Available online at www.sciencedirect.com

SCIENCE DIRECT®

ELSEVIER

The Journal of Systems and Software 75 (2005) 155–170

The Journal of Systems and Software

www.elsevier.com/locate/jss

A two-level scheduling method: an effective parallelizing technique for uniform nested loops on a DSP multiprocessor

Yi-Hsuan Lee, Cheng Chen *

Department of Computer Science and Information Engineering, National Chiao Tung University, 1001 Ta Hsueh Road, Hsinchu, Taiwan 30050, PR China

Received 24 June 2002; received in revised form 22 January 2003; accepted 10 February 2003
Available online 12 May 2004

Abstract

A *digital signal processor* (DSP), which is a special-purpose microprocessor, is designed to achieve higher performance on DSP applications. Because most DSP applications contain many nested loops and permit a very high degree of parallelism, the DSP multiprocessor has a suitable architecture to execute these applications. Unfortunately, conventional scheduling methods used on DSP multiprocessors allocate only one operation to each DSP every time unit, even if the DSP includes several function units that can operate in parallel. Obviously they cannot achieve full function unit utilization. Hence, in this paper, we propose a *two-level scheduling method* (*TSM*) to overcome this common failing. *TSM* contains two approaches, which integrates *unimodular transformations*, *loop tiling* technique, and conventional methods used on single DSP. Besides introducing algorithm, we also use an analytic module to analyze its preliminary performance. Based on our analyses the *TSM* can achieve shorter execution time and more scalable speedup results. In addition, the *TSM* causes less memory access and synchronization overheads, which are usually negligible in the DSP multiprocessor architecture.
© 2004 Elsevier Inc. All rights reserved.

Keywords: DSP multiprocessor; Scheduling; Uniform nested loop; Parallelize

1. Introduction

Most scientific and digital signal processing applications, such as image processing, weather forecasting, and fluid dynamics, are recursive or iterative (Hsu and Jeang, 1993; Kung, 1988). These applications are usually represented by nested loops, and most of their operations are multiplications and additions (Madisetti, 1995). The DSP is a special-purpose microprocessor, which is designed to achieve high performance on DSP applications with minimum silicon cost. Unlike general-purpose microprocessors, the DSP design is based on the Harvard architecture, and often includes several independent function units those are capable of operating in parallel (Eyre and Bier, 2000; Simar, 1998).

Because nested loops are the time-critical sections in such computation-intensive applications, their execution time will dominate the entire computational performance. To optimize the execution rate of such applications we need to explore the embedded parallelism of a loop (Passos et al., 1995; Passos and Sha, 1996). Conventional scheduling methods are divided into five categories; the transformation based technique is the most popular one (Parhi, 1999). Methods belonging to this category usually focus on scheduling multiplications and additions, and use *retiming* and *unfolding* techniques to restructure loop bodies with higher embedded parallelism (Chao and Sha, 1993; Leiserson and Saxe, 1991). They usually can obtain rate-optimal results, and require less time and space complexity.

Most DSP applications permit a very high degree of parallelism that can be exploited with *digital signal multiprocessor* (*DSMP*) systems (Madisetti, 1995). Many related scheduling methods for DSMPs have been proposed, but they only allow each DSP to execute one operation every time unit (Jeng and Chen, 1994; Koch

* Corresponding author. Tel.: +886-3-5712121x54734; fax: +886-3-5724176.
E-mail addresses: yslee@csie.nctu.edu.tw (Y.-H. Lee), cchen@csie.nctu.edu.tw (C. Chen).

0164-1212/$ - see front matter © 2004 Elsevier Inc. All rights reserved.
doi:10.1016/j.jss.2003.02.001

et al., 1997; Shatnawi et al., 1999). However, because a DSP often can execute several operations in parallel, these methods obviously cannot achieve full function unit utilization. In this paper, we propose a scheduling method that allocates more than one operation to a DSP in each time unit, thereby overcoming this limitation.

Our method, named *two-level scheduling method* (TSM), is motivated by the *skewed single instruction multiple data* (*SSIMD*) method (Barnwell et al., 1978). It contains two approaches. The first, directly called the *two-level scheduling method* (*TSM*), is integrated with *unimodular transformations* (Wolf and Lam, 1991) and the conventional single DSP scheduling method. The second is extended from *TSM* by adding *loop tiling* technique (Wolfe, 1996), which we name *two-level scheduling method with loop tiling* (*TSMLT*). Besides introducing the algorithm and principles, we use an analytical model to analyze their preliminary performances. From our analyses, both *TSM* and *TSMLT* can achieve higher functional unit utilization and shorter execution times than conventional methods. Their speedup results are also more scalable when the number of processing environments increases. Moreover, our methods cause less synchronization and memory access overheads, although they seem minor in DSMP architecture.

The remainder of this paper is organized as follows. Section 2 surveys the fundamentals and the background. Design issues, analytical modules and preliminary performance analyses of *TSM* and *TSMLT* are described in Sections 3 and 4, respectively. Finally, the conclusions and future work are presented in Section 5.

2. Fundamentals and background

In this section, we will model our scheduling problem and briefly survey some fundamentals. Loop transformation technique will be also introduced.

2.1. Modeling the problem (Passos and Sha, 1996, 1998)

Because DSP applications usually contain repetitive groups of operations, they can be easily represented by nested loops. Edwin Sha et al. model the nested loop as a *multi-dimensional data-flow graph* (*MDFG*) (Passos and Sha, 1996; Passos and Sha, 1998). An MDFG is defined as follows.

Definition 2.1. An MDFG $G = (V, E, d, t)$ is a node-weighted and edge-weighted directed graph, where V is the set of computation nodes, E is the set of dependency edges, d is a function from E to Z^n representing the multi-dimensional delays between the two nodes, where n is the number of dimensions, and t is a function from V to the positive integers, representing the computation time of each node.

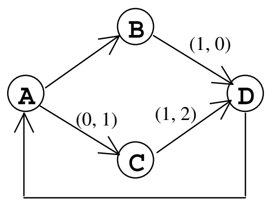
Fig. 1 shows a simple nested loop and its corresponding MDFG. An MDFG is *realizable* if there exists a schedule vector s such that $s * d \geqslant 0$, where d are loop-carried dependencies in G. A *schedule vector s* is the normal vector for a set of parallel equitemporal hyperplanes that define a sequence of execution (Lamport, 1974). An *iteration* is equivalent to the execution of each node in V exactly once. The period during which all nodes in an iteration are executed, according to data dependencies and without resource constraints, is called a *cycle period.* Cycle period is the maximum computational time among paths that have no delay (Passos and Sha, 1996; Passos and Sha, 1998). It is shown that the cycle period dominates the entire execution time of a nested loop. Notices that many MDFGs can represent a signal processing algorithm, depending on how we represent it by nested loops.

2.2. Retiming an MDFG (Leiserson and Saxe, 1991)

Retiming is a popular technique that reassigns delays to enhance the execution performance (Leiserson and Saxe, 1991). A *multi-dimensional retiming r* is a function from V to Z^n that redistributes nodes in consecutive iterations. A new MDFG $G_r = (V, E, d_r, t)$ is created after applying r such that each iteration still has one execution of each node. The *retiming vector* $r(u)$ of node u represents the offset between the original iteration and that after retiming. The delay vectors change accord-

```
for $i = 1$ to $m$ begin
    for $j = 1$ to $n$ begin
        $D[i, j] = B[i-1, j] \times C[i-1, j-2]$ ;
        $A[i, j] = D[i, j] \times 0.5$ ;
        $B[i, j] = A[i, j] + 1$ ;
        $C[i, j] = A[i, j-1] + 2$ ;
    end
end
        $D = \{(1, 0), (0, 1), (1, 2)\}$
```

(a)



(b)

Fig. 1. A simple nested loop and its corresponding MDFG.

ingly to preserve dependencies. Definitions and properties of retiming are shown below.

Definition 2.2. Given any MDFG $G = (V, E, d, t)$, retiming function r, and retimed MDFG $G_r = (V, E, d_r, t)$, we define the retimed delay vector for every edge, path, and cycle, respectively, by

(a) $d_r(e) = d(e) + r(u) - r(v)$ for every edge $u \xrightarrow{e} v, u, v \in V$ and $e \in E$.

(b) $d_r(p) = d(p) + r(u) - r(v)$ for every path $u \xrightarrow{p}, u, v \in V$ and $p \in G$.

(c) $d_r(l) = d(l)$ for any cycle $l \in G$.

Fig. 2 shows the retimed nested loop and MDFG in Fig. 1. A *prologue* is the set of instructions moved in each dimension that must be executed to provide necessary data for the iterative process. An *epilogue* is the complementary instruction set that will be executed to complete the process. If the nested loop contains sufficient iterations, the time required to run prologue and epilogue are negligible.

