Data Pipeline Optimization for Shared Memory Multiple-SIMD Architecture ⋆

Weihua Zhang[1,2], Tao Bao[1], Binyu Zang[1], Chuanqi Zhu[1]

[1]Parallel Processing Institute, Fudan University, Shanghai, China
[2]Key Laboratory of Computer System and Architecture, Institute of Computing Technology, Chinese Academy of Sciences
{zhangweihua,052053006,byzang,cqzhu}@fudan.edu.cn

Abstract. The rapid growth of multimedia applications has been putting high pressure on the processing capability of modern processors, which leads to more and more modern multimedia processors employing parallel single instruction multiple data (SIMD) units to achieve high performance. In embedded system on chips (SOCs), shared memory multiple-SIMD architecture becomes popular because of its less power consumption and smaller chip size. In order to match the properties of some multimedia applications, there are interconnections among multiple SIMD units. In this paper, we present a novel program transformation technique to exploit parallel and pipelined computing power of modern shared-memory multiple-SIMD architecture. This optimizing technique can greatly reduce the conflict of shared data bus and improve the performance of applications with inherent data pipeline characteristics. Experimental results show that our method provides impressive speedup. For a shared memory multiple-SIMD architecture with 8 SIMD units, this method obtains more than 3.6X speedup for the multimedia programs.

1 Introduction

In recent years, multimedia and game applications have experienced rapid growth at an explosive rate both in quantity and complexity. Currently, since these applications typically demand 10^{10} to 10^{11} operations to be executed per second [1], higher processing capability is expected. Generally speaking, there are two kinds of solutions to the issue - hardware solutions and software solutions. Hardware solutions such as application specific integrated circuits (ASICs) have the advantages of higher performance with lower power consumption; However, their flexibility and adaptability to new applications are very limited. As a result, it is much popular to handle the problem with software solutions which enhance the processing capability of general-purpose processor with multimedia extensions. In the past several years, the key idea in those extensions was to exploit subword

⋆ This research was supported by Specialized Research Fund for the Doctoral Program of Chinese Higher Education under Grant No. 20050246020 and supported by the Key Laboratory of Computer System and Architecture, Institute of Computing Technology, Chinese Academy of Sciences;

parallelism in a SIMD (Single Instruction Multiple Data) fashion, such as Intel's SSE, MIPS's MDMX, TI (Texas Instruments)'s TMS320C64xx series etc.

However, with various multimedia applications becoming more complicated, using only single SIMD unit as a multimedia accelerator can not satisfy the performance requirements of these applications. Although it can improve computing capability by increasing processing elements (PEs) in one SIMD unit, this approach is unacceptable from both hardware and software perspective. Therefore, multiple-SIMD architecture instead of single SIMD unit is becoming a dominant multimedia accelerator in modern multimedia processors. At present, there are two types of multiple-SIMD architectures: one is *shared memory multiple-SIMD architecture* (SM-SIMD)[3–9] , where multiple SIMD units share a common memory (cache) on chip. The other is *distributed memory multiple-SIMD architecture* (DM-SIMD)[10], on which each SIMD unit has its local memory.

Since SM-SIMD architecture can get smaller die size and less power consumption, SM-SIMD architecture is widely used in embedded SOCs [3–9]. Although the details of these SOCs are not completely the same, there are some common characteristics among them in order to fit mobile computing circumstance.

1. There is a shared memory (cache) on chip for better locality, and multiple SIMD units access shared memory through a shared data bus. Shared data bus can replicate one vector to all SIMD units at the same time.
2. There are limited registers in each SIMD unit.
3. Multiple SIMD units are controlled by a general purpose processor core. Most of SM-SIMD architectures use very long instruction word (VLIW) to exploit the parallelism among multiple SIMD units since such an approach offers the potential advantage of simpler hardware design compared with the superscalar approach while still exhibiting good performance through extensive compiler optimization.
4. There are interconnections among SIMD units to make one SIMD unit get data from the registers of its connected SIMD units. We call two SIMD units as *neighboring SIMD units* if there is an interconnection between them.

Table 1 lists the major products of SM-SIMD architecture with these characteristics.

