CODE OPTIMIZATION OF POLYNOMIAL APPROXIMATION FUNCTIONS ON CLUSTERED INSTRUCTION-LEVEL PARALLELISM PROCESSORS

Mei Yang[†], Jinchu Wang[‡], S. Q. Zheng[*], and Yingtao Jiang[†]

[†] Department of Electrical and Computer Engineering, University of Nevada, Las Vegas, NV 89154, USA

[‡] Hangzhou Fast Electronics Co. Ltd., Hangzhou, Zhejiang, 310011, China

[*] Department of Computer Science, The University of Texas at Dallas, Richardson, TX 75083, USA

Emails: [†]{meiyang, yingtao}@egr.unlv.edu, [‡]jinchuwang@hotmail.com, [*]sizheng@utdallas.edu

Abstract

In this paper, we propose a general code optimization method for implementing polynomial approximation functions on clustered instruction-level parallelism (ILP) processors. In the proposed method, we first introduce the parallel algorithm with minimized data dependency. We then schedule and map the data dependency graph (DDG) constructed based on the parallel algorithm to appropriate clusters and functional units of a specific clustered ILP processor using the proposed parallel scheduling and mapping (PSAM) algorithm. The PSAM algorithm prioritizes those nodes on the critical path to minimize the total schedule length and ensures that the resulted schedule satisfies the resource constraints imposed by a specific cluster ILP processor. As a result, our method produces the schedule lengths close to the lower bounds determined by the critical path lengths of the DDGs. Experimental results of typical polynomial mathematical functions on TI 'C67x DSP show that the proposed method achieves significant performance improvement over the traditional computation method.

Key Words: Polynomial approximation functions, instructional-level parallelism processors, DSPs, parallel functional units, data dependency graph, critical path

1 Introduction

In recent years, clustered instruction-level parallelism (ILP) processors have gained much interest from both academic and industrial communities due to the high performance brought by parallel execution of programs on multiple pipelined functional units [11]. In a clustered ILP processor, resources such as functional units, register files, and caches are partitioned or replicated and then grouped together as on-chip clusters [11]. These clusters are usually connected through a set of inter-cluster communication buses to allow cross data accesses. Fig. 1 shows a typical clustered ILP processor model.

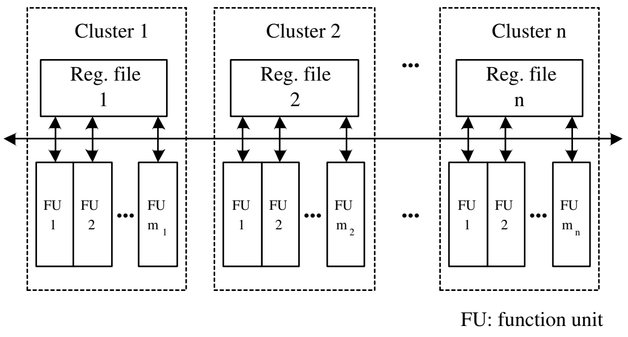


Figure 1: A typical clustered ILP processor model.

One typical ILP structure is the very long instruction word (VLIW) architecture [15], where one VLIW instruction is composed of multiple instructions, each running on a functional unit. Mainstream digital signal processing processors (DSPs) based on the VLIW architecture include TI's TMS320C6000 series [25], Motorola's DSP56000 series [14], and Analog Device's ADSP TigerSHARC series [2], etc., with the peak performance reaching up to several giga floating-point operations per second (GFLOPs). The advanced features of these DSPs make them very suitable for real-time applications featuring multi-channel and multi-function computation, such as voice and speech recognition, high-end graphics and imaging, and wide-band digital receiving, which involve the computation of many kinds of mathematical functions.

Efficient implementations of various mathematical functions are required by aforementioned real-time applications. Four kinds of commonly used mathematical functions are the exponential function, the logarithmic function, trigonometric functions, and inverse trigonometric functions [5]. They are generally implemented by the polynomial approximation method. The implementations of these mathematical functions in the library provided by DSP vendors, however, cannot meet stringent timing constraint imposed by some real-time applications, as the algorithms used by these library functions are not suitable for the DSPs featuring parallel functional units. The general optimization methods for ILP processors (such as software pipelining [10, 13] and loop unrolling [9]) do not fully explore the instruction-level parallelism when implementing polynomial approximation functions with high cross-iteration dependency on these DSPs.

Different from the goal of minimizing the total operation count for traditional sequential processors, the goal of code optimization for ILP processors is to minimize the number of execution cycles (i.e. schedule length) [16]. The code optimization needs to consider the resource constraints inherent in an ILP processor, such as the number of functional units in a cluster, limited number of functional units where the instructions can be executed, the number of cross data accesses, and the number of registers available in each functional unit. Several code optimization methods have been proposed for ILP processors or multiprocessors [8, 11, 18]. A list scheduling [17] based code generation framework for clustered ILP processors is proposed to combine the cluster assignment, register allocation, and instruction-level scheduling [11]. However, it fails to simultaneously consider the constraint that the number of cross data accesses is limited in one clock cycle. The scheme based on the split-node data acyclic graph is too complex for implementing polynomial approximation functions [8]. A simpler scheduling algorithm proposed in [18] is not suitable for heterogenous processors such as the various functional units typically seen in a DSP. More noticeably, all the above mentioned code optimization methods have no optimization of the computation algorithm.

In this paper, we propose a code optimization scheme to reduce the data dependency and improve the parallel scheduling for implementing polynomial approximation functions on clustered ILP processors, particularly on DSPs based on the VLIW architecture. In the proposed scheme, we first introduce the parallel algorithm with minimized data dependency. We then schedule and map the constructed data dependency graph (DDG) to appropriate clusters and functional units using the proposed parallel scheduling and mapping (PSAM) algorithm based on list scheduling [17]. The PSAM algorithm prioritizes those nodes on the critical path to minimize the total schedule length and ensures that the resulted schedule satisfies the resource constraints imposed by a specific cluster ILP processor. As a result, the proposed method produces the schedule lengths close to the lower bounds determined by the critical path lengths of the DDGs. To evaluate our method, we have implemented typical mathematical functions in assembly codes on a general-purpose floating-point DSP, TI's TMS320C67x ('C67x) processor. Experimental results show that our optimized codes achieve up to 79.5% performance improvement (in terms of the total number of clock cycles) over TI 'C67x library functions.