2.3. Loop transformations (Wolf and Lam, 1991; Wolfe, 1996)

Loop transformation is one of the basic techniques for parallel compiler design. It changes the execution sequence of the iterations to achieve the maximum degree of parallel operation for the program's execution in the multi-processor system. Many transformation techniques have been proposed. Among these, the *unimodular transformations* technique is one of the most important techniques (Wolf and Lam, 1991). In its model, a nested loop of depth n is represented as a finite convex polyhedron in the iteration space Z^n bounded by the loop bounds. Each iteration in the loop corresponds to a node in the polyhedron, and is identified by its index vector $[p_1 p_2 \cdots p_n^T]$. The loop-carried data dependency can be succinctly represented by the dependency vector $[d_1 d_2 \cdots d_n^T]$.

Loop tiling is a technique that decomposes a single loop into two nested loops; the outer loop steps between strips of consecutive iterations, and the inner loop steps between single iterations within a strip (Wolfe, 1996). Because this technique changes the execution sequence of the iterations, it can be used to increase data locality if the nested loop contains many iterations.

3. Two-level scheduling method

TSM contains two individual approaches that will be described in this and next sections. Because nested loops used in DSP applications are usually with depth of two, we use it as an example to explain our method. Nevertheless, *TSM* can be easily extended to cover nested loops with depths greater than two.

3.1. Problem definition

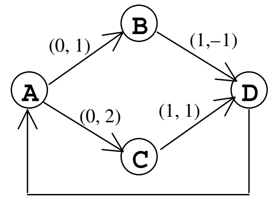
At first, the scheduling problem and system architecture are defined. We use a two-dimensional MDFG to model a nested loop with depth two, and execute it on DSMP architecture. Scheduling goals are to reduce the entire execution time and fully utilize function units. We also inherit architectural constraints from DSMP systems, and use these features to analyze our approaches (Madisetti, 1995).

3.2. Main stages of two-level scheduling method

The first approach is named the *two-level scheduling method* (*TSM*). Given a nested loop with depth two, the process of *TSM* contains two main stages. The first

```
for i = 1 to m begin
    D [i, 1] = B [i–1, 1] × C [i–1, –1] ;
    A [i, 1] = D [i, 1] × 0.5 ;
    for j = 1 to n–1 begin
        D [i, j+1] = B [i–1, j+1] × C [i–1, j–1] ;
        A [i, j+1] = D [i, j+1] × 0.5 ;
        B [i, j] = A [i, j] + 1 ;
        C [i, j] = A [i, j–1] + 2 ;
    end
    B [i, n] = A [i, n] + 1 ;
    C [i, n] = A [i, n–1] + 2 ;
end
```

(a)



(b)

Fig. 2. The retimed nested loop and MDFG of those in Fig. 1.

stage is to parallelize and allocate iterations to every DSP, and the second stage is to achieve the scheduling goals for each DSP. We explain these stages as follows.

In the first stage, we use unimodular transformations to parallelize the inner loop. Since this technique does not explain how to use its transformations, we design a simple algorithm *LP* listed in Fig. 3 to parallelize a uniform nested loop of depth two. WLOG, we only consider flow-dependencies. Notices that the transformation matrix to parallelize a nested loop is not unique, and algorithm *LP* can obtain one with minimum skew factor. Fig. 4 shows the parallelizing results in Fig. 1.

Then, iterations should be allocated to every DSP. Because the inner loop is parallelizable, iterations can be divided into *barrier sections*, where iterations within the same barrier section are independent. Clearly barrier sections must be executed in sequence to preserve dependencies. For reducing the entire execution time, iterations in a barrier section are evenly allocated. Suppose there are four DSPs and variables (m, n) equal (8, 7); Fig. 5 shows the allocation result of loop in Fig. 4(a).

In the second stage, we want to reduce the entire execution time and fully utilize function units. Since iterations inside each barrier section are independent, any conventional single DSP scheduling method can be applied for each DSP separately. In this paper we choose the *multi-dimensional rotation* method (Passos and Sha, 1998), because it can be directly applied on DSMP architecture and compared with our *TSM* (Tongsima et al., 1997). Fig. 6 shows the entire algorithm *TSM*.

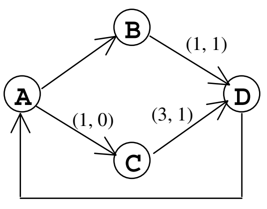
```
input : MDFG $G = (V, E, d, t)$
output : MDFG $G' = (V, E, d', t)$, matrix $T$
$T = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$; $G' = G$;
while ($\exists(a,0)$ and $(0,b)$ in $d'$, for $a,b > 0$) begin     /* Neither the outer nor inner loop can become parallel */
    $T = \begin{bmatrix} 1 & 0 \\ 1 & 1 \end{bmatrix}$;     /* Use loop skewing to eliminate this situation */
    $\forall d'(e) \in d'$  $d'(e) = T \times d'(e)$;
end
if ($\exists(0,a)$ in $d'$, for $a > 0$) begin     /* The inner loop can't become parallel */
  if ($\exists(b,-c)$ in $d'$, for $b,c > 0$)
    $T = T \times \begin{bmatrix} 1 & 0 \\ \lceil c+1/b \rceil & 1 \end{bmatrix}$;     /* Make all dependence with positive distances in all dimensions*/
  $T = T \times \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$;     /* Use loop permutation to make the inner loop become parallel*/
  $\forall d'(e) \in d'$  $d'(e) = T \times d'(e)$;
 end
return $G' = (V, E, d', t), T$;
```

Fig. 3. Pseudo codes of algorithm *LP*.

```
for $l = 2$ to $m+n$ begin
    dopar $k$ = max $(1, l-n)$ to min $(m, l-1)$ begin
        D [$k$, $l-k$] = B [$k-1$, $l-k$] × C [$k-1$, $l-k-2$] ;
        A [$k$, $l-k$] = D [$k$, $l-k$] × 0.5 ;
        B [$k$, $l-k$] = A [$k$, $l-k$] + 1 ;
        C [$k$, $l-k$] = A [$k$, $l-k-1$] + 2 ;
    end
end
```

D = {(1, 1), (1, 0), (3, 1)}

(a)



(b)

Fig. 4. The parallelized nested loop and MDFG of those in Fig. 1.

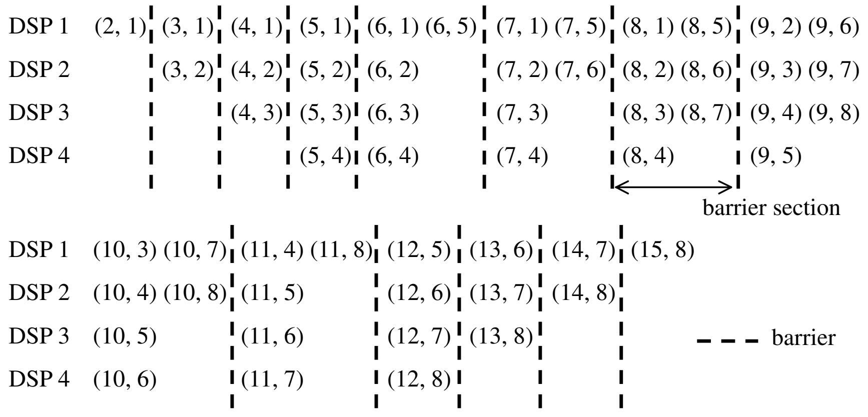

DSP 1 (2, 1) (3, 1) (4, 1) (5, 1) (6, 1) (6, 5) (7, 1) (7, 5) (8, 1) (8, 5) (9, 2) (9, 6)
DSP 2 (3, 2) (4, 2) (5, 2) (6, 2) (7, 2) (7, 6) (8, 2) (8, 6) (9, 3) (9, 7)
DSP 3 (4, 3) (5, 3) (6, 3) (7, 3) (8, 3) (8, 7) (9, 4) (9, 8)
DSP 4 (5, 4) (6, 4) (7, 4) (8, 4) (9, 5)
barrier section
DSP 1 (10, 3) (10, 7) (11, 4) (11, 8) (12, 5) (13, 6) (14, 7) (15, 8)
DSP 2 (10, 4) (10, 8) (11, 5) (12, 6) (13, 7) (14, 8)
DSP 3 (10, 5) (11, 6) (12, 7) (13, 8)
DSP 4 (10, 6) (11, 7) (12, 8)
barrier

Fig. 5. Allocation results of nested loop in Fig. 1(a) with $(m, n) = (8, 7)$.

```
input : nested loop with depth two L
G   convert (L);  (G',T)   LP (G);        /* Convert the nested loop to its
                                             corresponding MDFG */
Lp   unimodular transformations (L,T);
Allocate iterations to every DSP ;
for (each barrier section ) begin
     Each DSP can apply any appropriate scheduling and allocation
     method to execute iterations allocated to it
end
```

Fig. 6. Pseudo codes of algorithm *TSM*.

3.3. Preliminary performance analysis

In the following, we use an analytical model to analyze *TSM* and conventional methods used in DSMP. Based on the constraints of DSMP, we only consider the execution time and ignore both synchronization and memory access overheads.

3.3.1. Basic principles

Because retiming technique is widely used, we focus on its features to analyze execution performance. Scheduling result after applying retiming technique contains three main phases: prologue, repetitive patterns, and epilogue. If the given program is sufficiently large, the repetitive patterns will dominate the execution time.

