automatic synthesis is in the design of the control blocks of special-purpose microprocessors. In this paper,

* Corresponding author. Email: yjoyang@csie.ntu.edu.tw

† A preliminary version of the paper has appeared in Proceedings of MICRO-25.


*SSDI* 0165-6074(94)00024-7

we will propose a new global scheduling algorithm called GSSP (Global Scheduling for Structured Programs) for automatic synthesis of control blocks starting from behavior-level descriptions. As its name suggests, the main distinction of the GSSP algorithm is that it exploits the inheritances of structured programs. The input to the GSSP algorithm is a behavior description written in a structured hardware description language. Based on several interesting observations about the mobility of operations in the flow diagrams derived from structured programs, the GSSP algorithm works to maximize the speedup of the processor and minimize the size of the control block. If compared with existing global scheduling algorithms such as Trace scheduling [2], Tree compaction [3], and Percolation scheduling [5], GSSP consistently achieves better results in terms of the speedup of the processor and the size of the control block.

The rest of the paper is organized as follows. Section 2 presents the primitives to move operations between adjacent blocks. Section 3 discusses how to determine the global mobility of operations. Section 4 elaborates the global scheduling algorithm. Section 5 reports some experimental results and comparisons with other approaches. Finally, concluding remarks are given in Section 6.

## 2. Movement primitives

This section introduces the movement primitives that move operations between adjacent basic blocks in the flow diagrams derived from structured programs. These movements primitives form the basis of the global scheduling algorithm presented in the following sections.

### 2.1 Preprocessing

The input to the GSSP program is a behavior-level description written in a structured hardware description language. Fig. 1 shows the control statements of the input language. Along with the behavior description, the designer can specify the resource constraints such as the number and/or type of hardware modules that can be used to synthesize the circuit.

```
If Statement
Case Statement
For Statement
While Statement
Procedure Call Statement
Return Statement
```

Fig. 1. The control statements of the input language.

As mentioned earlier, the GSSP algorithm is distinctive in the exploitation of the inheritances of the input structured program. The most important inheritances exploited are:

(1) An *if* construct spreads a true part and a false part and these two parts meet at a joint-block. A *case* construct can be translated into nested ifs and treated accordingly.
(2) A loop has only one entry and one exit since there is no break statement in the input structured language.

Fig. 2(a) gives an example to be used throughout the paper to illustrate the GSSP algorithm. In this example, variables $i0$, $i1$, and $i2$ are inputs, and variables $o1$ and $o2$ are outputs. The behavioral description is firstly compiled into a *flow graph* which consists of a set of basic blocks linked by flow-of-controls. Fig. 2(b) shows the flow graph of this example.

The first step in the synthesis process is to transform a loop in the 'pre-test' form into an 'if construction' whose true part is the original loop in the 'post-test' form and whose false part is an empty block. In Fig. 2(b), operation $OP15$ is generated for the 'if construction'. If the control flows through the true part, the loop is executed at least once. Otherwise, the loop is not executed. In addition, a

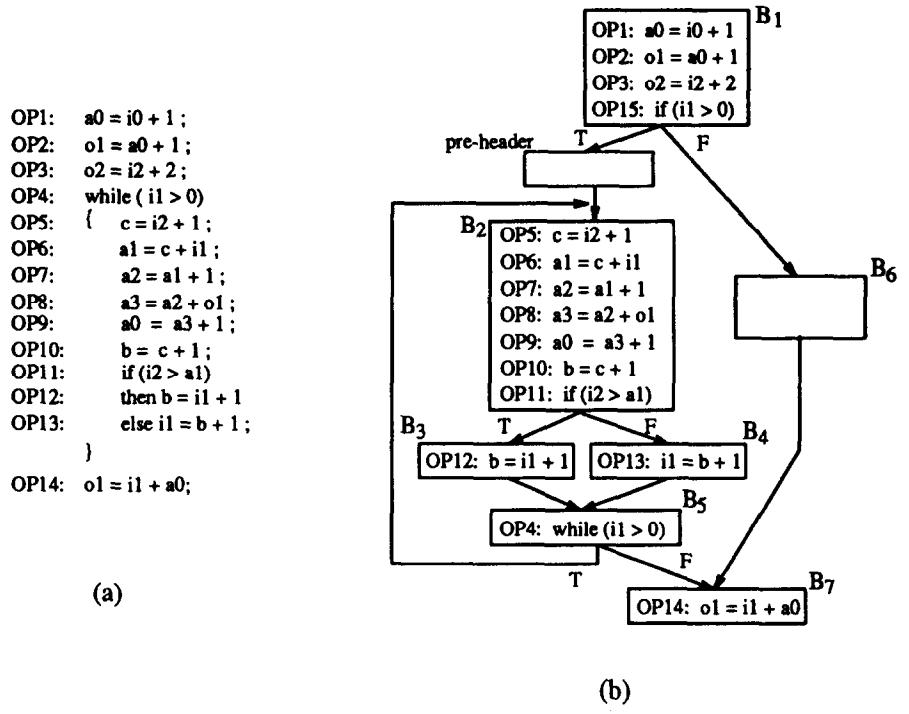



Fig. 2. An example used to illustrate the GSSP algorithm: (a) input behavior description; (b) flow graph.

pre-header is created. Initially, the pre-header is empty. The movement primitives discussed later may move some operations into the pre-header.

An operation is redundant if the value it defines will never be used under any combination of input values. Note that an operation which defines an output variable is not redundant. The GSSP algorithm assumes redundant operations are all removed during preprocessing.

### 2.2 *Movement primitives for an if construction*

A block that contains an *if* operation is called an if-block. An if-block $B_{if}$ has two immediate successors: (1) the true-block $B_{true}$, which is immediately executed after $B_{if}$ if the condition is true; (2) the false-block $B_{\text{false}}$, which is executed immediately after $B_{if}$ if the condition is false. Blocks $B_{true}$ and $B_{false}$ spread a *true part* $S_t[B_{if}]$ and a *false part* $S_f[B_{if}]$, respectively. $S_t[B_{if}]$ contains the blocks which will never be executed if the condition is false and $S_f[B_{if}]$ contains the blocks which will never be executed if the condition is true. We say that $S_t[B_{if}]$ and $S_f[B_{if}]$ are the *branch parts* of an *if* operation.