Table 1. Products of SM-SIMD architecture

Product	**Institute**	**Year**
MorphoSys [3]	University of California, Irvine	2000
HERA [4]	New Jersey Institute of Technology	2005
Imagine [5]	Stanford	2004
Motorola 6011 [6]	Motorola Inc.	2003
MGAP-2 [7]	Folsom Inc.	2000
Vision Chips [8]	University of Tokyo	2004
VIRAM [9]	UC Berkeley	2004

Scheduling for these shared bus and interconnected architectures is difficult because the compiler must simultaneously allocate many interdependent

resources: the SIMD units on which the operations take place, the register files to hold intermediate values, and the shared bus to transfer data between functional units and shared memory. These conditions put very high pressure on the optimizing algorithms of SM-SIMD architecture. Most prior VLIW scheduling algorithms, such as [17] and [18] can not deal with resource allocation. Although scheduling algorithm in [12] enables scheduling for architectures in which functional units are connected to multiple register files by shared buses and register file ports, the utilization of these resources is not considered. Optimizing algorithm in [11] solves the utilization issue of shared data bus to some extent through common sub-expressions elimination, but the locality of read-only operands is not exploited by the authors although this type of operands is very common in real multimedia applications. Scheduling algorithm in [13] is an efficiently algorithm that exploits how to improve the utilization of the resources based on the characteristics of multimedia application, but it is not presented in their works that how to exploit pipeline parallelism.

The major challenge to optimizing techniques for SM-SIMD architecture is to reduce the conflict of shared data bus and to improve the parallelism among multiple SIMD units. When there are interconnections among multiple SIMD units, some optimizations can be performed to reduce the conflict of shared data bus. If one operation executed on one SIMD unit can get its operand from the register of the neighboring SIMD unit through interconnection, it is better to get the data through interconnection rather than via shared data bus. The reason is that getting operands from neighborhood would bring no data bus conflict, which is the major motivation for the optimizing technique proposed in this paper. Data pipeline parallelism is that multiple SIMD units get data from their neighboring SIMD units and data flows among these SIMD units as in pipeline. In this paper, we present a novel algorithm, which transforms sequential multimedia programs into data pipeline forms to exploit data pipeline parallelism. While reducing the shared data bus conflict of multiple SIMD units, the algorithm also greatly improve the performance of the application programs with inherent data pipeline characteristics. This paper makes the following contributions:

- This paper presents a novel data pipeline optimization through exploiting the characteristics of read-read reuse in the multimedia applications and the interconnection characteristics in SM-SIMD.
- Based on the experimental results, this paper also gives out some advice on programming for SM-SIMD architecture.

The remaining of this paper is organized as follows. Section 2 gives out the problem overview for the pipeline scheduling. In section 3, we describe pipeline scheduling in detail. Section 4 introduces the experimental method and presents the analysis of the experimental results. And in section 5 we come to the conclusion.

2 Problems Overview

2.1 Constraint of Shared Data Bus on Parallelism

Because there are multiple SIMD units in SM-SIMD architecture, it is necessary to exploit the parallelism among SIMD instructions and map them to different SIMD units. However, many parallel SIMD instructions can not be executed in parallel because of the constraints of shared data bus. Below is an example of such condition.

```
for (i=0; i<100; i++)   // Loop1
    A[i,0:7] = B[i,0:7] + C[i,0:7];

Example code 1: A parallel code fragment.
```

The code in Example code 1 is a program after SIMD optimization. Loop1 is a parallel loop, whose different iterations can be dispatched to different SIMD units. When mapping these iterations to different SIMD units, all of them need to load their operands from the shared memory respectively. As a result, these instructions can only be executed in sequence since shared data bus can only satisfy one of their operand requirements in each cycle. Thus, it is useless to only identify the parallelism in the program to exploit the parallelism for SM-SIMD architecture. Multiple SIMD instructions can be executed in parallel only when there is no shared data bus conflict among them. Therefore it is important for SM-SIMD architecture to reduce the competition of shared data bus in order to fully utilize the computation resources.

2.2 Problem Overview

As analyzed in section 2.1, shared data bus would impede the parallelism among multiple SIMD units in SM-SIMD architecture. Therefore, how to reduce the conflict of shared data bus would greatly impact the parallelism among multiple SIMD units. The scheduling algorithm in [13] can reduce the conflict of shared data bus through replicating read-only data and increasing the register locality. Furthermore, the interconnections[1] among SIMD units could provide better solutions for some applications. One SIMD unit can get the data from the register of its neighboring SIMD unit. Such data-getting manner performs better than loading data from the shared memory because accessing the register of its neighborhood would provide no bus conflict. The goal of data pipeline optimization is to exploit pipeline parallelism, which can greatly reduce the conflict of shared data bus and improve the parallelism of SM-SIMD architecture.

[1] Transfers of values between SIMD units are accomplished through operations of explicit movements along the interconnections among SIMD units. The interconnection is assumed to have a bi-directional ring topology among SIMD units. In other words, one SIMD unit has connections with its two neighboring SIMD units. Though this assumption is not necessary, it simplifies the compiler algorithms and such topology is very popular in SM-SIMD architecture.

