Automatic Local Memory Management for Multicores Having Global Address Space

Kouhei Yamamoto[1], Tomoya Shirakawa[1], Yoshitake Oki[1], Akimasa Yoshida[1,2], Keiji Kimura[1], and Hironori Kasahara[1]

[1] Department of Computer Science and Engineering, Waseda University, Japan
[2] Graduate School of Advanced Mathematical Sciences, Meiji University, Japan
{yamamoto,tshira,okiyoshi}@kasahara.cs.waseda.ac.jp,akimasay@meiji.ac.jp,
kimura@apal.cs.waseda.ac.jp,kasahara@waseda.jp
http://www.kasahara.cs.waseda.ac.jp

Abstract. Embedded multicore processors for hard real-time applications like automobile engine control require the usage of local memory on each processor core to precisely meet the real-time deadline constraints, since cache memory cannot satisfy the deadline requirements due to cache misses. To utilize local memory, programmers or compilers need to explicitly manage data movement and data replacement for local memory considering the limited size. However, such management is extremely difficult and time consuming for programmers. This paper proposes an automatic local memory management method by compilers through (i) multi-dimensional data decomposition techniques to fit working sets onto limited size local memory (ii) suitable block management structures, called Adjustable Blocks, to create application specific fixed size data transfer blocks (iii) multi-dimensional templates to preserve the original multi-dimensional representations of the decomposed multi-dimensional data that are mapped onto one-dimensional Adjustable Blocks (iv) block replacement policies from liveness analysis of the decomposed data, and (v) code size reduction schemes to generate shorter codes. The proposed local memory management method is implemented on the OSCAR multi-grain and multi-platform compiler and evaluated on the Renesas RP2 8 core embedded homogeneous multicore processor equipped with local and shared memory. Evaluations on 5 programs including multimedia and scientific applications show promising results. For instance, speedups on 8 cores compared to single core execution using off-chip shared memory on an AAC encoder program, a MPEG2 encoder program, Tomcatv, and Swim are improved from 7.14 to 20.12, 1.97 to 7.59, 5.73 to 7.38, and 7.40 to 11.30, respectively, when using local memory with the proposed method. These evaluations indicate the usefulness and the validity of the proposed local memory management method on real embedded multicore processors.

Keywords: Parallelizing Compiler, Local Memory Management, Multicore, Global Address Space, DMA, Data Decomposition

1 Introduction

As modern embedded systems demand for more performance with lower power consumption, the architectural design of multicore processor has succeeded in pursuing both requirements. However, in embedded multicores for hard real-time control systems such as automobile engine control units, cache memory cannot be used to meet hard deadline constraints. In these systems, multicore architectures having local memories with addresses mapped to parts of global address space have been generally used. Examples of such embedded multicore processors are Renesas's RP2 [16] and V850E2/MX4 [19].

Local memory is a fast on-chip memory which can be explicitly controlled by software. Typically, local memory is reserved for data that is extensively reused throughout the entire program. A similar class of fast on-chip software controllable memory is scratch-pad memory [1, 12]. Although the functionality of scratch-pad memory is similar to local memory, scratch-pad memory is generally smaller in size and is specialized for data locality on a finer region of the program.

The low latency and software manageable characteristics of local memory offers guaranteed execution timing, which is a crucial property for real-time control embedded domains. Moreover, optimal mapping of data onto local memory through software implementations can achieve data locality and satisfy deadline requirements by removing runtime uncertainties by cache miss hits.

There remains a major obstacle when utilizing software based local memory for embedded systems with multicore processors: the mapping and decomposition of data onto local memory of each processor core. In other words, the use of local memory considering data locality requires comprehensive control of data placement and eviction by the programmer. To overcome this difficulty, a promising approach is to build a compiler algorithm to automatically decompose data and insert data transfer codes. Such compiler based approach will not only prevent error-prone code productions otherwise done by the programmer, but will also allow local memory optimizations to become available for a wide range of applications.

In this paper, a local memory management method with data decomposition for software controlled un-cached local memory on multicore processors is proposed to satisfy deadline constraints and obtain high performance. In particular, the method realizes local memory management techniques that determine data placement and replacement on local memory considering data locality over the whole input C program. The data decomposition process decomposes multi-dimensional arrays for each nested level. Additionally, data transfer costs between multiple processor cores are mitigated through Direct Memory Access (DMA) controllers. To allow automatic parallelization for various applications using local memories, the method is implemented on OSCAR compiler, a C and Fortran source-to-source multi-grain and multi-platform parallelizing compiler [14]. The effectiveness of the proposed method is demonstrated through several benchmark applications written in Parallelizable C [11] that have various data sizes and dimensions.

