Optimizing Address Assignment for Scheduling Embedded DSPs *

Chun Xue[1], Zili Shao[1], Edwin H.-M. Sha[1], and Bin Xiao[2]

[1] University of Texas at Dallas
Richardson, Texas 75083, USA
{cxx016000, zxs015000, edsha}@utdallas.edu
[2] Hong Kong Polytechnic University
Hung Hom, Kowloon, Hong Kong
csbxiao@comp.polyu.edu.hk

Abstract. DSP architecture typically provides indirect addressing modes with auto-increment and auto-decrement. Subsuming the address arithmetic into auto-increment and auto-decrement modes improves the size and performance of generated code. A lot of previous work has been done on address assignment optimization to achieve code size reduction by minimizing address operations for single functional unit processors. However, minimizing address operations alone may not directly reduce code size and schedule length for multiple-functional-unit processors. In this paper, we exploit address assignment and scheduling for multiple functional units processors. Our approach is to first construct a nice address assignment and then do scheduling. By fully taking advantage of the address assignment during scheduling, code size and schedule length can be significantly reduced. We propose a multiple-functional-unit algorithm to do both address assignment and scheduling. The experimental results show that our algorithm can greatly reduce code size and schedule length compared to the previous work.

1 Introduction

Microprocessors such as microcontrollers and digital signal processors(DSPs) are increasingly being used in embedded systems. Two major goals in embedded system design are to improve timing performance and to reduce code size for applications. DSP processors, such as TI TMS320C2x/5x/6x, AT&T DSP 16xx, provide dedicated address generation units (AGUs) for address calculation. AGUs can be used to reduce the number of address arithmetic instructions with the auto-increment and auto-decrement capabilities. When auto-increment or auto-decrement is used in an instruction, the value of the address register is modified in parallel with the instruction, hence the next instruction is ready to

* This work is partially supported by TI University Program, NSF EIA-0103709, Texas ARP 009741-0028-2001, NSF CCR-0309461, USA, and HK POLYU A-PF86 and COMP 4-Z077, HK.

be executed without extra instruction. With a careful placement of variables in memory, we can reduce the total number of the address arithmetic instructions, and in return, both the code size and the timing performance can be improved. Address assignment, i.e., optimization of memory layout of program variables for single functional unit processor has been studied extensively. However, for multiple functional units processors, little research has been done. In this paper, we focus on scheduling and address assignment for multiple functional units processors.

The address assignment was first studied by Bartley [2] and Liao et al.[1]. They modeled the problem as a graph theoretic optimization problem. Liao et al. formulated the address assignment problem as a Maximum Weighted Path Covering (MWPC) problem. Since the problem is NP-hard, a greedy heuristic algorithm based on Kruskal's maximum spanning tree algorithm was proposed to solve the problem. In their paper, both simple offset assignment (SOA) which use single address register and general offset assignment (GOA) which uses multiple address registers are discussed. Leupers and Marwedel [3] have extended the work done by Liao et al. by proposing a tie-breaking heuristic and a variable partitioning method to improve the quality of SOA/GOA solution. Leuper and David [4] have solved GOA problem with arbitrary register file sizes and auto-increment ranges. Gebotys [5] modeled the problem of assigning address registers to every variable given a fixed memory layout as a network flow problem and solved it optimally. All of these work that has been done is given an exact access sequence of the program variables. Address assignment is solved after the scheduling is completed. The goal of all these work is to minimize address operations to achieve code size reduction and performance improvement. It works on single functional unit processors. However, minimizing address operations alone may not directly reduce code size and schedule length for multiple functional units processors.

Some work has been done on combining scheduling and address assignment in code generation. Rao et al.[7] suggested modifying the variable access sequence using expression tree transformations and formulated it as the Least Cost Access Sequence(LCAS) problem and developed a heuristic algorithm to solve it. However, their solution is confined to single functional unit and single address register problem only. Lim et al.[8] addressed scheduling effect on the SOA problem. Their approach aimed to make graph sparser by an exhaustive search algorithm with pruning techniques. However, it is not true that sparser graphs always lead to cheaper MWPC cost than that of denser graphs.[9] Choi and Kim [9] proposed an algorithm that tightly couples offset assignment problem with scheduling to exploit scheduling on minimizing address instructions more effectively. Their algorithm only targets single functional unit and can not be directly applied to multiple functional units.