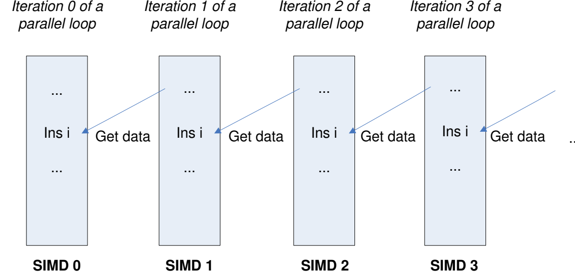

Fig. 1. Data pipeline.

In order to exploit the parallelism among multiple SIMD units, different iterations of a parallel loop are distributed to different SIMD units. Figure 1 shows different iterations of the parallel loop which are mapped to different SIMD units. If ins i executed on SIMD unit 0 can get its operand from the register of SIMD unit 1, ins i executed on SIMD unit k can get its corresponding operand from the register of SIMD unit k+1 as well. If these iterations can be scheduled consistently, data can be transferred among different SIMD units and thus be reused. Data flows through SIMD units as in a pipeline.

The program in Example code 2 is an example of such condition.

```
for (j=0; j<4; j++) {    // Loop2
    A[j,0,0:7] = m1[0,0:7] - m2[j,0:7];
    A[j,1,0:7] = m1[1,0:7] - m2[j+1,0:7];
    A[j,2,0:7] = m1[2,0:7] - m2[j+2,0:7];
    A[j,3,0:7] = m1[3,0:7] - m2[j+3,0:7];
}
```

Example code 2: An example for data pipeline.

In order to conveniently illustrate the problem in the following parts, we assume that there are 4 SIMD units in SM-SIMD architecture and it costs one cycle to finish the computation and getting data from the shared bus. If we schedule the code with the algorithm in [13] and distribute 4 iterations of Loop2 to 4 different SIMD units, 24 cycles are needed to finish 4 iterations of Loop2 (not including the cycles for writing back the results). However, 15 cycles are enough for the same work once data pipeline characteristics are exploited. The reason is that scheduling algorithm in [13] only exploits the parallelism based on replication, therefore only array m1 is reused. As a contrast, pipeline scheduling can not only exploit the reuse of array m1, but also reuse the elements of array m2 through data pipeline. Figure 2 illustrates the part of the execution process for the program.

3 Optimizing Algorithm

When there is a data pipeline between neighboring SIMD units, one SIMD unit should be the owner of an operand and the other is the consumer. In other words,

Cycle	SIMD 0	SIMD 1	SIMD 2	SIMD 3
1	*Load m2[0,0:7]*			
2		*Load m2[1,0:7]*		
3			*Load m2[2,0:7]*	
4				*Load m2[3,0:7]*
5	*Replicate m1[0,0:7]*			
6	m1[0,0:7]-m2[0,0:7]	m1[0,0:7]-m2[1,0:7]	m1[0,0:7]-m2[2,0:7]	m1[0,0:7]-m2[3,0:7]
7	*Get m2[1,0:7]*	*Get m2[2,0:7]*	*Get m2[3,0:7]*	*Load m2[4,0:7]*
8	*Replicate m1[1,0:7]*			
9	m1[1,0:7]-m2[1,0:7]	m1[0,0:7]-m2[2,0:7]	m1[0,0:7]-m2[3,0:7]	m1[0,0:7]-m2[4,0:7]
10	*Get m2[2,0:7]*	*Get m2[3,0:7]*	*Get m2[4,0:7]*	*Load m2[5,0:7]*
11	*Replicate m1[2,0:7]*			
12	m1[2,0:7]-m2[2,0:7]	m1[2,0:7]-m2[3,0:7]	m1[2,0:7]-m2[4,0:7]	m1[2,0:7]-m2[5,0:7]
13	*Get m2[3,0:7]*	*Get m2[4,0:7]*	*Get m2[5,0:7]*	*Load m2[6,0:7]*
14	*Replicate m1[3,0:7]*			
15	m1[3,0:7]-m2[3,0:7]	m1[3,0:7]-m2[4,0:7]	m1[3,0:7]-m2[5,0:7]	m1[3,0:7]-m2[6,0:7]

Fig. 2. Data pipeline scheduling.

after the data is used by the operation in one SIMD unit, the other SIMD unit can get it through the interconnection and reuse it. Such relationship among multiple operations executed in multiple SIMD units leads to a data pipeline and multiple SIMD units become the stages of data pipeline. In order to optimize the programs with such method, compilers need to identify data pipeline characteristics in the programs and schedule them based on the data pipeline flow relation. We call the data that can be transferred through interconnections as **pipe-data** and the two instructions, which use the pipe-data one after the other, as **pipeline instruction pair** in data pipeline optimization.