The rest of the paper is organized as follows. Section 2 introduces related works. Section 3 covers the proposed data decomposition method and local memory management method. Section 4 shows evaluation results of the proposed methods on benchmark applications. Section 5 concludes the paper.

2 Related Works

There have been many researches on local memory management methods.

In static data management, data partitions and allocations remain constant throughout the lifetime of the program. Avissar et al. proposed a compiler strategy that automatically partitions and allocates data onto different memory units [3]. Similar methods were reported by Steinke et al., utilizing a compiler extension technique for embedded systems to analyze the most frequently used data and variable within the application for static mapping onto local memory [4]. Steinke's analysis focuses mainly on reducing energy consumption by utilizing energy efficient local memories over caches. Panda et al. reported a method to partition scalars and array variables and map them onto on-chip scratchpad memory at compile time [1]. Excess variables that could not fit on scratch-pad memory are mapped onto off-chip memory. However, their approach is limited to single thread execution environments. For static allocation of data onto multicore processor environments, Che et al. presented an integer linear programming formulation and a heuristic technique to model code overlays and communication costs to maximize throughput of stream programs [5]. Their method shows improvement on stream programs for static allocations, but does not mention explicit mapping managements of data onto local memory. Similarly, Issenin et al. proposed a data reuse method for loops on multicore processor environments [9]. Their method focuses on data locality within loops, but does not consider locality between tasks of the entire program.

To achieve flexibility for allocated variables during the entire runtime of the program, several dynamic allocation algorithms for local memory are presented. Udayakumaran et al. proposed a dynamic allocation method that considers runtime behaviors of the program running on a single core processor [6]. Specifically, their method copies frequently accessed data onto scratch-pad memory by compiler codes dynamically and evict unused data to free scratch-pad space. However, the method is relevant only for single thread environments, neglecting communication and synchronization costs that occur for multicore processor environments. For multicore processor systems, Guo et al. proposed a data allocation algorithm for scratchpad memories to reduce memory access cost [7]. They incorporate a data duplication algorithm to extensively copy specific data onto remote processor core's scratch-pad memory to further reduce memory access costs. However, their method does not present explicit management techniques for mapping data onto local memory. Kandemir et al. proposed a data tiling strategy for multicore processor systems [8]. Their method focuses on array-intensive applications, and aims to increase inter-processor data sharing opportunities and minimize off-chip memory requests. Their technique, however,

considers data locality within loops and does not extract locality that spreads across the entire program.

As presented in this section, partitioning and allocating data onto software managed memory has been attempted by various researchers. However, the majority of the proposed solutions consider static or dynamic allocations of data that only assume single thread environments. Moreover, previous methods do not target data locality stretched across multiple coarse-grain tasks or local memory management techniques that extensively control the position of the stored data on local memory. Therefore, an integrated analysis of dynamically allocating and evicting data on coarse-grain tasks, including arrays accessed within nested loops, for software managed local memory under a multicore processor environment, to our knowledge, has not been attempted so far.

3 The Proposed Local Memory Management Method

The target architecture of the proposed method consists of multiple processor cores with an on-chip and/or off-chip centralized shared memory, or CSM. An example architecture is the OSCAR multicore architecture shown in Fig.1 [11]. Each processor core is equipped with local data memory, or LDM, for core private data and a distributed shared memory, or DSM, for data shared among processor cores. In embedded multicores, since the local memories are mapped to global address space, they can be recognized as distributed shared memory. Often, local memory is implemented by a single port memory and distributed shared memory is implemented using two ports memory. Within this memory architecture, the proposed method aims to exploit data locality of core private data on local memory. The main idea of the proposed method is to decompose data so that a working set can fit on LDM and the data on LDM can be reused among different coarse-grain tasks.

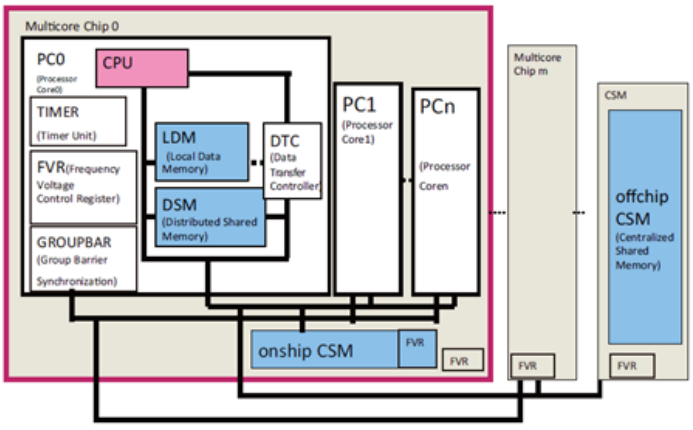

Fig. 1. Overview of the OSCAR Multicore Architecture

An overview of the proposed compiler local memory management method for multicores using adjustable block assignment and replacement technique is summarized below.

