IEICE TRANSACTIONS on Electronics

VOL. E103-C NO. 3
MARCH 2020

A PUBLICATION OF THE ELECTRONICS SOCIETY
The Institute of Electronics, Information and Communication Engineers
Kikai-Shinko-Kaikan Bldg., 5-8, Shibakoen 3 chome, Minato-ku, TOKYO, 105-0011 JAPAN

PAPER *Special Section on Low-Power and High-Speed Chips*

Local Memory Mapping of Multicore Processors on an Automatic Parallelizing Compiler

Yoshitake OKI[†a)], **Yuto ABE**[†], **Kazuki YAMAMOTO**[†], **Kohei YAMAMOTO**[†], **Tomoya SHIRAKAWA**[†], ***Nonmembers***, **Akimasa YOSHIDA**[††b)], ***Senior Member***, **Keiji KIMURA**[†c)], ***and*** **Hironori KASAHARA**[†d)], ***Members***

SUMMARY Utilization of local memory from real-time embedded systems to high performance systems with multi-core processors has become an important factor for satisfying hard deadline constraints. However, challenges lie in the area of efficiently managing the memory hierarchy, such as decomposing large data into small blocks to fit onto local memory and transferring blocks for reuse and replacement. To address this issue, this paper presents a compiler optimization method that automatically manage local memory of multi-core processors. The method selects and maps multi-dimensional data onto software specified memory blocks called Adjustable Blocks. These blocks are hierarchically divisible with varying sizes defined by the features of the input application. Moreover, the method introduces mapping structures called Template Arrays to maintain the indices of the decomposed multi-dimensional data. The proposed work is implemented on the OSCAR automatic parallelizing compiler and evaluations were performed on the Renesas RP2 8-core processor. Experimental results from NAS Parallel Benchmark, SPEC benchmark, and multimedia applications show the effectiveness of the method, obtaining maximum speed-ups of 20.44 with 8 cores utilizing local memory from single core sequential versions that use off-chip memory.

key words: *parallelization compiler, local memory management, multicore processor, global address space, data decomposition*

1. Introduction

Cache hierarchy has been widely utilized in embedded systems as a solution to mitigate the performance gap between computation time and off-chip memory access time. In systems with conventional cache memories, performance is achieved by mapping and reusing data on caches with reuse policies that did not necessarily match the characteristics of the input program. Although caches are automatically controlled by the hardware, its transparency due to cache hits and misses becomes difficult to model data access timings and predict program execution behaviors. As an alternative memory structure to hardware caches, modern multicore and embedded architectures often contain a software controllable on-chip Local Memory (LM), or scratch-pad memory (SPM), to control the data flow between off-chip global memory and on-chip memory. These memory structures allow programmers to have explicit control of the contents of LM and the data flow of the program, leading to precise predictions of program execution times.

Precise predictions of data access times are especially critical for real-time systems with hard deadline constraints. Unfortunately, since the software has full control of data, the programmer has to decide and manage what data stays in LM during the execution of the program. This is typically done by tagging the most recently or frequently used data inside loops in an attempt to exploit data locality. However, decisions must also consider data locality of the input program that exist between loop bodies and inside loops with multiple nest levels. Another challenge for managing LM is the mapping of data onto LM space. Since the available LM space is typically limited for embedded architectures, data mappings must be done in a way that maximizes LM usage without creating internal fragmentations or introducing unnecessary paddings.

To effectively manage LMs, the implementation and modification to the program code also becomes a burden to the programmer. A practical way to tackle these problems is to develop a LM management method to offload the code modification task from programmers. Automating the management prevents introducing errors to the code as well as opens opportunities to apply the LM management methods to various types of programs.

Previous studies on LM and SPM management [1]–[5] utilize data placement algorithms and Integer Linear Programming (ILP) methods to map frequently used data onto fast on-chip memory. However, these methods do not consider locality that exist across coarse grain program regions for multicore processor systems as well as efficient data mapping techniques for on-chip memory that maximizes LM utilization rate.

To address these challenges, this paper proposes an integrated compiler and LM mapping method to select and map data onto LM to reduce execution time of the program. The proposed method utilizes OSCAR, a source-to-source multi-grain and multi-platform automatic parallelizing compiler, to integrate the method and generate LM management code for multicore architectures.

The main contributions of this paper are as follows:

1. Data Selection: A compiler scheme to choose and decompose data from the input program for LM management. Data can be array elements from multi-

Manuscript received July 1, 2019.
Manuscript revised October 24, 2019.

[†]The authors are with Department of Computer Science and Engineering, Waseda University, Tokyo, 169–8555 Japan.
[††]The author is with School of Interdisciplinary Mathematical Sciences, Meiji University, Tokyo, 164–8525 Japan.
a) E-mail: okiyoshi@kasahara.cs.waseda.ac.jp
b) E-mail: akimasay@meiji.ac.jp
c) E-mail: kimura@apal.cs.waseda.ac.jp
d) E-mail: kasahara@waseda.jp

DOI: 10.1587/transele.2019LHP0010

level nested loops and loops that stretch across multiple loops.

2. Data Mapping: A LM mapping scheme to allocate data onto specific areas of limited LM space and maximize LM utilization rate using flexible block sizes with integer-divisible sizes. The scheme also introduces template structures to retain the original indices of the mapped array for improved program code readability.
3. Integration of the data selection and LM mapping schemes to an automatic parallelizing compiler and a comprehensive analysis of the method.

The rest of the paper is organized as follows. Section 2 overviews related work. Section 3 introduces the target architecture and the compiler model. Section 4 introduces the proposed method. Sections 5 and 6 presents the data selection and mapping schemes. Section 7 shows the evaluation results of the proposed method. Finally, Sect. 8 concludes the paper.

2. Related Work

There are several prior studies that have investigated optimization strategies for optimizing LM and SPM management.