Disregarding the result of *if* comparison, the flow of control meets at some point, which is the joint-block $B_{joint}$ of the branch. Block $B_{joint}$ spawns a set of blocks $S_j[B_{if}]$, called joint part of $B_{if}$. $S_j[B_{if}]$ is executed after the branch parts of $B_{if}$. Because blocks $B_{true}$, $B_{false}$ and $B_{joint}$ are all related to $B_{if}$, we call them the *related* blocks of $B_{if}$.

In Fig. 2, $S_t[B_1]$ contains blocks *pre-header*, $B_2$, $B_3$, $B_4$, and $B_5$. $S_f[B_1]$ contains block $B_6$, and $S_j[B_1]$ contains block $B_7$. The related blocks of $B_1$ are *pre-header*, $B_6$, and $B_7$.

*2.2.1 Upward movement primitives*

An operation within a block is said to be *upward movable* if the operation can be moved from the block to its predecessor block without changing the semantic of the program. Let $d(o_i)$ denote the variable defined by an operation $o_i$ and $in[B]$ be the set of variables which are live at the entry of block $B$. A variable $x$ is live at a point $p$ if and only if the value of $x$ is used along some path in the flow graph starting at $p$. The following lemmas state the properties of the upward movement.

**Lemma 1.** *An operation $o_i$ in $B_{true}$ ($B_{false}$) can be moved upward into $B_{if}$, if:*
(1) *it has no dependency predecessor in $B_{true}$ ($B_{false}$); and*
(2) $d(o_i) \notin in[B_{false}]$ ($d(o_i) \notin in[B_{true}]$).

**Lemma 2.** *An operation $o_i$ in $B_{joint}$ can be moved upward into $B_{if}$, if*
(1) *it has no dependency predecessor in $B_{joint}$; and*
(2) *it has not dependency predecessor in $S_t[B_{if}]$ and $S_f[B_{if}]$.*

**Lemma 3.** *No operation within $B_{joint}$ can be moved upward into the branch parts.*

Lemma 3 can be proved by assuming that if an operation in $B_{joint}$ is moved to the true part, then the semantics will be violated when the false part is executed and vice versa. These 3 lemmas show that an operation in $B_{joint}$, $B_{true}$ or $B_{false}$ is either movable to $B_{if}$ or not upward movable.

*2.2.2 Downward movement primitives*

An operation within a block is said to be *downward movable* if the operation can be moved from the block to its successor block without changing the semantic of the program. The following lemmas state the properties of the downward movement.

**Lemma 4.** *An operation $o_i$ in $B_{if}$ can be moved downward to $B_{true}$ ($B_{false}$), if*
(1) *it has no dependency successor in $B_{if}$; and*
(2) $d(o_i) \notin in[B_{false}]$ ($d(o_i) \notin in[B_{true}]$).

**Lemma 5.** *An operation $o_i$ in $B_{if}$ can be moved downward to $B_{joint}$, if*
(1) *it has no dependency successor in $B_{if}$; and*
(2) *it has no dependency successor in $S_t[B_{if}]$ and $S_f[B_{if}]$.*

From the lemmas above, it is not difficult to show that the conditions of moving an operation in $B_{if}$ to $B_{true}$, $B_{false}$ and $B_{joint}$ are *mutually exclusive.*

**Theorem 1.** *No operation in branch parts can be moved downward to $B_{joint}$.*

The theorem can be proved by assuming that if an operation in true part is moved to $B_{joint}$, then the semantic will be violated when false part is executed and vice versa. This property is very important in developing the global scheduling algorithm.

*2.3 Movement primitives for a loop construction*

Because a loop may be executed more than one times, the value that an operation defines will be changed during the loop execution unless the operation is a *loop invariant.* An operation is called a loop invariant if the value it defines is not changed as long as control stays within the loop.

Each loop has a single entry point, called the *loop header.* For example, in Fig. 2, block $B_2$ is the loop header. For a loop construction, the loop header is the only successor block of the pre-header. The conditions of moving an operation between the

loop header and pre-header are discussed in the followings.

**Lemma 6.** *An operation $o_i$ can be moved upward from the loop header to the pre-header, if*
(1) *$o_i$ is a loop invariant;*
(2) *it has no dependency predecessor in the loop header.*

**Lemma 7.** *An operation $o_i$ can be moved downward from the pre-header to the loop header, if*
(1) *$o_i$ is a loop invariant;*
(2) *it has no dependency successor in the pre-header.*

## 3. Global mobility of an operation

Based on the movement primitives presented in last section, we propose the Global As-Soon-As-Possible (GASAP) and the Global As-Late-As-Possible (GALAP) algorithms. Combining the results of GASAP and GALAP, we then can determine the global mobility of an operation.

### *3.1 The Global As-Soon-As-Possible (GASAP) algorithm*

The GASAP algorithm moves each operation upward as far as possible by applying the upward movement primitives repetitively. Fig. 3 outlines the GASAP algorithm. In Fig. 3, we give each basic block a unique identification number $ID(B)$. We order basic blocks so that $ID(B_i) < ID(B_j)$ if $B_j$ is a forward successor of $B_i$. The blocks are processed in the decreasing order of their $ID$ numbers. The operations in a block are processed sequentially starting from the first operation and ignoring the comparison operations.

```
Procedure GASAP()
{  for each block B (in decreasing ID number)
       for each operation in B (from head to tail)
       {  if B is a loop header
          then try to move the operation to the pre-header by lemma 6;
          else try to perform upward movement by lemma 1 or lemma 2;
          update variable live/dead information;
       }
}
```

Fig. 3. GASAP algorithm.

If the current block is a loop header, then we try to move the operations to pre-header according to Lemma 6. Otherwise, we try to perform an upward movement by applying Lemma 1 or Lemma 2. When an operation is moved, we append it to the end of the destination block and update the variable live/dead information of the related blocks accordingly. The time complexity of the algorithm is $O(bn)$ in worst case and $O(n)$ in average, where $b$ is the number of blocks and $n$ is the total number of operations.