1. Chooses block sizes for data transfer between shared memory and LDM specifically for each application.
2. Divides all data in the application into constant size aligned block structures called Adjustable Blocks. In contrast to other block allocation schemes such as buddy memory allocators where block sizes are restricted to multiples of powers of two and the granularity of the block is defined as a single page size, Adjustable Blocks divide data into integer divisible sizes and can further divide blocks into single word sizes for scalar variables.
3. Hierarchically decomposes multi-dimensional arrays by the outer-most loop dimension until the decomposed array fits inside the chosen block size.
4. Maps each decomposed array to assigned blocks on LDM considering locality optimizations.
5. Schedules eviction and reloading of blocks from LDM and shared memory. Blocks with dead variables or blocks with variables that will be reused in the most distant future have high replacement priorities.

3.1 Coarse-Grain Task Parallelization [14, 15]

The input C program is initially divided into coarse-grain tasks, or tasks with sufficient amount of work that can be efficiently scheduled to processors by the compiler. Coarse-grain tasks are also called Macro Tasks (MTs), and are divided into three categories: Basic Blocks (BBs), Repetition Blocks (RBs) for loops, and Subroutine Blocks (SBs) for functions. RBs and SBs are hierarchically decomposed into smaller MTs if coarse-grain task parallelism still exists within the task, as shown in RB number 7 in Fig.2. After all MTs for the input program are generated, they are analyzed to produce a Macro Flow Graph (MFG). MFGs depict the control flow and the data dependencies of the entire input program as a graph structure. Further, Macro Task Graphs (MTGs) are generated by analyzing the earliest executable condition [14] of every MT and analyzing the control dependencies and data dependencies among MTs on the MFG. MTGs illustrate parallelism among MTs and are utilized as the baseline structure for the proposed data localization method to extract data locality from the entire input program. An example MFG and MTG is illustrated in Fig.2 and Fig.3.

The scheduling of MTs to processor cores can be done either statically at compile time or dynamically at run time. The decision of static or dynamic scheduling of MTs depends on the topology and the branch structure of the MTG of the input program.

3.2 Data Decomposition Method

By analyzing the MTG of the input program, the data decomposition phase decomposes RBs connected by data dependence edges on the MTG so that data transfers among the data dependent RBs can be made through LDMs.

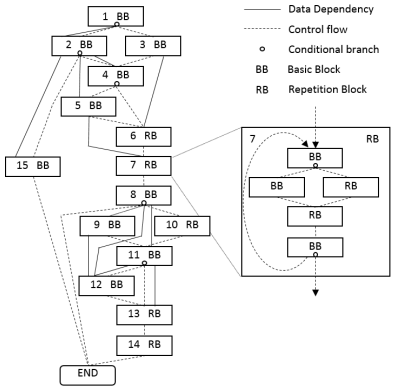

Fig. 2. Macro Flow Graph (MFG)

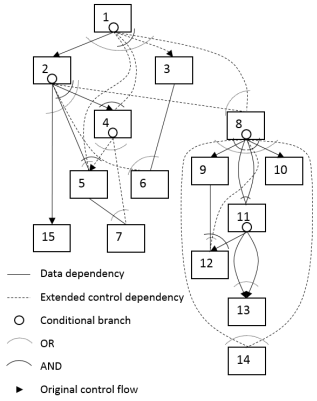

Fig. 3. Macro Task Graph (MTG)

The data decomposition process begins by creating groups of loops, or Target Loop Groups (TLGs), from the MTG that access the same arrays. The loops within these groups are then analyzed for dependencies through the Inter-Loop Dependency (ILD) analysis phase [15]. Once this dependency analysis completes, the number of required decompositions, namely the number of small loops each loop should be decomposed into, is decided from the available LDM size and the array sizes accessed by the decomposed loops.

3.2.1 Target Loop Group (TLG) Creation and Inter-Loop Dependency (ILD) Analysis

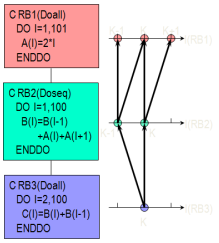

Fig. 4. Example of ILD Analysis

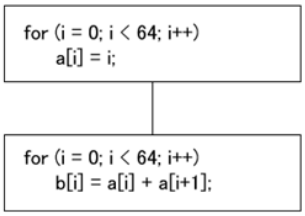

Fig. 5. A Simple TLG with Two Loops

Loops that access the same array are gathered into group of loops called TLGs. The loop with the largest estimated time within a TLG is chosen as the baseline loop for that specific TLG. This baseline loop is used as a criterion for the data dependency check on the ILD analysis phase. Fig.4 depicts an example where the baseline loop is chosen as RB3, which is data dependent on indices

