Automatic Parallelization of Hand Written Automotive Engine Control Codes Using OSCAR Compiler

Dan Umeda, Yohei Kanehagi, Hiroki Miakami, Akihiro Hayashi,

Keiji Kimura and Hironori Kasahara

Department of Cumputer Science and Engineering, Waseda University

3-4-1, Okubo, Shinjuku-ku, Tokyo 169-8555, Japan.

Email: {umedan, ykane, ahayashi, kimura, kasahara} @kasahara.cs.waseda.ac.jp

***Abstract*— The next-generation automobiles are required to be more safe, comfortable and energy-efficient. These requirements can be realized by integrated control systems with enhanced electric control units, or real-time control system such as an advanced engine control unit and an enhanced information system, human and other cars recognition, a navigation system considering traffic conditions in an emerging occasion from a natural disaster. For these purpose, performance enhancement of microprocessors is required to realize the next-generation automobiles integrated control system. However, the improvement of clock frequency and instruction-level parallelism such as Superscalar are difficult. And the performance of a single-core processor which controls power has reached the limits. Taking into account these factors, use of multi-core processors has been thought a promising approach to realize the next-generation automobiles integrated control system. However, automotive programs are difficult to parallelize because they have no loop parallelism that has been used in multi-core processors for a long time. This paper proposes to parallelize an automotive engine crankshaft control program which consists of conditional branches and arithmetic assign statements, basic blocks using automatic multigrain parallelizing compiler, or the OSCAR Compiler has been developed by the authors for more than 25 years. OSCAR compiler uses coarse grain task parallelism with newly developed a parallelism enhancing method like the branch duplication instead of loop parallelism. Performance of the hand-written engine control programs which was provided by Toyota Motor Corp, on the RP-X having eight SH4A processor cores developed by Renesas, Hitachi, Tokyo Institute of technology and Waseda University is evaluated. The evaluation shows speedups of 1.54 times with 2 processor cores compared with the case of an ordinary sequential execution. The proposed method successfully accelerated engine control program on a real multi-core processor.**

Keywords*— *Keywords: multi-core processor, automobile, automatic parallelization, embedded system

I. INTRODUCTION

Automobiles have become essentials in human lives. The technologies for automobiles have been advancing for the last several decades especially because of the advances of Electronic Control Units (**ECUs**). Those advances will allow us more safe, comfortable and energy efficient on the next-generation automobiles. These requirements can be realized by integrated control systems that fuse enhanced ECUs like engine control units and enhanced information system such as recognition systems for human and other cars and navigations systems considering traffic conditions including the occasions of natural disasters.

The sophisticated engine control algorithms and functions which will be used in the next-generation ECUs require performance enhancement on microprocessors to satisfy real-time constraints. However, performance improvement with single core processor has been limited by power consumption. Use of multi-core processors is a promising approach to realize the integrated control systems.

In terms of multi-core processors for the automotive control, the previous works have focused on improvement of reliability by performing redundant calculation [1][2] and throughput by functional distribution [3] rather than improvement of response time, or performance by parallel processing. In other words, they could not improve response time at all. To the best of our knowledge, parallel processing of the automotive control software to reduce response time has not been succeeded on multi-core processors because the software consists only of

conditional branches and small basic blocks. In the software, there are no loops, to which traditional parallelizing compiler have paralleled.

On the other hand, this paper has successfully parallelized the practical automotive engine control software using automatic multigrain parallelizing compiler, or the OSCAR compiler [4]. In this paper, a newly developed parallelism enhancement methods like the branch duplication instead of loop parallelism for engine control programs are proposed. These techniques applied to engine control software. The parallelized engine control software by the OSCAR compiler is evaluated on an embedded multi-core using SH-4A processor cores, RP-X.

The rest of this paper is organized as follows: Section II introduces the OSCAR Automatic Parallelizing Compiler. Section III proposes the method of parallel processing of engine control programs. Section IV describes evaluation of performance using our method. Finally, Section V is conclusions for this paper.