In this paper, we exploit the address assignment problem with scheduling for multiple functional units processors. In our approach, we first construct a nice address assignment and then do scheduling. In this way, we can fully take advantage of the obtained address assignment to significantly reduce code size

and schedule length when performing scheduling. Based on this method, we propose an algorithm which does address assignment and scheduling for multiple functional units processors. The experimental results show an average of 14%-18% code size and schedule length reduction compared with the traditional list scheduling, and an average of 7%-10% code size and schedule length reduction compared with the algorithm that directly applies Solve-SOA [1].

The remainder of the paper is organized as follows. Section 2 introduces the basic concepts and architecture model. Section 3 provides motivating examples. The algorithm are discussed in Section 4. Experimental results and concluding remarks are provided in Section 5 and 6, respectively.

2 Processor Model and Notations

The process model we use in this paper is given as follows. For each functional unit in a multiple functional units processor, there is an accumulator and an address register. Each operation involves an accumulator and, if any, another operand from the memory. Memory access can only occur indirectly via address registers, AR0 through ARk. Furthermore, if an instruction uses ARi for indirect addressing, then in the same instruction, ARi can be optionally post-incremented or post-decremented by one without extra cost. If an address register does not point to the desired location, it may be changed by adding or subtracting a constant, using the instructions ADAR and SBAR. The input of our algorithm is a DAG. A *Directed Acyclic Graph (DAG)*, $G = \langle V, E \rangle$, is a node-weighted graph, where V is the set of nodes and each node represents a computation, and $E \subseteq V * V$ is the edge set and an edge between two nodes denotes a dependency relation.

3 Motivating Examples

In this section, we provide a motivating example. For a given DAG, we compare the schedule length and code size generated by list scheduling, the Solve-SOA algorithm[1], and our algorithm.

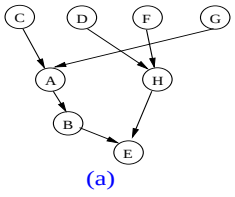


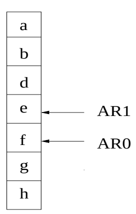
Fig. 1. (a)Input DAG (b) Function in each node

The input DAG that is shown in Figure 1(a) will be used through out this paper. Each node in the DAG is a computation. For example, node B denotes the computation of e = a + h. The list of each node and their corresponding computation is shown in Figure 1 (b).

Assume we have two functional units in our system. Using list scheduling that sets the priority of each node as the longest path from this node to a leaf node, we obtained the schedule shown in Figure 2-(1). The address assignment is simply using the alphabetical order shown in Figure 2-(2). The detail assembly code for this schedule is shown in Figure 2-(3). Each node in the scheduling in 2-(1) corresponds to several assembly instructions in 2-(3) to complete the computation defined by this node. For example, node C in 2-(1) corresponds to assembly code from line 0 to line 4 of functional unit 2 in 2-(3) that computes d = e + f. The resulting schedule length is 24 as shown in Figure 2.

FU1	G	A	B	H	E
FU2	C	D	F		

(1)



(2)