In order to implement this optimization, data pipeline optimizing performs the following steps, which is described in detail in the remainder of this section.

1. Determine candidate loop nests that will be executed on multiple SIMD units.
2. Analyze the live data to compute pipeline instruction pairs.
3. Determine the data flow directions of pipeline instruction pairs.
4. Eliminate redundant pairs which would cause unnecessary data transfers.
5. Transform some operations to communication operations.
6. Select the parallel loop, whose different iterations will be distributed to different SIMD units.
7. Allocate the resources for the iteration of the parallel loop.
8. Schedule the codes for multiple SIMD units.

3.1 Preliminary Optimizations

Code Partition Before our optimization, the programs have been already performed SIMD optimizations [14]. After SIMD optimizations, we use code partitioning to determine which segments of the program should be executed on

SM-SIMD architecture and which should be executed on the general purpose processor core. All sequential code, code for synchronization and controlling are mapped for execution on the general purpose processor core. The loops with SIMD operations are mapped for execution on SM-SIMD architecture.

Computation of Data Vector Reuse A data vector can be represented by four parameters: the data layout direction, the vector length, the address of its first element and the coefficient. Two data vectors are equal if and only if all these four parameters are equal. For two vectors of same array, when they have the same data layout direction and belong to the same uniformly generated set [19], when there is traditional temporal reuse between their first elements, their other corresponding elements also have traditional temporal reuse opportunity. Therefore, the first elements can be used as the representative elements of the data vectors to compute the data vector reuse under the constraints that these data vectors have the same data layout direction, the same vector length and the same uniformly generated set which their references are belonging to.

3.2 Live Data Analysis

In order to represent the instructions in pipeline instruction pairs, each instruction should have an exclusive symbol. Therefore, we construct the dependence directed acyclic graph (dependence-DAG) for the body of each candidate loop nest mapped to SM-SIMD architecture. Each SIMD instruction is assigned a sequential number based on its topological order in its individual dependence DAG.

If two instructions from a pipeline instruction pair are mapped to two neighboring SIMD units, they can communicate through the interconnection. To calculate the parallelism and perform the scheduling conveniently, a pipeline instruction pair should associated with several properties. We use the relation ⟨*first ins num, second ins num, dist, loop, array, subscript*⟩ to represent a pipeline instruction pair.

- *first ins num* is the smaller instruction number of the instructions in a pipeline instruction pair.
- *second ins num* is the larger instruction number of the instructions in a pipeline instruction pair.
- *loop* is the loop whose different iterations the pipeline instructions belong to.
- *dist* is the distance of loop iterations that carry this pipeline instruction pair.
- *array* is the array which the pipe-data belongs to.
- *subscript* is the subscript of pipe-data.

In a pair of two instructions, there are possibly more than one pipe-data among multiply operands. We mark each pipe-data in separate pipeline instruction pair.

3.3 Instruction Pair Direction

After pipeline instruction pairs are recognized, the data transfer direction that the pipe-data flows in a pipeline instruction pair should be determined. In other words, we need to decide which is the source of the instruction pair and which is the destination. In our algorithm, the data from a pipeline instruction pair flows from the instruction with smaller instruction number (first ins num) to the one with larger number (second ins num). The reasons are shown as follows. If data flows from an instruction with larger number to the one with smaller number, it is possible to have a cycle in the dependence DAG when a communication edge is added into the dependence DAG. Even a cycle is not involved, it is possible to lead to a deadlock when scheduling. Figure 3 is such a deadlock example (We assume the pipe-data flows along the direction of arrows. i, j, k and m are the instruction numbers of their corresponding instructions. Such representation will also be used in the following figures.). If instruction i needs the data of instruction m while instruction j is waiting for the data of instruction k, all of them would keep circular waiting and a deadlock would be formed. However, if the directions of data flow are reversed, the deadlock could be avoided. Moreover, if a data flows from an instruction with larger number to the one with smaller number, the scheduling of a lower level node in one dependence DAG depends on that of a deeper level node in the other dependence DAG. As a result, it is difficult to keep the load balance among the different DAGs when scheduling them to different SIMD units.