The rest of the paper is organized as follows. Section 2 provides the background information including polynomial approximation functions and the general computation process of these functions. Section 3 presents our code optimization method for parallel implementations of these polynomial approximation functions. Section 4 shows two typical working examples of the proposed method on TI 'C67x processor. Section 5 presents the experimental results and the comparison with TI 'C67x library functions. Section 6 concludes the paper.

2 Background

2.1 Polynomial Approximation Functions

The fundamentals of polynomial approximation mathematical functions are discussed in [1]. If a function $f(x)$ has continuous derivatives up to the $(n+1)^{th}$ order, then this function can be expanded in the following fashion [1] (Taylor Series):

$$f(x) = f(a) + \frac{f'(a)(x-a)}{1!} + \cdots + \frac{f^{(n)}(a)(x-a)^n}{n!} + R_n, \tag{1}$$

where R_n is the remainder after the $(n+1)^{th}$ term. If $\lim_{n\to\infty} R_n = 0$ and $a = 0$, the series is called the MacLaurin Series.

The MacLaurin Series of the commonly used mathematical functions can be found in [1, 6]. In this paper, we focus our study on the following typical mathematical functions pertaining to digital signal processing applications.

$$e^x = 1 + \frac{x}{1!} + \frac{x^2}{2!} + \cdots + \frac{x^n}{n!} + \cdots \qquad -\infty < x < \infty \tag{2}$$

$$\ln(1+x) = x - \frac{x^2}{2} + \frac{x^3}{3} - \cdots + (-1)^{n-1}\frac{x^n}{n} + \cdots \qquad -1 < x \leq 1 \tag{3}$$

$$\sin x = x - \frac{x^3}{3!} + \cdots + (-1)^n \frac{x^{2n+1}}{(2n+1)!} + \cdots \qquad -\infty < x < \infty \tag{4}$$

$$\cos x = 1 - \frac{x^2}{2!} + \frac{x^4}{4!} + \cdots + (-1)^n \frac{x^{2n}}{(2n)!} + \cdots \qquad \mid x \mid < \frac{\pi}{2} \tag{5}$$

$$\tan^{-1} x = x - \frac{x^3}{3} + \cdots + (-1)^n \frac{x^{2n+1}}{(2n+1)!} + \cdots \quad \mid x \mid < 1 \tag{6}$$

To achieve the required level of precision, these functions are typically approximated by polynomials of certain sufficient degrees [19, 27]. A desired function, say $f(x)$, is approximated by a polynomial approximation equation $P(n, x)$ such that

$$f(x) = P(n, x) + e(x), \qquad x_l < x < x_u, \tag{7}$$

where x_l and x_u are the lower and the upper bounds of x, respectively, and $e(x)$ is the error function usually minimized in the min-max (equi-ripple) sense [19]. Without the loss of generality, we present the polynomial approximation equations of *exponential*, *logarithm*, *sin*, *cos*, and *arctangent* functions in Equations (8)-(12). Note that *cos* function is generally implemented by calling *sin* function, as shown in Equation (11), where n is the highest power of the variable to bring the required error range down to ϵ.

$$exp(x) = \sum_{i=0}^{n} \{a[i]x^i\} + e(x) \quad 0 \le x \le \ln 2 \text{ and } \mid e(x) \mid \le \epsilon \tag{8}$$

$$ln(1+x) = \sum_{i=1}^{n} \{a[i]x^i\} + e(x) \quad 0 \le x \le 1 \text{ and } \mid e(x) \mid \le \epsilon \tag{9}$$

$$sin(x) = x \sum_{i=0}^{n} \{a[i]x^{2i}\} + xe(x) \quad \mid x \mid \le \frac{\pi}{2} \text{ and } \mid xe(x) \mid \le \epsilon \tag{10}$$

$$cos(x) = sin(x + \frac{\pi}{2}) \tag{11}$$

$$atan(x) = x \textstyle\sum_{i=0}^{n} \{a[i]x^{2i}\} + xe(x) \quad -1 \le \mid x \mid \le 1 \text{ and } \mid xe(x) \mid \le \epsilon \tag{12}$$

2.2 Implementation of These Functions

The implementation of each function consists of polynomial equation computation, pre-processing, and post-processing. In the following, we introduce the polynomial equation computation method that is commonly used. Discussion on the pre- and post-processing techniques can be found in [19].

The equation computation of each function involves iterations of multiplication and accumulation operations of $a[i]$ and x^i, as shown in Equations (8)-(12). Based on their different computation processes, we classify the polynomial equations discussed in Section 2.1 into two types, $f_1(x)$ and $f_2(x)$: $f_1(x)$ includes $exp(x)$ and $ln(1+x)$, and $f_2(x)$ includes $sin(x)$, $cos(x)$, and $atan(x)$. The general computation algorithm for $f_1(x)$ and $f_2(x)$ is based on Equation (13) and Equation (14), respectively, where k_i's represent the coefficients $a[i]$'s $(1 \leq i \leq n)$.

$$\begin{aligned} f_1(x) &= \sum_{i=0}^{n} k_i x^i \\ &= k_0 + k_1 x + k_2 x^2 + \cdots + k_{n-1} x^{n-1} + k_n x^n \\ &= k_0 + x(k_1 + x(k_2 + \cdots + x(k_{n-1} + x k_n) \cdots)) \end{aligned} \tag{13}$$

$$\begin{aligned} f_2(x) &= \sum_{i=0}^{n} k_i x^{2i+1} \\ &= k_0 x + k_1 x^3 + k_2 x^5 + \cdots + k_n x^{2n+1} \\ &= k_0 x + x(x^2(k_1 + x^2(k_2 + \cdots + \\ &\quad x^2(k_{n-1} + x^2 k_n) \cdots))) \end{aligned} \tag{14}$$

The algorithms employed in Equations (13) and (14) are only efficient for computing polynomial equations in sequential processors and they are not efficient for clustered ILP processors featuring parallel functional units. Unfortunately, the implementation of these functions with cross-iteration dependency cannot be optimized using the general optimization methods built in the compilers provided by the vendors of the clustered ILP processors. Although these optimization techniques, such as software pipelining [13], and loop unrolling [9], are suitable for applications with a large number of independent variables, such as FFT and FIR, they do not work well for applications with the cross-iteration dependency, such as computing Equations (13)-(14). Existing code optimization methods [8, 11, 18] based on list scheduling [17] are also not suitable for this application. In the next section, we will present our code optimization method for implementing polynomial approximation functions on clustered ILP processors.