Let us use the flow graph in Fig. 2(b) to demonstrate the operation of GASAP. The basic blocks are processed in the sequence of $B_7$, $B_6$, $B_5$, $B_4$, $B_3$, $B_2$, pre-header, and $B_1$. In blocks $B_7$, $B_6$, $B_5$, $B_4$, and $B_3$, no operation is upward movable. The next block processed is $B_2$. According to Lemma 6, operation $OP5$ is $B_2$ can be moved upward to the pre-header. Next, we move to the pre-header. According to Lemma 1, operation $OP5$ can be further moved upward to block $B_1$. After GASAP is completed, we obtain the new flow graph shown in Fig. 4.

### *3.2 The Global As-Late-As-Possible (GALAP) algorithm*

The GALAP algorithm moves each operation downward as far as possible by applying downward movement primitives repetitively. Fig. 5 outlines the GALAP algorithm. In GALAP, blocks are processed in the increasing order of their $ID$ numbers. The operations in a block are processed sequentially starting from the last operation and ignoring the comparison operations.

If the block being processed is a pre-header, we try to move the operations downward to the loop

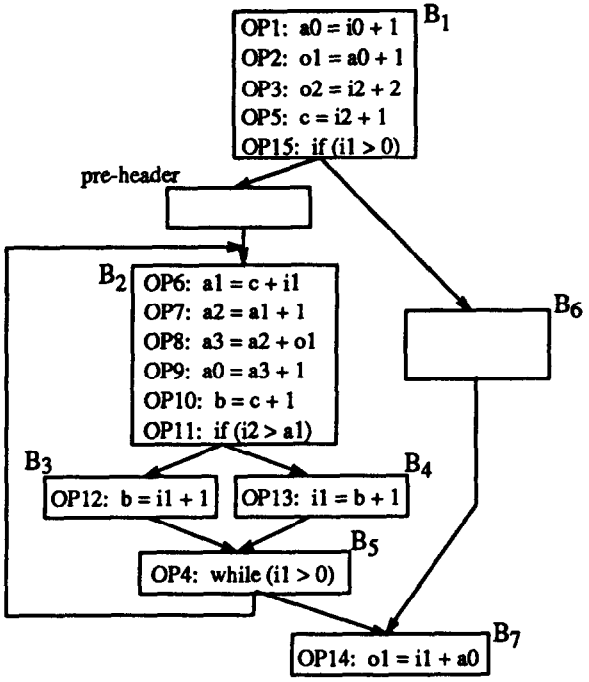



Fig. 4. Result of GASAP.

```
Procedure GALAP()
{  for each block in increasing ID number
      for each operation in B (from tail to head)
      {  if B is a pre-header
            then try to move the operation to the loop header by lemma 7;
            else try to perform downward movement by lemma 4 or lemma 5;
            update variable live/dead information;
      }
   }
}
```

Fig. 5. GALAP algorithm.

header based on Lemma 7. Otherwise, we try to move the operations downward to the related blocks by applying Lemma 4 and Lemma 5. When an operation is moved, the variable live/dead information of the related blocks are updated accordingly.

Let us use the flow graph in Fig. 2(b) to demonstrate the operation of the GALAP algorithm. The blocks are processed in the sequence of $B_1$, pre-header, $B_2$, $B_3$, $B_4$, $B_5$, $B_6$, and $B_7$. Because block $B_1$ is an if-block, we try to move the operations downward to blocks *pre-header*, $B_6$ and $B_7$. Firstly, operation $OP3: o2 = i2 + 2$ is moved to the head of $B_7$ by Lemma 5. Then, operation $OP2: o1 = a0 + 1$ is moved to the pre-header by Lemma 4. Operation $OP1: a0 = i0 + 1$ cannot be moved because variable $a0$ is used by $OP2$ in the pre-header and $OP14$ in block $B_7$.

Next, we move to the pre-header. Note that operation $OP2$ now is in the pre-header. Because operation $OP2$ is not a loop invariant, it cannot be further moved downward to the loop header, $B_2$. The next block processed is $B_2$. In $B_2$, operation $OP10$ can be moved downward to block $B_4$ and operations $OP9$, $OP8$, and $OP7$ can be moved downward to block $B_5$. We then examine blocks $B_3$, $B_4$, $B_5$, $B_6$, and $B_7$, and find that no operation in these blocks is downward movable. Fig. 6 shows the final result after applying the GALAP algorithm.

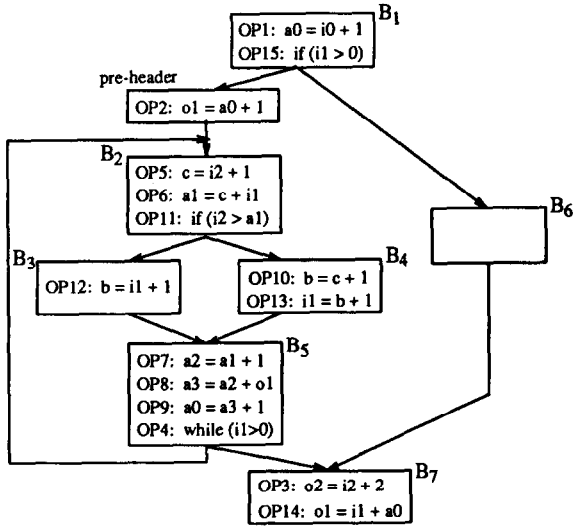



Fig. 6. Result of GALAP.

### 3.3 Scheduling strategy based on global mobility

Combining the results of GASAP and GALAP, we can determine the global mobility of an operation. The mobility of an operation is defined by the blocks which the operation may be scheduled into. Table 1 tabulates the global mobility of the operations in our example. The global mobility of operations determines the minimum number of control steps that each basic block must have. For instance, block $B_1$ must accommodate operations $OP1$ and $OP15$. Other operations such as $OP_2$, $OP3$, and $OP5$ may be scheduled into block $B_1$ but it is not mandatory.