Some variables are used in our analyses. $Prologue(n)$, $length(n)$, and $epilogue(n)$ represent corresponding execution lengths, where n is the number of resources. $List(n)$ is the execution length of a single repetitive pattern produced by the *list scheduling method* (Man et al., 1986), which is a simple scheduling method without the retiming technique and usually cannot obtain the optimum result. *R retiming depth*, $d(n)$, is the number of iterations that must be moved into the prologue and epilogue.

The following are our assumptions. The input nested loop is depth two with loop bounds of the outer and inner loops of m and n, respectively. The system architecture contains N identical DSPs, $DSP1 \sim DSPN$, and each DSP contains k function units. Within each barrier section, if the iterations cannot be allocated equally, we simply allow $DSP1$ to execute the additional iterations.

3.3.2. Preliminary analysis

From our observations, conventional multiple-DSP scheduling methods based on a retiming technique allocate an operation to every DSP in each time unit and never change the execution sequence of the iterations. Therefore, their execution time is the same in our analytical model, which is described in Lemma 3.1.

Lemma 3.1. *After applying any conventional multiple-DSP scheduling methods, their execution time is*

$$prologue(N) + (mn - d(N)) \times length(N) + epilogue(N) \tag{3.1}$$

Proof. This formula is trivial for the retiming process.

We classify *TSM* into two cases, named *TSM* 1 and *TSM*2, based on whether the nested loop can be parallelized directly or not. In other words, transformation matrices T obtained from algorithm *LP* are identity and $\begin{bmatrix} w & 1 \\ 1 & 0 \end{bmatrix}$, $w \geqslant 0$, for *TSM* 1 and *TSM*2, respectively. Lemmas 3.2–3.4 describe their execution time. □

Lemma 3.2. *After applying TSM and the transformation matrix T is identity, its execution time is*

$$m \times (prologue(k) + (n/N - d(k)) \times length(k) + epilogue(k)) \tag{3.2}$$

Proof. Because the inner loop can be executed in parallel directly, the iteration space remains square as shown in Fig. 7(a). It can be divided into m equal barrier sections with n independent iterations. Hence, formula (3.2) can be obtained directly. □

Lemma 3.3. *After applying TSM and the transformation matrix T is* $\begin{bmatrix} w & 1 \\ 1 & 0 \end{bmatrix}$, $w \geqslant 0$, *the number of iterations executed by DSP*1 *is*

$$\begin{cases} \left(1 + \left\lfloor \frac{m-1}{N} \right\rfloor\right) \times \left\lfloor \frac{m-1}{N} \right\rfloor \times Nw + \left\lceil \frac{m-1}{N} \right\rceil \times ((mw - w)\,\mathbf{mod}\ Nw) \times 2 + \left\lceil \frac{m}{N} \right\rceil \times (w + n - mw) \\ \quad if\ \ wm + 1 \leqslant w + n \\ \left(1 + \left\lfloor \frac{n}{Nw} \right\rfloor\right) \times \left\lfloor \frac{n}{Nw} \right\rfloor \times Nw + \left\lceil \frac{n}{Nw} \right\rceil \times (n\,\mathbf{mod}\ Nw) \times 2 + A \times \left\lceil \frac{B+1}{N} \right\rceil - \left(C \times \left\lfloor \frac{A}{w} \right\rfloor + \min(C, D)\right) \times \left(\left\lceil \frac{B+1}{N} \right\rceil - \left\lceil \frac{B}{N} \right\rceil\right) \\ \quad if\ \ wm + 1 > w + n \end{cases} \tag{3.3}$$

where variables (A, B, C, D) *are equal to* $(mw - w - n, \lfloor n/w \rfloor, (n + 1)\mathbf{mod}\ w, A\mathbf{mod}\ w)$.

Proof. After loop transformations, the iteration space is as shown in Fig. 7(b) or (c) based on the relationships of variables w, m, and n. It can be divided into $wm + n - w$ barrier sections, but each contains unequal iterations. Hence, we calculate the number of iterations allocated to *DSP*1 for the longest execution time. The result can be obtained from formula (3.3), and the detailed calculations are omitted here. □

Lemma 3.4. *After applying TSM and the transformation matrix T is* $\begin{bmatrix} w & 1 \\ 1 & 0 \end{bmatrix}$, $w \geqslant 0$, *its execution time is*

$$List(k) \times B + (prologue(k) + epilogue(k)) \times C + length(k) \times (A - B - d(k)C) \tag{3.4}$$

where variables (B, C) *equal* $(d(k)Nw(1 + d(k)), wm + n - w - 2d(k)Nw)$, *and A is the number of iterations executed by DSP*1 *calculated from formula* (3.3).

Proof. Similarly to Lemma 3.3, the entire execution time is dominated by *DSP*1. Iterations allocated to *DSP*1 are divided into $wm + n - w$ unequal barrier sections with independent iterations. We can only apply the retiming technique if the barrier section contains more than $d(k)$ iterations. Otherwise, it must be simply scheduled using the *List scheduling method*. The execution time of this case is given by formula (3.4), and the detailed calculations are also omitted. □

3.3.3. Experimental results

In this section, six DSP applications selected from (Passos and Sha, 1998; Lee et al., 2001; Yu et al., 1997) are used to compare conventional methods and *TSM*. As before, the multi-dimensional rotation method is applied to the conventional scheduling method and the second stage of *TSM*. Based on characters in each application, we use DSPs that contain different numbers of multipliers and adders listed in Table 1 to balance their utilization.

Fig. 8 shows scheduling results of each example. It is clear that *TSM* can obtain much shorter execution time, because *TSM* allocates more than one operation to a DSP in each time unit to increase the function unit utilization. Besides, as shown in Fig. 9, speedup results of *TSM* are usually more scalable. The reason is that the conventional method uses all the DSPs to execute a

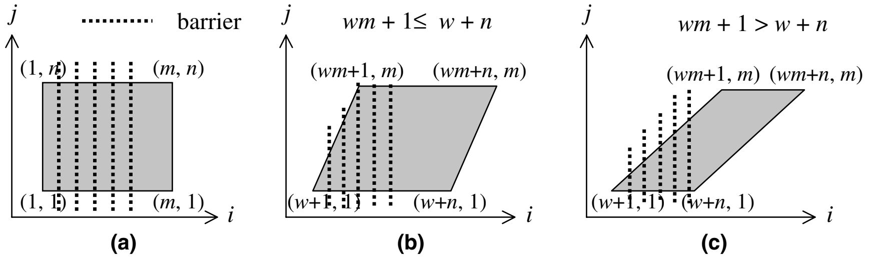


Fig. 7. Three kinds of parallelized iteration space.

Table 1
Characters and resource constraints for each DSP application

	No. of multiplications	No. of additions	No. of multipliers	No. of adders
Transmission lines [19]	4	8	1	2
Model B	8	4	2	1
Wave digital filter [9]	2	2	1	1
Model A [24]	3	4	1	1
Infinite impulse filter [9]	8	8	1	1
Floyd–Steinberg algorithm [25]	4	13	1	2