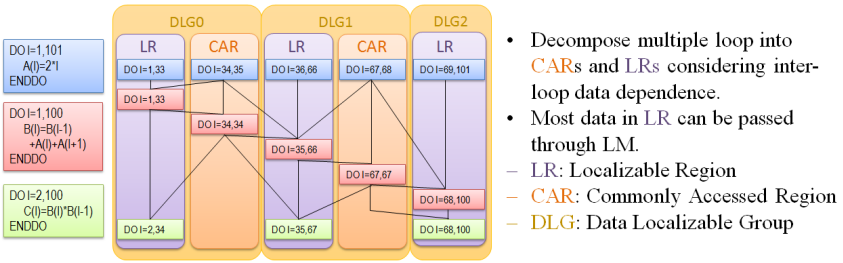

Fig. 6. Example of Localizable Regions (LR) and Commonly Acccessed Regions (CAR)

i and i-1 of RB2 and i-1, i, and i+1 of RB1. The ILD analysis phase resolves data dependencies between loops within the generated TLGs and detects relevant iterations of those loops that have dependence with the iterations of the baseline loop. Moreover, the ILD analysis phase detects Commonly Accessed Regions (CAR), or array regions accessed by multiple processors, and Localizable Regions (LR), or array regions accessed by a single processor, of each TLG. Data reuse can be performed on arrays accessed by LRs that stretch across multiple loops within a TLG, since LRs encompass loop regions that can be safely kept in LDMs of each processor. An example diagram of IR and CAR is shown in Fig.6. Fig.5 shows an example of a TLG. In this example, the second loop is chosen as the baseline loop since its estimated cost is larger than the first loop. For the indices of array a, iteration i of the baseline loop has dependencies on iteration i of the first loop and iteration i+1 of the current loop.

3.2.2 Decomposition Count

The working set size of data shared across multiple decomposed small loops after decomposition must be strictly less than the available LDM size of the target processor core. To mitigate the algorithmic complexity for parameter calculations, the presented method chooses decomposition counts, namely the number of small data portions each data should be decomposed into, that allows all decomposed arrays within a TLG to simultaneously exist on LDM. By simplifying the decomposition decision algorithm, the method guarantees mapping of arbitrary sized arrays onto LDM with low overhead.

3.2.3 Extending the Data Decomposition Method to Multi-dimensional Loops

Previous data decomposition schemes that exploit data locality mostly focus on dividing the outer-most loop of a nested loop. Hence, these methods can not treat cases where decomposition of the outer loop fails to generate array sizes smaller than the available LDM size. In contrast, the data localization method presented in this paper is safely applicable to loops with arbitrary dimensions.

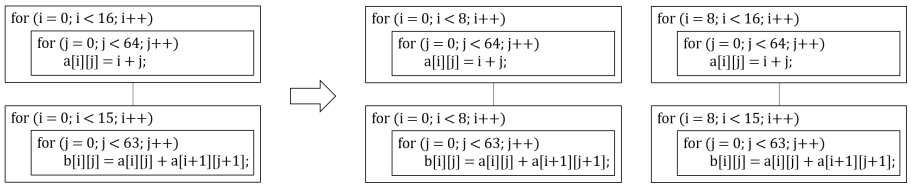

Fig. 7. Decomposition of Only the Outer Loop

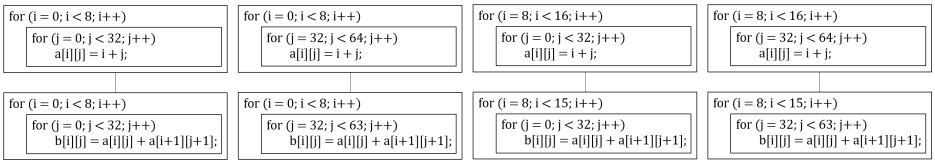

Fig. 8. Decomposition of Outer and Inner Loops

Fig.7 depicts an example of decomposing only the outer-most loop of a nested loop. In this example, dividing the outer-most loop still leaves behind a 64 iteration inner loop, which accesses an array shared between two loops. Previous data decomposition methods will fail to place the array onto LDM if this target array size is larger than the available LDM size. Fig.8 illustrates an example of decomposing both the outer and the inner loop of the original loop code of Fig.7. By calculating the necessary decomposition count from the total array size and the LDM size, the decomposition process not only terminates on the outer-most loop, but continues inwardly onto inner loops and decides the decomposition count for each nest level. By hierarchically dividing each nest level, data size of the accessed array can be significantly reduced, ultimately allowing programs with large data size to adjustably fit on LDM.