3 Proposed Work

Our goal of optimization is to obtain a minimum schedule length by fully exploring the parallelism of polynomial equation computation on clustered ILP processors. To achieve this, we propose a code optimization method consisting of two phases:

1. Find a parallel algorithm to compute the polynomial equation with minimized data dependency.

2. Schedule and map the data dependency graph constructed based on the parallel algorithm to appropriate clusters and functional units of the targeted clustered ILP processor with minimized schedule length subject to all the resource constraints.

Each phase is explained in detail as follows.

3.1 Parallel Algorithms

$f_1(x)$ and $f_2(x)$ can be rewritten as shown in Equations (15)-(16), where $n = 2m$.

$$\begin{aligned} f_1(x) &= \sum_{i=0}^{n} k_i x^i \\ &= k_0 + \sum_{i=1}^{m} x^{2i-1}(k_{2i-1} + k_{2i}x) \\ &= k_0 + x(k_1 + k_2 x) + \cdots + x^{2i-1}(k_{2i-1} + \\ &\quad k_{2i}x) + \cdots + x^{2m-1}(k_{2m-1} + k_{2m}x) \end{aligned} \tag{15}$$

$$\begin{aligned} f_2(x) &= \sum_{i=0}^{n} k_i x^{2i+1} \\ &= k_0 x + \sum_{i=1}^{m} x^{4i-1}(k_{2i-1} + k_{2i}x^2) \\ &= k_0 x + x^3(k_1 + k_2 x^2) + \cdots + x^{4i-1}(k_{2i-1} + \\ &\quad k_{2i}x^2) + \cdots + x^{4m-1}(k_{2m-1} + k_{2m}x^2) \end{aligned} \tag{16}$$

Take $f_1(x)$ as an example. The computation of Equation (15) involves m groups of operations: $x(k_1+k_2x)$, $x^3(k_3+k_4x)$, $\cdots$, and $x^{2m-1}(k_{2m-1}+k_{2m}x)$. Each group is composed of the computation of x^{2i-1}, $1 \le i \le m$, and three two-operand multiplication or addition operations. More importantly, the data dependency between each group is minimized so that the computation of the m groups can be performed in parallel.

3.2 Scheduling And Mapping Data Dependency Graphs

For a given program, we use the data dependency graph to describe its data flow. Based on the parallel algorithm, we generate the linear assembly code (i.e., the assembly code with no cluster and functional unit assigned and no register allocated), construct the DDG, and schedule and map the DDG to the targeted clustered ILP processor.

3.2.1 Construction of Data Dependency Graph

Definition 1 *A Data Dependency Graph $G = (V, E, D, T, Y)$ is a node-weighted and edge-weighted directed graph, where V is the set of instructions, E is the set of edges connecting two nodes with data dependency, $D(e)$ represents the delay slots of the instruction of the node where edge e originates from, $T(u)$ represents the execution time (functional unit latency) of node u, and $Y(u)$ denotes the instruction type of node u.*

A node in the DDG can be any instruction of a DSP. We assume that $Y(u) = 1$ for additions (ADD) represented as ellipses, $Y(u) = 2$ for multiplications (MPY) represented as rectangles, $Y(u) = 3$ for memory operations (LOAD or STORE) represented as triangles, and so on. Each node u is denoted as a pair $(d, T(u))$, where d is the destination operand. To have a complete DDG, we typically add one dummy node l representing instruction NULL, which takes 0 clock cycle to execute and has no destination operand. A *source node* is one with no edge terminating at it. A *sink node* is one with no edge originating from it. In a DDG, the dummy node is the only sink node.

An edge in the DDG shows the data dependency between two nodes, and the label on it denotes its weight $D(e)$. A *critical path* is defined as the longest path from any source node to the sink node in a DDG. The length of a path is the sum of weights of all nodes and all edges on the path. The critical path length determines the lower bound of the running time of a program.

3.2.2 The PSAM Algorithm

When scheduling and mapping a DDG to clusters and functional units of the targeted clustered ILP processor, the data dependency relations defined by the DDG must be satisfied. Meanwhile, the schedule must satisfy all the resource constraints of the targeted clustered ILP processor, including the number of functional units in a cluster, the functional units that each instruction can be executed in, the number of cross data accesses allowed in one clock cycle, and the number of registers available in each functional unit.

As a matter of fact, the corresponding DDG's of all applications discussed in this paper are directed acyclic graphs (DAGs). It is well known that optimal scheduling of nodes in a DAG to a set of processors is an NP-hard problem [18]. In this section, we propose a heuristic scheduling and mapping algorithm, named as the Parallel Scheduling and Mapping (PSAM) algorithm, to schedule and map the DDG to clusters and functional units of a specific clustered ILP processor satisfying all the aforementioned resource constraints.

To keep track of the schedule of functional units, we introduce the following variables. y denotes a functional unit with $y.CID$ representing the cluster that y belongs to. $I(y,t)$ denotes the node that is assigned to functional unit y to run at clock cycle t; its initial value is set to NIL. $F(Y(u),t)$ denotes the set of available functional units in which instructions of type $Y(u)$ can be executed at t; it is initialized as all possible functional units that instructions of type $Y(u)$ can be executed in. In the PSAM algorithm, we assume that one DDG $G = (V, E, D, T, Y)$ is represented using adjacency lists. Each node in G is also associated with the following data structures.

$C(u)$: The cluster that node u is assigned to.

$P(u)$: The priority value of node u. It is defined as the length of the longest path starting from u to the sink node l. Note that, for a DDG, the length of a path is the summation of the weights of all the nodes and edges on the path.

$R(u)$: The register allocated to the destination operand of u.

$S(u)$: The clock cycle that node u is scheduled in.

$col(u)$**:** The color of node u. If the node is already scheduled, ready to be scheduled, or not ready to be scheduled, it is colored as BLACK, GRAY, or WHITE, respectively.

$\pi(u)$**:** The parent node of node u, which is determined as the predecessor node of u on the path to be scheduled. Initially, $\pi(u) = NIL$ for each node $u \in V$.