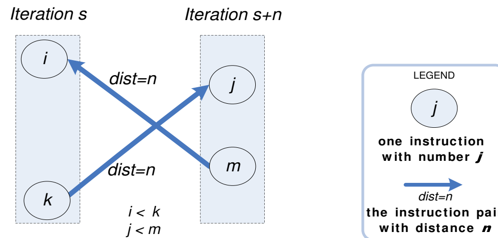

Fig. 3. A deadlock example.

As a result, the direction of pipeline instruction pair flows from a smaller instruction to a larger one. It is possible to have two instructions trying to share data with each other holding the same instruction number, which means they are two instances from the same instruction. However, we do not construct pipeline instruction pair for such instructions, because the data could be reused with the replication method in [13], as the case array m1 at cycle 11 shown in Figure 2. In other words, the first ins number will be always smaller than the second one in a pipeline instruction pair.

In the following parts of this paper, we also refer to the instruction with first ins num as the **start instruction** and the instruction with second ins num as the **end instruction**.

3.4 Redundant Communication Elimination

While pipeline instruction pairs are used, it is possible to have some redundant pipeline instruction pairs, which would cause unnecessary data communications thus should be eliminated. Figure 4 is such an example. In this example multiple instruction pairs share the same pipe-data. Assume instruction i and instruction j are the first pipe-data requiring instructions in those two iterations. After data communication is finished between them, all other instruction pairs are redundant because the pipe-data can be saved in the local register.

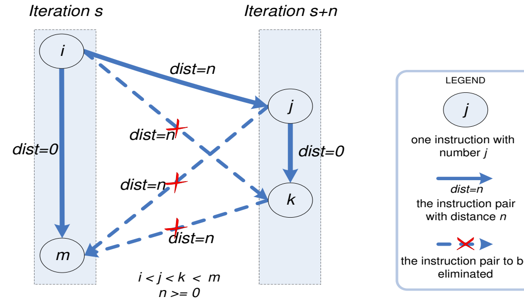

Fig. 4. An example of additional copy pair.

Indeed, for the same reuse data, it only needs to be transferred once between the neighboring dependence DAGs. Therefore, it is enough that only the pipeline instruction pair with the smallest instruction number in the different dependence DAGs is maintained. As the redundant pairs, other data pipeline pairs using this pipe-data can be eliminated. After the elimination, we add the communication edge into the dependence DAG for each pipeline instruction pair. The weight of each edge is the distance between them.

3.5 Computation-communication Transformation

Sometimes, some computing instructions can be transformed into communication operations. When two instructions satisfy the following conditions, one of the two instructions can be replicated by a communication operation. First, all operands of the two instructions are pipe-data. Second, their operations are the same. In such condition, we call the end instruction of the two instructions in the data pipeline pair as **comp-commu instruction**. A comp-commu instruction can be replicated by the operation that gets the result from the start instruction,

if the cycle of executing the comp-commu instruction is less than the cycle of getting the result directly. We use the following steps to process such condition. For a comp-commu instruction P:

1. Get the pipe-data that would be used by other pairs that P is the start instruction.
2. Compute the cycles (cyc_{comp}) that get all other operands (possibly no other operands need to be gotten) of P and finish the operation of P.
3. Compute the cycles (cyc_{comm}) that directly get the result of P.
4. Compare cyc_{comp} with cyc_{comm}. If $cyc_{comp} < cyc_{comm}$, we compute the result of P through step 2. Otherwise, we transform the operation of P into the operations in step 3.

3.6 Parallel Loop Selection

In this part, we select the loop whose different iterations would be distributed to multiple SIMD units. Before we select the loop, we compute the amounts of different distance replication data for each loop based on the algorithm in [13]. And then, a **replication weight** is assigned to each loop. The replication weight is the maximal amount value among different distance replication data. It can be represented as <*amount, rep-dist*>. rep-dist is the distance of replication data with the maximal amount. Then we select the loop, which is permutable with the innermost loop and with the maximal replication weight value, as parallel loop in order to utilize the parallelism based on replication as much as possible. If multiple loops have the same replication weights, we compute the amount of pipeline instruction pairs of these loops, whose distances are all equal to the value of rep-dist. We select the one with the maximal pipeline amount from the loops with the maximal replication weight as the **parallel distributed loop**.

Once the parallel distributed loop is selected, it is changed as the innermost parallel loop and performed loop mining optimization. The mining distance is equal to rep-dist. And only the pipeline instruction pair, which carried by parallel distributed loop and having the same distance with rep-dist, would be exploited in the scheduling algorithm.

3.7 Register Allocation

Once selecting the parallel distributed loop, we deal with the problem of limited register number in this section, we can allocate resource based on the requirement of the reuse data vector and average instruction parallelism in one iteration of the parallel distributed loop. We also use the interconnection characteristics in register allocation algorithm.