For single core environments, Li et al. [6] proposed a SPM partitioning method that splits arrays and inserts data transfer instructions between on-chip and off-chip memory. Their method adopts graph coloring techniques to map arrays to the partitioned SPMs. The method proposed in this paper is similar in the sense that the target on-chip memory is partitioned to map arrays. However, the proposed method in this paper is also applicable for multicore processor environments. Panda et al. [7] proposed an off-line variable partitioning method for SPM and off-chip memory to map constants and variables onto SPM and larger arrays onto off-chip memory. Avissar et al. [8] proposed an automated compiler optimization technique that performs data partition and allocation into multiple memory units. Similarly, Hiser and Davidson [9] proposed an algorithm to partition and assign variables into arbitrary memory hierarchies. Steinke et al. [10] proposed a compiler strategy for embedded systems that statically map most frequently used data onto on-chip LM. Sjodin and Platen [1] proposed an ILP formulation method to allocate variables to on-chip memory and pointers to cheap pointer types. Although ILP approaches give an exact method, they can be expensive when applied to large program arrays whose live ranges span across multiple functions.

For multicore environments, Che et al. [11] proposed an integrated model using ILP and heuristics for allocating data onto SPM. Their proposed method incorporates code overlay costs and inter-core communication overhead costs for multicore processor environments. Although their model exhibits performance on stream applications, it does not present explicit data mapping and management methods for LM. Kandemir et al. [3] proposed a compiler optimization technique that reduces power consumption and memory access latency. Their method utilizes data tiling for array partitioning to optimize inter-core communications and data sharing opportunities. Their method, however, mainly considers data locality within loops and does not consider locality among coarser tasks. Similarly, Ozturk [2] proposed a compiler technique for multi-processor systems using ILP methods to map objects to different memory hierarchies. Guo et al. [12] proposed regional data placement algorithms to reduce memory access costs, where the algorithms reduces the total memory access cost of parallelizable regions which have single or multiple copies of data in SPMs of each core. However, this method does not incorporate explicit data management methods for data onto LM. Meftali et al. [13] proposed an ILP model for memory allocation to all types of memory on a multiprocessor environment. Issenin et al. proposed a data reuse method for loops structures on multicore processor systems [4]. Their method exploits data locality within loops, but does not consider data locality that exists on a coarse grain of the program. Angiolini et al. [14] proposed a SPM partitioning algorithm that maps memory locations to partitioned SPM locations for multicore processors using a Dynamic Programming approach. According to their analysis, the presented algorithm has a time and space complexity that is polynomial in the input data size. Verma et al. [15] proposed a data allocation algorithm for SPM that inserts data copy instructions at runtime using variable liveness analysis and ILP methods. Similarly, Suhendra et al. [5] proposed an ILP-based SPM optimization method that perform task mapping, scheduling, SPM partitioning, and data mapping.

To summarize, previous optimization techniques do not focus on extracting locality across coarse grain program regions. Parallelization granularities in multicore processors include loop-level parallelization such as do-loops and coarse grain parallelism among multiple loops. To obtain performance in scientific and embedded systems, it is critical to exploit localities that cover all types of parallelization granularity levels.

This paper develops a LM management method that allocates decomposed data extracted from coarse-grain tasks onto LM. The proposed method extracts locality from single and multi-dimensional arrays within nested loops, and maps them onto memory blocks with block sizes decided from the features of the input program. This paper builds upon the work by Yoshida et al. [16] and Yamamoto et al. [17] by considering the available LM size and implementing additional memory block size choices for applications that require more LM space to create an improved and efficient mapping of data onto LM. Moreover, this paper provides additional analysis for evaluation from the previous works.

3. Target Architecture and Compiler Model

In this paper, the target architecture has processor-coupled LMs that are visible and controllable by software. The proposed method assumes architectures that have several clusters of multicore processors and a shared centralized off-chip memory. Each multicore processor consists of software controllable LM units reserved for private data and a distributed shared memory for data shared between cores. The proposed method utilizes these software controllable LM units to exploit data locality present in core-private data. The OSCAR multicore processor is an example architecture that implements this processor and memory structure [18].

To generalize LM management, the proposed method utilizes the OSCAR multi-grain and multi-platform automatic parallelizing compiler's coarse-grain parallelization technique. The OSCAR compiler takes sequential C and Fortran programs as input, and converts them into parallelized programs that can be compiled into executables for multicore processors.

The OSCAR Compiler detects parallelism that exists in multiple parallelism granularity levels. The levels of parallelism include loop parallelism, coarse grain parallelism, and fine grain parallelism. To fully extract the parallelism of the input program, the compiler detects data locality that exists in all levels of parallelism. To integrate the proposed method with the compiler, the method utilizes structures generated by the OSCAR compiler. The OSCAR compiler divides the input program into coarse-grain tasks called Macro Tasks (MT), which will be the base structure for the proposed method. The details of the parallelizing techniques and the utilized structures of the OSCAR compiler are beyond the scope of this paper.

4. Data Selection and Mapping Scheme for LM

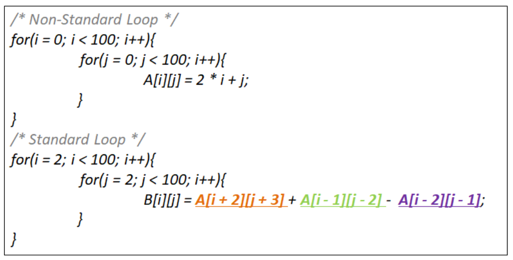
The main idea of the proposed method is to decompose data appropriately to safely fit them onto LM space to achieve locality. Specifically, the method can be divided into the following two schemes.

1. Data Selection Scheme: The first scheme selects and decomposes data used inside each coarse-grain tasks. The scheme selects data that exploits locality across multiple coarse-grain tasks. Moreover, the scheme decomposes these data to map them onto LM space for multicore processors.
2. Data Mapping Scheme: The second scheme allocates the decomposed data onto LM utilizing flexible block sizes and template mapping structures. The memory blocks are hierarchically divided and determined by application features to maximize the utilization rates of LM. The proposed mapping structures create a mapping of the decomposed data onto LM space to maintain the original indices of the decomposed data and to improve the program readability.

