International Journal of Advanced Culture Technology Vol.9 No.4 282-287 (2021)
DOI https://doi.org/10.17703/IJACT.2021.9.4.282

IJACT 21-12-33

A Study on Effect of Code Distribution and Data Replication for Multicore Computing Architectures

Doosan Cho

Prof., Dept. of Electrical and Electronic Engineering, Sunchon National Univ., South Korea
dscho@scnu.ac.kr

Abstract

A multicore system must be able to take full advantage of the program's instruction and data parallelism. This study introduces the data replication technique as a support technique to maximize the program's instruction and data parallelism. Instruction level parallelism can be limited by data dependency. In this case, if data is replicated to each processor core and used, instruction level parallelism can be used to the maximum. The technique proposed in this study can maximize the performance improvement effect when applied to scientific applications such as matrix multiplication operation.

Keywords: *Memory, compiler, optimization, data allocation, loop pipelining, program transformation*

1. INTRODUCTION

In the past 20th century, CPU performance's improvement doubled every 18 months according to a technological innovation called Moore's Law. As the 21st century began, the CPU performance did not improve in proportion to the increase in the number of transistors per same area. As shown in Figure 1, this is called Moore's Gap. The cause of CPU performance stagnation is that power consumption increases exponentially as the density increases, and as a result, heat generation increases. Therefore, the increase in the performance of a single CPU is blocked by a technological barrier called the power wall, leading to the limited performance improvement according to Moore's Law. As a result, the technique of improving the performance of a single CPU has been replaced by a technology paradigm to multiple CPUs with decent performance.

When comparing the performance of the latest single CPU and the performance of multiple CPUs, the difference was near zero, so the multi-CPU technology developed rapidly. The first technique for developing multicore is called Hyper-Threading. This technology was a form of sharing a cache memory and a bus, and after that, a multi-CPU similar to the present was a Pentium dual-core processor that appeared in 2005. Next, multi-core processors for smartphone, notebooks such as ATOM processors and Cortex appeared, and multi-core architectures using tens or hundreds of CPU cores became the mainstream of the computing market.

The biggest problem with multi-core architectures was memory bandwidth. In order for a large number of processor cores to work, data is inputted to the processor core in the form of streaming, but since too much data is moved to multiple processors at once, the bottleneck of the data bus bandwidth degrades the overall performance of the system. To solve this problem, various types of memory structures have been proposed and

Manuscript received: October 16, 2021 / revised: November 2, 2021 / accepted: December 2, 2021
Corresponding Author: dscho@scnu.ac.kr
Tel: +82-61-750-3577, Fax: +82-61-750-3570
Professor, Dept. of EE, Sunchon National Univ., Korea

Copyright©2021 by The International Promotion Agency of Culture Technology. This is an Open Access article distributed under the terms of the Creative Commons Attribution Non-Commercial License (http://creativecommons.org/licenses/by-nc/4.0)

are currently being used. Among these types, a widely used technique is a hybrid architecture that is parallelized on-chip local memory and distributed shared memory used by each core. Although this type of memory structure provides proper system performance, there is an overhead in which optimal data arrangement must be managed by system software such as a compiler and an operating system. In this study, we will discuss techniques for optimizing this overhead by a compiler.

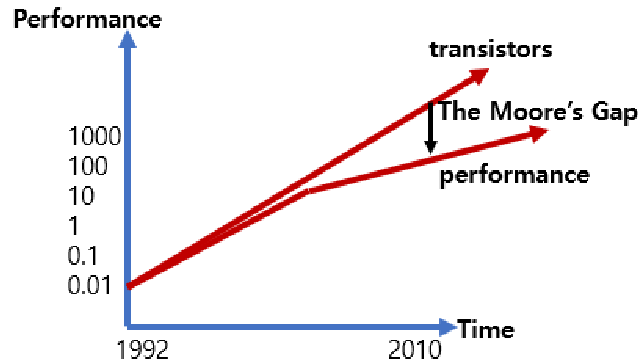

Figure 1. Gap between performance and number of transistors

Technical limitations caused by the increase in processor density include power consumption and heat generation problems as well as leakage current problems. The leakage current problem began to emerge in the microprocessor that was more advanced than the 180nm fabrication technology, and it became a serious problem, exceeding 50% of the power consumption in 10nm fabrication or less. Various studies have been conducted to solve these problems. Among the various solutions, non-volatile memory technology is in the spotlight. Since the nonvolatile memory has a near-zero leakage current, it has developed into a hybrid memory that is mixed with the existing memory technology in the on-chip memory or main memory. Non-volatile memory technology also includes technical limitations in terms of delay and power consumption for write operations. Since these shortcomings could be solved with hybrid technology, current system chips support a memory structure in which non-volatile memory, SRAM, and DRAM are mixed.