3.3 Scheduling of Decomposed Loops

Decomposed loops with common iteration ranges are placed and executed on the same processor core to achieve data locality. Decomposed loops with accessing the same iteration ranges are grouped together into Data Localizable Groups (DLGs) [15]. An example of DLG is shown in Fig.6. Once DLGs are generated, the decomposed small loops within each DLG are statically scheduled to the same processor core.

3.4 Local Memory Management

The challenge of mapping and evicting decomposed data for LDM still remains. To address this problem, the LDM memory management phase of the method utilizes scheduling results of DLGs to make appropriate mapping decisions on LDM and insertion choices of data transfer codes for every decomposed data.

After mapping decisions are determined from the DLG scheduling phase, the method adopts Adjustable Blocks and Template Arrays for the actual mapping of the decomposed data onto LDM [10]. Adjustable Blocks are hierarchical structures of constant size blocks and are used to flexibly choose appropriate memory block sizes for each application program. Template Arrays are mapping structures that maps multi-dimensional arrays to specific one-dimensional blocks of LDM, and are also used to maintain code readability of the indices of the arrays.

3.4.1 Adjustable Blocks

The decomposition count of data varies with the characteristic and the complexity of the application, which require the LDM management method to handle arrays with arbitrary sizes. However, simply adopting memory blocks with varying sizes is insufficient, since the data placement and eviction process induces memory fragmentation. To avoid such inefficiency, the proposed method maps data onto LDM using hierarchically aligned constant size blocks called Adjustable Blocks [10]. The basic structure of Adjustable Block is depicted in Fig.9. Adjustable Blocks allow flexible selection of block sizes depending on the data size present in the input program, and can be further divided into smaller blocks with integer divisible sizes of the parent block, unlike buddy memory allocators where block sizes are limited to multiples of powers of two. The constant size blocks and the hierarchical structure of Adjustable Blocks allow efficient mapping of blocks with varying size and dimension onto LDM as well as avoiding performance critical fragmentations of LDM. For the current implementation of the method, the block sizes of Adjustable Blocks are reduced by powers of 2 for each level down the hierarchy.

When the Adjustable Block size for each application program is decided, the LDM address space is decomposed into a set of blocks. During parallel execution, the decomposed data are loaded to a block and evicted from the block managed by the compiler.

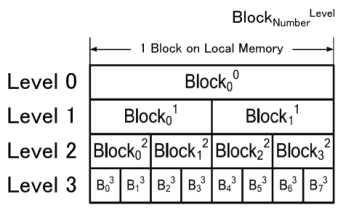

Fig. 9. Hierarchical Structure of Adjustable Blocks

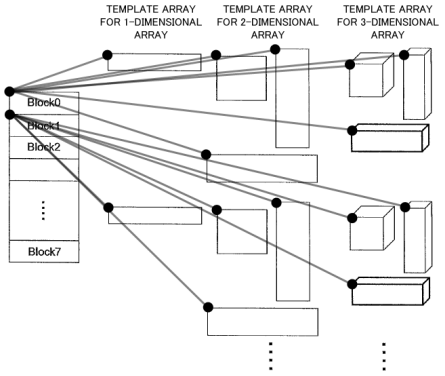

Fig. 10. Overview of Template Arrays

3.4.2 Template Arrays

LDM can be represented as a one dimensional array. Therefore, when a multi-dimensional array is allocated onto LDM, the index calculation of the array becomes complex. To overcome this complexity, the method introduces an array mapping technique called Template Arrays [10]. Fig.10 displays an overview of Template Arrays. The basic idea of Template Arrays is that each block on LDM corresponds to multiple empty arrays with varying dimensions. These arrays have an additional dimension augmented to its structure to store the corresponding block number. By maintaining block numbers for every array on each block, the method manages to systematically decide which region and block of LDM memory is appropriate for the target decomposed array. Moreover, by choosing a block that has the same dimension with the target array, the mapping provides better readability for the array indices.

3.4.3 Block Eviction Policy

To adjustably utilize LDM during the runtime of the program, the proposed method appropriately evicts memory blocks guaranteed to be unused or to be reused latest in the future from LDM to off-chip shared memory to create new spaces for incoming variables, unlike Least Recently Used (LRU) policies where variables with the longest unused period are evicted. The live and dead information of each variable is analyzed by the OSCAR compiler. In order to minimize data transfer latencies and fully utilize data locality, data with high probability of being accessed again continues to reside on LDM. In particular, the method adopts the following block eviction priority policy to maintain data locality, listed from most to least significance:

1. Dead variables (variables that will not be accessed further in the program)
2. Variables that are accessed only by other processor cores
3. Variables that will be later accessed by the current processor core
4. Variables that will immediately be accessed by the current processor core

3.5 Data Transfer Between Off-chip Memory

Data transfer codes between LDM and off-chip shared memory is inserted according to the scheduling results of the DLGs as presented in section 3.3. The method assumes DMA controllers as the underlying data transfer hardware to allow fast and asynchronous burst transfers between processor cores. The current implementation of the method explicitly inserts data transfer codes before MTs that load data and after MTs that store data. Overlapping of data transfers and task executions is not achieved due to a hardware bug in the RP2 multicore processor used in this evaluation. Still, this MT-granularity data transfer policy minimizes synchronization overheads and maintains data coherence with other processing cores that work on the same array.