With global mobility, we then attempt to minimize the control steps in frequently executed blocks. The basic strategy is to move each mobile operation from a frequently executed block to a less frequently executed block unless the mobile operation does not increase the control steps of the original containing block, i.e. the mobile operation can be scheduled for parallel execution with an operation that is mandatorily placed in the block. This strategy is further elaborated in the following:

- *Minimize the control steps in the if-blocks:* An if-block has larger execution probability than its branch parts. In order to achieve maximal speedup, we should move as many operations to the branch parts as possible. Based on the discussion in Section 2, we know downward movement primitives tend to move operations to the branch parts. Hence, in order to minimize the control steps in an if-block, we perform GALAP first. Then, the operations in the branch parts are moved upward to the if-block only if they will not increase the number of control steps in the if-block.
- *Minimize the control steps in the inner loops:* Since inner loops are most frequently executed, we should not schedule a mobile operation into an inner loop unless the number of control steps will not increase. Therefore, all the loop invariants should be moved upward to the pre-header before we schedule the loop body. After the loop body is scheduled, the loop invariants can be moved into the loop again if they do not require extra control steps.

Table 1
Mobility of operations in the example

| Operation | Global mobility | Operation | Global mobility |
|---|---|---|---|
| $OP1$ | $B_1$ | $OP9$ | $B_2, B_5$ |
| $OP2$ | $B_1$, pre-header | $OP10$ | $B_2, B_4$ |
| $OP3$ | $B_1, B_7$ | $OP11$ | $B_2$ |
| $OP4$ | $B_5$ | $OP12$ | $B_3$ |
| $OP5$ | $B_1$, pre-header, $B_2$ | $OP13$ | $B_4$ |
| $OP6$ | $B_2$ | $OP14$ | $B_7$ |
| $OP7$ | $B_2, B_5$ | $OP15$ | $B_1$ |
| $OP8$ | $B_2, B_5$ | | |

## 4. The global scheduling algorithm

In this section, we will propose a new global scheduling algorithm based on the global mobility of operations. Our goal is to achieve maximum speedup and minimum size of control code at the same time.

A pseudo code of the global scheduling algorithm is given in Fig. 7. With the output of GALAP, the algorithm starts from the inner-most loop and

```
Procedure Global_Scheduling()
{
   for each loop L (from inner to outer)
      {
         □ detect the loop invariants and put them in the pre-header;
         □ call the procedure Schedule_Nested-ifs to schedule the loop L;
           (Top-down scheduling the loop L)
         □ call the procedure Re_Schedule to reschedule the loop L;
           (Bottom-up rescheduling the loop L)
         □ treat the loop body L as a supernode;
      }
   call the procedure Schedule_Nested-ifs to schedule the flow graph.
}
```

Fig. 7. Global scheduling algorithm.

proceeds outward. Note that the GALAP moves every operation downward to the inner-most loop in which it can stay. The loop invariants must be moved upward to the pre-header before we schedule the loop body. Once the scheduling of an inner loop is completed, it will be treated as a supernode when the scheduler goes outward.

The scheduling of a loop consists of two phases. In the first phase (top-down scheduling), procedure *Schedule_Nested-ifs* works to schedule the loop body. Section 4.1 below details the operation of procedure *Schedule_Nested-ifs*. In the second phase (bottom-up rescheduling), procedure *Re_Schedule* works to schedule as many loop invariants as possible in the loop body without increasing the number of control steps. Section 4.2 below details the operation of procedure *Re_Schedule*.

Let us use the behavior description in Fig. 2(a) as an example to illustrate the global scheduling algorithm. The output of GALAP is shown in Fig. 6. With the output of GALAP, the algorithm starts from the loop containing blocks $B_2$, $B_3$, $B_4$, and $B_5$. Firstly, loop invariant *OP5* is moved from block $B_2$ to the pre-header. Procedure Schedule_Nested-ifs then schedules the loop body. Next, procedure Re_Schedule reschedules *OP5* into the loop again. After the loop is scheduled, it is treated as a supernode when we move outward and call procedure Schedule_Nested-ifs to schedule blocks $B_1$, pre-header, $B_6$, and $B_7$.

### 4.1 Top-down scheduling of a loop

The pseudo code of procedure Schedule_Nested-ifs is outlined in Fig. 8. The blocks are processed in the increasing order of their *ID* numbers (top-down manner). Due to the GALAP process, every operation has been downward moved to the

```
Procedure Schedule_Nested-ifs()
{
    for each block in increasing ID number
    {
        **Backward List Scheduling**
        for control step C_step = S_last downto 1
            • try to assign "must" operations to the step C_step

        **Forward List Scheduling **
        for control step C_step = 1 to S_last
        {
            • schedule all critical "must" operations to the step C_step
            • try to schedule "may" operations to the step C_step
            • try to schedule non-critical "must" operations to the step C_step
            • duplication or renaming if necessary
        }
    }
}
```

Fig. 8. The procedure Schedule_Nested-ifs.

latest block in which the operation can be scheduled. Therefore, in the output of GALAP, every operation is called a 'must' operation for the block it belongs to because we cannot schedule the operation in a later block. The operation that can be moved *upward* to a block $B$ is called a 'may' operation for block $B$. The idea behind the scheduling algorithm is to schedule each block with the minimal number of control steps, which is determined by the height of the data dependence graph of the 'must' operations in the block. Meanwhile, we try to schedule as many 'may' operations into a block as long as the number of control steps does not increase.

The scheduling of a block consists of two phases. In the first phase, a *backward list scheduling* is performed to determine the deadline for each "must" operation and the minimum number of control steps of the block. In the second phase, a *forward list scheduling* is performed to include as many 'may' operations as possible under the constraint that the number of control steps does not increase. The operations of these two phases are elaborated in the following.

#### 4.1.1 Backward list scheduling phase

In the backward list scheduling phase, only the 'must' operations in a block are taken into consideration. The 'must' operations are assigned to control steps by a backward (or bottom-up) list scheduling. The backward list scheduling determines the latest step $BLS(o_i)$ that a 'must' operation $o_i$ must be assigned to and the minimal number of the control steps of the block.

#### 4.1.2 Forward list scheduling phase

The forward list scheduling phase reschedules a block by a forward list scheduling. The goal is to include as many 'may' operations as possible into a block without increasing the control steps of the block. The goal can be achieved by scheduling each 'must' operation $o_i$ no later than the control step $BLS(o_i)$. Because backward list scheduling assigns 'must' operations to control steps as late as possible, a 'must' operation $o_i$ scheduled no later than the control step $BLS(o_i)$ in the forward list scheduling phase will not increase the control steps of the block.