II. OSCAR Automatic Parallelizing Compiler

This section describes the overview of the OSCAR compiler. The OSCAR Complier realizes an automatic parallelization of programs written in Parallelizable C [5], which is very close to MISRA C [6][7][8] used in automobile industry for reliability and productivity. The OSCAR Compiler's input is a sequential program and its output includes an executable for a target multi-core and a multi-platform parallelized C code using OSCAR API [9][10], which allows us to execute the parallelized C code on various multi-core processors include ARM, Intel, IBM, Renesas Electronics, Tilera, Fujitsu, and so on. The OSCAR Compiler exploits multigrain parallelism including coarse-grain parallelism, loop level parallelism and fine-grain parallelism [11]. First of all, the OSCAR Compiler decomposes a program into coarse grain tasks, namely macro-tasks (MTs), such as basic blocks (BBs), loops (RBs), and function call or subroutine calls (SBs) as shown as Fig 1. Macro-tasks can be hierarchically defined inside each sequential loop or function. After generation of macro-tasks, data dependencies and control flow among macro-tasks are analyzed in each nested layer, and hierarchical macro-flow graphs (MFGs) representing control flow and data dependencies among macro-tasks are generated. A Macro-Flow Graph (Fig. 2a) represents control flow and data dependencies among Macro-Tasks. Nodes represent Macro-Tasks, solid edges represent data dependencies among Macro-Tasks, and dotted edges represent control flow. Small circle inside a node represents a conditional branch inside the Macro-Tasks. Though arrows if edges are omitted in the Macro-Flow Graph, it is assumed that the directions are downward.

Then, to exploit coarse grain task parallelism among macro-tasks MTs associated with both the control dependencies and the data dependencies, the Earliest Executable Condition analysis [12] is applied to each macro-flow graph. By this analysis, a macro-task graph (MTG) is generated (Fig. 2b). Macro-Task Graphs represent coarse grain task parallelism among Macro-Tasks. Nodes represent Macro-Tasks. A small circle inside a node represents conditional branches. Solid edges represent data dependencies. Dotted edges represent extended control dependencies. Extended control dependency means ordinary normal control dependency and the condition on which a data dependence predecessor of a Macro-Task is not executed. Solid and dotted arcs connecting solid and dotted edges have two different meanings. A solid arc represents that edges connected by the arc are in AND relationship. A dotted arc represents that edges connected by the arc are in OR relationship. Though arrows of edges are omitted assuming downward direction, edges having arrow represents original control flow edges, or branch direction in Macro-Flow Graph. If SB or RB has nested inner layer, Macro-Tasks and Macro-Task Graphs are generated hierarchically. If the Macro-Task Graph has only data dependencies, the compiler schedules Macro-Tasks to Processor Groups at compile time (static scheduling). The static scheduling scheme can minimize data transfer, task assignment and synchronization overhead. If the Macro-Task Graph has conditional branches among Macro-Tasks, the dynamic scheduling developed by the authors' group is usually applied. However, in this paper, OSCAR compiler uses only static scheduling to minimize run-time overhead considering the grain of the automotive codes as described in the next section.

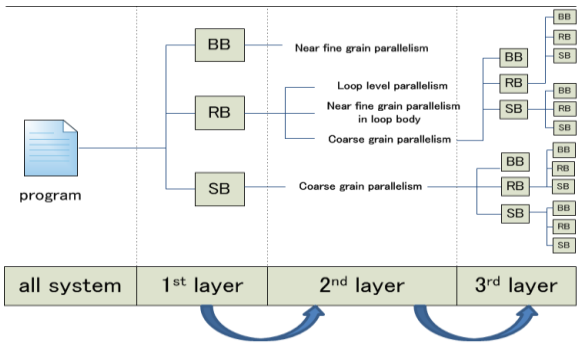

Fig. 1. Exploitation of multigrain parallelism including coarse-grain parallelism