3.6 Code Compaction Method

3.6.1 Overview of the Code Compaction Method

The LDM management approach presented by previous researches produces duplicated code for each decomposed loop. This straightforward scheme generates multiple copies of the loop body with different lower and upper bounds, effectively creating unique loop codes for each decomposition count. To prevent such

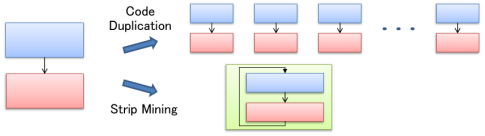

Fig. 11. Overview of the Strip Mining Technique for Nested Loops

```
for (ii = 0; ii < 15; ii+=8)
  for (jj = 0; jj < 63; jj+=32)
    for (i = ii; i < min(15,ii+8+1); i++)
      for (j = jj; j < min(63,jj+32+1); j++)
        a[i][j] = i + j;
    for (i = ii; i < min(15,ii+8); i++)
      for (j = jj; j < min(63,jj+32); j++)
        b[i][j] = a[i][j] + a[i+1][j+1];
```

Fig. 12. Code Compaction Applied to the Nested Loop on Fig.7

code bloat, the proposed method adopts code compaction techniques based on strip mining [13]. Fig.11 depicts the strip mining scheme incorporated to the method. By applying mid-grain parallelization to the outer-most blocking loop, proper mapping onto processor cores and execution order can be guaranteed without applying scheduling.

3.6.2 Code Compaction Method for Multi-dimensional Loops

To utilize code compaction techniques for multi-dimensional loops, iteration ranges among multiple loops within TLGs must first be aligned by loop peeling [17, 18]. After peeling the excessive iteration ranges for every loop, the target loops are fused as a single MT. Fig.12 shows an example with multi-dimensional loops, illustrating the code compaction method applied to the original loop code on Fig.7. Since the first and the second loops within the TLG on Fig.7 has different iteration ranges, the iteration of the first loop with indices i = 15 and j = 63 will be peeled to match up with the smaller iteration ranges of the second loop. Following this loop peeling, the method then performs loop decomposition. If the decomposition count is 2, each loop nest will be divided into 2 pieces, consequently performing strip mining with block sizes of 8 as the outer loop and 32 as the inner loop.

4 Evaluations

To show the effectiveness of the method, this section presents evaluation results on several benchmark applications. The method was implemented on the

OSCAR automatic parallelization compiler and tested on Renesas's RP2 SH4A processor based 8 core homogeneous multicore processor [16]. The RP2 multicore processor is based on the OSCAR multicore architecture shown in the previous section. Each processor core of RP2 is based on SH4A with 600MHz, and has dedicated LDM to freely load and evict data during program execution. To share data among processor cores, each core has access to a processor wide distributed shared memory. An overview of the RP2 architecture is depicted in Fig.13. RP2 is equipped with LDM (OLRAM) with a 1 clock cycle latency, distributed shared memory (URAM) on each processor core with a 2 clock cycle latency, and a 128MB DDR2 CSM with a 55 clock cycle latency. Data cache, or D$, is not used for the evaluation. Every processor core is connected with SHwy, which is Renesas's standard bus.

