International Journal of Reliable Information and Assurance
Vol.7, No.2 (2019), pp.21-26
http://dx.doi.org/10.21742/ijria.2019.7.2.04

Implementation of Low Cost Memory Subsystem for Low-end IoT Devices

Jonghee M. Youn[1] and Doosan Cho[2]

[1]*Computer Engineering, Yeungnam Univ., South Korea*
[2]*Electrical & Electronic Engineering, Sunchon National Univ., South Korea*
[1]*youn@yu.ac.kr*, [2]*dscho@scnu.ac.kr*

Abstract

The increasingly popular IoT devices and cloud computing devices are being developed in various models from high to low price, but the low-cost market is still growing more actively. In these devices, where internet communication is a key feature, the most expensive components are memory and screen panels. Currently, screen panels are limited in LCD and OLED technology, so the choice is small, but memory includes flash memory, hard disk, DRAM, SRAM, SDRAM, multi-bank memory, and on-chip memory. Therefore, each type is selected and configured according to requirements such as function, power consumption, performance, and cost. The choice of memory architecture available for low-cost IoT devices is quite limited, with a small configuration of SRAM and some flash memory or DRAM. In the case of hard real-time IoT devices, it is very difficult to meet the deadlines in such a memory structure, and developers apply various system optimizations to solve them. Normally, multibank DRAM is selected at the hardware design stage. Parallel access to as many bank memories as possible in the same space can significantly improve system performance. If the hardware is selected as multi-bank memory, there must be system software to support it. In other words, a compiler must be provided to generate program code for parallel memory access. This is because traditional compilers generate program code for sequential access. In this paper, we propose a parallel memory access program code generation method for multi-bank memory support of low-cost IoT devices. The proposed method solves the data placement problem for multi-bank memory and maximizes system performance by actively using multi-bank memory.

Keywords: *Energy consumption, IoT system, Heterogeneous memory system, Load/store data dependence graph, Compiler technique, System optimization*

1. Introduction

To improve system performance, one of the most efficient hardware implementation is multi-bank memory, because multiple data can be accessed simultaneously in the same time. However, to get the most benefit out of these hardware features, it needs to properly assign the data so that the data can be used at the same time and generate memory access instructions for using the data. Many commercial processors use multi-bank memory such as the Motorola DSP56000, Analog Devices ADSP2106x, and NEC μPD77016. They are supported by an optimizing compiler to assign data properly onto multiple memory banks.

Article history:
Received (July 13, 2019), Review Result (September 6, 2019), Accepted (October 29, 2019)

Print ISSN: 2288-8950, eISSN: 2207-5143 IJRIA

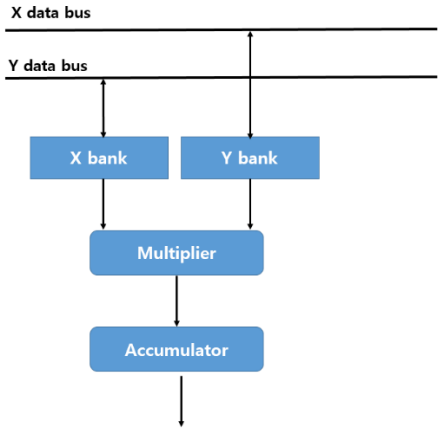

Figure 1. ALU configuration in DSP56000 (free scale semiconductor)

[Figure 1] shows a diagram of the memory bank and data ALU of the DSP56000 manufactured by Freescale Semiconductor. Two X memory banks and two Y memory banks are sending operands of data directly to the multiplier and to the accumulator. To process this process, the optimizing compiler must distribute data to be used at the same time onto the X bank and Y bank separately. The two operands would be used at the same instruction as its operands. If they are not separately assigned to X/Y banks, then, the multiplier should wait to complete the sending process from one single bank. This case yields the worst execution time and power consumption, since the long sending process consumes more time and power. Eliminating this inefficiency is the task of the compiler. Thus, compiler's role is very important in the multi-bank memory. The problem is that the existing compiler does not configure the hardware structure like multi-bank memory to generate efficient data assignment and instruction code. Traditional compilers determine data placement in the order of alphabetical or variable declaration order, making it difficult to fully exploit the benefits of multibank memory.