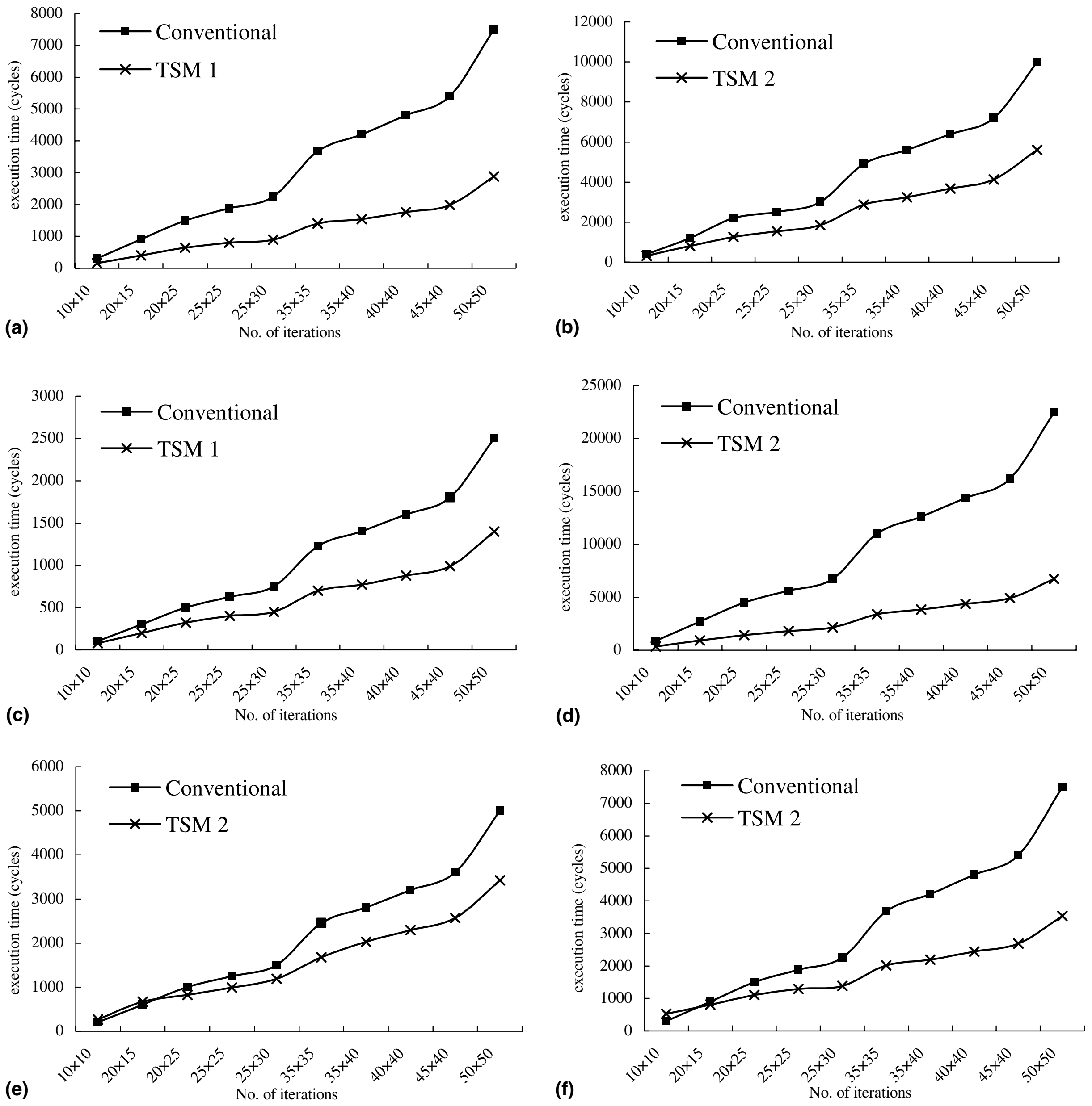


Fig. 8. Sheduling results. (a) Transmission lines, (b) infinite impulse filter, (c) wave digital filter, (d) Floyd–Steinberg algorithm, (e) model A, (f) model B.

single iteration, its speedup will be restricted by the essential parallelism of the application. However, *TSM* allocates iterations to every DSP instead of operations, and its speedup will not be restricted. Therefore,

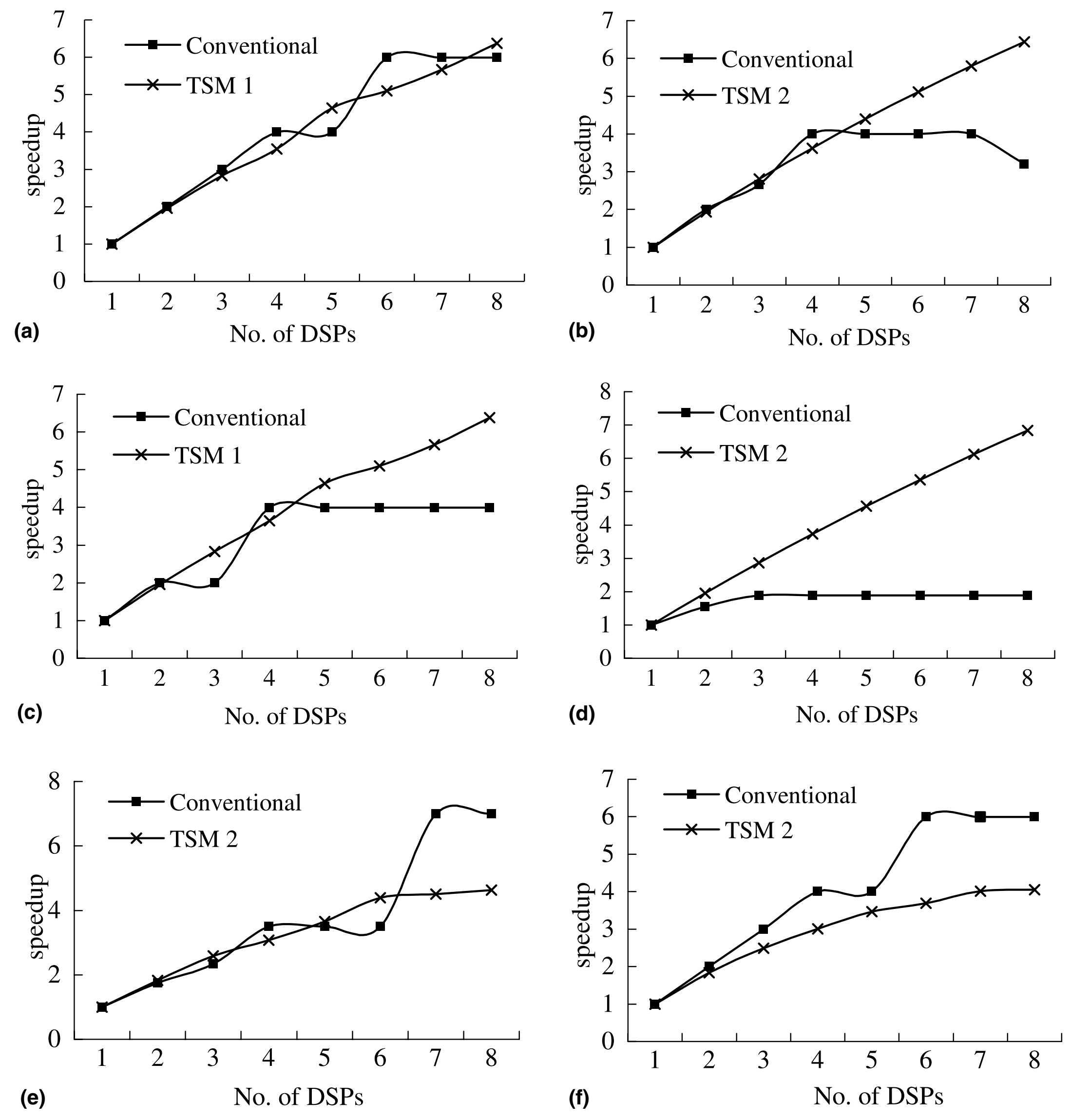


Fig. 9. Speedup results. (a) Transmission lines, (b) infinite impulse filter, (c) wave digital filter, (d) Floyd–Steinberg algorithm, (e) model A, (f) model B.

although the exact speedup value of *TSM* may be smaller, it can achieve better speedup performance for larger system sizes.

3.3.4. Formal evaluation

In the following, we analyze formulas (3.1)–(3.3) using a formal approach. In total there are 15 variables, and we partition them into two groups by considering their features. The first group contains variables (N, k, w, m, n), which can be obtained directly once the input problem and system architecture have been specified. Others belong to the second group, which are calculated based on the behavior of the problem, the applied scheduling method, and the first group variables. We add a new variable *Ops* to represent the number of operations in a single iteration, and list the approximate values of these 16 variables in Table 2.

We rewrite formulas (3.1), (3.2), and (3.4) as follows using Table 2:

$$\text{execution time of conventional methods} = mn \times \lceil \text{Ops}/N \rceil \tag{3.5}$$

$$\text{execution time of TSM1} = mn/N \times \lceil \text{Ops}/k \rceil \tag{3.6}$$

$$\text{execution time of TSM2} = (A - B + C_2 B) \times \lceil \text{Ops}/k \rceil \tag{3.7}$$

where variable A is the result calculated from formula (3.3), and variable B equals to $d(k)Nw(1 + d(k))$

For a specific system architecture, results calculated from formulas (3.6) and (3.7) are, in general, smaller than (3.5). It means that *TSM* can usually achieve shorter execution times for a given architecture. As

Table 2
Variables and their approximate values

Variable	Approximate value	Comments
N, k, w, m, n	N, k, w, m, n	Obtains directly from the input problem and system architecture
List(N)	$C_1 \times length(N)$	C_1, C_2 are constants ($C_1, C_2 \geqslant 1$)
List(k)	$C_2 \times length(k)$	
Length(N)	**Ceiling** (Ops/N)	Assumes that all resources are fully utilized
Length(k)	**Ceiling** (Ops/k)	
$d(N)$	$d(N)$	Unpredictable from the input problem and system architecture
$d(k)$	$d(k)$	
Prologue(N)	$0.5 \times d(N)length(N)$	$d(N)$ iterations are moved into prologue and epilogue
Epilogue(N)	$0.5 \times d(N)length(N)$	$\Rightarrow prologue(N) + epilogue(N) \geqslant d(N)length(N)$
Prologue(k)	$0.5 \times d(k)length(k)$	$d(k)$ iterations are moved into prologue and epilogue
Epilogue(k)	$0.5 \times d(k)length(k)$	$\Rightarrow prologue(k) + epilogue(k) \geqslant d(k)length(k)$

variable N increases, because formulas (3.6) and (3.7) will not be restricted by variable *Ops*, *TSM* could achieve scalable speedup results. Finally, if variable k increases, the execution time with conventional methods cannot be decreased, but with *TSM*, it can be. The reason is conventional methods always allocate only one operation to a DSP for each time unit, but *TSM* allocates more. This analysis is consistent with evaluation results in Section 3.3.3.

3.4. Summary

We briefly summarize advantages and disadvantages of *TSM*. Obviously *TSM* can achieve better function unit utilization and performance in terms of execution time and speedup. Furthermore, since *TSM* uses both unimodular transformations and conventional single DSP scheduling, it can simultaneously exploit the degree to which the program can be made parallel and instruction-level parallelism.