Therefore, the multi-core system chip contains various software challenges. In most multimedia data streaming applications, the operation of the program code is predictable at compile time, so program task assignment through profiling is possible. On the other hand, in multicore, data access patterns are dynamically determined during program execution, so it is difficult to optimally allocate data through profiling. Therefore, data allocation in multi-cores for shared distributed on-chip memory in a hybrid form must be performed dynamically. In this study, we propose a technique that solves the data allocation problem in multi-core for a hybrid type of shared distributed on-chip memory.

This study introduces a data replication technique to support parallel execution with the goal of improving performance. Since data replication increases the memory space used, an overhead of increasing power consumption occurs due to this. In order to reduce this overhead, it is necessary to simultaneously apply a compiler technique to reduce memory power consumption, such as [1][2][3][4]. In addition, it is expected that better performance will be improved if it is applied together with compiler techniques that can obtain additional performance improvement such as [5][6][7][8]. The study of [9][10] introduces a technique for inducing improvement in memory system performance using artificial intelligence or machine learning techniques. In the future, it is expected that related research will proceed in the direction of using these AI-related techniques together.

This paper is organized as follows. The next chapter explains the technical details that is proposed in this

work. In Chapter 3, the experimental results are examined, and the conclusion is described.

2. THE PROPOSED TECHNIQUE

Figure 2 shows the targeted multi-core processor in this study. There are 4 processing cores in a row and 4 local memory connected to them, they are connected by a shared data bus. Directly connected local memory can be accessed in 1 cycle, and other cores' remoted local memory accesses takes 3 cycles. Figure 3 is an example code of matrix multiplication. The result matrix stores the multiplication result, and the matrices matrix1 and matrix2 are the operand matrices. Matrix multiplication operation is performed with a triple ***for*** loops, and let's look at an example of mapping of the example code in Figure 3 to the multicore shown in Figure 2. Since this study is not about the code mapping algorithm, the code mapping is manually done. The focus of this study is the data replication and allocation algorithm. Figure 4 shows the order of operation in which matrix multiplication is performed.

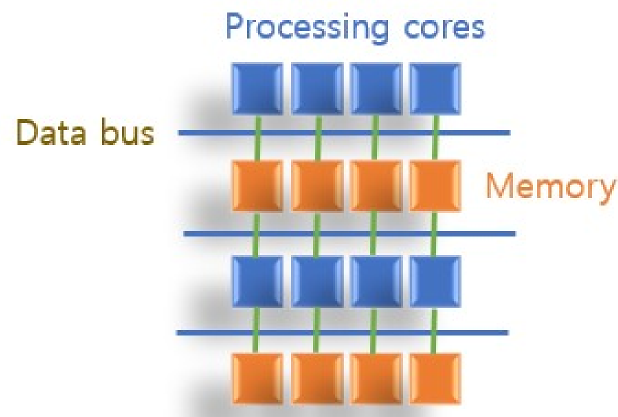

Figure 2. Multiprocessor core architectures

```
for (int i = 0; i < Row_Size; i++) {
        for (int j = 0; j < Column_Size; j++) {
                result[i][j] = 0;
                for (int k = 0; k < Row_Size; k++) {
                        result[i][j] += matrix1[i][k] * matrix2[k][j];
                }
        }
}
```