The proposed technique performs data placement optimization for various memory architectures composed of multiple memory banks. To explain this effectively, this study is organized as follows. The next section looks at the relevant research trends. We will look first at existing research and identify the need for our research. Section 3 describes the proposed technique in detail. Basically, our technique uses compiler analysis techniques, thus, it can be used as part of the compiler or post pass optimization of the compiler. We will examine the experimental results in section 4 and then finally make a conclusion.

2. Related works

The earliest work on the problems discussed in this paper was discussed by Powell, Lee and Newman [1]. After the scheduling and register allocation steps, their paper [1] states that the placement of local data variables to dual memory banks. It uses meta-assembly code what they defined. Data is alternately assigned to the X and Y banks according to the access order of the program code without special analysis. Saghir, Chow and Lee's research discussed variable partitioning techniques for virtual VLIW DSP architectures [2][3]. These studies introduce two algorithms. Data partitioning and replication are the core of the algorithm. Data partitioning / replication is commonly used to improve performance by using locality in common memory systems. These techniques can be implemented in the data mapping part of the compiler to

improve existing techniques or back-end optimization to improve performance by rearranging the data mapping determined by the existing techniques. A study by Sudarsanam and Malik [4][5] introduces memory bank allocation and register allocation algorithms using graph labeling. In order to find the optimal solution, they adopted simulation annealing and presented reliable experimental results.

In the paper by Leupers and Kotte [6], various search/combinations are attempted to find an optimizing solution with dry-run for the search space. By doing so, a solution can be determined after executing for the whole search space. This is only available when it is guaranteed that the space of the solution search/combination can be terminated in a finite time. A representative search method is integer linear programming.

The most recent memory bank optimization study is [7][8][9]. Cho, Paek and Whalley [7] proposed an optimal data and register allocation algorithm for high performance architectures. It is difficult to use with existing techniques because it is developed with very complicated and sophisticated algorithm. The partitioning task uses heuristics to select the maximum spanning tree of the interference graph.

Certain architectures support parallel load stores, Pande and Greenland [8] proposed a register allocation algorithm that provides sufficient support for these hardware resources. The first step is to use the PostPath Optimizer, which compiles once and binds the load/store of the generated code to the possible range.

They apply a two-color graph coloring algorithm to propose a partitioned motion schedule graph. Zhuge, Xiao and Sha's research [9] provides two algorithms. They are related to optimize data partitioning and code scheduling. The algorithms provide potential parallel data accesses that can practically occur in scheduling. The problem is formulated as a variable-independent graph modified with the moving window used by removing some edges that cannot be scheduled in the same control stage. Greedy strategy is used to split the graph into several separate sets.

Most of the previous studies used some sort of graph with different optimization methods. It is designed as a post-pass backend step that is applied mostly in assembly code. This has the advantage of being able to analyze all data accesses, but in general it is very difficult not to affect register allocation and scheduling. The approach we propose works with a high-level programming language. Information from sophisticated program analysis can find any possible data parallelism among data accesses. Data accesses are analyzed, and the dependence between data and usage instructions is analyzed at once, enabling global data optimal placement. For pointer variables that use aliases, data access analysis is not always possible, but as described in this conclusion, the results of the application are very effective.

3. The proposed technique

Definition 1. Load-Store Dependence Graph.

This graph consists of a set of edge and node. Each node in the set N represents load-store instructions. Each edge in the edge set represents dependence relation between nodes. A weight on edges represents the number of coalesced nodes.

The graph construction process is as the following. First, a data dependency graph is constructed. This uses the existing algorithm [10]. The data dependency graph constructed based on the target loop code is the starting point for constructing the load-store dependence graph that we proposed. With the data dependence graph, the beginning point is merging all arithmetic nodes of the graph to load-store nodes. The arithmetic nodes that depend on the load instruction are merged into the node corresponding to the load. At this point, the edge connected

to the other node is connected to the corresponding load node. Repeating this task completes the load/store only dependency graph. [Figure 2] illustrates the dependency graph building process.