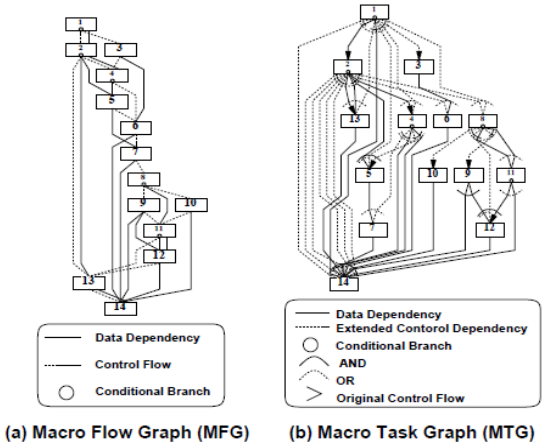

Fig. 2. Macro-flow graph and macro-task graph

III. Parallelization of Engine Control Programs

This section introduces the parallel processing method for automotive programs using the OSCAR compiler. This method

successfully extracts the suitable parallelism for two processor cores that automotive companies are targeting for the ECUs, and enables to parallelize with low overhead.

In this paper, a crankshaft control program which is one of the important engine control code is parallelized. One of the most important characteristics from the point of parallel processing, they have no loop parallelism that has been used in multi-core or multi-processors for a long time. Therefore, the proposed method extracts more parallelism among functions and conditional branches as much as possible, and minimizes run-time overhead as less as possible using restructurings.

A. *Characteristics of Engine control programs*

The hand-written engine programs have the following characteristics to control systems strictly.

- They are composed of small basic blocks which cost less than 100 clock cycles.
- There are composed of many conditional branches related to sensors and control modes.
- They have only a few parallelizable loops with very small execution cost.

All these characteristics prevent ordinary parallelizing compilers from parallelizing engine control programs because of their too fine granularity, complicated control flow structures and little loop parallelism. In order to parallelize those programs with low run-time overhead, the following three items are applied as the basic parallelizing strategy:

- Coarse grain task parallelization among basic blocks
- Static scheduling of basic blocks on MTG to processor cores
- Code restructuring to improve parallelism considering complicated control flow structure

These items are described in following sections.

B. *Coarse Grain Parallelization for Engine Control programs*

The target engine control programs like crankshaft control are composed of conditional branches and small basic blocks without parallelizable loops that have been used in multi-core processors for a long time. Almost all basic blocks cost less than 100 clock cycles. Also fine grain parallelization cannot be applied because they have many conditional branches which prevent compilers from fine grain parallelization.

For these reasons, current product compilers cannot parallelize this kind of automotive control programs. Also, accelerators cannot be applied to this application because of conditional branches. The traditional loop parallelization technique widely used for multi-core processors can not apply the target engine control program since the program is composed of a series of conditional branches, assignment statements, and subroutine calls.

The coarse grain task parallel processing by the OSCAR compiler [4] is suitable since function calls and basic blocks can be defined as a task in addition to loops. Fig. 3 shows a MTG of the crankshaft control program which is parallelized in this paper. In this graph, yellow nodes represent subroutine blocks and red nodes represent basic blocks. In this paper, though the OSCAR compiler can apply near the fine grain parallel processing and loop iteration level parallel processing hierarchically, proposed method utilizes only coarse grain parallelization in the OSCAR compiler because they have no parallelizable loop and many conditional branches.