$pre(u)$**:** The list of predecessors of node u. For a node u with no predecessor, such as the source node, $pre(u) = NIL$.

$adj(u)$**:** The list of descendants of node u, which is one of the input to the PSAM algorithm. For a node u with no descendant, such as the sink node, $adj(u) = NIL$.

The PSAM algorithm uses a priority queue Q [4] to manage the set of nodes ready to be scheduled in a non-increasing sorted list of their priority values ($P(u)$'s) and instruction types ($Y(u)$'s). Assuming that the number of cross data accesses is limited to x, we use $X(t)$ to record the number of cross data accesses at clock cycle t, $0 \leq X(t) \leq x$.

Registers will also be allocated during the process of scheduling and mapping of nodes to clusters and functional units. Considering a node (a two-operand or three-operand instruction) in the DDG, the source operand (if not an instant value) must be the destination operand of its predecessor node. Hence, as long as we allocate the register for the destination operand of each node, all the operands will get their register allocated. For example, for the node of instruction ADD $src1, src2, dst$, only a register for dst needs to be allocated since registers for $src1$ and $src2$ are allocated in its predecessor nodes. For each register r, we use $C(r)$ to denote the cluster it belongs to. We use $A(y, t)$ to represent the set of registers available in functional unit y at t; it is initialized as the set of all registers in functional unit y.

The PSAM algorithm consists of three major steps.

Step 1: Compute $P(u)$ as the length of the longest path from u to l for each node $u \in V$ using the Bellman-Ford algorithm.

Step 2: For each node $u \in V$, initialize $pre(u)$, $\pi(u)$, and $col(u)$. Initialize Q by inserting all source nodes.

Step 3: Schedule nodes in Q to appropriate clusters and functional units according to the non-increasing order of their priority values.

If there is an available functional unit at some cluster in the current and following clock cycles covering the functional unit latency of the instruction, schedule it in the current clock cycle and insert its successor node(s) into Q if all the predecessor nodes of the successor node are scheduled; otherwise, reinsert the node back into Q. The scheduling process is continued until Q is empty.

The pseudocode of the PSAM algorithm is listed below. Using a breadth-first search approach and a priority queue, PSAM prioritizes those nodes on the critical path to minimize the total schedule length while satisfying the data dependency relations defined by the DDG. PSAM also ensures that the resulted schedule satisfies the resource constraints imposed by a specific cluster ILP processor. In most cases, the PSAM algorithm yields a near-optimal schedule. Additionally, the PSAM algorithm minimizes the number of registers by reusing registers allocated to a node's parent node or its predecessor node.

Algorithm: Parallel scheduling and mapping (PSAM)

```
Input: DDG $G = (V, E, D, T, Y), x$
Output: Schedule $I$ and register allocation $R$
//Step 1:
$\forall u \in V$, compute $P(u)$ as the length of the longest path
from $u$ to $l$ using the Bellman-Ford algorithm
/Step 2:
for each node $u \in V - l$ do
    set $pre(u)$ as the set of nodes $u$ is adjacent to
    $\pi(u) \leftarrow NIL$
    $R(u) \leftarrow NIL$
    if there is no $v$ such that $u \in adj(v)$ then
        $col(u) \leftarrow$ GRAY
        ENQUEUE($Q, u$)
    else $col(u) \leftarrow$ WHITE
end-for
for $t \leftarrow 0$ to $M$ ($M$ is a large number) do
    $X(t) \leftarrow 0$
end-for
$t \leftarrow 0$
//Step 3:
while $Q \neq \emptyset$ do
    $Q' \leftarrow \emptyset$
    while $Q \neq \emptyset$ do
        $u \leftarrow$ DEQUEUE($Q$)
        $t \leftarrow \max(t, S(\pi(u)) + T(\pi(u)) + D(\pi(u), u))$
        //if no functional unit available, reinsert $u$ back to $Q$
        if $| U = \{y \mid y \in F(Y(u), t)$ and $I(y, t) = NIL\}| = 0$ then
            ENQUEUE($Q$, $u$)
            break
        //for each source node, assign one available functional unit
        //and register
        else if $\pi(u) = NIL$ and find $y \in U$ such that $X(t') < x$
            for all $t' \in [t..t + T(u) - 1]$ then
            $C(u) \leftarrow y.CID$
            Find $r \in A(y, t)$ and allocate $r$ to $R(u)$
        //for other nodes, assign $u$ the same functional unit and
        //register as its parent node as much as possible
        **else if** $\pi(u) \neq NIL$ **and** find $y \in U$ such that
            $y.CID = C(\pi(u))$ **then**
            $C(u) \leftarrow y.CID$
            **if** $adj(\pi(u)) = \{u\}$ **then** $R(u) \leftarrow R(\pi(u))$
            **else** Find $r \in A(y,t)$ and allocate $r$ to $R(u)$
        **else if** find $y \in U$ such that $X(t') < x$ for all
            $t' \in [t..t + T(u) - 1]$ **then**
            $C(u) \leftarrow y.CID$
            **if** find $v \in pre(u) - \pi(u)$ **and** $adj(v) = \{u\}$ **then**
                $R(u) \leftarrow R(v)$
            **else** Find $r \in A(y,t)$ and allocate $r$ to $R(u)$
        $S(u) \leftarrow t$; $col(u) \leftarrow$ BLACK
        //update corresponding variables
        **for** $t' \leftarrow t$ **to** $t + T(u) - 1$ **do**            $I(Y,t') \leftarrow u$
            $F(Y(u),t') \leftarrow F(Y(u),t') - \{y\}$
            **if** $(\pi(u) \neq NIL$ **and** $y.CID \neq C(\pi(u)))$ **or**
                $(\exists v \in pre(u)$ **and** $y.CID \neq C(v))$ **then**
                $X(t') \leftarrow X(t') + 1$
        **end-for**
        **for** $t' \leftarrow t$ **to** $t + T(u) + D(u) - 1$ **do**
            $A(y,t') \leftarrow A(y,t') - R(u)$
        **end-for**
        **for** each $v \in adj(u)$ **do**
            **if** $col(v) \neq$ BLACK **then**
                **if** $S(u) > S(\pi(v))$
                    **then** $\pi(v) \leftarrow u$
            //insert $v$ to $Q$ if all its predecessor nodes are scheduled
            **if** $\forall w \in pre(v), col(w) =$ BLACK **and** $v \neq l$ **then**
                $col(v) \leftarrow$ GRAY
                ENQUEUE$(Q',v)$
        **end-for**
    **end-while**
    $Q \leftarrow Q'$
    $t \leftarrow t + 1$
**end-while**
```