While we are scheduling operations to a control step $C_{step}$, a 'must' operation $o_i$ becomes a *critical* 'must' operation if $C_{step} = BLS(o_i)$. In event resource conflict occurs, the following priority is applied to resolve the problem:

(1st) critical 'must' operations
(2nd) 'may' operations, and
(3rd) non-critical 'must' operations.

Because the critical 'must' operations have the highest priority, the number of control steps will not increase. This method will maximize resource utilization without increasing the control steps of a block. As more 'may' operations are moved upward, the number of 'must' operations of later blocks are reduced. If there are unused resources left in a step, we will invoke transformations *duplication* and *renaming* to make use of them.

*Duplication:* Duplication moves an operation from the joint part to the true part and the false part at the same time. In general, any operation in the joint part can be duplicated and moved to branch parts if no data precedence constraint is violated. Moreover, for a nested-ifs structure, an operation may be duplicated many times. However, in order to avoid unlimited increase of control space, we may want to limit the number of times by which an operation can be duplicated.

*Renaming:* A ready operation $o_i$ cannot be moved from $B_{true}$ to $B_{if}$ if $d(o_i) \in in[B_{false}]$. However, if we rename the variable $d(o_i)$ as $d(o_i)'$, the renamed operation becomes movable. An assignment operation $d(o_i) = d(o_i)'$ must be added to $B_{true}$ in order to preserve the semantics. Because an assignment operation has shorter execution delay and uses less resources, the scheduling result of block $B_{true}$ can be better.

*4.1.3 Time complexity*

Let $n$ denote the number of operations and $b$ denote the number of blocks. Since the length of a ready queue is limited by $O(n)$, the search for a candidate operation to be scheduled is $O(n)$. The time taken to carry out book-keeping after an operation is scheduled is at most $O(b)$. Therefore, the time complexity for scheduling an operation is $O(n + b)$. And, the time complexity of the Schedule_Nested-ifs procedure is $O(n^2 + nb)$.

*4.2. Bottom-up rescheduling of a loop*

For a loop body, procedure Schedule_Nested-ifs determines the number of control steps for each block and assign each operation $o_i$ to a control step $Step(o_i)$. Upon the completion of Schedule_Nested-ifs, all the loop invariants are moved to the preheader. What procedure Re_Schedule tries to do after Schedule_Nested-ifs is done is to move as many loop invariants as possible back to the loop body under the constraint that the number of control steps does not increase. The pseudo code of procedure Re_Schedule is outlined in Fig. 9. In Re_Schedule, blocks are processed in a decreasing order of their $ID$ numbers (bottom-up manner) and the process starts from the last control step and proceeds upward to the first control step. Similar to that in forward list scheduling, an operation $o_i$ becomes a *critical* operation of a control step $C_{step}$ if $C_{step} = Step(o_i)$. In event resource conflict occurs, operations are assigned to control steps according to the following priority:

(1st) 'critical' operations of the control step,
(2nd) loop invariants, and
(3rd) the other operations of the loop.

Once the scheduling of an inner loop is done, it is treated as a supernode when the scheduling procedure moves to the outer loops.

*4.3 Example of global scheduling*

Let us use the program in Fig. 2(a) as an example to illustrate the global scheduling algorithm. The global mobility of operations is shown in Table 1. Assume we are given two ALUs as the resource constraint.

The output of GALAP is shown in Fig. 6. With the output of GALAP, our algorithm starts from the loop that contains blocks $B_2$, $B_3$, $B_4$, and $B_5$.

```
Procedure Re_Schedule()
{
    for each block in decreasing ID number
    {
        for control step C_step = S_last downto 1
        {
            • schedule all critical operations to the step C_step
            • try to schedule loop invariants to the step C_step
            • try to schedule non-critical operations to the step C_step
        }
    }
}
```

Fig. 9. The procedure Re_Schedule.

Firstly, loop invariant $OP5$: $c = i2 + 1$ is moved to the pre-header. The flow graph is shown in Fig. 10(a). Then, procedure Schedule_Nested-ifs is invoked to schedule the loop body. The procedure processes the blocks according to the following sequence $B_2$, $B_3$, $B_4$, and $B_5$.

When procedure Schedule_Nested-ifs processes block $B_2$, there are only two operations