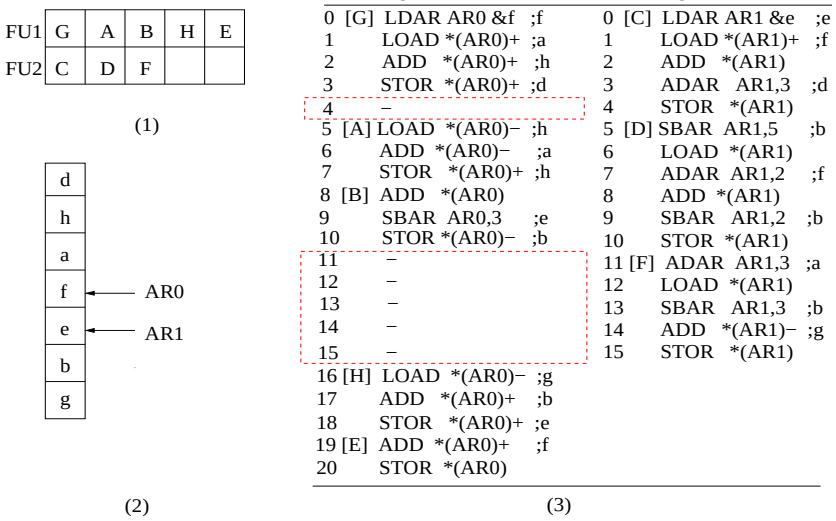
FU1	FU2
0 [G] LDAR AR0 &f ;f	0 [C] LDAR AR1 &e ;e
1 LOAD *(AR0)	1 LOAD *(AR1)− ;f
2 ADAR AR0,5 ;a	2 ADD *(AR1)
3 ADD *(AR0)	3 ADAR AR1,2 ;d
4 SBAR AR0,7 ;h	4 STOR *(AR1)
5 STOR *(AR0)	5 [D] ADAR AR1,2 ;b
6 [A] ADAR AR0,4 ;d	6 LOAD *(AR1)
7 LOAD *(AR0)	7 SBAR AR1,4 ;f
8 SBAR AR0,4 ;h	8 ADD *(AR1)
9 ADD *(AR0)	9 ADAR AR1,4 ;b
10 ADAR AR0,7 ;a	10 STOR *(AR1)+ ;a
11 STOR *(AR0)	11 [F] LOAD *(AR1)− ;b
12 [B] SBAR AR0,7 ;h	12 ADD *(AR1)
13 ADD *(AR0)	13 SBAR AR1,5 ;g
14 ADAR AR0,3 ;e	14 STOR *(AR1)
15 STOR *(AR0)	
16 [H] ADAR AR0,3 ;b	
17 LOAD *(AR0)	
18 SBAR AR0,5 ;g	
19 ADD *(AR0)	
20 ADAR AR0,5 ;b	
21 STOR *(AR0)	
22 [E] SBAR AR0,3 ;e	
23 ADD *(AR0)− ;f	
24 STOR *(AR0)	

(3)

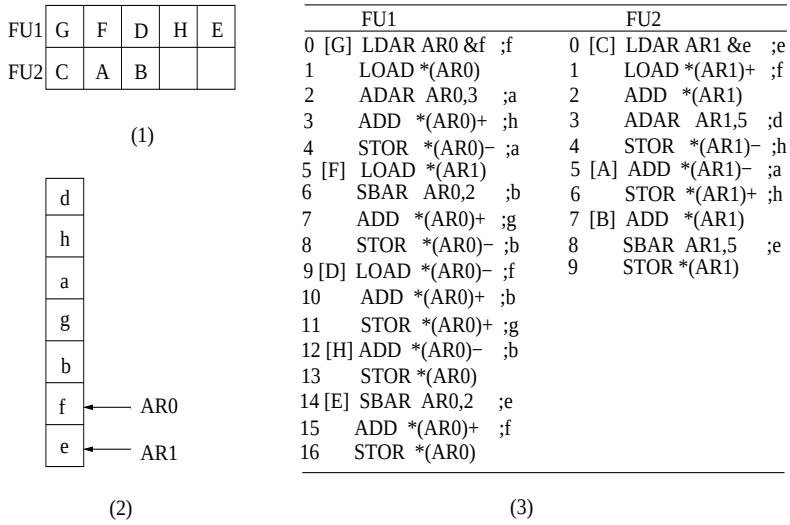
Fig. 2. List Scheduling without address assignment. Total schedule length=24 (1)Node Scheduling (2)Address Assignment (3)Assembly Code

Based on the scheduling from 2, the Solve-SOA algorithm [1] is applied to generate a better address assignment as shown in Figure 3-(a)-(2). With this new address assignment, some address arithmetic operations are saved. We obtain a schedule with a total schedule length of 20 as shown in Figure 3-(a)-(3).

As we can see from the schedule with assembly code in Figure 3-(a)-(3), after applying address assignment using the SOA algorithm [1], the number of address operations are reduced, namely ADAR and SBAR. However, the total schedule length is not reduced as much as the number of address operations. For multiple functional units, even we save address operations in one function unit, we may not reduce the total schedule length or code size because of the dependency constraints as shown in the dashed boxes in Figure 3-(a)-(3). It is because that the other functional unit may have operations in parallel that has not been completed yet. This characteristic of multiple functional units imply that we can not achieve good result with a fixed schedule.