Figure 3. Matrices multiplication code

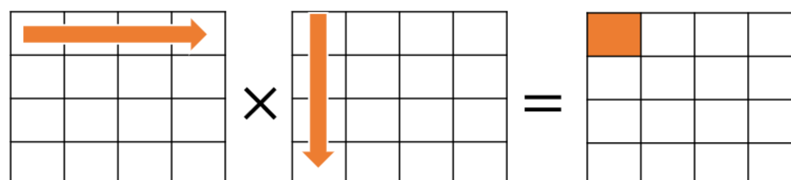

Figure 4. The order of matrix multiplication

With the matrix 1, data is accessed in the order of rows and the multiplication is performed by the order. With the matrix 2, data is accessed in the order of columns and the multiplication is performed by the column order. As shown in Figure 4, when the first row of matrix 1 and the first column of matrix 2 are multiplied, the elements of row 1, column 1 of the result matrix are calculated by the multiplication. If four processing cores are participated in the multiplication with the size of 4x4 matrix, to perform parallel execution, the result matrix should be partitioned into quarters by the row order, and they should be allocated to the four processing cores.

In order for the calculation to proceed in this way, matrix 1 must be divided into quarter of rows and should be allocated to the corresponding local memory of the allocated processing core. The matrix 2 must be replicated to the corresponding local memory of the four processing cores. To perform these procedures, the replication occupies additional memory space ,16 data x 4 bytes = 64bytes, that is the overhead of the proposed technique. However, the proposed technique is 4 times faster than the single core's process. To take this speedup with the overhead, the detail of the proposed data replication algorithm is described in Figure 5.

```
ALGORITHM
Input: program code, architecture information
Output: replicated data & distributed code

int main(void)
{
    list_core=0;
    list_memory=0;
    D_graph = Dependency_graph(program_code);

    while( is_empty(D_graph) ){
        if( is_evenly_distributed(list_core) )
            return 0;
        else{
            if(is_distribution(D_graph)){
                    list_core = get_operation(D_graph);
                    list_memory = distribution_data(D_graph);
            }
            if(is_replication(D_graph)){
                    list_core = get_operation(D_graph);
                    list_memory = replication_data(D_graph);
            }
        }
    }
}
```

Figure 5. The proposed algorithm

The proposed algorithm uses program code and multiprocessor architecture information as inputs. Architecture information includes the number of processor cores, the size of connected local memories, and access latency. The output provides replicated data and distributed code results. D_graph is a data structure containing code and data dependency information of the given program code. The data format stored in

D_graph is shown below. As shown in the below code, result(0,0) indicates the value of 0,0 position data in the result matrix. The arrow indicates that the data of result(0,0) is calculated as the product of the row index 0,1,2,3 data of matrix1 and the row index 0,4,8,12 data of matrix2. In this way, D_graph stores dependency information between matrices' data.

Data format in D_graph
result(0,0) → matrix1(0,1,2,3) * matrix2(0,4,8,12)
result(0,1) → matrix1(0,1,2,3) * matrix2(1,5,9,13)
result(0,2) → matrix1(0,1,2,3) * matrix2(2,6,10,14)
result(0,3) → matrix1(0,1,2,3) * matrix2(3,7,11,15)
…

Based on D_graph information, the first row of matrix1 is used to calculate the first row of the result matrix. In addition, the calculation requires the entire matrix2 data. Therefore, matrix2 is required to replicate to finish the calculation in the proposed algorithm. As shown in the D_graph information, matrix1 should be distributed to processing core's local memory. The replication of matrix2 modifies the data dependencies that is constructed due to matrix2. After the data dependency's modification, matrix multiplication can be evenly distributed to given processing cores in the order of calculation of matrix1 and result matrices. The distribution result of the multiplication is stored to ***list_core***. Data required for the distributed multiplication stored to ***list_memory***. Because list_memory stores the data required for each core's multiplication in the same order as list_core, there is no more record to label for replication or distribution. In the ***while*** statement, the loop finishes when there are no more distributed multiplication in D_graph.

3. RESULT

For the evaluation of the proposed algorithm, an experiment was conducted with matrix multiplication programs of various sizes. The experimental environment consists of AMD 3.4GHz 16 core processor, 64GB memory, Windows 10, visual studio version 16.11.2. Table 1 lists the experimental results. In the experiment, 1x1, 4x4, 16x16, and 32x32 size matrix multiplication programs were used. The first two rows show the results of execution by assigning to a single core measured in nanoseconds. The 3rd and 4th rows show the result in nanoseconds when it is executed with the same amount of computation divided into 4 cores. When a 4x4 matrix was allocated to 4 cores by 1x4 in the second column and the third row, a 35.8% improvement in execution performance was possible. Since one 4x4 matrix has to be copied to 4 cores in order to perform matrix operation on 4 cores, memory space of 4 x 64-byte matrices is additionally used. If the 16x16 matrix is divided into 4x16 matrices and executed, the execution time can be improved by 22.9%. Since one 16x16 matrix has to be copied to 4 cores, an additional 1024 byte x 4 memory is used. In case of 32x32 matrix multiplication, performance improvement was possible by 20.3%, and 4096byte x 4 memory space was additionally used.

