International Journal of Internet, Broadcasting and Communication Vol.9 No.1 29-34 (2017)
https://doi.org/10.7236/IJIBC.2017.9.1.29

IJIBC 17-1-5

A Study on Efficient Use of Dual Data Memory Banks in Flight Control Computers

Doosan Cho

Electrical & Electronic Engineering, Sunchon National University, Korea
dscho@scnu.ac.kr

Abstract

Over the past several decades, embedded system and flight control computer technologies have been evolved to meet the diverse needs of the mobile device market. Current embedded systems are at the heart of technologies that can take advantage of small-sized specialized hardware while still providing high-efficiency performance at low cost. One of these key technologies is multiple memory banks. For example, a dual memory bank can provide two times more memory bandwidth in the same memory space. This benefit take lower cost to provide the same bandwidth. However, there is still few software technologies to support the efficient use of multiple memory banks. In this study, we present a technique to efficiently exploit multiple memory banks by software support. Specifically, our technique use an interference graph to optimally allocate data to different memory banks by an optimizing compiler. As a result, the execution time can be improved upto 7% with the proposed technique.

Keywords: *Optimization, Low Power, Embedded System, Data Memory, Performance*

1. Introduction

Many embedded applications operate in a real-time environment. This means that the embedded system must provide a certain level of performance to meet the real-time constraint. To satisfy this requirement, embedded system should be designed to an application-specific hardware including programmability. Normally, it achieved by a compiler for optimizing the use of such specialized hardware. Optimizing compilers consist of various hardware-specific optimization techniques such as instruction scheduling and register allocation, but there are still empty space for optimizing memory system such as multiple memory banks.

It is a well-known fact that there are lots of data parallelism in multimedia applications. Figure 1 shows some frequently used code for multimedia applications. This is DSP56000 assembly code generated from a TI's (Texas Instruments) native compiler. On lines 1, 5, and 7, you can see that move operations perform data accesses like load/store instructions. The *x:* or *y:* of the move instructions indicates dual memory banks

Manuscript Received: Dec. 29, 2016 / Revised: Jan. 10, 2017 / Accepted: Jan. 20, 2017
Corresponding Author: dscho@scnu.ac.kr
Tel:+82-61-750-3577, Fax: +82-62-750-3570
Department of Electrical & Electronic Engineering, Sunchon National University

X, Y included in TI's DSP56000. The move command moves data from each bank to a register through directives x: or y :. As shown in the code, the move command can access data from the dual banks (X and Y) simultaneously.

```
move                x:(r0)+, x0 y:(r4)+,y0
mpyr x0,y0,a        x:(r0)+, x0 y:(r4)+,y0
do      #N-1, end
mpyr x0,y0,a        a, x:(r1)+ y:(r4)+,y0
move                x:(r0)+,x0
end
move                a,x:(r1)+
```

Figure 1. N real multiplies code from DSP56000

Like VLIW (Very Large Instruction Word) architectures, DSP56000 can also execute multiple operations in a single instruction independently. In Figure 1, the first **move** instruction reads the data from address of r0 register and address of r4 register in the memory banks, X & Y. They load data into the registers x0 and y0, and perform operations to increase the address of the registers r0 and r4 by a word. The second instruction **mpyr x0,y0,a** multiplies the two loaded values, x0 and y0, and then it stores them. The other operations on the same line execute reload the next two values x0, y0 from addresses (r0)+, (r4)+. At this point, we focus the fact that if two values (x0 and y0) are stored in two different memory banks, then the data can be loaded at the same time. However, if the two values are stored in the same bank, they must be loaded serialized memory accesses. As a result, it is required two times more memory accesses by the serialization. Dual memory banks have two read port to support memory accesses. Therefore, optimizing compiler should efficiently analyze the program code and generate concurrently accessible data placing into different memory banks to optimize performance.