(a) Scheduling with SOA address assignment. Total schedule length=20
(1)Node Scheduling (2)Address Assignment (3)Assembly Code



(b) Scheduling with new address assignment. Total schedule length=16
(1)Node Scheduling (2)Address Assignment (3)Assembly Code

Finally, the schedule generated by our algorithm is shown in Figure 3-(b). With a different address assignment as shown in Figure 3-(b)-(2), we generate a schedule with schedule length of 16.

From this example, we can clearly see that minimizing address operations alone can not directly reduce schedule length and code size for multiple functional units processors. In this paper, we use an approach that does address assignment first and then perform scheduling based on the obtained assignment to reduce code size and schedule length.

4 Address Assignment and Scheduling for DSP

In this section, we first analyze the relationship between scheduling and address assignment, and show how to generate an address assignment before scheduling. Then, we propose an algorithm for multiple functional units processors.

4.1 Address Assignment before Scheduling

Most of previous work [1,5,6,7,8,9] considers the address assignment after the scheduling is done. That is, given a known access sequence based on a fixed schedule, find a good address assignment that reduces the number of address operation code. As shown in Section 3, minimizing address operations alone can not directly reduce schedule length and code size for multiple functional units processors. We use an approach that obtains an address assignment first and then perform scheduling based on the address assignment to solve this problem. Next we show how to generate a good address assignment given a DAG as input. We propose an algorithm mSOA which improves the Solve-SOA algorithm [1] so that it can handle partial access sequence. The algorithm is shown in Algorithm 4.1.

Algorithm 4.1 mSOA(Modified Solve-SOA)

Require: DAG $G = \langle V, E \rangle$, i number of functional units
Ensure: An address assignment
  for all $u \in G$ **do**
    access_sequence += access_sequence(u) + " | "
  end for
  access_graph ← Generate_Access_Graph(access_sequence)
  address_assignment ← Find_Maximum_Weight_Path_Cover(access_graph)

The input of Solve-SOA is a complete access sequence based on a fixed schedule. In our algorithm, the schedule is not known yet. So we will have a partial access sequence as the input which only includes access sequence within each node. To represent this type of partial access sequence, the input access sequence of mSOA includes a special symbol " | ", which means there is no relation between the two neighbor variables.

The mSOA algorithm takes this partial access sequence as input, and generate an address assignment as output. In order to generate the output, we first construct an access graph G=(V,E) from the partial access sequence input. In

this access graph G, each node $v \in V$ corresponds to a unique variable in the partial access sequence. An edge $e(u, v) \in E$ exists with weight w(e) if variable u and v are adjacent to each other w(e) times in the partial access sequence. If there is a " | " symbol between u and v in the partial access sequence, it means u and v are not adjacent to each other, so it will not be counted in the weight of w(e). From this access graph G, we find a Maximum Weight Path Cover using the Solve-SOA algorithm [1]. With this path cover, we obtain the address assignment output.

The cost we obtain in mSOA is actually the *Lower Bound* of the number of address operations for any schedule. The reason is as follows. As we perform scheduling, we may add more weight and edges to the access graph constructed in mSOA. No matter what we do in the scheduling step, we will not remove any edges or weight from this access graph. Hence we will never have an address assignment that have a lower cost then the cost we obtain in mSOA. With this lower bound, our goal in the scheduling step is to leverage the generated address assignment as much as possible.

4.2 Algorithm for Multiple Functional Units

In this section, we propose an algorithm, Multiple Functional Unit Scheduling (MFSchSOA), to minimize the schedule length and code size by minimizing address operations for multiple functional units. MFSchSOA algorithm is shown in Algorithm 4.2.

Due to the the dependency in a DAG, we can only schedule a node after all its parent nodes have been scheduled. The scheduling problem with address operation minimization is to find a matching between available functional units and ready nodes in such a way that the schedule based on this matching minimizes the total number of address operations in every scheduling step. This is equivalent to the min-cost weighted bipartite matching problem. Thus, in MFSchSOA algorithm, we repeatedly create a weighted bipartite graph G_{BM} between the set of available functional units and the set of nodes in the Ready_List, and assign nodes based on the min-cost maximum bipartite matching M. In each scheduling step, the weighted bipartite graph, $G_{BM} =< V_{BM}, E_{BM}, W >$, is constructed as follows: $V_{BM} = FU\_SET \cup L_{RD}$ where $FU\_SET =< F_1, F_2, \cdots, F_N >$ is the set of currently available functional units and L_{RD} is the set of all nodes in Ready_List; for each FU $F_i \in FU\_SET$ and each node $u \in L_{RD}$, an edge $e(F_i, u)$ is added into E_{BM} and $W(F_i, u) = WCF(Last\_Var(F_i), First\_Var(u), Priority(u))$, where $Last\_Var(F_i)$ is the last variable accessed in the functional unit i, $First\_Var(u)$ is the first variable that will be accessed by node u. Priority(u) is the longest path from node u to a leaf node. $WCF(X, Y, Z)$ is a weight function as defined as follows:

$$WCF(X, Y, Z) = \begin{cases} Z - 2 & \text{the distance between X and Y} = 0 \\ Z - 1 & \text{the distance between X and Y} = 1 \\ Z & \text{Otherwise} \end{cases}$$

In this way, ready nodes with highest priority are considered first. Given the same priority, nodes with address operation savings have more advantage.

Algorithm 4.2 MFSchSOA(Multiple Functional unit Scheduling with SOA)

Require: DAG $G = \langle V, E \rangle$, i number of functional units
Ensure: A schedule with minimum schedule length
  $s \leftarrow$ Generate_Seq(Dag)
  $a \leftarrow$ mSOA(s)
  for all $u \in G$ **do**
    $priority(u) \leftarrow$ the longest path to leaf from u;
  end for
  $Ready\_Set \leftarrow$ all nodes that are ready to be scheduled in G;
  while $Ready\_Set \neq \phi$ **do**
    $Cur\_avail\_FU \leftarrow$ current available Functional units ;
    Construct $G_{BM} = \langle V_{BM}, E, W \rangle$, where: $V_{BM} = FU\_SET \cup L_{RD}$; $E = \{(F_i, u) | \forall F_i \in FU\_SET, u \in L_{RD}\}$ and $W(F_i, u)) = WCF(Last\_Var(F_i), First\_Var(u), Priority(u))$; where:

$$WCF(X, Y, Z) = \begin{cases} Z - 2 \text{ the distance between X and Y} = 0 \\ Z - 1 \text{ the distance between X and Y} = 1 \\ Z \quad \text{Otherwise} \end{cases}$$

    $M \leftarrow Min\_Cost\_Bipartite\_Matching(G_{BM})$;
    for all $e(F_i, u) \in M$ **do**
      Schedule Node u to functional unit F_i
    end for
    $Ready\_Set \leftarrow$ all nodes that are ready to be scheduled in G;
  end while

The technique proposed by Fredman and Tarjan [10] can be used to obtain a min-cost maximum bipartite matching in $O(n^2 \log n + nm)$, where n is the number of nodes and m is the number of edges of a bipartite graph. Let N be the number of functional units. In every scheduling step, we need at most $O(|V|^2 \log |V| + |V|^2)$ to find a minimum weight maximum bipartite matching, since the number of nodes is $N + |V|$ and the number of edges is $N * |V|$ in the bipartite graph. Thus, the complexity of MFSchSOA is $O(|V|^3 \log V)$, since the scheduling step is at most $|V|$ and N is a constant.

5 Experiments

In this section, we experiment with our algorithms on a set of benchmarks programs. The algorithms are implemented in C language on redhat linux platform. The input of our experiments are dependency graph of operations, and the output is the schedule length with the detailed schedules.

We compare our MFSchSOA with the list scheduling, and the algorithm that directly applies Solve-SOA on multiple functional units processors. Table 1 to table 3 shows the comparison of the results in terms of total schedule length among list scheduling (Column "List_Sch"), simSOA(Column "simSOA") and

MFSchSOA (Column "MFSchSOA") when the number of functional units equal 2, 3, and 4 respectively.

The experimental results show an average of 14%-18% code size and schedule length reduction compared with the traditional list scheduling, and an average of 7%-10% code size and schedule length reduction compared with the algorithm that directly applies Solve-SOA [1].

6 Conclusion

In this paper, we show that we can improve both performance and code size when we combine scheduling with address assignment for multiple functional units DSP processors. Specifically, we can generate an address assignment, and then utilize this address assignment during the scheduling. Hence we minimized the address operations needed and significantly reduced schedule length. In this paper, we did not consider the effect of reordering of input-operands of commutative operations, and possible application of multiple address registers per functional unit, which will be our next research topic.