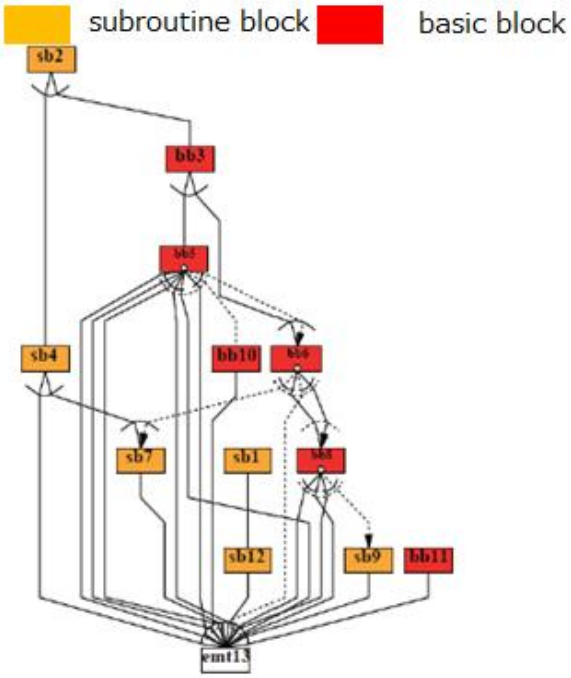

Fig. 3. Macro-task graph of a crankshaft control

C. *Static Scheduling for engine control programs*

It is required for engine control software to guarantee real-time constraints because this software has hard real-time constraint and to minimize run-time overhead because this program runs in the cycle of a few millisecond on a embedded processor. For these reason, static scheduling is applied in this paper instead of dynamic scheduling which has run-time overhead. However, these MTs in Fig.3 which have conditional branches such as bb5, bb6 and bb8 cannot be scheduled statically because the compiler cannot see if the branch is taken or not at compile time. The OSCAR usually applies dynamic scheduling for a program which has conditional braches.

For application which it to be applied static scheduling, compiler can hide all conditional branches in MTG using task fusion. Fig.4 shows MTG using task fusion. A block5 in this MTG is merged block using task fusion and has conditional branches inside the merged MT. In Fig.3, the OSCAR compiler fuses a group of conditional branches of bb5 to bb11. The OSCAR compiler generates MTG which has only data dependencies and enables to assign tasks in Fig.4 statically. Also, profiling based cost is used to enhance static scheduling in this paper.

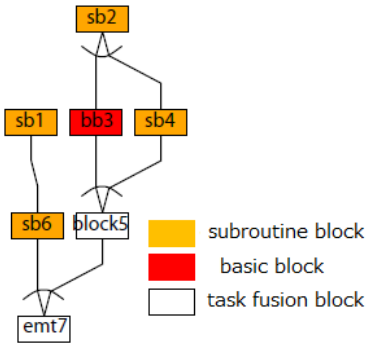

Fig. 4. Macro-task graph of a crankshaft program using task fusion

D. Restructuring For Engine Control programs

As shown in Fig. 4, sb1 and sb6 seem to be able to be executed in parallel with other tasks because sb1 and sb6 has no dependency on other tasks. However, because the execution times of sb1 and sb6 occupy just 1% of the whole execution time and the critical path composed of sb2, sb4 and block5 in this graph occupies about 99% of the whole execution time, an inline expansion is applied to sb2 and sb4 in order to exploit more parallelism over hierarchies or nested levels.

In the proposed method, to improve coarse grain task parallelism the OSCAR Compiler uses a selective inline expansion method. This selective inline expansion chooses function calls which have a coarse grain parallelism inside callee functions and also have large execution cost. Then, it applies inline expansion to the selected function calls to exploit sufficient parallelism keeping the code size as small as possible.

Fig. 5 shows a MTG of the restructured program by which a selective inline expansion was applied. As shown in Fig. 5, more coarse grain parallelism is exploited than that of the MTG in Fig. 4. However, the execution time of block36 still occupies about 70% of the whole execution time. The theoretical maximum speed-up ratio which is calculated as (1) is about 1.1 times because the critical path in this MTG occupies about 90% of the whole execution time.

Then, the block36 becomes the next target since this large execution cost and still has coarse grain parallelism inside it. Fig. 6 shows a simplified image of the block36. There are sb2, sb3 and sb4 inside a then-clause of the if-statement as a shown in the left side of Fig. 6. These subroutine calls are assigned onto the same processor core though there is no dependence among them. Fig.7 shows a MTG of example source code in Fig.6. These coarse grain parallelisms among them cannot be exploited since the if-statement and these subroutine calls are packed into the same MT to minimize scheduling overhead.

Here, this if-statement is duplicated to exploit coarse grain task parallelism among sb2, sb3 and sb4. As a shown in Fig. 6, the if-statement is duplicated for each subroutine call, then each duplicated if-statements and corresponding subroutine call are packed into same MT. Thus, coarse grain parallelism among those subroutine calls can be efficiently exploited. Such the duplication can be applied when variables used in a condition-expression is not changed in a then-clause like the sb2, sb3 and sb4 in this example.