The complexity of each step of the PSAM algorithm is analyzed as follows. The first step takes $O(\mid V \mid^3)$ time by the Bellman-Ford algorithm [4]. The second step takes $O(\mid V \mid^2)$ time. The third step takes $O(p \mid V \mid + \mid V \mid^2)$ time, where p is the maximum number of registers available, because the breadth-first search takes $O(\mid V \mid + \mid V \mid^2)$ time [4] and the schedule of all nodes needs $O(p \mid V \mid)$ time. Putting everything together, the complexity of the PSAM algorithm is $O(\mid V \mid^3)$.

The PSAM algorithm differs from other list scheduling algorithms [8, 11, 18] by performing the cluster and functional unit assignments and register allocation subject to all resource constraints pertaining to a cluster ILP processor. The PSAM algorithm is also suitable for scheduling and mapping for other applications with directed acyclic DDGs on clustered ILP processors.

4 Examples

In this section, we show how the proposed method works for the code optimization of two functions, *logrithm* and *arctangent* as the example of $f_1(x)$ and $f_2(x)$ respectively, on TI 'C67x DSP.

TI 'C67x is a general-purpose floating-point DSP family featuring the VelociTI VLIW architecture [25]. Its CPU has two data paths (clusters) (A and B) operating in parallel, and each cluster has four parallel functional units (.L, .S, .M, and .D) and one register file containing 16 32-bit registers. A cross data bus exists to connect the functional units of one cluster to registers on the other for exchanging data between the two register files, and the number of cross data accesses in one clock cycle is limited to two. Each functional unit (FU) is controlled by a 32-bit instruction. Thus, the VLIW structure of 'C67x with 256-bit-width instruction allows execution of up to 8 32-bit instructions in one clock cycle (CLK). Each 'C67x instruction has a functional unit latency (UL) of 1 clock cycle followed by various delay slots (DS), each corresponding to 1 clock cycle in 'C67x. Table 1 shows the description of commonly used floating-point instructions. As shown in the table, the number of functional units each instruction can be executed in is limited. The schedule of a program on 'C67x needs to satisfy all these resource constraints.

To optimize the codes for *logarithm* and *arctangent* functions, following the first phase

Instruction	**Syntax**	**Description**	**FU**	**CLKs**	**UL**	**DS**
ADDSP	(.unit) $src1$, $src2, dst$	$dst = src1+$ $src2$;32-bit addition	.L1, .L2	4	1	3
MPYSP	(.unit) $src1$, $src2, dst$	$dst = src1\times$ $src2$;32-bit multiplication	.M1, .M2	4	1	3
LDW	(.unit)$*$+B14/ B15[$ucst15$], dst	$dst = mem$; Load 32 bits from memory	.D1, .D2	5	1	4
STW	(.unit) src,$*$+B14/ B15[$ucst15$]	$mem = src$; Store 32 bits into memory	.D1, .D2	1	1	0

Table 1: Description of commonly used floating-point instructions.

of the proposed method, we derive the two functions as shown in Equations (17) and (18). It was shown that $n = 8$ is enough to satisfy the required error range for most practical applications [19, 27].

$$\begin{aligned} ln(1+x) &= k_1x + k_2x^2 + k_3x^3 + k_4x^4 \\ &\quad +k_5x^5 + k_6x^6 + k_7x^7 + k_8x^8 \\ &= x(k_1 + k_2x) + x^3(k_3 + k_4x) + \\ &\quad x^5(k_5 + k_6x) + x^7(k_7 + k_8x) \end{aligned} \tag{17}$$

$$\begin{aligned} atan(x) &= x + k_1x^3 + k_2x^5 + k_3x^7 + k_4x^9 \\ &\quad +k_5x^{11} + k_6x^{13} + k_7x^{15} + k_8x^{17} \\ &= x + x^3(k_1 + k_2x^2) + x^7(k_3 + k_4x^2) \\ &\quad +x^{11}(k_5 + k_6x^2) + x^{15}(k_7 + k_8x^2) \end{aligned} \tag{18}$$

We then construct the DDGs from the linear assembly codes generated according to Equations (17) and (18). The DDG of Equation (17) implemented on 'C67x is shown in

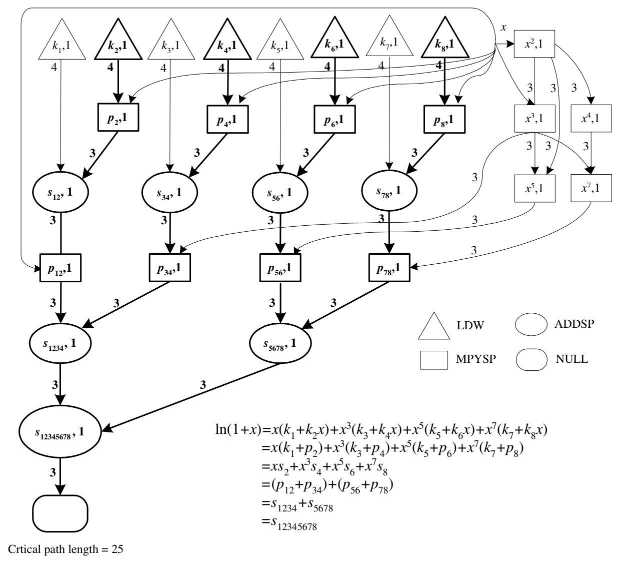
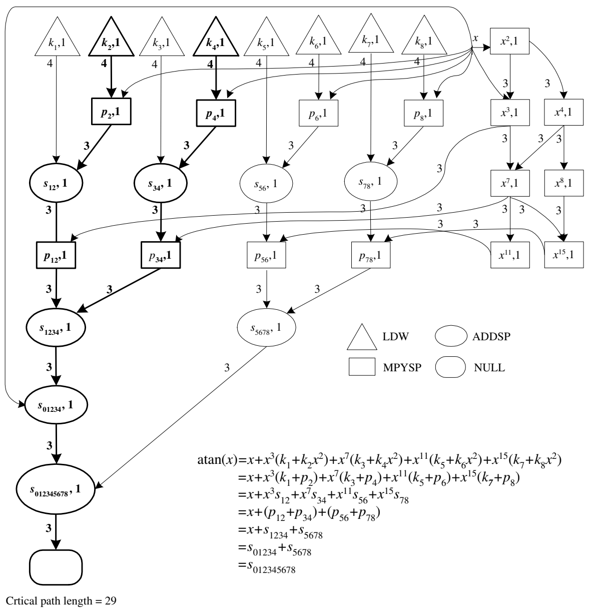


Figure 2: The DDG of Equation (17).