References

1. Liao, S., Devadas S., Keutzer K., Tjiang S., Wang A.: Storage Assignment to Decrease Code Size. ACM Transactions on Programming Languages and Systems (TOPLAS) **18** (1996) 235-253
2. Bartley D.: Optimizing stack frame accesses for processors with restricted addressing modes. Sofware-Practice & Experience John Wiley & Sons, Inc. (1979)
3. Leupers R., Marwedel P.: Algorithm for Address Assignment in DSP code Generation. IEEE/ACM International conference on Computer-aided design November (1996) 109-112
4. Leupers R., David F.: A Uniform Optimization Technique for Offset Assignment Problem. International Symposium on System Synthesis December (1998) 3-8
5. Gebotys C.: DSP Address Optimization using a Minimum Cost Circulation Technique. IEEE/ACM International conference on Computer-aided Design November (1997) 100-103
6. Saip H., Lucchesi C.: Matching algorithm for bipartite graphs. DCC-93-03(Departamento de Cincia da Computao, Universidade Estudal de Campinas)", http://www.dee.unicamp.br/ic_tr_ftp/ALL/Abstrace.html March (1994)
7. Rao A., Pande S.: Storage Assignment using Expression Tree Transformation to Generate Compact and efficient DSP Code. ACM SIGPLAN on Programming Language design and implementation (1999)
8. Lim S., Kim J., Choi K.: Scheduling-based Code Size Reduction in Processors with Indirect Addressing Mode. International Symposium on Hardware/software Codesign April (2001) 165-169
9. Choi Y., Kim T.: Address Assignment Combined with Scheduling in DSP Code Generation. ACM IEEE Design Automation Conference June (2002) 225-230
10. Fredman M., Tarjan R.: Fibonacci Heaps and Their Uses in Improved Network Optimization Algorithms. Journal of the ACM 34 (1987) 596-615

FUs=2					
Benchmarks	**List_Sch**	**simSOA**	**MFSchSOA**	**%LS**	**%SOA**
IIR Filter	24	21	19	**20.8**	**9.5**
IIR-UF2	44	41	39	**11.3**	**4.9**
IIR-UF3	65	62	58	**10.7**	**6.5**
Diff. Equ.	28	24	23	**17.9**	**4.2**
All-Pole	67	55	53	**20.9**	**3.6**
4-stage Lat	70	65	59	**15.7**	**9.2**
8-stage Lat	119	108	106	**10.9**	**1.8**
Elliptical	107	106	89	**16.8**	**16.0**
Voltera	73	71	66	**9.6**	**7.0**
Average Reduction				**14.9**	**7.0**

Table 1. The comparison on schedule length for MFSchSOA, simSOA, and List Scheduling when there are 2 functional units.

FUs=3					
Benchmarks	**List_Sch**	**simSOA**	**MFSchSOA**	**%LS**	**%SOA**
IIR Filter	22	19	17	**22.7**	**10.5**
IIR-UF2	30	27	26	**13.3**	**3.7**
IIR-UF3	44	42	40	**10.0**	**4.8**
Diff. Equ.	19	19	16	**15.8**	**15.8**
All-Pole	67	54	51	**23.9**	**5.6**
4-stage Lat	51	47	43	**15.7**	**8.5**
8-stage Lat	94	89	83	**11.7**	**6.7**
Elliptical	83	75	72	**13.3**	**4**
Voltera	62	62	55	**11.3**	**11.3**
Average Reduction				**15.3**	**7.9**

Table 2. The comparison on schedule length for MFSchSOA, simSOA and List Scheduling when there are 3 functional units.

FUs=4					
Benchmarks	**List_Sch**	**simSOA**	**MFSchSOA**	**%LS**	**%SOA**
IIR Filter	22	18	14	**36.4**	**22.2**
IIR-UF2	22	21	20	**9.0**	**4.8**
IIR-UF3	38	35	32	**15.8**	**8.6**
Diff. Equ.	19	18	16	**15.8**	**11.1**
All-Pole	67	54	49	**26.7**	**9.3**
4-stage Lat	49	46	39	**20.4**	**15.2**
8-stage Lat	96	86	81	**15.6**	**5.8**
Elliptical	80	74	72	**10.0**	**2.7**
Voltera	62	58	52	**16.1**	**10.3**
Average Reduction				**18.4**	**10.0**

Table 3. The comparison on schedule length for MFSchSOA, simSOA and List Scheduling when there are 4 functional units.