Table 1. Experimental result

Matrix size	1x1	4x4	16x16	32x32
Execution time (nano-second)	481100	2042600	23421400	109096000
Distributed matrix size	N/A	1x4	4x16	8x32
Execution time (nano-second)	N/A	731400	5365500	22215000
Improvement (%)	N/A	35.8%	22.9%	20.3%

As a result of confirming through the above experiment, if additional available memory is sufficient, significant execution performance improvement can be obtained if parallel operation is supported by multiple cores using the data replication technique. In the experiment of this study, an average of 26.3% performance improvement was possible. The proposed algorithm is expected to obtain good results in scientific applications where arithmetic operations such as matrix operations are repeatedly performed.

4. CONCLUSION

In the current IT industry, where artificial intelligence and big data applications are mainstream, the demand for code generation to effectively accelerate the frequent matrix operations is increasing. The development of compiler technology that can satisfy the market needs will help advance the AI learning process and big data application technique. In this study, a data parallelism support technique specialized for matrix operation was proposed to satisfy these market requirements. Using the proposed technique, an average performance improvement of 26.3% was obtained.

ACKNOWLEDGEMENT

This paper was supported by Sunchon National University Research Fund in 2021. (Grant number: 2021-0225)

REFERENCES

[1] D. Cho, "A Study on Improvement of Low-power Memory Architecture in IoT/edge Computing," *Journal of the Korean Society of Industry Convergence*, Vol. 24, No. 1, pp. 69-77, Feb. 2021. DOI: https://doi.org/10.21289/KSIC.2021.24.1.69

[2] J. Yoon, D. Cho, "A spill data aware memory assignment technique for improving power consumption of multimedia memory systems," *Multimedia Tools and Applications*, Vol. 78, No. 5, pp. 5463-5478, Mar. 2019. DOI: https://doi.org/10.1007/s11042-018-6783-x

[3] J. Cho, J. Yoon and D. Cho, "Improvement Energy Efficiency for a Hybrid Multibank Memory in Energy Critical Applications," *Technical gazette*, vol.27, no. 6, pp. 1946-1955, 2020.

[4] J. Yoon, D. Cho, "Improving memory system performance for multimedia applications," *Multimedia Tools and Applications*, Vol. 76, No. 4, pp. 5951-5963, 2017. DOI: https://doi.org/10.1007/s11042-015-2807-y

[5] J. Cho, J. Lee, D. Cho, "Efficient memory design for medical database," *Basic & Clinical Pharmacology & Toxicology*, Vol. 125, pp. 198, July. 2019.

[6] J. Cho, J. Lee, D. Cho, "Low-End Memory Subsystem Optimization Process," *International Journal of Smart Home,* Vol. 13, No. 2, pp. 11-16, Oct. 2019. DOI: http://dx.doi.org/10.21742/ijsh.2019.13.2.02

[7] J. Cho, D. Cho, Y. Kim "Study on LLVM application in Parallel Computing System," *The Journal of the Convergence on Culture Technology*, Vol. 5, No. 1, pp. 395-399, Feb. 2019.

[8] J. Cho, D. Cho, "Development of a Prototyping Tool for New Memory Subsystem*," International Journal of Internet, Broadcasting and Communication*, Vol. 11, No. 1, pp. 69-74, Jan. 2019.

[9] D. Cho, "Study on Memory Performance Improvement based on Machine Learning," *The Journal of the Convergence on Culture Technology*, Vol. 7, No. 1, pp. 615-619, Feb. 2021.

[10] D. Cho, "Memory Design for Artificial Intelligence," *International Journal of Internet, Broadcasting and Communication*, Vol. 12, No. 1, pp. 90-94, Dec. 2020. DOI: https://doi.org/10.7236/IJIBC.2020.12.1.90