Fig. 8 shows a MTG of the program after the inline expansion and the conditional branch duplication have been applied. In this graph, the average cost of MT is about 3,000 clock cycles. In addition, maximum cost of MT is about 10,000 clock cycles, and minimum cost is less than 100 clock cycles. The static scheduling for coarse grain parallelism imposes little synchronization overhead for such the task granularity. The critical path in this graph occupies about 60% of the whole execution time. The proposed methods reduce the critical path from 99% to 60%. Selective inline expansion and conditional branch duplication allow us to exploit remarkable coarse grain parallelism. Finally theoretical maximum speedup ratio which is calculated as (1) is about 1.6 times in this graph because the critical path occupies about 60% of the whole execution time.

Theoretical Maximum Speedup Ratio

=(Whole Execution Time)/(Execution Time on the Critical Path) (1)

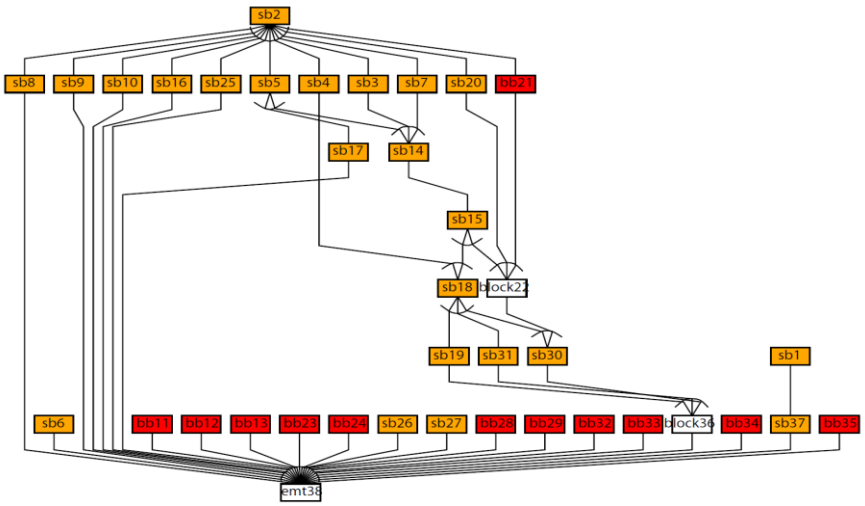

Fig. 5. Macro-task graph of the automotive engine control software after inline expansion