But *TSM* still has some potential problems. From the definition of retiming technique, it requires several consecutive iterations to redistribute operations. In *TSM*, however, if the nested loop is transformed, barrier sections at the beginning and end cannot apply retiming technique because they will not contain enough iterations.

In view of this problem, loop tiling technique seems useful. Given a nested loop, this technique will divide its iteration space into tiles contained consecutive iterations. Therefore, if we allocate tiles to a DSP instead of iterations, the retiming technique can always be applied in *TSM*.

4. Two-level scheduling method with loop tiling

The second approach of TSM is the *two-level scheduling method with loop tiling* (*TSMLT*). We still use a nested loop of depth two and the architecture features defined above to explain and analyze *TSMLT*. Similarly to *TSM*, *TSMLT* also can be easily extended to cover nested loops with depths greater than two.

4.1. Loop tiling steps

Notice that not any nested loop can be tiled. If two iterations A and B contains dependency with distance $(a, -b)$ for $a, b > 0$, Fig. 10 shows the dependency will be violated after loop tiling. Algorithm *LT* listed in Fig. 11 is designed to remove such dependencies.

We also need to select the appropriate tile size. The tile size can be obtained by any one of proposed method, and we simply assume it is $P \times Q$, where P and Q are positive integers. WLOG, we let $P \times Q \geqslant d$ because d is usually smaller. After loop tiling, algorithm *TD* listed in Fig. 12 is used to transfer intra-tile (original) dependencies to inter-tile dependencies. The tiled nested loop and iteration space with tile size 4×3 are shown in Fig. 13.

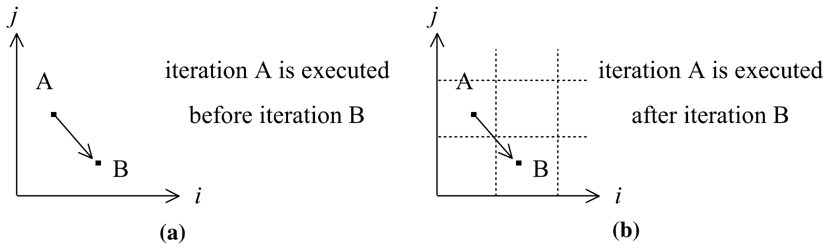


Fig. 10. Iteration spaces with and without loop tiling.

```
input  : MDFG  $G = (V, E, d, t)$
output  : MDFG  $G' = (V, E, d', t)$, matrix  $T$
  $T = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$; $G' = G$;
  if ($\exists(a, -b)$ in $d'$, for $a, b > 0$) begin
    $T = \begin{bmatrix} 1 & 0 \\ \lceil b/a \rceil & 1 \end{bmatrix}$;          /* Use loop skewing to eliminate
                                                   dependences with distance (a, b) */
    $\forall d'(e) \in d'$  $d'(e) = T \times d'(e)$; end
return  $G' = (V, E, d', t), T$;
```

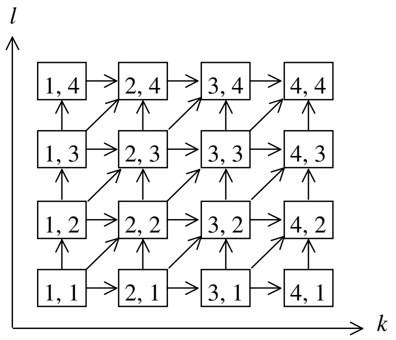
Fig. 11. Pseudo codes of algorithm *LT*.

```
input  : MDFG  $G = (V, E, d, t), P, Q$
output  : dependence  set $d'$
$i = 0$; $j = 0$; $d' = \phi$;
if ($\exists(a, 0)$ in $d$, for $a > 0$)
  for $i = \lfloor a/P \rfloor$ to $\lfloor (a + P - 1)/P \rfloor$ $d' = d' \cup (a, 0)$;     /* Transfer dependencies with
                                                             distance (a, 0) */
if ($\exists(0, a)$ in $d$, for $a > 0$)
  for $i = \lfloor a/Q \rfloor$ to $\lfloor (a + Q - 1)/Q \rfloor$ $d' = d' \cup (0, a)$;     /* Transfer dependencies with
                                                             distance (0, a) */
if ($\exists(a, b)$ in $d$, for $a, b > 0$) begin
  for $i = \lfloor a/P \rfloor$ to $\lfloor (a + P - 1)/P \rfloor$
    for $j = \lfloor b/Q \rfloor$ to $\lfloor (b + Q - 1)/Q \rfloor$
                                                          /* Transfer dependencies with
      $d' = d' \cup (i, j)$; end                                distance (a, b) */
return  $d'$;
```

Fig. 12. Pseudo codes of algorithm *TD*.

```
for k = 1 to m by 4 begin
    for i = k to min (m, k+4–1) begin
        for l = 1 to n by 3 begin
            for j = l to min (n, l+3–1) begin
                D [i, j] = B [i–1, j] × C [i–1, j–2] ;
                A [i, j] = D [i, j] × 0.5 ;
                B [i, j] = A [i, j] + 1 ;
                C [i, j] = A [i, j–1] + 2 ;
            end
        end
    end
end
```

(a)



(b)

Fig. 13. The tiled nested loop and iteration space of nested loop in Fig. 1(a).

4.2. Main stages of two-level scheduling method with loop tiling

Given a nested loop with depth two, the process of *TSMLT* contains three main stages. The first stage is to tile the nested loop. The second stage is to parallelize and allocate tiles to every DSP, and the final stage is to achieve the scheduling goals for each DSP. All steps of the first stage are shown in Section 4.1, and we explain other two stages in detail as follows.

Essentially, the process of second and third stages of *TSMLT* is the same as for *TSM*, using the tile instead of the iteration. In the second stage, although algorithm *LP* can be applied directly to parallelize tiles, it can be simplified to algorithm *TP* listed in Fig. 14 by ignoring dependency with distance $(a, -b)$, $a, b > 0$.

```
1 input : dependence set $d$
2 output : matrix $T$
3 if ($\exists(0,a)$ in $d$, for $a>0$) begin
4     if ($\exists(b,0)$ in $d$, for $b>0$) $T=\begin{bmatrix}1&0\\1&1\end{bmatrix}$;  /* Neither the outer nor inner loop can become parallel and use loop skewing to eliminate this situation */
5     else $T=\begin{bmatrix}1&0\\0&1\end{bmatrix}$;
6     $T=T\times\begin{bmatrix}0&1\\1&0\end{bmatrix}$;  /* Use loop permutation to make the inner loop become parallel*/
7 end else $T=\begin{bmatrix}1&0\\0&1\end{bmatrix}$;
8 return $T$;
```

Fig. 14. Pseudo codes of algorithm *TP*.

For convenience, we use the same mechanism as *TSM* to allocate tiles to every DSP in *TSMLT*. Similarly, tiles inside each barrier section are independent and any appropriate single DSP scheduling method can be applied. We still use the nested loop in Fig. 1(a), with variables (m,n,P,Q) equal (8, 7, 4, 3), as an example. Suppose that there are four DSPs, Fig. 15 shows its parallelized iteration space and allocation result. Fig. 16 contains the entire algorithm *TSMLT*.

4.3. Preliminary performance analysis

In the following, we analyze *TSMLT* using the same analytical model as above. Variable definitions, input nested loop, and system architecture are all as in Section 3.3. After tiling a nested loop with tile size $P\times Q$, it is clear that not all tiles contain exactly $P\times Q$ iterations. Although some tiles may be smaller, for convenience we assume all contain $P\times Q$ iterations. Hence, analysis result in this section can be treated as the worst case, and the actual execution time is not greater to it.

4.3.1. Preliminary analysis

Basically, *TSMLT* can be classified into four cases according to whether the nested loop is directly tiled or not and the tiles parallelized or not. However, generally the intra-tile dependency distance is smaller than the tile size in each dimension, so inter-tile dependencies with distances (1, 0) and (0, 1) are always be generated. Therefore, we classify *TSMLT* into two cases just based on whether the nested loop is tiled directly or not, and assume the result of algorithm *TP* is always $\begin{bmatrix}1&1\\1&0\end{bmatrix}$.

*TSMLT*1 means that the nested loop can be tiled directly. This case is similar to *TSM*2, but using tiles instead of iterations. Use of Lemmas 4.1 and 4.2 gives the execution time of *TMLT*1.