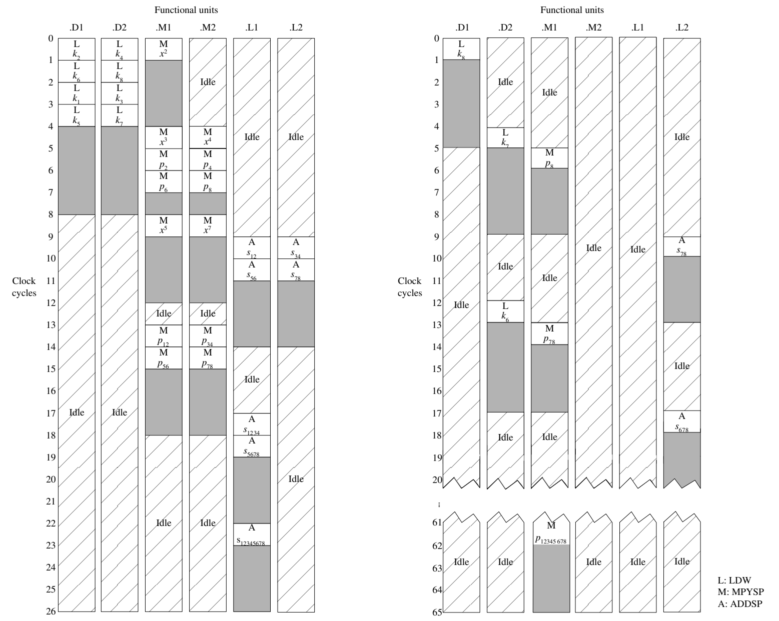


LDW
MPYSP
ADDSP
NULL
atan(x)=x+x3(k1+k2x2)+x7(k3+k4x2)+x11(k5+k6x2)+x15(k7+k8x2)
=x+x3(k1+p2)+x7(k3+p4)+x11(k5+p6)+x15(k7+p8)
=x+x3s12+x7s34+x11s56+x15s78
=x+(p12+p34)+(p56+p78)
=x+s1234+s5678
=s01234+s5678
=s012345678
Crtical path length = 29

Figure 3: The DDG of Equation (18).

Fig. 2, which has 28 nodes. Note that there is no need to load x since it is the function argument. The latency of the critical path of Fig. 2 (as shown by dark paths) is 25 CLKs, which determines the lower bound of parallel scheduling without considering any resource constraint. The DDG of Equation (18) implemented on 'C67x is shown in Fig. 3, which has 31 nodes and the critical path length of 29 CLKs.

Applying the PSAM algorithm with G set as the DDG in Fig. 2 and $x = 2$, we obtain the scheduling and mapping of *logrithm* function shown in Fig. 4(a), in which each column represents a functional unit and each entry shows the instruction that is assigned to the functional unit at each clock cycle. Functional units .S1 and .S2 are not included since they are not used by any instruction in our examples. The empty entries represent idle clock cycles. The shaded entries are the delay slots of an instruction. Using the schedule produced by the PSAM algorithm, the computation of Equation (17) only takes 26 CLKs, which is only 1 CLK longer than the lower bound given by the critical path length of Fig. 2.



$$\begin{aligned}\ln(1+x) &= x(k_1+k_2x)+x^3(k_3+k_4x)+x^5(k_5+k_6x)+x^7(k_7+k_8x)\\ &= x(k_1+p_2)+x^3(k_3+p_4)+x^5(k_5+p_6)+x^7(k_7+p_8)\\ &= xs_2+x^3s_4+x^5s_6+x^7s_8\\ &= (p_{12}+p_{34})+(p_{56}+p_{78})\\ &= s_{1234}+s_{5678}\\ &= s_{12345678}\end{aligned}$$

(a)

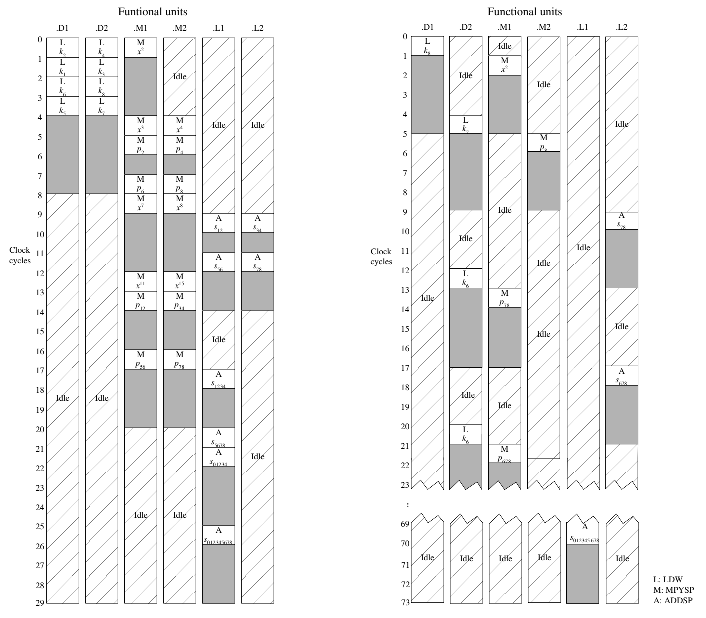
$$\begin{aligned}\ln(1+x) &= k_1x+k_2x^2+k_3x^3+k_4x^4+k_5x^5+k_6x^6+k_7x^7+k_8x^8\\ &= x(k_1+x(k_2+x(k_3+x(k_4+x(k_5+x(k_6+x(k_7+xk_8)))))))\\ &= x(k_1+x(k_2+x(k_3+x(k_4+x(k_5+x(k_6+x(k_7+p_8)))))))\\ &= x(k_1+x(k_2+x(k_3+x(k_4+x(k_5+x(k_6+xs_{78}))))))\\ &= x(k_1+x(k_2+x(k_3+x(k_4+x(k_5+x(k_6+p_{78}))))))\\ &= x(k_1+x(k_2+x(k_3+x(k_4+x(k_5+xs_{678})))))\\ &= \ldots\\ &= p_{12345678}\end{aligned}$$

(b)

Figure 4: (a) Optimized parallel scheduling and mapping of Fig. 2. (b) Scheduling and mapping of the DDG constructed from the linear assembly code based on Equation (13).