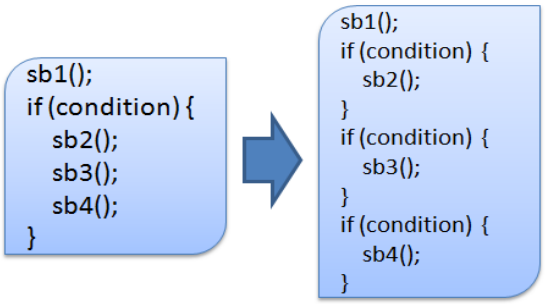

Fig. 6. Exmaple source code of Conditional branch duplication

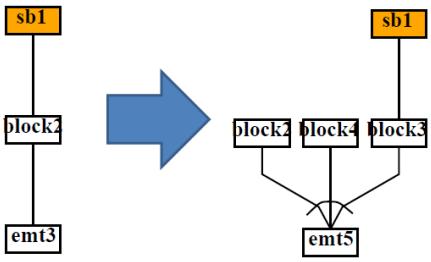

Fig. 7. Exmaple MTG of Conditional branch duplication

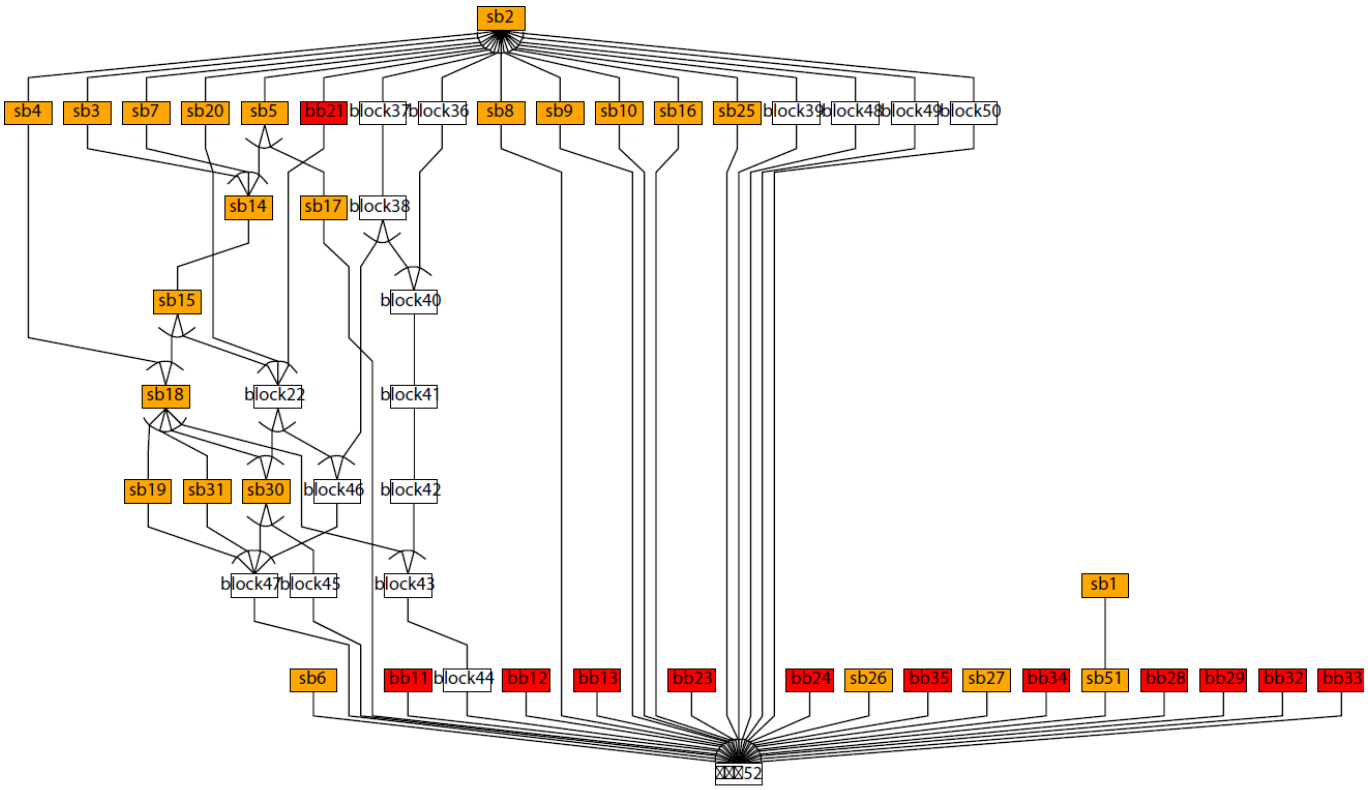

Fig. 8. Macro-task graph of the automotive engine control software after inline expansion and conditional branch duplication

IV. Performance Evaluation of Parallelized Engine Control Code on The RP-X

This paper uses the embedded multi-core processor RP-X developed by the authors with Hitachi and Renesas Electronics, Tokyo Institute of Technology [13] to evaluate the performance of the parallelized crankshaft control program which is one of the most important engine control programs.

A. Evaluation Environment

The RP-X processor has eight 648MHz SH-4A general-purpose processor cores, four 324MHz FEGA accelerator cores, two matrix processor "MX-2" and the video processing unit "VPU5", as shown in Fig. 9. Each SH-4A core consists of a 32KB instruction cache, a 32KB data cache, a 16KB local instruction/data memory (ILM and IDM in Fig.9.), a 64KB distributed shared memory (URAM in Fig.9), centralized shared memory (CSM) and a data transfer unit. The RP-X can change the clock frequency of processor cores, such as 648MHz, 324MHz, 162MHz and 81MHz.