Register Requirement Before resource allocation, we first need to compute the register requirement of one iteration, which is the number of maximally simultaneously live variables. In order to compute this value, we construct the

interference graph based on the algorithm in [16]. The degree of a node in the interference graph represents the number of simultaneously live variables with this nod. Assume the degree of the interference graph is N, then the register requirement equals to N+1. Assume the total register number in N_r SIMD units can finish the allocation with no live data spilled out in one iteration of the distributed loop.

Resource Allocation If there does not exist instruction level parallelism in an iteration of the distributed loop, only regarding the SIMD units as register resources would waste the computation resources. Therefore we compute the average instruction parallelism of the iteration of the distributed loop for later resource allocation. Assume the value of the average SIMD instruction level parallelism is equal to N_i.

After computing average instruction parallelism, we allocate resources by considering both the instruction parallelism and the register requirement at the same time. We select the minimal value of N_i and N_r as the number for resource allocation. The main goal of our scheduling algorithm is to find as much parallelism as necessary to saturate the available hardware. Therefore, if there is instruction parallelism in an iteration with register pressure, exploiting some of them can also lower the pressure on the requirement of the registers and reduce the number of operations for spilled out. In other words, when guaranteeing the utilization of the computation resources, we also try to satisfy the register requirement of the reused data vectors because it can lower the competition for the shared data bus. Suppose N_s is allocated for each iteration. After the resource allocation for the single iteration of the distributed loop, we can get the value that how many iterations (N_p) of the distributed loop can be executed on SM-SIMD architecture. Namely, $N_p = \lfloor NUM_{SIMD}/N_s \rfloor$.

Once N_p is gotten, we do strip mining to adjust the innermost loop with only N_p iterations. After the strip mining, the innermost loop would be transformed into two parts. The one with N_p iterations would be the distributed loop in the innermost level, and the other part would be interchanged outside the localized vector space in order to maintain the locality in the localized vector space.

3.8 Scheduling Algorithm

Once the previous steps are finished, we schedule the code for SM-SIMD architecture. We use a scheduling algorithm to generate codes for SM-SIMD architecture. The scheduling algorithm itself is known to be an NP-complete problem. We propose a heuristic algorithm. According to the scheduling algorithm, the parallel distributed loop is unrolled by a factor of N_p and distributed to the corresponding allocated resources. The scheduling algorithm is shown as follows.

- Construct the dependence DAG of an iteration of the parallel distributed loop and make $N_p - 1$ additional copies to form a parallel dependence DAG, whose N_p sub-DAGs correspond to the DAGs of N_p iterations of the parallel distributed loop.

– Mark the replication parallel point and the locality parallel point in those sub-DAGs based on the algorithm in [13].
– Add **communication node** between the instructions in the same pipeline instruction pair and connect communication nodes with the start instruction node and the end instruction node. The directions of the connecting edges are the same as the ones of the corresponding pipeline instruction pairs. When scheduling codes, we also use the communication node as the synchronization node.
– The DAG is traversed to generate code for SM-SIMD architecture. When traversing the DAG, the multiple sub-DAGs have the arbitrary scheduling sequence before reaching a synchronization point instruction. If an instruction is not marked as a synchronization point, all its instances mapped to different SIMD units would be executed in sequence. If one of sub-DAGs reaches a synchronization point, we stop the scheduling and move to the next one until all the scheduling of the sub-DAGs reach there or no other synchronization point can be reached. Then we generate a parallel control instruction for all different instances of this synchronization instruction and execute them in parallel on different SIMD units, if the type of the synchronization point is not communication node. Otherwise, a communication VLIW instruction is generated to make the multiple SIMD units get data from their neighboring SIMD units.
– Repeat this process until the scheduling is finished.

4 Experimental Results

We implement a detailed performance simulator based on Morphosys[3] [2] by extending Simplescalar-3.0d. Morphosys is chosen as a basic underlying hardware because of the following reasons. First, it is a typical SM-SIMD architecture and some industry SOCs are implemented using the similar techniques as in Morphosys. Second, many detailed resources about its design are available. Therefore, it can make the simulation more faithful.