Register	Corresponding variable	Register	Corresponding variable
A_0	x	B_0	k_4, p_4, s_{34}, p_{34}
A_1	k_2, p_2, s_{12}, p_{12}, s_{1234}, $s_{12345678}$	B_1	k_8, p_8, s_{78}, p_{78}
A_2	x^2	B_2	x^4, x^7
A_3	k_6, p_6, s_{56}, p_{56}, s_{5678}	B_3	k_3
A_4	x^3	B_4	k_7
A_5	k_1		
A_6	x^5		
A_7	k_5		

Figure 5: Register allocation for *logrithm* function generated from the PSAM algorithm.



$$\begin{aligned}\mathrm{atan}(x)&=x+x^3(k_1+k_2x^2)+x^7(k_3+k_4x^2)+x^{11}(k_5+k_6x^2)+x^{15}(k_7+k_8x^2)\\&=x+x^3(k_1+p_2)+x^7(k_3+p_4)+x^{11}(k_5+p_6)+x^{15}(k_7+p_8)\\&=x+x^3s_{12}+x^7s_{34}+x^{11}s_{56}+x^{15}s_{78}\\&=x+(p_{12}+p_{34})+(p_{56}+p_{78})\\&=x+s_{1234}+s_{5678}\\&=s_{01234}+s_{5678}\\&=s_{012345678}\end{aligned}$$

(a)

$$\begin{aligned}\mathrm{atan}(x)&=x+k_1x^3+k_2x^5+k_3x^7+k_4x^9+k_5x^{11}+k_6x^{13}+k_7x^{15}+k_8x^{17}\\&=x+x(x^2(k_1+x^2(k_2+x^2(k_3+x^2(k_4+x^2(k_5+x^2(k_6+x^2(k_7+x^2k_8))))))))\\&=x+x(x^2(k_1+x^2(k_2+x^2(k_3+x^2(k_4+x^2(k_5+x^2(k_6+x^2(k_7+p_8))))))))\\&=x+x(x^2(k_1+x^2(k_2+x^2(k_3+x^2(k_4+x^2(k_5+x^2(k_6+x^2s_{78})))))))\\&=x+x(x^2(k_1+x^2(k_2+x^2(k_3+x^2(k_4+x^2(k_5+x^2(k_6+p_{78})))))))\\&=x+x(x^2(k_1+x^2(k_2+x^2(k_3+x^2(k_4+x^2(k_5+x^2k_{678}))))))\\&=\ldots\\&=s_{012345678}\end{aligned}$$

(b)

Figure 6: (a) Optimized parallel scheduling and mapping of Fig. 3 using the PSAM algorithm. (b) Scheduling and mapping of the DDG constructed from the linear assembly code based on Equation (14).

As a comparison, Fig. 4(b) shows the scheduling and mapping of the DDG constructed from the linear assembly code generated according to Equation (13), in which there is no instruction executed in parallel and it takes 65 CLKs to finish the equation computation. Fig. 5 shows the resulted register allocation.

Similarly, we obtain the parallel scheduling and mapping of Fig. 3 in Fig. 6(a). Fig. 6(b) shows the scheduling and mapping of the DDG constructed from the linear assembly code

Register	**Corresponding variable**	**Register**	**Corresponding variable**
A_0	x	B_0	k_4, p_4, s_{34}, p_{34}
A_1	k_2, p_2, s_{12}, p_{12}, s_{1234}, s_{01234}, $s_{012345678}$	B_1	k_3
A_2	x^2	B_2	x^4
A_3	k_1	B_3	k_8, p_8, s_{78}
A_4	x^3	B_4	k_7
A_5	k_6, p_6, s_{56}	B_5	x^8, x^{15}, p_{78}
A_6	x^7		
A_7	k_5		
A_8	x^{11}, p_{56}, s_{5678}		

Figure 7: Register allocation for *arctangent* function generated by the PSAM algorithm.

Function	**Description**	**OI (CLKs)**	**'C67x LF (CLKs)**	**Improvement %**
expf	*exponential* function	74	213	65.3%
logf	natural *logarithm* function	44	136	67.6%
log10f	10 base *logarithm* function	47	154	69.5%
sinf	*sin* function	70	173	60%
cosf	*cos* function	74	183	60%
atanf	*arctangent* function	79	265	70.2%
atan2f	*arctangent* function with two arguments	87	424	79.5%

Table 2: Comparison of optimized implementations of typical mathematical functions vs. TI 'C67x library functions in terms of the total clock counts.

generated according to Equation (14). The schedule length of $atan(x)$ is reduced from 73 CLKs to 29 CLKs, which is equivalent to the lower bound determined by the critical path length of Fig. 3. The resulted register allocation is shown in Fig. 7.

It is worthy to point out that code optimization of pre- and post-processing is also important for code efficiency. The details of optimizations of pre- and post-processing techniques for implementing these functions on 'C67x can be found in [28].

5 Experimental Results

To evaluate the performance of the proposed code optimization method, experiments of typical mathematical functions, $ln(1+x)$, $exp(x)$, $sin(x)$, $cos(x)$, and $atan(x)$, are conducted on TI 'C67x development tools, Code Composer Studio2.0 (CCS2.0). $exp(x)$ is optimized similar to $ln(1+x)$ while $sin(x)$ and $cos(x)$ are optimized similar to $atan(x)$. We generate all assembly codes, compile, and debug them on CCS2.0. Our programs achieve the same precision as 'C67x library functions do. Programs of our optimized implementations are profiled and the number of clock cycles of each program with C overhead is measured in CCS2.0. Table 2 compares the maximum running times (all pre-processing and post-processing are counted) of our optimized implementations (OI) of these functions vs. 'C67x library functions (LF) in terms of the number of clock cycles (CLKs). Due to pipeline flushing, missing pipeline conflicts, and extra program fetches, the listed results may have 1 to 2 CLKs error when halting the simulator [22].