In this paper, the clock frequency of processor cores is set to 81MHz in order to bring close to its actual automotive control unit. Table I shows the minimum access costs of local data memory (LDM), distributed shared memory (DSM), and centralized shared memory (CSM). LDM access needs 1 clock cycle, local DSM access needs 1 clock cycle, remote DSM access needs clock cycles, and off-chip CSM access needs 8 clock cycles at 81MHz.

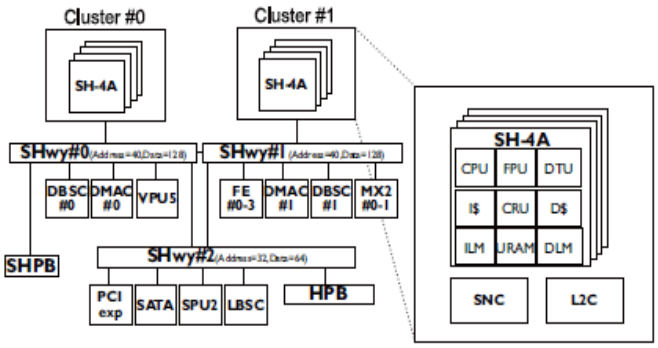

Fig. 9. The embedded multi-core processor RP-X

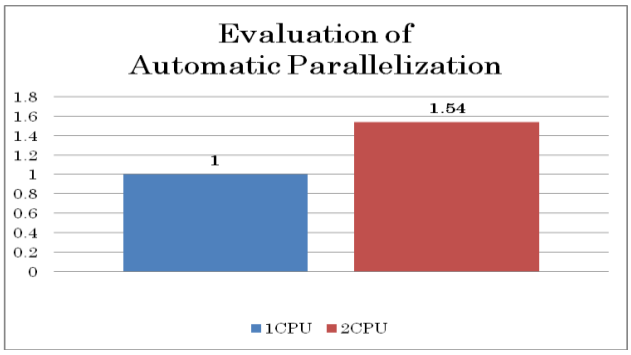

Fig. 10. The evaluation of automatic parallelization of the automotive engine control software on RP-X

TABLE I. MINIMUM ACCESS COSTS FOR LDM, DSM, AND CSM

Type of Memory	Latency(clock cycle)
LDM	1
DSM(local access)	2
DSM(remote access)	4
CSM	8

B. Performance Evaluation by the OSCAR Compiler

In this evaluation, a crank shaft control program which is working on current automobiles is evaluated. LDM is used for data which are accessed on only one core, DSM is used for synchronization and off-chip CSM is used for others. More parallelism is extracted to use selective inline expansion and conditional branch duplication as described in section III.D. The OSCAR compiler parallelizes this program using coarse grain parallelization as described in section III.B and schedule tasks to each processor cores statically utilizing task fusion as described in section III.C. In this paper, only two SH-4A cores on RP-X are used because next-generation automobiles plan to use a dual-core processor. In this static schedule, all MTs on the critical path are run by only CPU0.

Fig. 10 shows the result of the evaluation of the automotive engine control software parallelized by the OSCAR Compiler. The proposed method attains speedups of 1.54 times with 2 processor cores compared with the sequential execution. This result is near the theoretical speedup ratio mentioned before (1.6 times), though it is a little lower than the theoretical speedup ratio because of thread generation overhead and memory access overhead for shared memory. This result has shown possibility that the engine control codes are parallelized by automatic compiler on actual multi-cores and more sophisticated control programs are applicable because performance is improved using a multi-core processor.

V. CONCLUSIONS

This paper has proposed the parallelization scheme by the OSCAR compiler of the automotive engine control software which has been unable to parallelize before The original hand-written sequential engine control program provided by Toyota Motor Corporation which has a lot of small basic blocks and conditional branches is restructured with selective inline expansion and the conditional branch duplication in order to exploit coarse grain task parallelism. The OSCAR compiler has parallelized this program using coarse grain parallelization and scheduled statically using task fusion. The parallelized program has been evaluated on 2 processor cores on the embedded multi-core RP-X with LDM, DSM and CSM because next-generation automobiles plan to use a dual-core processor. This evaluation shows performance improvement of 1.54 times speed-up using 2 cores compared with sequential execution automatically. This paper has succeeded to attain a close result to the theoretical speedup ratio of 1.6. The result shows that the OSCAR Compiler can exploit parallelism from the automotive engine control software, which is composed of a series of conditional branches, assignment statements and subroutine calls. In addition, this result has possibility to realize more sophisticated control unit for safety, comfortable and energy efficient driving which current ECUs cannot attain because performance of an ECU can be improved using a multi-core processor.