Lemma 4.1. *After applying TSMLT and the transformation matrix T obtained from algorithm LT is identity, the number of tiles executed by DSP*1 *is*

$$\begin{cases} \left(1+\left\lfloor\frac{m'-1}{N}\right\rfloor\right)\times\left\lfloor\frac{m'-1}{N}\right\rfloor\times N+\left\lceil\frac{m'-1}{N}\right\rceil\times((m'-1)\mathbf{mod}\ N)\times 2+\left\lceil\frac{m'}{N}\right\rceil\times(n'-m'+1) & if\ m'\leqslant n' \\ \left(1+\left\lfloor\frac{n'}{N}\right\rfloor\right)\times\left\lfloor\frac{n'}{N}\right\rfloor\times N+\left\lceil\frac{n'}{N}\right\rceil\times(n'\mathbf{mod}\ N)\times 2+(m'-n'-1)\times\left\lceil\frac{n'+1}{N}\right\rceil & \\ -(n'+1)\times(m'-n'-1)\times\left(\left\lceil\frac{n'+1}{N}\right\rceil-\left\lceil\frac{n'}{N}\right\rceil\right) & if\ m'>n' \end{cases} \tag{4.1}$$

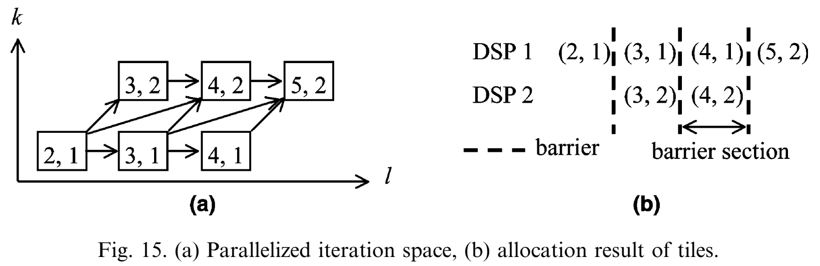


Fig. 15. (a) Parallelized iteration space, (b) allocation result of tiles.

```
input : nested loop with depth two $L$, tile size $P \times Q$
$G = convert(L)$;  $(G', T) = LT(G)$;     /* Convert the nested loop to its
                                             corresponding MDFG */
$L' = unimodular\ transformations(L, T)$;
$L_T$ = loop tiling with tile size $P \times Q$;
$d' = TD(G', P, Q)$;  $T' = TP(d')$;
$L_P = unimodular\ transformations(L_T, T')$;
Allocate tiles to every DSP ;
for (each barrier section) begin
    Each DSP can apply any appropriate scheduling and allocation
    method to execute tiles allocated to it
end
```

Fig. 16. Pseudo codes of algorithm *TSMLT*.

where variables (m', n') *equal* $(\lceil m/P \rceil, \lceil n/Q \rceil)$.

Proof. After loop tiling it will contains $\lceil m/P \rceil \times \lceil n/Q \rceil$ tiles. Because the transformation matrix T' obtained from algorithm *TP* is always $\begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}$ and we use similar allocating mechanism as *TSM*, this situation is similar to Lemma 3.3. Hence, we inherit the result from formula (3.3) and use variables (m', n', w) instead of $(m, n, 1)$. □

Lemma 4.2. *After applying TSMLT and the transformation matrix T obtained from algorithm LT is identity, its execution time is*

$$(prologue(k) + epilogue(k)) \times (m' + n'1)$$
$$+ length(k) \times [APQ - d(k)(m' + n' - 1)] \qquad (4.2)$$

where variables (m', n') *equal* $(\lceil m/P \rceil, \lceil n/Q \rceil)$, *and A is the number of tiles executed by DSP*1 *calculated from formula* (4.1).

Proof. As in Lemma 3.4, the number of tiles allocated to *DSP*1 are obtained from formula (4.2) and represented by A. These tiles are partitioned into $m' + n' - 1$ barrier sections with unequal numbers of independent tiles. Since we assume that $P \times Q \geqslant d(k)$, the retiming technique can be applied in every barrier section. Clearly therefore, the entire execution time of this case can be calculated by formula (4.2). □

*TSMLT*2 is the most complex because the nested loop must be skewed twice and permutated. Unfortunately, as shown in Fig. 17(a), tiles may not form a regular parallelogram after the first skewing and tiling. In other words, its slope (skew factor) is not a constant, so we cannot calculate exactly the number of tiles allocated to *DSP*1. For this problem, we must determine a constant skew factor to estimate this irregular tiling result. If we can obtain this constant skew factor, this case will be reduced to the previous case and results of Lemmas 4.1 and 4.2 can be used immediately.

We have tried two methods. First we intuitively set the final skew factor equal to the maximum slope value, but the estimated tiling result was very different from the real iteration space as shown in Fig. 17(b). Second we selected the average slope value even if it was fractional. As shown in Fig. 17(c), this estimated tiling result is much more realistic. Notice that we simply use this average skew factor to represent an irregular parallelogram; the original tiling result is not changed. We choose the later method and use Lemma 4.3 to calculate the average skew factor.

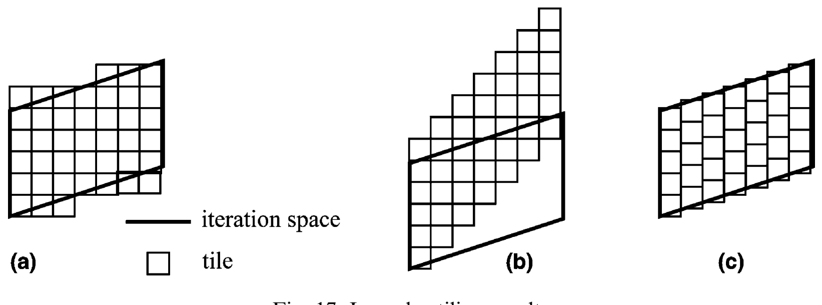


Fig. 17. Irregular tiling results.

Lemma 4.3. *Applying TSMLT and supposing that the nested loop must be skewed with factor* w, $w > 0$, *before tiling. After allocating tiles to every DSP, the average skew factor of the irregular tiling result is* $\lceil mw + 1 - w/Q \rceil - 2/\lceil m/p \rceil - 1$.

Proof. We select iterations (1, 1) and (m, 1) in the original nested loop to calculate the average skew factor. After loop tiling, these two iterations will reside in tiles (1, 1) and $(\lceil m/P \rceil, \lceil mw + 1 - w/Q \rceil)$. Then, after parallelizing tiles, they are shifted to tiles (1, 2) and $(\lceil m/P \rceil, \lceil mw + 1 - w/Q \rceil + \lceil m/P \rceil)$. Hence, the average skew factor is $\lceil mw + 1 - w/Q \rceil + \lceil m/P \rceil - 2/\lceil m/P \rceil - 1$, which is calculated from the distance between these two tiles.

Using the constant skew factor from Lemma 4.3, we can analyze *TSMLT*2 by a similar method to the previous one. Although this result is not exact, its inaccuracy may be tolerated because the estimated tiling result is realistic enough. Lemmas 4.4 and 4.5 give the execution time of *TSMLT*2. □

Lemma 4.4. *Applying TSMLT and the transformation matrix T obtained from algorithm LT is* $\begin{bmatrix} w & 1 \\ 1 & 0 \end{bmatrix}$, $w \geqslant 0$, *the number of tiles executed by DSP*1 *is*

$$\begin{cases} \left(1 + \left\lfloor \frac{m'-1}{N} \right\rfloor\right) \times \left\lfloor \frac{m'-1}{N} \right\rfloor \times Nw' + \left\lceil \frac{m'-1}{N} \right\rceil \times ((m'w' - w')\,\mathbf{mod}\ Nw') \times 2 + \left\lceil \frac{m'}{N} \right\rceil \\ \quad \times (w' + n' - m'w') & \textit{if } w'm' + 1 \leqslant w' + n' \\ \left(1 + \left\lfloor \frac{n'}{Nw'} \right\rfloor\right) \times \left\lfloor \frac{n'}{Nw'} \right\rfloor \times Nw' + \left\lceil \frac{n'}{Nw'} \right\rceil \times (n'\,\mathbf{mod}\ Nw') \times 2 + A \times \left\lceil \frac{B+1}{N} \right\rceil \\ \quad -\left(C \times \left\lfloor \frac{A}{w'} \right\rfloor + \min(C, D)\right) \times \left(\left\lceil \frac{B+1}{N} \right\rceil - \left\lceil \frac{B}{N} \right\rceil\right) & \textit{if } w'm' + 1 > w' + n' \end{cases} \tag{4.3}$$

where variables (m', n', w', A, B, C, D) *equal* $(\lceil m/P \rceil, \lceil n/Q \rceil, \lceil mw + 1 - w/Q \rceil + \lceil m/P \rceil - 2/\lceil m/P \rceil - 1, m'w' - w' - n', \lfloor n'/w' \rfloor, (n' + 1)\,\mathbf{mod}\ w', A\,\mathbf{mod}\ w')$.

Proof. As in Lemma 4.1, we obtain formula (4.3) from formula (3.3) and use the variables (m', n', w') defined above in place of (m, n, w). □

Lemma 4.5. *Applying TSMLT and the transformation matrix T obtained from algorithm LT is* $\begin{bmatrix} w & 1 \\ 1 & 0 \end{bmatrix}$, $w \geqslant 0$, *its execution time is*

$$(prologue(k) + epilogue(k)) \times (m' + n' - w') + length(k) \times [APQ - d(k)(m' + n' - w')] \tag{4.4}$$

where variables (m', n', w') *equal* $(\lceil m/P \rceil, \lceil n/Q \rceil, \lceil mw + 1 - w/Q \rceil + \lceil m/P \rceil - 2/\lceil m/P \rceil - 1)$, *and A is the number of tiles executed by DSP*1 *calculated from formula* (4.3).