The following sections explain the proposed method in detail.

5. Data Selection Scheme for LM

The first scheme of the proposed method is to select appropriate data to exploit data locality for multicore processors, and decompose the selected data to fit them onto LM space.



```
/* Non-Standard Loop */
for(i = 0; i < 100; i++){
        for(j = 0; j < 100; i++){
                A[i][j] = 2 * i + j;
        }
}
/* Standard Loop */
for(i = 2; i < 100; i++){
        for(j = 2; j < 100; i++){
                B[i][j] = A[i + 2][j + 3] + A[i - 1][j - 2] -  A[i - 2][j - 1];
        }
}
```

Fig. 1 An example TLG from a sample program code with two loops.

5.1 Data Selection from Loops by Target Loop Groups

The first step for efficient LM management is to select data that exploits data locality from the input program. To ensure data locality and continuous memory access, array elements accessed by multiple adjacent loops must be mapped onto a common processor core's LM. The proposed scheme selects array elements within multi-level nested loops and array elements that are accessed by multiple loops. In the OSCAR compiler, these loops correspond to coarse grain tasks, or MTs called Repetition Blocks (RBs).

To keep track of the loops and its data, the proposed scheme introduces a structure called Target Loop Group (TLG). TLGs help organize localities within programs by combining adjacent loop bodies that access common array elements. A TLG extracted from a sample program code is shown in Fig. 1. The loops within each TLG have the following properties:

1. Loops that are adjacent on the original input program are grouped into a single TLG
2. Indices of the arrays inside the loops are represented as a linear function of the loop index variable
3. There is a unique standard loop defined as the loop with the largest estimated cost within each TLG. The cost is calculated based on arithmetic, load, and store instructions. Loops that are not chosen as the standard loop are called non-standard loops.

5.2 Dividing Loop Iteration Ranges for Locality through Inter-Loop Dependency Analysis

Data locality is exploited when array elements on LM are continuously accessed by loops on the same processor core. While gathering loops into TLGs keeps the arrays close together for locality, the loops must have a common iteration range for continuous access and sharing of array elements between processor cores. To solve this, the Inter-Loop Dependency analysis (ILD) detects data dependence between each loops and calculates appropriate iterations ranges for each TLG. The detailed steps of the ILD are shown in the following sub-sections.

The ILD begins by setting the TLG's standard loop as a

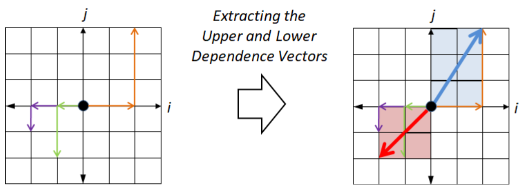


Fig. 2 Direct Inter-Loop Data dependence (DirILD) of the loops from the TLG of Fig. 1.

reference point to analyze data dependence between loops. The main idea is that the standard loop retains its original iteration ranges, while the other loops in the TLG adjusts to the standard loop's iteration range while considering data dependence among loops.

To analyze data dependence within TLGs, the ILD initially calculates the indices for each loop, or RB, that has dependence with its adjacent loops. This is done by extracting inter-loop dependence which has data dependence between adjacent loops. These are called Direct Inter-Loop Data dependence (DirILD) vectors, and are calculated for each adjacent loop pair in every TLG. DirILD vectors from the example TLG of Fig. 1 is shown in Fig. 2 by the red and blue arrows.

For multi-dimensional arrays, the ILD creates DirILD vectors that contain the data dependence for each array dimension as its components. DirILDs are the building blocks for calculating the iteration ranges of the non-standard loops that has direct and indirect data dependence with the iterations of the standard loop.

Next, the adjusted iteration ranges of the non-standard loops are calculated by the Direct and Indirect Inter-Loop Data dependence (DI_ILD). DI_ILD vectors are calculated by the following equation, where s is the standard loop Repetition Block (RB) number and SucDep(x) is a set of successor RBs of block x that have dependence with x.

$$\begin{aligned} & for\ i = s - 1\ to\ 1: \\ & DI\_ILD(RB_i, RB_s) = \bigcup_{RB_j \in SucDep(RB_i)} \\ & \qquad \bigcup_{t \in DI\_ILD(RB_j, RB_s)} \ \bigcup_{x \in DirILD(RB_i, RB_j)} x + t \end{aligned} \tag{1}$$

The DI_ILD vectors represent the relative inter-loop dependence of the non-standard loops with the standard loop. The inverse relationship of the DI_ILD vectors is represented as the Inverse DI_ILD (IDI_ILD) vector, and is calculated by utilizing the DI_ILD vectors as shown in the following formula, where i is the target RB number.

$$IDI\_ILD(RB_i, RB_s) = \bigcup_{x \in DI\_ILD(RB_i, RB_j)} -x \tag{2}$$

Once the relationships of the iterations between the non-standard loops and the baseline standard loop are resolved, the scheme now calculates the new iteration ranges for each loop. The IDI_ILD vectors are used to calculate the Converted Index Range (CIR) of each loop. CIRs represent the converted iteration ranges of the loops with respect to the iteration ranges of the standard loop. The equation for CIRs is shown below, where IR(x) represents the iteration range of block x.

$$\begin{aligned} & CIR(RB_i) = \{x \in IR(RB_i) \mid \\ & min(IR(RB_i)) + max(IDI\_ILD(RB_i, RB_s)) \\ & \quad \le x \le max(IR(RB_i)) + min(IDI\_ILD(RB_i, RB_s))\} \end{aligned} \tag{3}$$

Furthermore, the CIRs of the individual loops are combined to generate a common iteration range for all of the loops within TLGs. Creating a common iteration range for each TLG is a key step to extract locality within the loops. This iteration range is called the Group Converted Index Range (GCIR) of a TLG, and is represented by the following equation.

$$GCIR = \bigcup_{1 \le i \le s} CIR(RB_i) \tag{4}$$

Since the GCIRs encapsulate a common iteration range of the loops, the scheme can split the loops into iteration ranges that are accessed by a single core or shared between multiple cores. Localizable Regions (LRs) are loop iteration regions that contain arrays accessed only by a single processor core. In other words, these regions contain loop iterations that can continuously reside on the same core's LM for data locality. In contrast, Commonly Accessed Regions (CARs) are loop iteration regions that contain arrays accessed and shared by multiple processor cores.