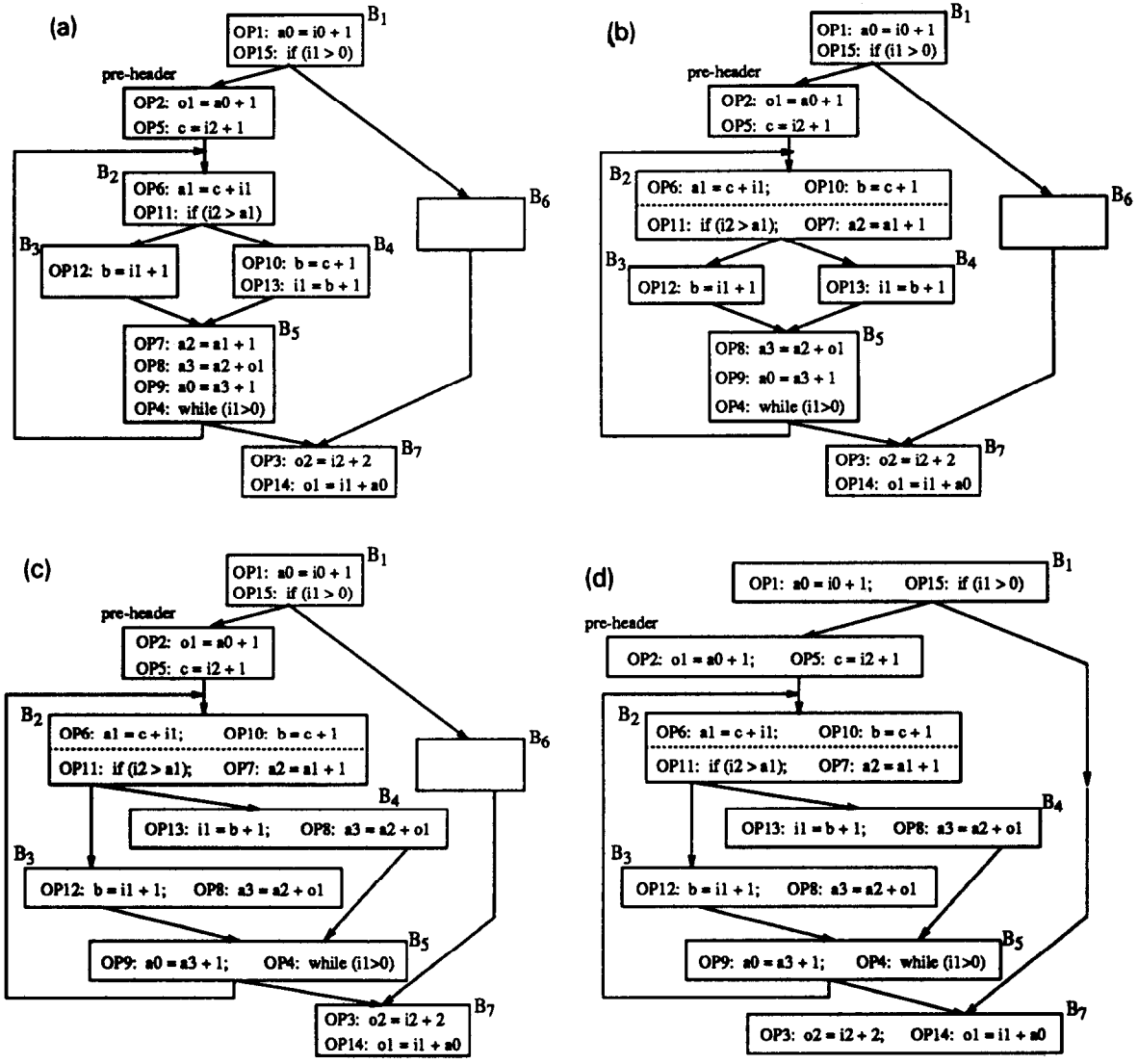



Fig. 10. Snapshots of our global scheduling algorithm on the example.

$OP6$: $a1 = c + i1$ and $OP11$:*if*$(i2 > a1)$ left in the block since $OP5$ has been moved to the pre-header. The two operations are the 'must' operations of block $B_2$. The backward list scheduling schedules the two 'must' operations in two control steps. Operation $OP11$ is scheduled in the control step 2 and operation $OP6$ is scheduled in the control step 1. After the backward list scheduling completes its task and determines that the minimal number of control steps for block $B_2$ is 2, the forward list scheduling is invoked. Because operation $OP6$ is a critical 'must' operation for control step 1, it must be placed here. However, the hardware resources are not fully utilized at control step 1. We can place one more operation here. According to global mobility information, operation $OP10$: $b = c + 1$ is a 'may' operation for block $B_2$ and it is ready. Therefore, we move operation $OP10$ upward from block $B_4$ to block $B_2$ and place it in control step 1. Now, we move to control step 2. Since operation $OP11$ is a critical 'must' operation for control step 2, its location is sealed. Then, according to global mobility information, operation $OP7$: $a2 = a1 + 1$ is a 'may' operation for block $B_2$ and can be scheduled for parallel execution with $OP11$. Hence, it is moved upward and placed in the second slot in control step 2. At this point, Schedule_Nested-ifs has finished processing block $B_2$. The resulting flow graph is shown in Fig. 10(b).

Now, Schedule_Nested-ifs moves to block $B_3$. In block $B_3$, there is only one operation $OP12$: $b = i1 + 1$. Operation $OP12$ is a 'must' operation for block $B_3$. Hence, the backward list scheduling assigns it to control step 1. When the forward list scheduling processes block $B_3$, it finds that operation $OP12$ is a critical 'must' operation. Hence, the forward list scheduling places $OP12$ in control step 1 and searches for another operation to fill the second slot. At this point, operation $OP8$: $a3 = a2 + o1$ in block $B_5$ is ready but it is not a 'may' operation for the block $B_3$. One way to resolve this problem is to duplicate $OP8$ and place one copy to block $B_3$ and another copy to block $B_4$. As a result, Schedule_Nested-ifs completes the scheduling of block $B_3$ with one control step and invokes one operation duplication.

Similarly, procedure Schedule_Nested-ifs processes blocks $B_4$ and $B_5$ and completes the first phase (top-down) of loop scheduling with the flow graph shown in Fig. 10(c).

Next, the second phase (bottom-up) of loop scheduling is performed. Procedure Re_Schedule is called to reschedule the loop. The procedure processes the blocks according to the following sequence $B_5$, $B_4$, and $B_2$. We try to move loop invariant $OP5$ into the loop. However, because the resources have been fully utilized in each control step, operation $OP5$ cannot be included in the loop.

After the loop is scheduled, it is treated as a supernode. The scheduling of the loop will never be changed again. Then, the algorithm goes outward to schedule the flow graph. Procedure Schedule_Nested-ifs is called to process the blocks according to the following sequence $B_1$, pre-header, $B_6$ and $B_7$.

Let us examine how procedure Schedule_Nested-ifs processes block $B_1$. There are two operations $OP1$: $a0 = i0 + 1$ and $OP15$: *if*$(i1 > 0)$ in block $B_1$. Both are 'must' operations for the block. The backward list scheduling assigns the two operations to the same control step because these two operations can be executed concurrently. Then, the forward list scheduling is performed. The two operations $OP1$ and $OP15$ are scheduled to the control step 1 because they are critical 'must' operations. Since not resource is available, no 'may' operation can be moved upward to the block. The scheduling of block $B_1$ is finished with one control step.

Similarly, procedure Schedule_Nested-ifs processes blocks pre-header, $B_6$ and $B_7$ and completes its task with the final result shown in Fig. 10(d). The algorithm schedules the program with 8 control words. Since operation $OP8$ is duplicated during

scheduling, there are 16 operations in the final result. Because the hardware parallelism is 2, the size of control code is minimal. Furthermore, each iteration of the inner loop only takes 4 control steps. It is easy to show that the final scheduling for the loop is optimal.