Noticeably, the overall improvement of our optimized implementations of these functions over 'C67x library functions is more than 60% (up to 79.5% in some cases). As all polynomial approximation functions can be categorized into $f_1(x)$ and $f_2(x)$, it is expected that performance of other polynomial approximation mathematical functions implemented in 'C67x library can also be significantly improved by the proposed code optimization method.

6 Concluding Remarks

In this paper, we considered the code optimization of polynomial approximation functions on clustered ILP processors and proposed a general code optimization method. In the proposed method, we first find the parallel algorithm with minimized data dependency. According to the parallel algorithm, the linear assembly code for the targeted clustered ILP processor is generated and the DDG is constructed. We then schedule and map the DDG to appropriate clusters and functional units of the targeted clustered ILP processor using the proposed PSAM algorithm. The PSAM algorithm prioritizes those nodes on the critical path to minimize the total schedule length while satisfying the data dependency relations defined by the DDG. The advantage of the PSAM algorithm over other list scheduling based algorithms is it considers all resource constraints imposed by a specific cluster ILP processor, including the number of functional units in a cluster, the functional units that each instruction can be executed in, the number of cross data accesses allowed in one clock cycle, and the number of registers available in each functional unit. Through the examples of several typical polynomial approximation functions on TI '67x, we showed that our method produces the schedule lengths close to the critical path lengths of the DDGs and achieves significant performance improvement over the traditional computation method. The proposed method and the PSAM algorithm are particularly useful for many real-time digital signal processing applications.

References

[1] M. Abramovitz and I.A. Stegun, *Handbook of Mathematical Functions* (New York: Dover Publications, 1965).

[2] Analog Device, TigerSHARC Processor [Online], Available: http://www.analog.com/processors/processors/tigersharc /index.html.

[3] Cody and Waite, *Software Manual for the Elementary Functions* (Prentice Hall, 1980).

[4] T.H. Cormen, C.E. Leiserson, R.L. Rivest, and C. Stein, *Instruction to Algorithms, 2nd Edition* (The MIT Proces, Cambridge, MA, 2001).

[5] D. Das, K. Mukhopadhyaya, and B.P. Sinha, Implementation of four common functions on an LNS co-processor, *IEEE Trans. Computers*, *44*(1), 1995, 155-161.

[6] Engineering Fundamentals, Taylor Series [Online], Available: http://www.efunda.com.

[7] J. Eyre and J. Bier, The evolution of DSP processors, *IEEE Signal Processing Magazine*, *1*(2), 2000, 43-51.

[8] S. Hanono and S. Devadas, Instruction selection, resource allocation and scheduling in the VIV retargetable code generator, *Proc. the 35th ACM/IEEE Design Automation Conf.*, 1998, 510-515.

[9] J.L. Hennessy and D.A. Patterson, *Computer Architecture: A Quantitative Approach, 3rd Ed.* (Elsevier Science & Technology Books, 2002).

[10] Y.T. Hwang and Y.C. Chuang, High performance code generation for VLIW digital signal processors, *Proc. IEEE Workshop on Signal Processing Systems (SiPS)*, 683-692.

[11] K. Kailas, K. Ebcioglu, and A. Agrawala, CARS: a new code generation framework for clustered ILP processors, *Proc. the 7th Int'l Symp. on High-Performance Computer Architecture*, 2001, 133-143.

[12] B. Kruatrachue and T. Lewis, Grain size determination for parallel processing, *IEEE Software*, *5*(1), 1988, 23-31.

[13] M. Lam, Software pipelining: an efficient scheduling techniques for VLIW machines, *Proc. the SIGPLAN'88 Conf. on Programming Language Design and Implementation*, 1988, 318-328.

[14] Motorola 1995. *DSP 56K Central Architecture Overview.*

[15] B. Rau and J. Fisher, Instruction-level parallel processing: History, overview, and perspective, *Journal of Supercomputing*, *7*(1/2), 1983, 9-50.

[16] M. Schlansker, T.M. Conte, J. Dehnert, K. Ebcioglu, J.Z. Fang, and C.L. Thompson, Compilers for instruction-level parallelism, *Computer*, *30*(12), 1997, 63-69.

[17] R. Sethi, *Algorithms for Minimal-Length Schedules* (John Wiley & Sons, Inc., New York, 1976), Chapter 2. Computer and job-shop scheduling theory.

[18] S. Shang, S. Sun, and Q. Wang, An efficient parallel scheduling algorithm of dependent task graphs, *Proc. the 4th Int'l Conf. on Parrallel and Distributed Computing, Applications and Technologies*, 2003, 595-598.

[19] Texas Instruments, *A Collection of Functions for the TMS320C30* (Texas Instruments, 1990).

[20] Texas Instruments, *TMS320C3x General-Purpose Applications User's Guide* (Texas Instruments, 1990).

[21] Texas Instruments, *TMS320C62x/C67x Technical Brief* (Texas Instruments, 1998).

[22] Texas Instruments, *Code Composer Studio User's Guide* (Texas Instruments, 2000).

[23] Texas Instruments, *TMS320C6000 Optimizing Compiler User's Guide* (Texas Instruments, 2000).

[24] Texas Instruments, *TMS320C6000 Programmer's Guide* (Texas Instruments, 2000).

[25] Texas Instruments, TMS320C6701, Floating-Point Digital Signal Processor [Online], Available: http://focus.ti.com/docs/prod.

[26] Texas Instruments, *TMS320C6000 CPU and Instruction Set Reference Guide* (Texas Instruments, 2000).

[27] Texas Instruments 2001. TMS320C62x/C67x library functions package with Code Composer Studio1.2.

[28] M. Yang, Y. Wang, J. Wang, and S.Q. Zheng, Optimized scheduling and mapping of logarithm and arctangent functions on TI TMS320C67x processor, *Proc. IEEE Intl Conf. on Acoustics, Speeach, and Signal Processing (ICASSP)*, 2002, 31563159.