Proof. This proof is the same as for Lemma 4.2. □

4.3.2. Experimental results

In the following, we use the same examples and architectures as in Section 3.3.3 to compare conventional methods, *TSM*, and *TSMLT*. From Fig. 18, *TSMLT* outperforms conventional method but not so clear-cut with *TSM*. The main reason is that although retiming technique can always be applied in *TSMLT*, more iterations will be allocated to *DSP*1. This situation will lower the function unit utilization and lengthen the execution time. Due to this tradeoff, performance improvement between *TSM* and *TSMLT* becomes uncertain. Based on our observations, if both $d(k)$ and the skewing factor from algorithm *LP* are large, *TSMLT* performs better.

In Fig. 18, we also consider results for different tile sizes. Usually, smaller tile sizes achieve better performance. It is consistent with the tradeoff mentioned above, because smaller tile size allows more even allocation of tiles (iterations). Speedup results of *TSMLT* shown in Fig. 19 are similar to those shown in Fig. 12.

4.3.3. Formal evaluation

We analyze *TSMLT* in the same manner as in Section 3.3.4. Formulas (4.2) and (4.4) can be rewritten using the approximate values of variables listed in Table 2:

$$\text{execution time of TSMLT1} = APQ \times \lceil \text{Ops}/k \rceil \tag{4.5}$$

where variable A is the result calculated from formula (4.1).

$$\text{execution time of TSMLT2} = APQ \times \lceil \text{Ops}/k \rceil \tag{4.6}$$

where variable A is the result calculated from formula (4.3).

It is interesting that formulas (4.5) and (4.6) are similar. Comparing to formula (3.5), *TSMLT* usually can obtain shorter execution time with specific architecture. Speedup results of *TSMLT* are also more scalable for similar reasons as for *TSM*, no matter how much we increase N or k. The comparison between *TSM*

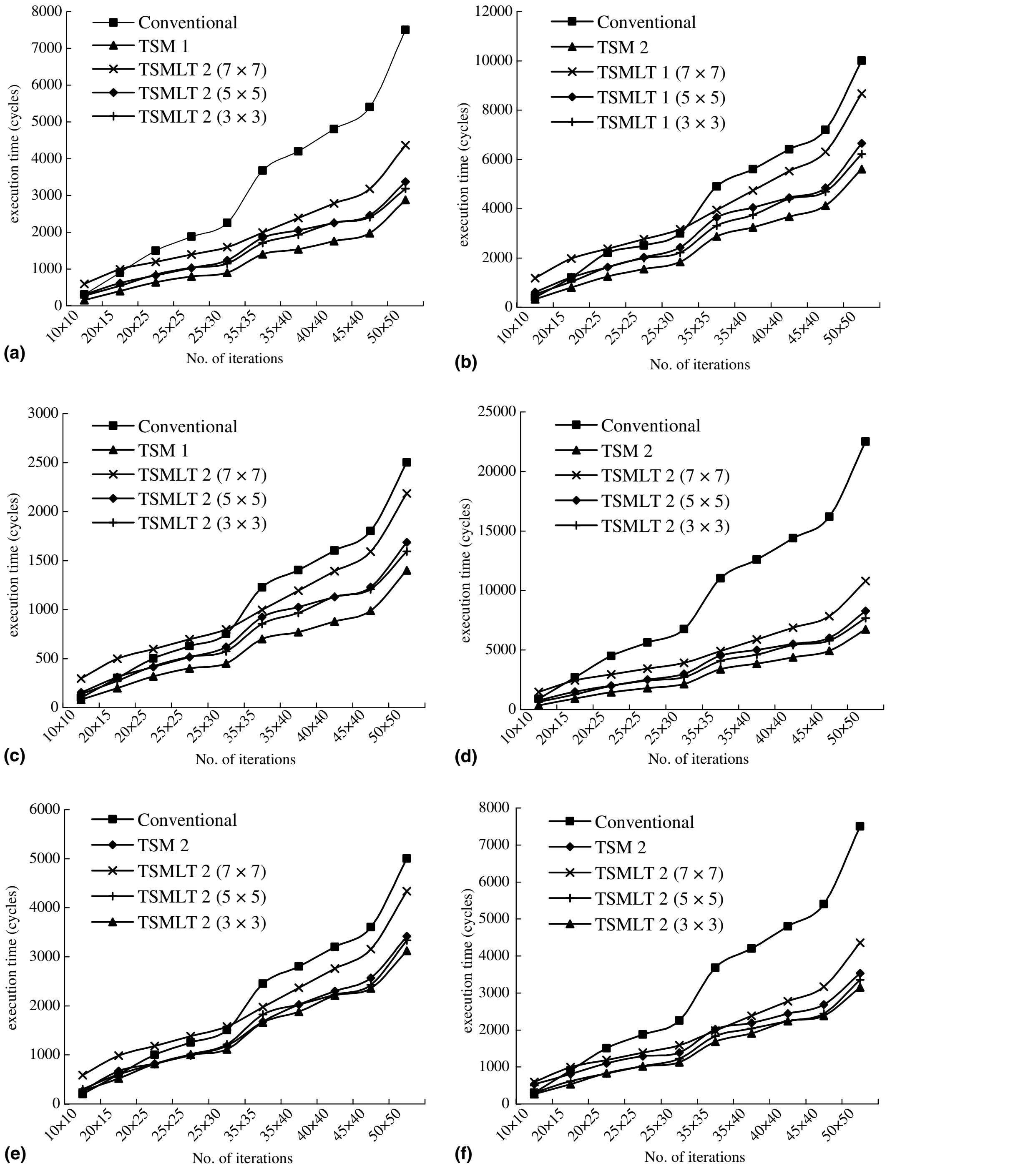


Fig. 18. Scheduling results. (a) Transmission lines, (b) infinite impulse filter, (c) wave digital filter, (d) Floyd–Steinberg algorithm, (e) model A, (f) model B.

and *TSMLT* is uncertain, because the number of iterations (tiles) allocated to *DSP*1 cannot be easily obtained from above formulas. Usually, if a nested loop can be scheduled using *TSM* 1, it can achieve shorter execution time no matter it belongs to which case of *TSMLT*. On the other hand, if a nested loop must be scheduled using *TSM*2, and both $d(k)$ and the skewing factor from algorithm *LP* are large, *TSMLT* may have better results. This analysis is also consistent with above evaluations.

4.4. Summary

We summarize advantages and disadvantages of *TSMLT* briefly. Similar to *TSM*, *TSMLT* can achieve

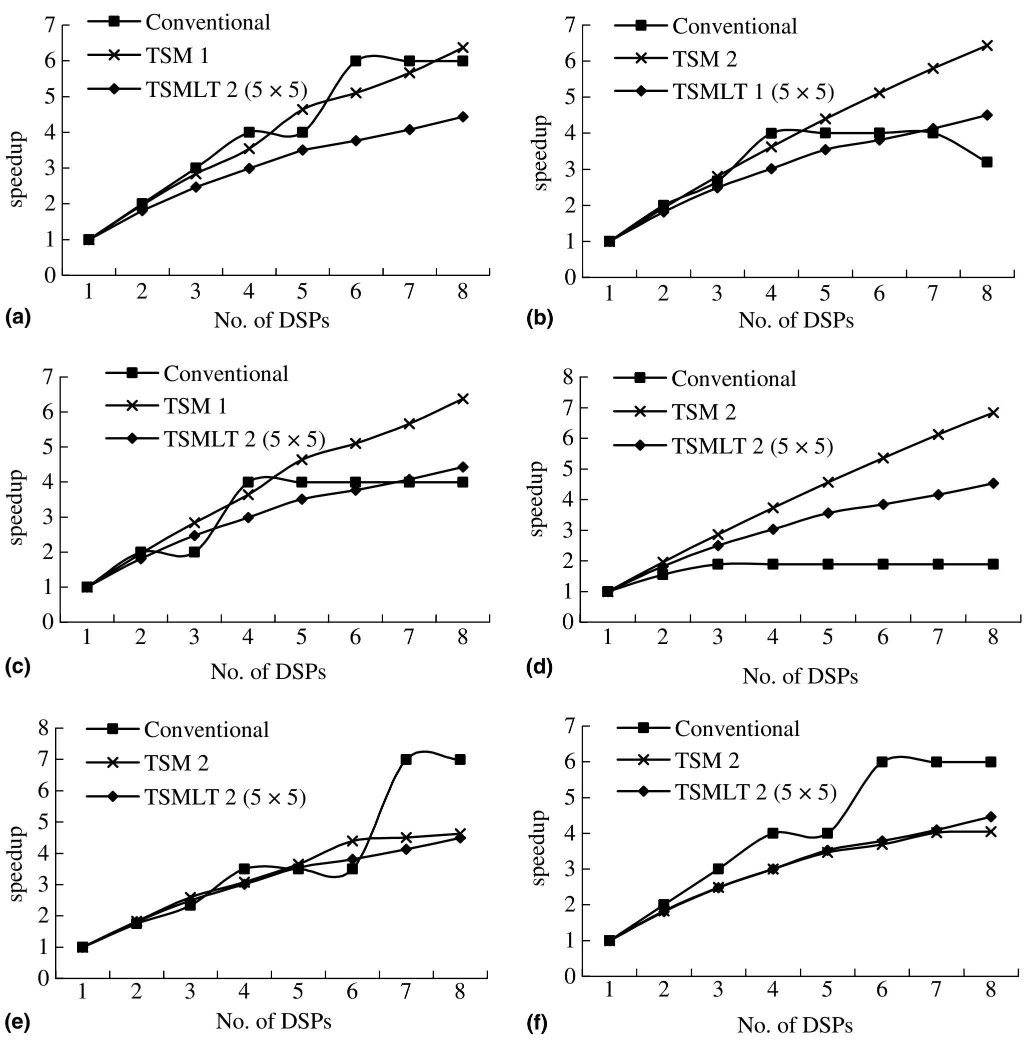


Fig. 19. Speedup results. (a) Transmission lines, (b) infinite impulse filter, (c) wave digital filter, (d) Floyd–Steinberg algorithm, (e) model A, (f) model B.