## 5. Experiments

The GSSP algorithm has been implemented in C language on a SUN 4/40 workstation and has been integrated into a high-level synthesis system. This section reports the experimental results on five real programs. These five programs are Roots [5], Linear Predictive Coding (LPC) [6], Knapsack [7], MAHA's example [8], and Wakabayashi's example [9]. The characteristics of these five benchmark programs are summarized in Table 2. In the first three examples, we compare GSSP with Trace Scheduling (TS) and Tree Compaction (TC). In the last two examples, GSSP is compared against MAHA [8], the conditional resource sharing in [9], the path-based approach [10], and the approach in [11].

Table 2
Summary of test programs

| Programs | #block | #if | #loop | #op | #op/block |
|---|---|---|---|---|---|
| Roots | 10 | 3 | 0 | 22 | 2.2 |
| LPC | 19 | 6 | 5 | 63 | 3.32 |
| Knapsack | 34 | 11 | 6 | 84 | 2.47 |
| MAHA | 19 | 6 | 0 | 22 | 1.1 |
| Wakabayashi | 7 | 2 | 0 | 16 | 2.3 |

### 5.1 Result on roots

Program 'Roots' computes the roots of a second order equation. This example is taken from [5], where it is used to illustrate the trace scheduling algorithm. It consists of several branches. We follow the assumption in [5] that each operation takes one cycle. The critical path in this example is the trace with the highest execution probability.

Table 3 tabulates the results with constraints on the number of ALUs (#alu), the number of multipliers (#mul), and the number of latches (#latch). The results include the comparison of (i) the total number of control words and (ii) the number of control steps in the critical path. As shown in Table 3, GSSP consistently gives better results than Trace Scheduling and Tree Compaction in terms of the total number of control words and the number of control steps in the critical path. Tree compaction is designed to reduce the control overhead of Trace Scheduling by limiting the range of scheduling. The results show that it does use less control words but suffers having a longer critical path.

Table 3
Results of roots

| | | | # of control words | | | # of control steps in the critical path | | |
|---|---|---|---|---|---|---|---|---|
| #alu | #mul | #latch | GSSP | TS | TC | GSSP | TS | TC |
| 1 | 1 | 1 | 11 | 14 | 13 | 9 | 11 | 11 |
| 1 | 2 | 1 | 10 | 14 | 13 | 8 | 9 | 10 |
| 2 | 1 | 1 | 10 | 12 | 12 | 8 | 11 | 11 |
| 2 | 2 | 1 | 8 | 12 | 12 | 6 | 9 | 10 |

### 5.2 Results on LPC and Knapsack

Programs LPC and Knapsack contain several loops. Because the inner loops contain only straight line codes, they can be perfectly optimized by GSSP, Trace Scheduling, and Tree Compaction. Therefore, we only compare the number of control words. Table 4 and Table 5 show the results of LPC and Knapsack, respectively. The results are derived under constraints on the number of multipliers (#mul), the number of comparators (#cmpr), the number of ALUs (#alu) and the number of latches (#latch) and with the assumption that multiplication takes two clock cycles to complete. The results show that GSSP achieves the best result among the three and Tree Compaction is better than Trace Scheduling.

Table 4
Results of LPC

| | | | | # of control words | | |
|---|---|---|---|---|---|---|
| #mul | #cmpr | #alu | #latch | GSSP | TS | TC |
| 1 | 1 | 1 | 1 | 52 | 71 | 69 |
| 1 | 1 | 1 | 2 | 52 | 71 | 69 |
| 1 | 1 | 2 | 1 | 50 | 69 | 66 |
| 1 | 1 | 2 | 2 | 50 | 69 | 66 |

Table 5
Results of Knapsack

| | | | | # of control words | | |
|---|---|---|---|---|---|---|
| #mul | #cmpr | #alu | #latch | GSSP | TS | TC |
| 1 | 1 | 1 | 1 | 63 | 74 | 69 |
| 1 | 1 | 2 | 1 | 60 | 73 | 68 |
| 1 | 1 | 1 | 2 | 55 | 66 | 63 |
| 1 | 1 | 2 | 2 | 52 | 63 | 60 |

### 5.3 Results on MAHA and Wakabayashi's examples

In the high-level synthesis of a digital system, it is very common to make the result of an operation forwarded to its successor on the same control step. Usually, a finite-state-machine is used to control the operations and data transfers. Several high-level synthesis algorithms, e.g. MAHA [8], Cyber [9], path-based approach [10], and [11], have been proposed for handling behavioral description

Table 6
Results of MAHA's example

| | | | | #C_STEP | | | |
|---|---|---|---|---|---|---|---|
| | #add | #sub | cn | states | long | short | avg |
| GSSP | 1 | 1 | 1 | 6 | 6 | 2 | 3.5 |
| | 1 | 1 | 2 | 5 | 5 | 2 | 3.375 |
| | 2 | 3 | 3 | 3 | 3 | 1 | 1.3125 |
| [11] | 1 | 1 | 1 | 8 | 8 | 3 | — |
| | 1 | 1 | 2 | 6 | 5 | 2 | — |
| | 2 | 3 | 3 | 3 | 3 | 2 | — |
| Path [10] | 1 | 1 | 2 | 9 | 5 | 2 | — |
| | 2 | 3 | 5 | 4 | 3 | 1 | — |
| MAHA [8] | 1 | 1 | 2 | 8 | 8 | — | — |
| | 2 | 3 | 3 | 4 | 4 | — | — |

Table 7
Results of Wakabayashi's example

| | #alu | #add | #sub | cn | #C_STEP states | #1 | #2 | #3 | avg |
|---|---|---|---|---|---|---|---|---|---|
| GSSP | 0 | 1 | 1 | 1 | 7 | 7 | 4 | 4 | 4.75 |
| | 0 | 1 | 1 | 2 | 7 | 7 | 4 | 3 | 4.25 |
| | 2 | 0 | 0 | 2 | 6 | 6 | 4 | 3 | 4.00 |
| [11] | 0 | 1 | 1 | 1 | 7 | 7 | 4 | 3 | 4.75 |
| | 0 | 1 | 1 | 2 | 7 | 7 | 4 | 3 | 4.25 |
| | 2 | 0 | 0 | 2 | 6 | 6 | 5 | 3 | 4.25 |
| Path [10] | 0 | 1 | 1 | 2 | 8 | 7 | 6 | 3 | 4.75 |
| | 2 | 0 | 0 | 2 | 6 | 6 | 5 | 3 | 4.25 |
| Cyber [9] | 0 | 1 | 1 | 1 | 7 | 7 | 5 | 4 | 5.00 |

containing several branches. We obtain the results on MAHA and Wakabayashi's examples by first applying GSSP to partition the operations into states and then using the global slicing technique [12] to merge the mutually exclusive states on the branch parts.