ACKNOWLEDGMENT

This work was supported by Toyota Motor Corporation. I would like to express appreciation to Mr. Mitsuo Sawada from Toyota Motor Corporation.

REFERENCES

[1] K Seo, T Chung, H Heo, and K Yi. Coordinated implementation and processing of a unified chassis control algorithm with multi-central processing unit. JAUTO1346 IMechE Vol. 224 Part D: J. Automobile Engineering, 2009.

[2] Kyungil Seo, Taeyoung Chung, Hyundong Heo, Kyongsu Yi, and Naehyuck Chang, “An Investigation into Multi-Core Architectures to Improve a Processing Performance of the Unified Chassis Control Algorithms,” SAE Int.J.Passeng.Cars-Electron.Electr.Syst., pp. 53-62, 2010.

[3] Dinesh Padole, and Preeti Bajaj, "FUZZY ARBITER BASED MULTI CORE SYSTEM-ON-CHIP INTEGRATED CONTROLLER FOR AUTOMOTIVE SYSTEMS: A DESIGN APPROACH," CCECE, pp. 1937-1940, 2008.

[4] Kasahara, H., Obata, M., Ishizaka, K., "Automatic coarse grain task parallel processing on smp using openmp. Proc of The 13th International Workship on Languages and Compilers for Parallel Computing, " 2000

[5] M. Mase, Y. Onozaki, K. Kimura, and H. Kasahara, "Parallelizable c and its performance on low power high performance multicore processors," In Proc. of 15th Workshop on Compilers for Parallel Computing, Jul. 2010.

[6] The Motor Industry Software Reliability Association, "Guidelinesfor the use ofthe C Language in Vehicle Based Software," Oct. 1998.

[7] The Motor Industry Software Reliability Association, "MISRA-C 2004 Guidelinesfor the use of the C language in critical systems," Oct. 2004.

[8] The Motor Industry Software Reliability Association, "MISRA-C 2013 Guidelinesfor the use of the C language in critical systems," Mar. 2013.

[9] K. Kimura, M. Mase, H. Mikami, T. Miyamoto, J. Shirako, and H. Kasahara. Oscar api for real-time low-power multicores and its performance on multicores and smpservers. Lecture Notes in Computer Science, 5898:188-202, 2010.

[10] A. Hayashi, Y. Wada, T. Watanabe, T. Sekiguchi, M. Mase, J. Shirako, K. Kimura, and H. Kasahara. Parallelizing compiler framework and api for power reduction and software productivity of real-time heterogeneous multicores. Lecture Notes in Computer Science, 6548:184-198, 2011.

[11] H. Kasahara, H. Honda, A. Mogi, A. Ogura, K. Fujiwara, and S. Narita, "A multi-grain parallelizing compilation scheme for OSCAR(Optimally scheduled advanced multiprocessor," In Proceedings of the Fourth International Workshop on Languages and Compilers for Parallel Computing, pp. 283-297, August 1991.

[12] H. Honda, M. Iwata, and H. Kasahara, "Coarse grain parallelism detection scheme of a Fortran program," Trans. of IEICE, Vol.J73-D-1, No.12, pp. 951-960, Dec. 1990.

[13] Y. Yuyama, M. Ito, Y. Kiyoshige, Y. Nitta, S. Matsui, O. Nishii, A. Hasegawa, M. Ishikawa, T. Yamada, J. Miyakoshi, K. Terada, T. Nojiri, M. Satoh, H. Mizuno, K. Uchiyama, Y. Wada, K. Kimura, H. Kasahara, and H. Maejima, "A 45nm 37.3gops/w heterogeneous multi-core soc," IEEE International Solid-State Circuits Conference, ISSCC, pp. 100-101, Feb. 2010.