2. Background

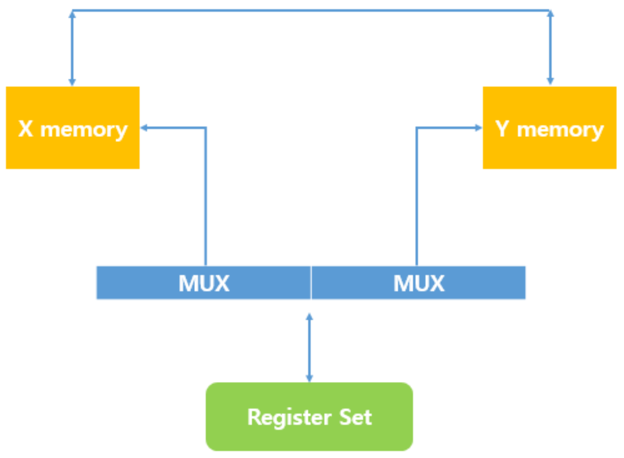


Figure 2. X/Y Data Memory Bank

A sort of processor architecture performs data load/store operations in parallel for improving memory system performance [1]. As an example, we choose TI's DSP56000 describing in this paper. As shown in Figure 2, the X/Y data memory banks allows two memory accesses concurrently. Due to this point of the DSP56000 architecture, two memory access operations can be concurrently executed in one instruction cycle. However, there is lack of compiler support to place such accessible data in parallel onto different memory bank, since data placement is determined by the compiler's code generator [2][3]. Thus, the compiler should be aware such hardware feature to efficiently use that for performance speedup.

3. Utilization of dual memory banks

Ideally, independently accessible data should be allocated to different memory banks for simultaneous accesses [4][5]. This is possible by dividing such data into ***n*** pieces and assigning them to ***n*** different memory banks when the number of memory banks is ***n***. This data partitioning / allocation is possible by solving the following two problems. First, we have to define some certain relations between independently accessible data. In other words, we need a criterion that can guide to determine places of such data to multiple memory banks at the allocation step. The generated decision from the first step, it maximizes system performance since they exploit the memory port maximally. Second, we need to define the rules so that as many pieces of partitioned data occur as possible to ensure the best performance.

To that end, we design our data allocation technique based on an undirected interference graph to determine partitioning/allocation data from/to multiple memory banks in a program. In our graph, a node means a unique variable used of a target program, and edge means an interference relation, which indicates simultaneously accessible them. That is, nodes connected to the edge must be allocated to a set of different memory bank so that they can be simultaneously accessed to maximize system performance. An example of the construction of an interference graph is shown in Figure 4.

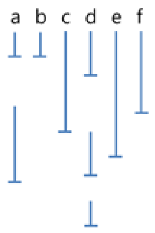
```
while(instructions not empty){
    if(operation == move){    //load or store
        nodeBuild(variables);
        if(node_i and node_j access different variables/arrays)
            if(they have the same live-range)
                add edge to graph;
    }
}
```

Figure 3. Algorithm for building interference graph

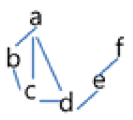
Figure 3 is a pseudocode to describe the construction of an interference graph. Figure 4(a) shows an example code for concurrently executing instructions that target the TI's DSP56000 architecture. Figure 4(b) shows an example of live range for each variable of the example code. Figure 4(c) illustrates the constructed interference graph for variables that can be simultaneously accessed on separate memory banks, and the allocation results for the two banks (X, Y). Although we omitted in the illustrated example, the proposed interference graph includes weights on the edges, and the weights are incremented by 1 according to the number of interferences between the variables.

```
MOVE              a,r0 b,r1
MOVE              c,r2 d,r3
MAC    r0,r1,r2   e,r4 f,r5
MAC    r3,r4,r5   r2,a c,r6
ADD    r2,r6,r7   r5,d e,r8
MOVE              a,r9 d,r10
MAC    r8,r9,r10
MOVE              r10,d
```

(a) An example code



(b) live-range of each variables



X	Y
a	b
d	c
f	e

(c) A result of interference graph and bank assignment

```
MOVE              X:a,r0 Y:b,r1
MOVE              X:d,r3 Y:c,r2
MAC    r0,r1,r2   X:f,r5  Y:e,r4
MAC    r3,r4,r5   r2,X:a Y:c,r6
ADD    r2,r6,r7   r5,X:d Y:e,r8
MOVE              X:a,r9
MOVE              X:d,r10
MAC    r8,r9,r10
MOVE              r10,X:d
```

(d) Optimized data placement

Figure 4. An example of building interference graph

Figure 3 is a pseudocode to describe the construction of an interference graph. Figure 4(a) shows an example code for concurrently executing instructions that target the TI's DSP56000 architecture. Figure 4(b) shows an example of live range for each variable of the example code. Figure 4(c) illustrates the constructed interference graph for variables that can be simultaneously accessed on separate memory banks, and the allocation results for the two banks (X, Y). Although we omitted in the illustrated example, the proposed interference graph includes weights on the edges, and the weights are incremented by 1 according to the number of interferences between the variables.

Finally, Figure 4(d) shows the result code reflecting the bank allocation result. Our bank assignment algorithm based on the interference graph is based on the minimum cost partitioning technique. The least cost partitioning problem is NP-complete, and we applied a heuristic to select the nodes with the largest weight (cost) first. That is, in the interference graph, nodes are selected with the highest weight. And then the selected nodes are removed from the graph. It repeats until there is no nodes. At the each step, the selected nodes are assigned to separated memory banks.