Before evaluating the experiments, we first analyze the benchmarks used in [20] and [21]. Only the benchmarks with inherent data pipeline characteristics are selected because it is unnecessary to optimize the programs without such characteristics. We select *impeg2* from BMW (Berkeley Multimedia Workload) [20], *me*, *cfa* and *dct* programs in [21] as test benchmarks. However, there are some sequential optimizations in *impeg2*, which impede the exploitation for the parallelism. The program is originally programmed for general purpose processor (GPP) platforms. Due to the limited computational resource in GPP, programmers try to improve performance by reducing the proportion of computation, for example by adding extra if statement for some specific inputs and return

[2] There are 8 SIMD units in Morphosys. Each SIMD unit consists of eight 32-bit processing element and the register file of each SIMD units includes 4 registers. Eight SIMD units shared one 256-bit data bus and the communication channel between two neighboring SIMD units is 128-bit.

their results in order to skip the complex computation parts. Such techniques indeed speed up those programs on GPP platforms. However, they are impeding compilers to exploit the parallelism in the programs. We rewrite a new *impeg2* version - *impeg2pa*, which remains the original application algorithm but with no extra sequential optimizations.

In the experiments, we optimize these five programs in two methods and compare the performance of the optimized programs. In order to have a uniform criteria, all speedups are computed through the results of extension architecture divided by sequential results. In order to compare the results of *impeg2* and *impeg2pa*, we compute their speedups against the same sequential program. The optimizing methods are:

- Automatically scheduling with *Agassiz*[3]: Agassiz converts the original programs into the optimized ones with embedded assembly codes through the algorithm in [13]. The GCC compiling tool chains of SimpleScalar can thus generate the machine codes to run on the simulator.
- Manually scheduling with data pipeline: We optimize the programs with data pipeline optimization for SM-SIMD architecture by embedding assembly instructions manually, then compile the optimized programs with GCC compiling tool chains of SimpleScalar.

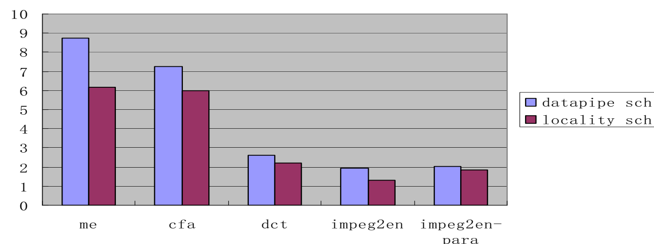

Fig. 5. Speedup comparison

The experimental results are shown in figure 5.There are eight SIMD units in Morphosys that we use as underlying architecture. Average utilization of the SIMD units in SM-SIMD architecture is used to illuminate the average busy ratio of eight SIMD units after scheduling. Avg P-SIMD shows how many instructions that SM-SIMD architecture can averagely finish in one cycle. First, Avg P-SIMD is computed. We gather the total cycles (C) consumed by SM-SIMD architecture not including the idle cycles and also the total number of the SIMD instructions (I) in each program. The average amount of SIMD instructions that SM-SIMD

[3] Agassiz is a source to source compiler tool for C programs developed by Department of Computer Science and Engineering, University of Minnesota at Twin-Cities and Parallel Processing Institute, Fudan University.

architecture can execute per cycle can be computed through the equation Avg P-SIMD = I / C. Thus the average utilization equals to (Avg P-SIMD/8)*100%. The detailed results are shown in Table 2.

Table 2. The average utilization of SM-SIMD architecture

	cfa	*me*	*dct*	*impeg2*	*impeg2pa*
Avg P-SIMD data pipe	6.173	5.924	4.274	2.2085	5.130
Avg P-SIMD locality sche	2.381	2.267	3.200	1.580	1.580
Avg Util data pipe	77.2%	74.1%	53.4%	27.6%	64.1%
Avg Util locality sche	29.8%	28.3%	40%	19.7%	19.7%

From the results of speedup and average utilization, one of the observations is that data pipeline scheduling can get higher speedup and better average utilization for the applications with inherent data pipeline characteristics. The core parts of *me* and *cfa* are similar and are particularly suitable for data pipeline optimization, therefore their speedups are perfect. For the application of *dct*, its speedup is lower because the optimized proportion in its code is smaller. *me* is the core of *impeg2* algorithm. But in *impeg2* there exists many sequential optimizations which cause large obstacles to parallel optimizing. Moreover, the sequential part for controlling in *impeg2* is much more than that in *me*, thus the speedup of *impeg2* is lower than *me*.

Another interesting observation is that the speedup of *impeg2pa* is 14% higher than that of *impeg2*. The reason is that there are no sequential optimizations in *impeg2pa* which impede the exploitation for the parallelism in the program. The inherent data pipeline characteristics in *impeg2pa* can be fully exploited by the data pipeline optimization, therefore the speedup of *impeg2pa* is higher than that of *impeg2*. Based on this observation, we think sometimes it is better to write the application programs according to their original algorithms, which is easier for compilers to perform scheduling and generate better optimized codes. Otherwise, the codes of some applications should be re-written for higher performance.