Initially, the GCIR is split into P equal ranges, where P is the number of available processor cores. To generate the CARs, the scheme takes the overlapping iteration regions, or the iteration regions that are shared and accessed by different processor cores, of the adjacent iteration ranges for each divided loop. The CAR for processors P and P + 1 are calculated by the following equation.

$$\begin{aligned} & IR(RB_i^{<P,P+1>}) \\ & = \bigcup_{t \in DGCIR^P} \left(\bigcup_{t \in ILD(RB_i, RB_s)} x + 1 \right) \bigcap \\ & \qquad \bigcup_{t \in DGCIR^{P+1}} \left(\bigcup_{t \in ILD(RB_i, RB_s)} x + 1 \right) \end{aligned} \tag{5}$$

The LRs are calculated by subtracting the overlapping iteration regions from the divided loops and taking the iteration ranges that are accessed only by a single processor core. The formula for LR for processor P is shown below.

$$\begin{aligned} IR(RB_i^{<P>}) = & \bigcup_{t \in DGCIR^P} \left(\bigcup_{t \in ILD(RB_i, RB_s)} \right) \\ & - IR(RB_i^{<P-1,P>}) - IR(RB_i^{<P,P+1>}) \end{aligned} \tag{6}$$

Figure 3 shows an ILD analysis step with LRs and CARs on a single dimension array with two loops. Figure 4 also

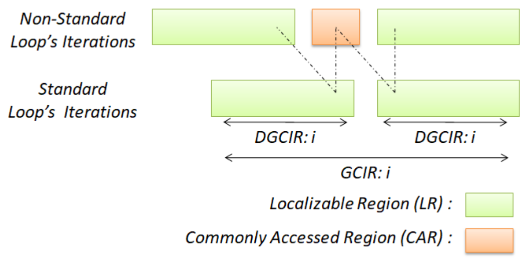

Non-Standard Loop's Iterations
Standard Loop's Iterations
DGCIR: i
DGCIR: i
GCIR: i
Localizable Region (LR) :
Commonly Accessed Region (CAR) :

Fig. 3 ILD on a 1D array with two loops.

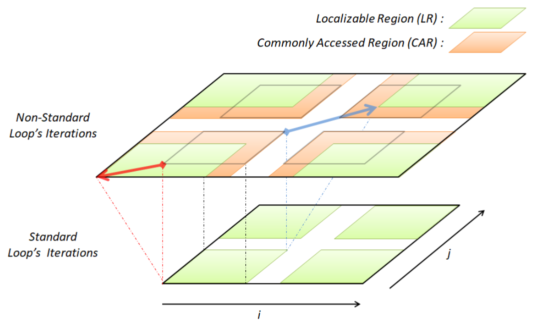

Localizable Region (LR) :
Commonly Accessed Region (CAR) :
Non-Standard Loop's Iterations
Standard Loop's Iterations
j
i

Fig. 4 ILD on a 2D array with two loops.

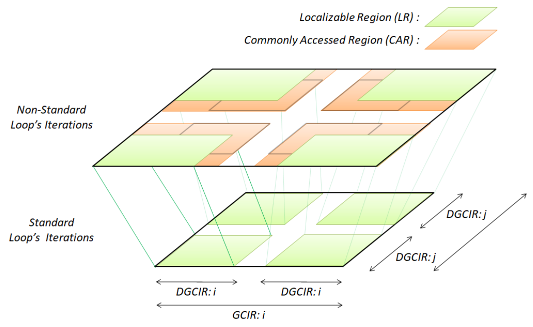

Localizable Region (LR) :
Commonly Accessed Region (CAR) :
Non-Standard Loop's Iterations
Standard Loop's Iterations
DGCIR: j
DGCIR: j
DGCIR: i
DGCIR: i
GCIR: i

Fig. 5 GCIRs and DGCIRs from Fig. 4.

shows an ILD analysis step with LRs and CARs on a two dimensional array with two loops. The extracted GCIRs and DGCIRs from the example ILD step from Fig. 4 is shown in Fig. 5.

5.3 Decomposing Data of Localizable Regions and Commonly Accessed Regions

A key idea of the proposed scheme is to decompose arrays into smaller sub-arrays that fit safely onto LM space. Once adjacent loops are gathered into TLGs and data dependencies of loop iteration regions are analyzed to form LRs and CARs, the scheme decomposes the corresponding arrays of the analyzed loop iterations.

To create a LM space aware working set size, the iterations of the LRs and CARs are equally divided by a decomposition count. The decomposition count defines the number of smaller sub-arrays each array should be divided into. To mitigate the calculation cost, the proposed scheme defines the decomposition count of the arrays to be large enough to allow all arrays existing in each TLG to reside simultaneously onto each processor core's LM. The decomposition count for each TLG is decided by the following steps.

1. Estimate the array access size, or the working set size, of the loops within each TLG.
2. Compare the size of the available LM with the estimated working set size.
3. Choose a decomposition count that divides the arrays into sub-arrays that have the largest possible size or matches the available LM size.

For multi-dimensional arrays, the decomposition count is calculated by dividing the outermost loop level, or at the RB granularity level, to calculate the decomposition count. However, if dividing only the outer-most loop layer fails to create an array size small enough for LM, the scheme recursively decomposes each layer of the loop, deciding the decomposition count for each loop-nest level. This corresponds to recursively executing steps 2 and 3 of the decomposition count deciding mechanism shown above.