4. Experimental results

To measure the performance of the proposed algorithm, we used DSP kernel codes such as adpcm, edge_detect, compress, and histogram [6][7]. The results are also measured using a simulator of TI's DSP56000 architecture. The comparison code uses the basic technique, thus all data are into a single memory bank. A dual port memory bank is used for comparison with the ideal performance result (ideal).

As you can see in Figure 5, the result is improved by an average of 7.5%. The results vary greatly depending on the nature of the code. The degree of parallelism of the memory operation decreases depending on the distribution of the control code in the loop. Therefore, the effectiveness of the proposed technique is diminished. On the other hand, it is confirmed that the kernel code with a large memory access operation can achieve a large performance improvement. When the performance of the basic technique is taken as 1, the ideal technique and the results of the proposed technique are shown in Figure 5. It is confirmed that almost the same result can be obtained compared with the ideal result.

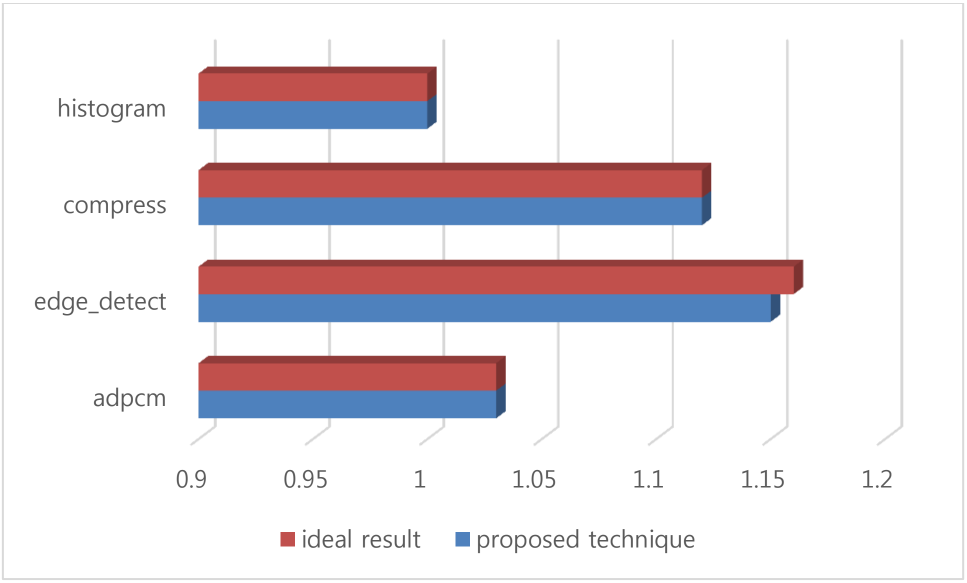


Figure 5. Result of performance improvement

5. Conclusion

Many digital signal processor manufacturers offer dual memory bank systems, which allow applications to simultaneously use multiple memory banks. Unfortunately, until now there has been no compiler technique to support the effective use of this hardware feature. In this paper, we propose a software technique to support the efficient use of these hardware features. Using our proposed scheme, system performance can be improved by taking full advantage of dual bank memory.

Acknowledgement

This paper was supported by(in part) Sunchon National University Research Fund in 2016.

References

[1] A. Appel, J. Davidson, and N. Ramsey, “The Zephyr Compiler Infrastructure,” Technical Report at http://www.cs.virgina.edu/zephyr, University of Virginia, 1998.

[2] G. Araujo, S. Devadas, K. Keutzer, S. Liao, S. Malik, A. Sudarsanam, S. Tjiang, and A. Wang, "Challengers in code generation for embedded processors," Kluwer Academic Publishers, p48-64, 1995.

[3] G. Araujo and S. Malik, “Code Generation for Fixed-point DSPs,” ACM Transactions on Design Automation of Electronic Systems, 3(2):136–161, April 1998.

[4] J. Cho, J. Kim, and Y. Paek. "Efficient and Fast Allocation of on-chip dual memory banks," IINTERACT, Feb, 2002.

[5] C. Fraser, “A Retargetable Compiler for ANSI C,” ACM SIGPLAN Notices, 26(10):29–43, Oct. 1991.

[6] S. Jung and Y. Paek, “The Very Portable Optimizer for Digital Signal Processors,” In International Conference

on Compilers, Architectures and Synthesis for Embedded Systems, pages 84–92, Nov. 2001.

[7] R. Leupers and P. Marwedel, “Algorithms for Address Assignment in DSP Code Generation,” In Internaltional Conference on Computer-Aided Design, 1996.