5 Conclusions

The experimental results demonstrate that data pipeline optimization techniques are very effective to optimize real-life applications. Furthermore, when writing programs for SM-SIMD architecture, it is better to remain the original structure according to the application algorithms, which is much easier for compilers to exploit the parallelism in the programs and thus generate better codes.

References

1. Diefendorff, K.; Dubey, P.K. "How multimedia workloads will change processor design" Computer, Sept., 1997, PP. 43-45
2. Rixner, S.; Dally, W.J. "Register organization for media processing", 6the International Symposium on High-Performance Computer Architecture, 2000, 375-386

3. H. Singh, M. H. Lee, N. Bagherzadeh, MorphoSys: An Integrated Reconfigurable System for Data-Parallel and Computation-Intensive Applications. IEEE Transaction on Computers, 2000, 49, 5, 465-481
4. Xiaofang Wang; Ziavras, S.G, A framework for dynamic resource assignment and scheduling on reconfigurable mixed-mode on-chip multiprocessors, IEEE International Conference on Field-Programmable Technology, 2005 51 - 58
5. K.B. Dally, W.J. Kapasi, J. Owens, J.D. Towles, B. Chang, A. Rixner, S. Imagine: media processing with streams, IEEE Micro, 2001, 21(2), pp.35-46
6. http://www.motorala.com
7. Eric S. Gayles, Thomas P. and Mary Jane Irwin. The Design of the MGAP-2: A Micro-Grained Massively Parallel Array. IEEE Transaction on Very Large Scale Integration(VLSI) Systems, Vol. 8, No. 6, Dec 2000
8. T. Komuro, M. Ishikawa. A Dynamically Reconfigurable SIMD Processor for a Vision Chip. IEEE Journal of Solid-State Circuits, Vol. 39, No. 1, Jan 2004
9. J. Gebis, S. William, C. Kozyrakis, D. Patterson, VIRAM1: A Media-Oriented Vector Processor with Embedded DRAM", 41st Design Automation Student Design Contenst, San Diego, CA, June 2004
10. H.Peter Hofstee, Power Efficient Processor Architecture and The Cell Processor, 11th International Conference on High-Performance Computer Architecture, San Francisco,USA,February 2005
11. G. Venkataramani, W. Najjar, F. Kurdahi, N. Bagherzadeh. Automatic compilation to a coarse-grained reconfigurable system-opn-chip. November 2003 ACM Transactions on Embedded Computing Systems (TECS), Vol.2 Issue 4
12. P. Mattson, W. J. Dally, S. Rixner, U. J. Kapasi, J. D. Owens Communication Scheduling, Proceedings of the ninth international conference on Architectural support for programming languages and operating systems Nov. 2000
13. W. Zhang, X. Qian, Y. Wang, B. Zang, C. Zhu, Optimizing Compiler for Shared-Memory Multiple SIMD Architecture, ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems 2006, Ottawa, Canada.
14. W.H Jiang, C. Mei, B. Huang, B.Y Zang, C.Q Zhu "Boosting the Performance of Multimedia Applications Using SIMD Instructions" The 15th International Conference on Compiler Construction. April 2005 Edinburgh, Scotland
15. David A. Padua and Michael J.Wolfe, Advanced Compiler optimizations for Supercomputers, Communications of the ACM, 29(1986) 1184-1201.
16. S.S. Muchnick. Advanced Compiler Design and Implementation, Mor Kaufma, 1997.
17. A.Capitanio, N.Dutt, and A.Nicolau, "Partitioned register files for VLIWs: A preliminary analysis of trade-offs." Proceedings of the 25th Annual International Symposium on Microarchitecture, Dec., 1992, pp. 292-300.
18. M.Fernandes, J.Llosa, and N.Topham, Distributed modulo scheduling." Proceedings of the 5th Annual International Conference on High Performance Computer Architecture, Jan., 1999, pp. 130-134.
19. M. E.Wolf, M.S.Lam, A Data Locality Optimizing Algorithm, ACM SIGPLAN Conference on Programming Language Design and Implementation. 1991, 30-44.
20. N.T.Slingerland and A.J. Smith, Multimedia Instruction Sets for General Purpose Microprocessors : A Survey. Technical Report CSD-00-1122, Univ. of California at Berkeley Computer Science, Dec. 2000.
21. D. Talla, L.K. John, and D.C. Burger. Bottlenecks in Multimedia Processing with SIMD-Style Extensions and Architectural Enhancements, IEEE Transactions on Computers, August, 2003, 52(8), pp.1015-1031.