5.4 Task Scheduling with Data Localizable Groups

The decomposed LRs and CARs within TLGs achieve data locality only when they are allocated onto LM space of the same processor core in multicore systems. To schedule them onto appropriate processor cores, the proposed scheme defines LRs and CARs as new coarse-grain tasks, or MTs. By defining these divided LRs and CARs into generalized tasks, the proposed scheme creates opportunities for other compiler optimization techniques such as task fusion and restructuring. Moreover, the proposed scheme gathers MTs (LRs and CARs) into Data Localizable Groups (DLGs) to allow simplified locality-aware scheduling.

6. Data Mapping Scheme for LM

The second scheme of the proposed method is the mapping of the decomposed array onto LM space.

This step is invoked after creating DLGs and DLG scheduling results. This step performs memory mapping of the decomposed array. The data mapping step utilizes software configurable memory blocks called Adjustable Blocks, and mapping structures called Template Arrays. Adjustable Blocks are hierarchically divisible memory block structures that divide LM into constant sized blocks that suit the decomposed array. Template Arrays are mapping structures that create mappings of arrays to blocks on LM to maintain the original array indices for improved code readability.

6.1 Flexible Memory Block Sizes

The mapping of data depends on the data usage characteristics and the features of the input program. A straightforward approach to allocate these data onto LM is to map them through arbitrary sized memory blocks. However, utilizing

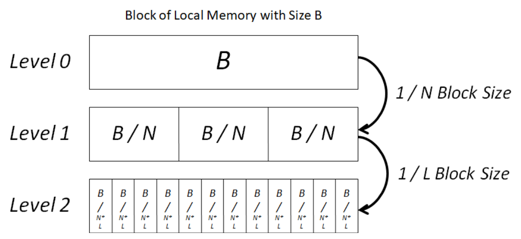


Fig. 6 Structure of Adjustable Blocks, where N and L are integers.

arbitrary size LM blocks could trigger performance degradations due to internal memory fragmentations. To mitigate these inefficiencies, the proposed scheme allocates data onto LM through flexible memory block structures called Adjustable Blocks. A sample structure of Adjustable Blocks is shown in Fig. 6.

Adjustable Blocks are built from constant size memory blocks which are hierarchically aligned to map and match data with varying sizes and dimensions. This hierarchical characteristic with constant memory block sizes is the key feature to achieve efficient utilization rates of LM.

The main advantage of Adjustable Blocks is the software controllable feature of memory block sizes which can be adjusted according to the requested data sizes of the input program. The memory blocks of Adjustable Blocks are aligned in specific levels in the hierarchy, and can be divided into smaller sub-blocks with powers-of-two divisible sizes or integer divisible sizes of its parent block.

Each level in the hierarchy has a distinct block size chosen from each decomposed data sizes of the input program. The choice of integer divisible block sizes of Adjustable Blocks bring flexibility to the input program, differing from buddy memory allocators used in Operating Systems as well as from previous methods of LM managements where memory block sizes were always chosen to be multiples of powers-of-two.

Before choosing the memory block sizes for each Adjustable Block level, the decomposed arrays are sorted by descending order in size and arranged into a list. Once sorted, each array is checked whether its parent array, or the preceding array in the list, is an integer multiple of the current array. If it is an integer multiple, the current array is mapped to a new level with a memory block size divided by that integer factor.

To illustrate the effectiveness of integer divisible blocks, an example is shown below. In previous researches where memory block sizes were always adjusted to multiples of powers-of-two, multi-dimensional arrays with dimensions [5][5][9] require memory blocks with dimensions of [8][8][16]. The array sizes become $5 * 5 * 9 = 225$ for the original array, and $8 * 8 * 16 = 1024$ for the adjusted array. With this approach, the utilization rate of memory only achieves $225/1024 = 0.22$

Low utilization rates become a bottleneck especially for embedded systems where LM space is typically limited. In contrast, if memory blocks with block sizes adjusted to multiples of powers-of two fail to allocate LM space, the proposed scheme divides memory blocks into integer divisible sizes of its parent memory block to reduce the memory allocation size required by the input application.

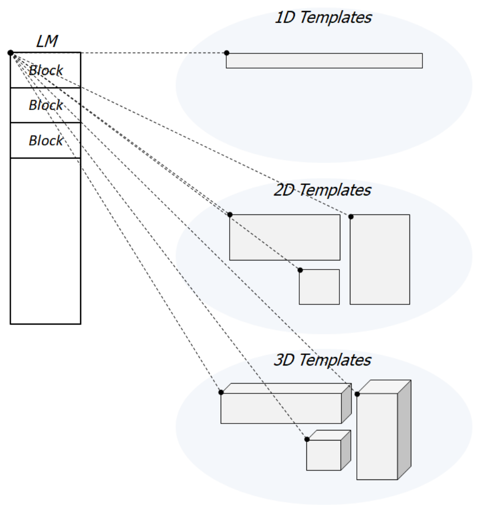


Fig. 7 Overview of Template Arrays for each dimension.

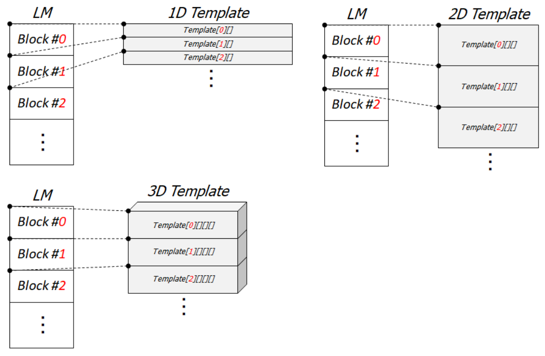


Fig. 8 Mapping of LM arrays onto Template Arrays.