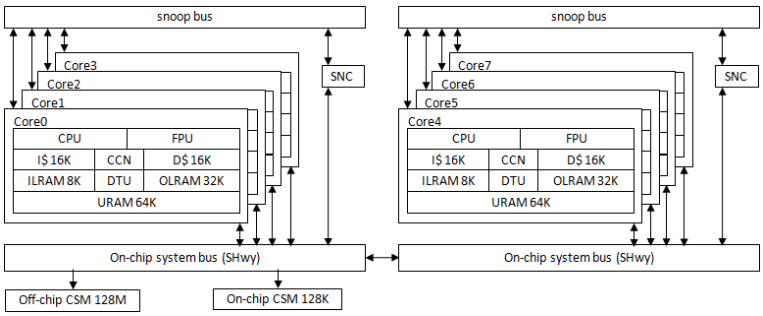

Fig. 13. Architecture of the RP2 Multicore Processor

4.1 Tested Applications

To evaluate the performance of the proposed data localization method, 5 sequential programs written in Parallelizable C [11], such as the example code in Fig.7 used for the explanation of the proposed method, an AAC encoder, a MPEG2 encoder, SPEC95 Tomcatv, and SPEC95 Swim were used. Tomcatv and Swim are chosen from the SPEC95 benchmark suite since both applications in this version have data size small enough to fit into the limited off-chip CSM size of RP2. The method applied one-dimensional decomposition to AACenc and Mpeg2enc, and two-dimensional decomposition to the sample program, Tomcatv, and Swim. These applications were compiled by the OSCAR source-to-source automatic parallelization compiler for multiple platforms with the proposed method integrated as part of OSCAR's analysis phase, followed by a backend compilation process by a native compiler for each target multicore processor to generate machine codes. The 4 programs, except the example program of Fig.7, are explained below.

– AACenc is an AAC encoder application provided by Renesas Technology. For evaluation, a 30 second audio file was used as input to generate an audio file with a bit rate of 128Kbps.
– Mpeg2enc is a MPEG2 encoder application which is part of the MediaBench benchmark suite. For evaluation, a 30 frame video with a resolution of 352 by 256 pixels was used as input.
– Tomcatv is a loop-intensive benchmark application from the SPEC CPU95 benchmark suite. Before performing the LDM management method, loop fusion and variable renaming were applied.
– Swim is a benchmark application that performs 2 dimensional array computations from the SPEC CPU95 benchmark suite. Before performing the LDM management method, loop distribution and loop peeling were performed.

4.2 Evaluation Results

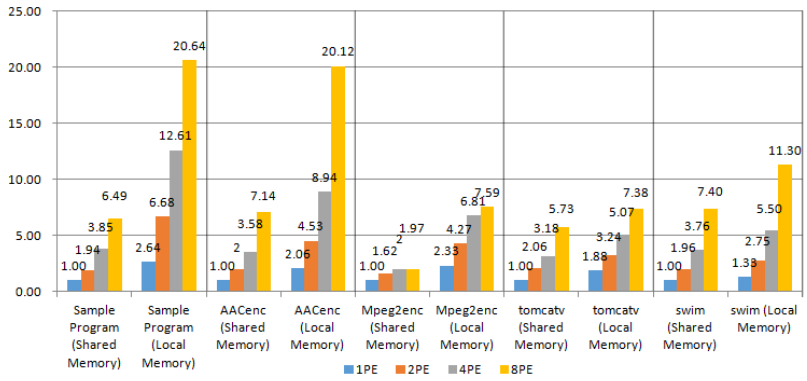

Fig. 14. Speedups of the Proposed Method (Local Memory) Compared to Executions Utilizing Shared Memory (Shared Memory) on Benchmarks Applications using RP2

Fig.14 shows the experimental results of the applications on the RP2 8 core processor. Since, to our knowledge, there are no other open-source compilers that explicitly manage LDM, the results compare executions of the applications that utilize the proposed LDM management method and off-chip CSM.

In the sample program of Fig.7, the parallelized program by the OSCAR compiler using off-chip CSM, or DDR2 memory, achieved speedups of 3.85 for 4 cores and 6.49 for 8 cores. On the other hand, the proposed LDM management method obtained better speedups, such as 2.64 for single core, 12.61 for 4 cores, and 20.64 for 8 cores, compared to single core executions using off-chip CSM.

For AACenc, the speedups using the off-chip CSM was 3.58 for 4 cores and 7.14 for 8 cores compared with single core environment. By contrast, the speedups for AACenc using the proposed LDM management method were 2.06

for 1 core, 8.94 for 4 cores, and 20.12 for 8 cores. For Mpeg2enc, the speedups obtained using the off-chip memory were 2.00 on 4 cores and 1.97 on 8 cores against sequential execution. The proposed method outperformed off-chip memory solutions by obtaining speedups of 2.33 for single core, 6.81 for 4 cores, and 7.59 for 8 cores. In Tomcatv, speedups achieved by utilizing the CSM were 3.18 for 4 cores and 5.73 for 8 cores. Compared to the CSM environment, the proposed method obtained higher speedups of 1.88 for 1 core, 5.07 for 4 cores, and 7.38 for 8 cores. For Swim, speedups using the off-chip CSM were 3.76 for 4 cores and 7.40 for 8 cores against 1 core execution. In contrast to those results, the proposed method showed speedups of 1.33 for 1 core, 5.50 for 4 cores, and 11.30 for 8 cores. The evaluations show that the proposed LDM management method achieves scalable speedups for embedded and scientific applications.