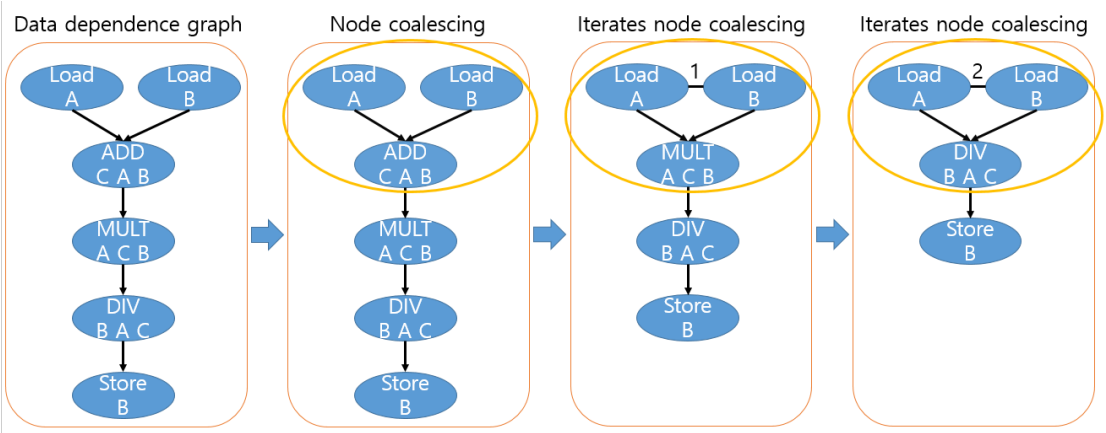

Figure 2. The building process of the load/store dependence graph

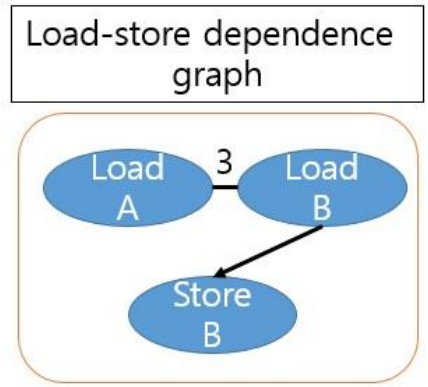

Figure 3. An example of load/store dependence graph

[Figure 2] shows the data dependency graph on the most left. The first two load instructions create the variable A B, and store the final calculation result through ADD, MULTIPLY, and DIVIDE operations. In the second picture of [Figure 2], the Add node is dependent on load A and load B. Therefore, it is coalesced with the two loads. At this point, edge is created by merging Add between two nodes. The weight 1 indicates that the coalesced node is merged by the two loads. Repeat this process to coalesce the Mult and Div nodes. Eventually, the weight of 3 creates between the two load nodes. This indicates that since the load coalesced three nodes, they must be executed simultaneously before the three operations. [Figure 3] shows the final load-store dependence graph. The Store node only stores variable B, so the edge is connected only to load B node. The proposed technique uses this load-store graph to determine data placement more effectively in multibank on-chip memory.

Based on the load-store dependency graph, the proposed technique optimizes data to place on multiple memory banks. The goal of the technique is to minimize memory access execution time. To this end, we formulate this data allocation problem mapping into a simple node assignment problem.

PROBLEM:

MINIMIZE_EXECUTION_TIME (ASSIGNMENT (LOAD_STORE_GRAPH(N)))

The proposed technique operates the assignment by level ordering traversal in the load store dependence graph. The nodes having edges are then allocated to different memory banks as long as there is space available. The reason for allocating nodes with edges to different memory banks is to achieve the goal of improving system performance by allowing them to be accessed concurrently.

```
NODE ASSIGNMENT ALGORITHM
G=Load_Store_Dependence_Graph
PREV_BEST = 0
WHILE( 1 )
                ITERATE(G)
                  VISIT( G(N) )
                  BEST_SOLUTION = ASSIGNMENT (MAPPING(N, banks))
      STOP(NO_MORE_NODE(G))
IF ( BEST_SOLUTION >= PREV_BEST )
PREV_BEST = BEST_SOLUTION
ELSE
Return PREV_BEST
```

Algorithm 1. The proposed node assignment technique

[Algorithm 1] describes the algorithm for finding the solution to the problem defined PROBLEM. The first input to the algorithm is the load-store dependence graph. Algorithm visits all nodes in a leveling traversal manner. Each level is allocated to the memory bank in consideration of the edges of the nodes. If nodes have edges, assign them to different banks as much as they are needed at the same time, even if they are not in different banks. Record the determined bank mapping in BEST_SOLUTION. Repeat this process until there are no nodes in the graph. Current BEST_SOLUTION stores to PREV_BEST if the gain (execution time reduction) is greater compared to PREV_BEST. Again, calculate the solution with a different mapping than before, and update the result by comparing PREV_BEST with the gain. Repeat this process until there is no more the best solution to update PREV_BEST.