6.2 Templates for Arrays

For conventional mapping techniques, allocating multi-dimensional data onto LM required complicated index calculations with offsets variables since memory is internally represented as a single dimensional array. To reduce such calculation complexity and overhead, the proposed scheme introduces an array mapping technique called Template Arrays. Template Arrays are structures that prepare the same size, dimension, and type as the chosen Adjustable Block size of the input program. An overview of Template Arrays for 1D, 2D, and 3D arrays is shown in Fig. 7, and an assignment example of Template Arrays is shown on Fig. 8.

Template Arrays holds a mapping between memory blocks on LM and empty array entries with multiple dimensions. These empty arrays have an additional dimension augmented to store the block number that corresponds to the original block on LM. By using these block numbers as tags, the scheme decides the region and block of memory appropriate for the decomposed data. Moreover, by preparing the same size and dimensions as the original arrays, Template Arrays keeps a mapping that maintains the dimensions of the original array without changing the indices. This preserves the original array indices, avoiding complex index calculations for performing multi-dimensional array accesses.

7. Evaluations

In this section, the effectiveness of the proposed local memory management method is shown through evaluation results on benchmark applications. The method was implemented on the OSCAR automatic parallelizing compiler and executed on the Renesas RP2 8-core multicore processor.

7.1 Architecture of the Renesas RP2 Multicore Processor

The Renesas RP2 multicore processor is an embedded processor based on the OSCAR multicore architecture [19]. The chip was developed by Renesas Electronics, Hitachi, and Waseda University and was supported by NEDO Multicore Processors as a real-time consumer electronics project. An overview of the RP2 architecture is shown in Fig. 9. The RP2 processor has two SMP clusters, each with 4 SH4A cores running at 600MHz. Each processor core has its own private LM. The access latency of this LM is 1 clock cycle. Each processor core can also access a common processor-wide 128MB off-chip centralized shared memory (CSM), which has 55-clock cycle latency.

7.2 Benchmark Applications

To evaluate the performance of the proposed method, the following 6 applications written in C were used: BT from Nas Parallel Benchmark (NPB), Tomcatv and Swim from SPEC95, AACEncoder from Renesas Electronics, Mpeg2Enc from MediaBench Benchmark Suite, and Tracking from SD-VBS. The applications were first converted to Parallelizable C [18], which is a dialect of C similar to MISRA-C.

The applications were then compiled by the OSCAR source-to-source automatic parallelizing compiler, where the proposed method ran as part of the OSCAR compiler's analysis and optimization phase. Finally, the parallelized output C program was compiled into binaries by the Renesas SuperH C Compiler as a backend compiler.

7.3 Evaluation Results

Figure 10 presents the evaluation results of the benchmark applications executed on the RP2 mutlicore processor. For the figures shown in this section, the results using only the off-chip shared memory of RP2 are labeled as "Parallelized", and the results utilizing the proposed LM management method are labeled as "Parallelized with LM".

Loop intensive applications such as BT, Swim, and Tomcatv showed large speedups utilizing the proposed method compared against executions that only utilize off-chip shared memory. For example, the speedups of the off-chip shared memory version for BT were 3.55 for 4 cores, and 6.66 for 8 cores. On the other hand, the speedups for BT using the presented method were 1.70 for 1 core, 3.99 for 2 cores, 6.71 for 4 cores, and 12.79 for 8 cores. The speedup comes from decomposing multi-dimensional arrays and keeping the working set reside on the low-latency LM.

The evaluation results of the proposed method for media applications such as AACenc and Mpeg2enc also show speedups against its shared memory execution counterparts. In AACenc, the speedups of the off-chip shared memory version were 3.42 for 4 cores and 6.84 for 8 cores compared to the single core environment. In contrast, the speedups for AACenc with LM management were 1.99 for 1 core, 4.37 for 2 cores, 8.72 for 4 cores, and 20.44 for 8 cores.

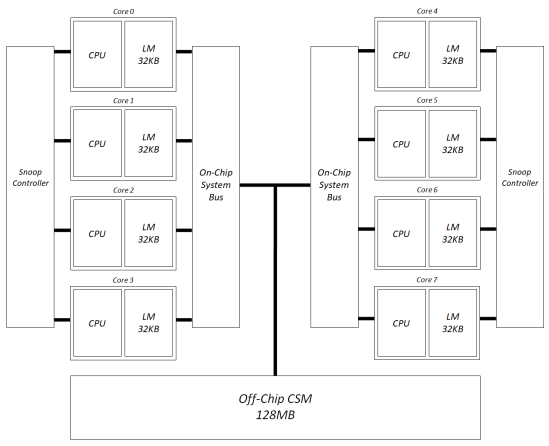


Fig. 9 The RP2 Multiprocessor with the OSCAR Multicore Architecture.

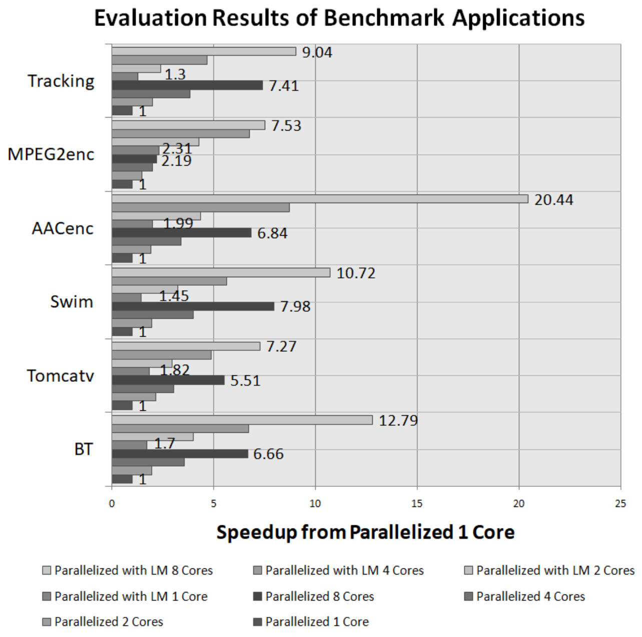


Fig. 10 Speedups of the proposed method with LM.