5 Conclusions

This paper has proposed automatic local memory management method with data assignment to adjustable blocks chosen for each application utilizing data assignment units between off-chip shared memory and local memory. The method also incorporates multi-dimensional templates that allow programmers to understand the parallelized program using local memory management. Utilizing local memory is necessary to satisfy deadline requirements for applications of embedded systems, such as automobile engine control programs, with multicore processors. This software managed local memory control approach successfully decomposes large size data into smaller chunks so that the working set fits on local memory, while avoiding fragmentation and maintaining readability of code using Adjustable Blocks and Template Arrays. Data transfer between local memory and off-chip memory is managed through insertion of data transfer codes between coarse-grain tasks. Additionally, the proposed method allows reuse of data on local memory over different loops. The proposed method further integrates code compaction technique to mitigate code bloat, allowing the technique to successfully decompose multi-dimensional arrays. The method was implemented on the OSCAR source-to-source parallelization compiler to automatically generate data locality optimized code. Evaluations were performed on the RP2 8 core multicore processor equipped with off-chip shared memory and local memory. For the sample program in Fig.7, the proposed local memory management method achieved a speedup of 20.64 times for 8 cores against sequential execution using off-chip shared memory of RP2. Similarly, on 8 cores using local memory, AACenc, Mpeg2enc, Tomcatv, and Swim obtained speedups of 20.12, 7.59, 7.38, and 11.30, respectively, against 1 core execution using the off-chip shared memory. These results reveal that the proposed automatic local memory management method is effective for reducing execution times for embedded applications with deadline constraints.

Acknowledgments. This work was partly supported by JSPS KAKENHI Grant Number JP15K00085.

References

1. Panda, P. R. et al.,"Efficient utilization of scratch-pad memory in embedded processor applications," Proc. of European conference on Design and Test, 1997.
2. Kandemir, M. et al.,"Dynamic management of scratch-pad memory space," Proc. of Design Automation Conference, 2001.
3. Avissar, O. et al.,"An optimal memory allocation scheme for scratch-pad-based embedded systems," ACM Transactions on Embedded Computing Systems, 2002.
4. Steinke, S. et al.,"Assigning program and data objects to scratchpad for energy reduction," Proc. of Design, Automation and Test in Europe Conference and Exhibition, 2002.
5. Che, W. et al.,"Compilation of stream programs for multicore processors that incorporate scratchpad memories," Proc. of Design, Automation and Test in Europe Conference and Exhibition, 2010.
6. Udayakumaran, S. et al.,"Dynamic allocation for scratch-pad memory using compile-time decisions," ACM Transactions on Embedded Computing Systems, 2006.
7. Guo, Y. et al.,"Data placement and duplication for embedded multicore systems with scratch pad memory," IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, 2013.
8. Kandemir, M. et al.,"Exploiting shared scratch pad memory space in embedded multiprocessor systems," Proc. of Design Automation Conference, 2002.
9. Issenin, I. et al.,"Multiprocessor system-on-chip data reuse analysis for exploring customized memory hierarchies," Proc. of Design Automation Conference, 2006.
10. Kasahara, H. et al.,"U.S. Patent No. 8,438,359," Washington, DC: U.S. Patent and Trademark Office, 2013.
11. Kimura, K. et al.,"Oscar api for real-time low-power multicores and its performance on multicores and smp servers," International Workshop on Languages and Compilers for Parallel Computing, 2009.
12. Banakar, R. et al.,"Scratchpad memory: design alternative for cache on-chip memory in embedded systems," Proc. of international symposium on Hardware/software codesign, 2002.
13. Wolfe, M.,"More iteration space tiling," Proc. of ACM/IEEE conference on Supercomputing, 1989.
14. Kasahara, H. et al.,"A multigrain parallelizing compilation scheme for OSCAR (optimally scheduled advanced multiprocessor)," International Workshop on Languages and Compilers for Parallel Computing, 1991.
15. Yoshida, A. et al.,"Data-localization for fortran macro-dataflow computation using partial static task assignment," Proc. of international conference on Supercomputing, 1996.
16. Ito, M. et al.,"An 8640 mips soc with independent poweroff control of 8 cpu and 8 rams by an automatic parallelizing compiler," Proc. of IEEE International Solid State Circuits Conference, 2008.
17. Kennedy, K. et al.,"Optimizing Compilers for Modern Architectures: A Dependence-Based Approach," Morgan Kaufmann Publishers Inc., San Francisco, CA, USA. 2001.
18. Padua D. et al.,"Advanced compiler optimizations for supercomputers," Communications of the ACM 29, 1986.
19. https://www.renesas.com/en-in/products/microcontrollers-microprocessors/v850/v850e2mx/v850e2mx4.html