4. Conclusion

In this paper, we propose a compiler technique that solves data location optimization problems for various commercialized processor architectures with multiple memory banks. We designed the technique by converting the existing data allocation problem into a simple graph node allocation problem, unlike the existing techniques. The advantage of our proposed method is that we can plant our technology on various architecture platforms by redefining the problem of NP complete as a very simple graph node allocation problem. A simple problem definition can provide a range of choices to support a variety of applications. The problem is optimized by adding new variables to the user's needs. The problem in this study is defined for loop code, so the problem itself is small. Therefore, it is possible to find the optimal solution using perfect search. As a result, optimal performance can be provided. In this study, experiments were conducted on small loop codes, but it is expected that it is possible to conduct experiments

using large benchmarks related to multimedia [11] or network communications [12] and to develop more generalized improved algorithms.

Acknowledgements

This work was supported by Basic Science Research Program through the National Research Foundation of Korea (NRF) funded by the Ministry of Education (NRF - 2018R1D1A1 B07050054).

References

[1] D.B. Powell, E.A. Lee, and W.C. Newman, "Direct synthesis of optimized DSP assembly code from signal flow block diagrams," In Proceedings of the IEEE International Conference on Acoustics, Speech, and Signal Processing (ASSP), vol.5, pp.553-556, **(1992)** DOI: 10.1109/ICASSP.1992.226560

[2] M.A.R. Saghir, P. Chow, and C.G. Lee, "Automatic data partitioning for HLL DSP compilers," In Proceedings of the 6th International Conference on Signal Processing Applications and Technology, pp.866-871, **(1995)**

[3] M.A.R. Saghir, P. Chow, and C.G. Lee, "Exploiting dual data-memory banks in digital signal processor," In ACM SIGOPS Operating Systems Review, Proceedings of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems, vol.30, no.5, pp.234-243, **(1996)** DOI: 10.1145/248208.237193

[4] A. Sudarsanam and S. Malik, "Memory bank and register allocation in software synthesis for ASIPs," In Proceedings of the IEEE/ACM International Conference on Computer Aided Design, pp.388-392, **(1995)**

[5] A. Sudarsanam and S. Malik, "Simultaneous reference allocation in code generation for dual data memory bank ASIPs," Journal of the ACM Transactions on Automation of Electronic Systems (TODAES), vol.5, pp.242-264, **(2000)**

[6] R. Leupers and D. Kotte, "Variable partitioning for dual memory bank DSPs," In Proceedings of the IEEE International Conference on Acoustics, Speech, and Signal Processing (ASSP), **v**ol.2, pp.1121-1124, **(2001)** DOI: 10.1109/ICASSP.2001.941118

[7] J. Cho, Y. Paek, and D. Whalley, "Efficient register and memory assignment for non-orthogonal architectures via graph coloring and MST algorithm," In Proceedings of the International Conference on the LCTES and SCOPES, Berlin, Germany, vol.37, no.7, pp.130-138, **(2002)** DOI: 10.1145/513829.513853

[8] X. Zhuang, S. Pande, and J.S. Greenland, "A framework for parallelizing load/stores on embedded processors," In: Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT), Virginia, **(2002)** DOI: 10.1109/PACT.2002.1106005

[9] Q. Zhuge, B. Xiao, and E.H.-M. Sha, "Variable partitioning and scheduling of multiple memory architectures for DSP," In Proceedings of the IEEE International Parallel and Distributed Processing Symposium (IPDPS), **(2002)**

[10] Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren, "The program dependence graph and its use in optimization," ACM Trans. Program. Lang. Syst., vol.9, no.3, pp.319-349, **(1987)**

[11] Chunho Lee, M. Potkonjak and W. H. Mangione-Smith, "MediaBench: a tool for evaluating and synthesizing multimedia and communications systems," Proceedings of 30th Annual International Symposium on Microarchitecture, Research Triangle Park, NC, USA, pp.330-335, **(1997)** DOI: 10.1109/MICRO.1997.645830

[12] Poovey Jason, Conte Thomas, Levy Markus, and Gal-On Shay, "A benchmark characterization of the EEMBC benchmark suite," Micro, IEEE, vol.29, no.5, pp.18-29, **(2009)** DOI: 10.1109/MM.2009.74