Figures 11 and 12 shows the total memory read and write overheads for the "Parallelized" and "Parallelized with LM" versions for singe core executions. For each application, the versions utilizing the on-chip LM shows a significant decrease in overhead latencies for both memory reads and writes compare to their off-chip shared memory counterparts. The read overhead of the LM version of Mpeg2enc decreased to 37% compared to the shared memory version. Similarly, the write overhead for BT's LM version was reduced to 35% relative to its shared memory version. The decrease in memory overhead comes from the proposed method minimizing the use of shared memory that incurs 55-clock cycle latency for every memory access.

The CSM access frequencies of core 1 for multicore environments of the benchmark applications are shown in Figs. 13 to 18. The results for versions using only the off-chip CSM are labeled as "Parallelized", and the results utilizing the proposed LM management method are labeled as "Parallelized with LM". The results show the relative frequency compared to the 1 Core Parallelized version. For the evaluated RP2 environment, the latencies is 55 clock cycles for CSM accesses and 1 clock cycle for LM accesses. For each application, the counts of CSM data transfer decreases as the number of processor core increases. When utilizing the proposed LM management method, the CSM accesses further decreases since portions of the accesses are replaced with the faster LM accesses with 1 clock cycle. For the shared memory version of Swim, the number of CSM data transfer decreases to 50%, 25%, and 13% for 2, 4, and 8 core processor environments compared to the 1 core Parallelized environment. In contrast, the LM management version of Swim has a greater decrease in CSM data transfers of

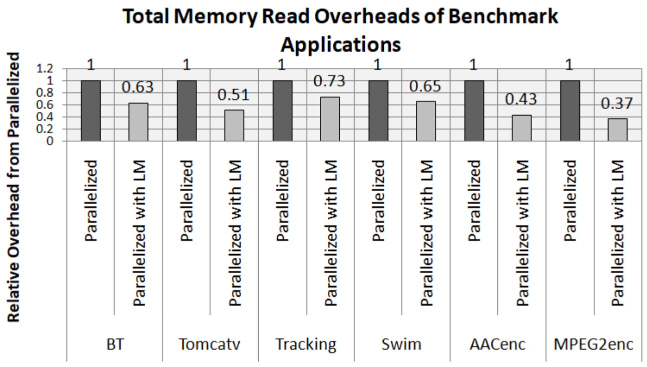


Fig. 11 Total memory reads of the proposed LM and the CSM versions.

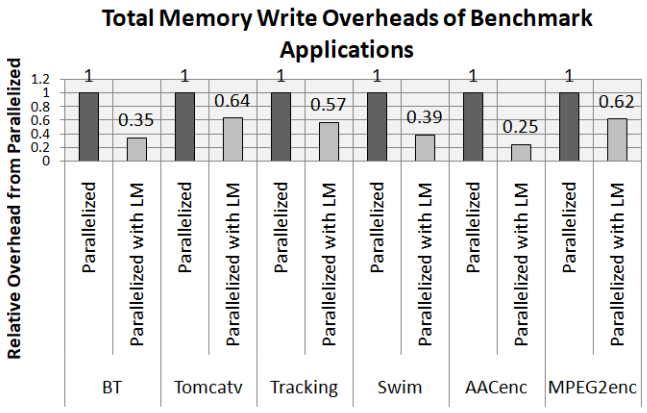


Fig. 12 Total memory writes of the proposed LM and the CSM versions.

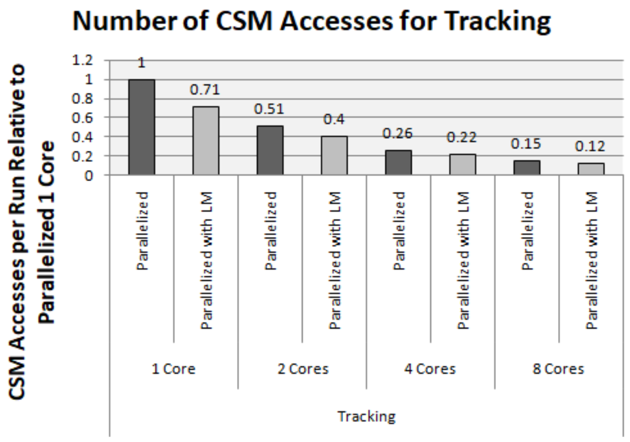


Fig. 13 Number of CSM accesses for Tracking.

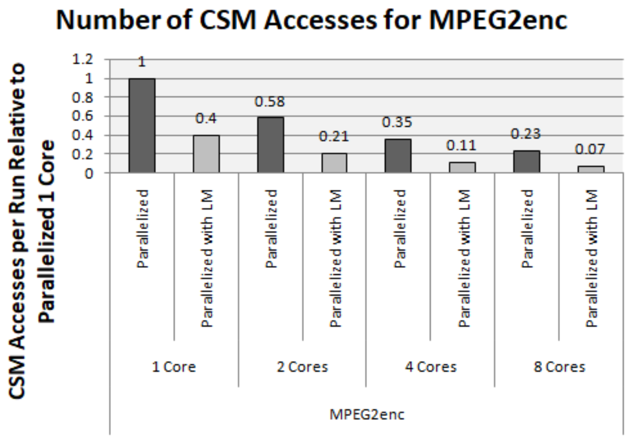


Fig. 14 Number of CSM accesses for MPEG2enc.