higher function unit utilization, shorter execution time, and more scalable speedup results. Parallel degree and instruction level parallelism are also exploited. Moreover, *TSMLT* can takes advantage of data locality because it applies the loop tiling technique. Compared with *TSM*, its performance is uncertain and may depend on the input nested loop.

In order to simplify above analyses, we employ an ideal model, which ignores both memory access and synchronization overheads. However, they cannot be entirely eliminated in real system. Thus, decreasing the demands of synchronization and memory access are design issues for any scheduling method.

TSM and *TSMLT* both predominate in synchronization overheads. It needs synchronizations after every time unit in conventional methods, but only after every barrier section in our methods. Since the number of barrier sections is much less than execution time units, our methods will cause less synchronization overheads.

As for memory access overheads our methods also have advantages. Generally, fetching an operand from adjacent DSPs or remote memory is much slower than local storage. In DSP applications, we find that a variable is usually defined and used in the same iteration. In this situation, conventional methods will cause many inter-DSP communications or remote memory accesses, because they separate operations in the same iteration into all DSPs. However, in *TSM* and *TSMLT*, data transference is unnecessary within barrier section, because those iterations and tiles are independent. Thus, our methods also can cause less memory access overheads.

5. Conclusions and future work

In this paper, we have proposed a two-level scheduling method to schedule a nested loop on DSMP, and use an analytical model to analyze the preliminary performance. Our method contains two approaches *TSM* and *TSMLT*, which integrate unimodular transformations, conventional scheduling method used on single DSP, and loop tiling technique. From our analyses, both can achieve shorter execution times, higher function unit utilization, and more scalable speedup, and *TSMLT* also advantageously use data locality. Comparing *TSM* and *TSMLT* show that neither is always better than the other, which may depends on the input nested loop.

Besides previously listed features, there are still several promising issues for future research. First is the construction of a simulation and evaluation environment. We already have an environment that can schedule MDFG using some conventional methods on DSMP. After integrating unimodular transformations and loop tiling techniques, it will help us evaluate *TSM* and *TSMLT* more accurately. Second is the reorganization of our methods. *TSM* and *TSMLT* combine three methods but leave essential algorithms unchanged. In the future, we propose designing other scheduling methods, which preserve features but would not directly integrate these methods.

Acknowledgements

This research was supported by the National Science Council of the Republic of China under contract number: NSC 89-2213-E009-200.

References

Barnwell III, T.P., Gaglio, S., Price, R.M., 1978. A multi-microprocessor architecture for digital signal processing. In: Proceedings of International Conference on Parallel Processing, August.

Chao, L.F., Sha, E.H.-M., 1993. Static scheduling of uniform nested loops. In: Proceedings of 7th International Parallel Processing Symposium, Newport, CA, USA, April, pp. 1421–1424.

Eyre, J., Bier, J., 2000. The evolution of DSP processors. IEEE Signal Processing Magazine 17 (2), 43–51.

Hsu, Y.C., Jeang, Y.L., 1993. Pipeline scheduling techniques in high-level synthesis. In: Proceedings of 6th Annual IEEE International ASIC Conference and Exhibition, Rochester, NY, USA, pp. 396–403.

Jeng, L.G., Chen, L.G., 1994. Rate-optimal DSP synthesis by pipeline and minimum unfolding. IEEE Transactions on Very Large Scale Integration (VLSI) Systems 2 (1), 81–88.

Koch, P., Larsen, N., Bauer, T., Ejlersen, O., 1997. GENETICAS: A multi-DSP scheduling technique based on genetic algorithms. In: Proceedings of 30th Asilomar Conference on Signals, Systems and Computers, Pacific Grove, CA, USA, vol. 2. pp. 1391–1395.

Kung, S.Y., 1988. VLSI Array Processors. Prentice Hall, Englewood, NJ.

Lamport, L., 1974. The parallel execution of DO loops. Commun. ACM SIGPLAN 17 (2), 82–93.

Lee, Y.H., Tsai, M.L., Chen, C., 2001. RPUSM: An effective instruction scheduling method for nested loops. In: Proceedings of National Computer Symposium, Workshop on Computer Architecture and Parallel Systems, Taiwan, December, pp. C025–C036.

Leiserson, C.E., Saxe, J.B., 1991. Retiming synchronous circuitry. Algorithmica 6 (1), 5–35.

Madisetti, V.K., 1995. VLSI Digital Signal Processors: An Introduction to Rapid Prototyping and Design Synthesis. Butterworth-Heinemann, Boston.

Man, H., Rabaey, J., Six, P., Claesen, L.J., 1986. Cathedral-II: a silicon compiler for digital signal processing. IEEE Design and Test 3 (6), 13–25.

Parhi, K.K., 1999. VLSI Digital Signal Processing Systems: Design and Implementation. Wiley Inter-Science, New York.

Passos, N.L., Sha, E.H.-M., 1996. Achieving full parallelism using multi-dimensional retiming. IEEE Transactions on Parallel and Distributed Systems 7 (11), 1150–1163.

Passos, N.L., Sha, E.H.-M., 1998. Scheduling of uniform multi-dimensional systems under resource constraints. IEEE Transactions on VLSI Systems 6 (4), 719–730.

Passos, N.L., Sha, E.H.-M., Chao, L.F., 1995. Optimizing synchronous systems for multi-dimensional applications. In: Proceedings of European Design and Test Conference, pp. 54–58.

Shatnawi, A., Ahmad, M.O., Swamy, M.N.S., 1999. Scheduling of DSP data flow graphs onto multiprocessors for maximum throughput. In: Proceedings of IEEE International Symposium on Circuits and Systems, Orlando, FL, USA, vol. 6. pp. 386–389.

Simar, R. Jr., 1998. Codevelopment of the TMS320C6x VelociTI architecture and compiler. In: Proceedings of IEEE International Conference on Acoustics, Speech, and Signal Processing, Seattle, WA, USA, vol. 5. pp. 3145–3148.

Tongsima, S., Sha, E.H.-M., Passos, N.L., 1997. Communication-sensitive loop scheduling for DSP applications. IEEE Transactions on Signal Processing 45 (5), 1309–1322.

Wolf, M.E., Lam, M.S., 1991. A loop transformation theory and an algorithm to maximize parallelism. IEEE Transactions on Parallel and Distributed Systems 2 (4), 452–471.

Wolfe, M., 1996. High Performance Compilers for Parallel Computing. Addison-Wesley, Redwood City, CA, USA.

Yu, T.Z., Sha, E.H.-M., Passos, N.L., Ju, R., 1997. Algorithm and hardware support for branch anticipation. In: Proceedings of IEEE Great Lakes Symposium on VLSI, pp. 163–168.

Cheng Chen is a professor in the Department of Computer Science and Information Engineering at National Chiao Tung University, Taiwan, ROC. He received his B.S. degree from the Tatung Institute of Technology, Taiwan, ROC in 1969 and M.S. degree from the National Chiao Tung University, Taiwan, ROC in 1971, both in electrical engineering. Since 1972, he has been on the faculty of National Chiao Tung University, Taiwan, ROC. From 1980 to 1987, he was a visiting scholar at the University of Illinois at Urbana Champaign. During 1987 and 1988, he served as the chairman of the Department of Computer Science and Information Engineering at the National Chiao Tung University. From 1988 to 1989, he was a visiting scholar of the Carnegie Mellon University (CMU). Between 1990 and 1994, he served as the deputy director of the Microelectronics and Information Systems Research Center (MISC) in National Chiao Tung University. His current research interests include computer architecture, parallel processing system design, parallelizing compiler techniques, and high performance video server design.

Yi-Hsuan Lee is a Ph.D. candidate in Computer Science and Information Engineering at National Chiao Tung University, Taiwan, ROC. She received her B.S. degree in Computer Science and Information Engineering at National Chiao Tung University, Taiwan, ROC in 1999. Her current research interests include computer architecture, parallelizing compiler techniques, multi-processor scheduling problem, task scheduling for heterogeneous systems, and scheduling problem in DSP architecture.