Under the constraints on the number of adders (#add), the number of subtracters (#sub), and the number of operations that can be chained together in a control step (cn), Table 6 compares the results of various approaches on the MAHA's example. There are 12 execution paths in the MAHA's example. The 'long' and 'short' in Table 6 denote the number of control steps of the longest and shortest paths, respectively, and 'avg' is the average control steps of all paths. The results show GSSP requires fewest states in the finite state machine and optimizes each path with fewest control steps.

Table 7 compares the results of various approaches on the Wakabayashi's example. There are three paths (denoted as #1, #2 and #3) in the Wakabayashi's example. Again, GSSP requires fewest states in the finite state machine and optimizes each path with fewest control steps.

## 6. Conclusion

In this paper, we propose a new global scheduling algorithm called GSSP (Global Scheduling for Structured Programs) for automatic synthesis of the control blocks of special-purpose microprocessors starting from behavior-level descriptions. The main distinction of the GSSP algorithm is that it exploits the inheritances of structured programs. The GSSP algorithm works based on several interesting observations about the mobility of operations in the flow diagrams derived from structured programs. The optimization goal is to maximize the speedup of the processor and minimize the size of the control block. As reported in Section 5, GSSP consistently achieves better results in terms of the speedup of the processor and the size of the control block when compared with existing scheduling algorithms.

## References

[1] M.C. McFarland, A.C. Parker and R. Camposano, The high-level synthesis of digital system, *Proc. IEEE* (Feb. 1990) 301–318.

[2] J.A. Fisher, Trace scheduling: A technique for global microcode compaction, *IEEE Trans. Comput.* (June 1981) 478–490.

[3] J. Lah and D.E. Atkins, Tree compaction of microprograms, *Proc. 16th Annual Microprogramming Workshop*, (Oct. 1983) 23–33.

[4] A. Nicolau, Uniform parallelism exploitation in ordinary programs, *Proc. Int. Conf. on Parallel Processing* (Aug. 1985) 614–618.

[5] F. Gasperroni, *Compilation Techniques for VLIW Architectures* (Courant Institute of Mathematical Science, New York University, March 1989).

[6] M.M. Jamali, P. Bumrungthum and N. Mohankrishnan, A parallel algorithm for linear predictive coding, *Signal Processing IV: Theories and Applications* (1988) 759–762.

[7] E. Horowitz and S. Sahni, *Fundamentals of Computer Algorithms* (Computer Science Press, 1978) 355.

[8] A.C. Parker, J. Pizarro and M.J. Mlinarr, MAHA: A program for data path synthesis, *Proc. 23rd Design Automation Conf.* (July 1986) 461–466.

[9] K. Wakabayashi and T. Yoshimura, A resource sharing and control synthesis method for conditional branches, *Proc. Int. Conf. on Computer-Aided Design* (Nov. 1989) 62–65.

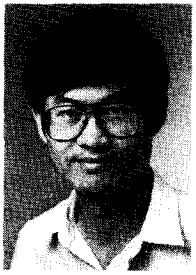
[10] R. Camposano and R.A. Bergamasch, Synthesis using path-based scheduling: Algorithms and exercises, *Proc. 27th Design Automation Conf.* (June 1990) 450–455.

[11] T. Kim, J. Liu and C.L. Liu, A scheduling algorithm for conditional resource sharing, *Proc. Int. Conf. on Computer-Aided Design* (Nov. 1991) 84–87.

[12] C.J. Tseng, Bridge: A versatile behavioral synthesis system, *Proc. 25th Design Automation Conf.* (June 1989) 415–420.

**Shih-Hsu Huang** received the B.S. degree in computer science and information engineering from National Chiao Tung University, Hsinchu, Taiwan, in 1989, and the M.S. degree in computer science from National Tsing Hua University, Hsinchu, Taiwan, in 1991. He is currently a Ph.D. student in the Department of Computer Science and Information Engineering at National Taiwan University, Taipei, Taiwan. His research interests include instruction scheduling and VLSI synthesis. He can be reached by e-mail at huang@solar.csie.ntu.edu.tw

**Cheng-Tsung Hwang** received the B.S. degree in computer science form Tung Hai University, Taichung, Taiwan, in 1987, and the Ph.D. degree in computer science from National Tsing Hua University, Hsinchu, Taiwan, in 1992. He is currently serving as Associate Professor of Computer Science at Providence University, Taichung, Taiwan. From 1991 to 1992, he has been doing postgraduate research in the Department of Computer Science at University of California, Riverside. His research interests include silicon compilation and optimization in VLSI design. He can be reached by e-mail at jthuang@cs.nthu.edu.tw



**Yu-Chin Hsu** received the B.S. degree in computer science from National Taiwan University, Taipei, Taiwan, in 1981, and the M.S. and Ph.D. degrees in computer science from the University of Illinois at Urban-Champaign in 1986 and 1987, respectively. He is currently serving as Associate Professor of Computer Science at University of California, Riverside. Previously, Dr. Hsu served on the faculty of Tsing Hua University, Hsinchu, Taiwan. His research interests include automated synthesis of digital systems from VHDL description. He co-received an Outstanding Young Author Award from IEEE Circuits and Systems Society in 1990. He can be reached by e-mail at hsu@cs.ucr.edu

**Yen-Jen Oyang** received the B.S. degree in information engineering from National Taiwan University in 1982, the M.S. degree in computer science from California Institute of Technology in 1984, and the Ph.D. degree in electrical engineering from Standford University in 1988. He is currently an Associate Professor in the Department of Computer Science and Information Engineering, National Taiwan University. His research interests include computer architecture, distributed systems, and VLSI system design. He can be reached by e-mail at yjoyang@csie.ntu.edu.tw