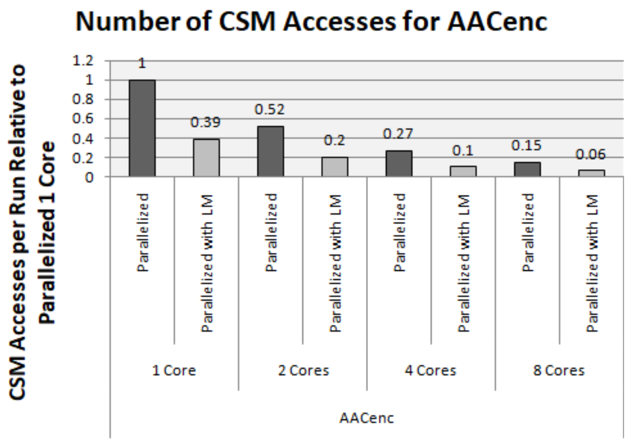


Fig. 15 Number of CSM accesses for AACenc.

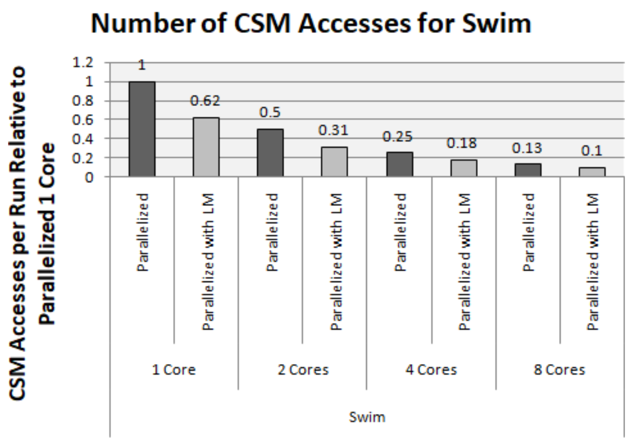


Fig. 16 Number of CSM accesses for Swim.

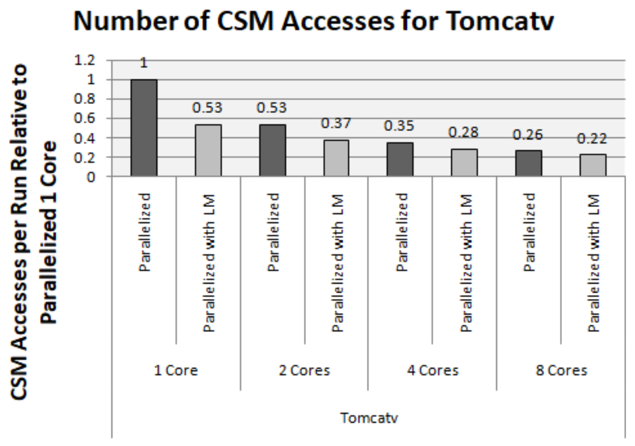


Fig. 17 Number of CSM accesses for Tomcatv.

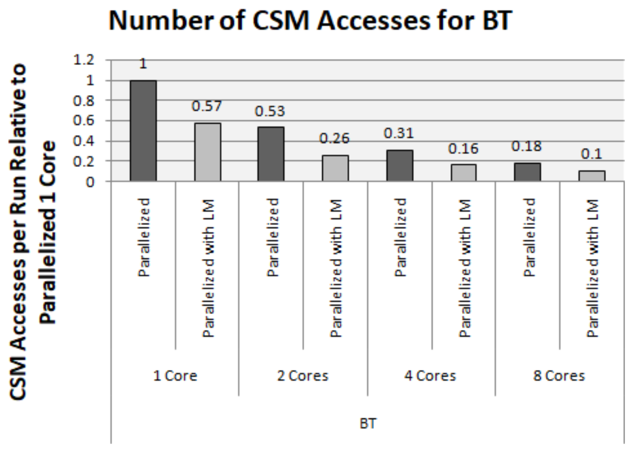


Fig. 18 Number of CSM accesses for BT.

31%, 18%, and 10% for 2, 4, and 8 core environments compared to the 1 core Parallelized environment. For AACenc's shared memory version, the CSM data transfers are 52%, 27%, and 15% for 2, 4, and 8 core environments against the 1 core Parallelized environment. For AACenc's LM management version, these CSM data transfers are further reduced to 20%, 10%, and 6% for 2, 4, and 8 cores against the 1 core Parallelized environment. The decreasing trend in CSM data transfer frequency shows that the application has a smaller portion of data transfer to and from CSM as more processor cores are added. Moreover, for the proposed LM management method versions, the accesses of the 55 clock cycle CSM is reduced compared to the shared memory counterparts.

Table 1 Local memory usage of the benchmark applications.

Applications	Local Memory Sizes
Tracking	24.6KB
Mpeg2Enc	32KB
AACenc	30.5KB
Swim	30.7KB
Tomcatv	32KB
BT	30KB

By mapping data and utilizing LM, the applications with the proposed method show improved scalability with increased processor core counts. The speedups of the proposed LM management method from the shared memory versions come from successfully mapping working sets onto LM using Adjustable Blocks. Since the sizes of these blocks are extracted from the features of the input program, they generate a unique configuration for each application. The maximum LM usage for the benchmark applications are summarized in Table 1. Since the RP2 processor has a 32KB LM, the method successfully allocates and uses LM for each application that fit onto the available LM size. Given n as the block size of the first level, the Adjustable Block sizes for Tracking are

$$Level2 : n/2 \tag{7}$$
$$Level3 : n/2/2, \tag{8}$$

and the block sizes for Tomcatv are

$$Level2 : n/2 \tag{9}$$

For Mpeg2enc, the Adjustable Block sizes begin with a size of 16384 bytes, and halves each level until the block size becomes 4 bytes. Similarly, AACenc has a block size of 4096 bytes for first level, and halves until the block size becomes 4 bytes. Swim only allocates 1 level. As presented in Sect. 6.1, the proposed Adjustable Blocks can also generate integer divisible block sizes which are not multiples of powers of two. For BT, the block sizes are

$$Level2 : n/3 \tag{10}$$
$$Level3 : n/3/5. \tag{11}$$

The required block sizes for BT are 960 Bytes for the third level, 4800 Bytes for the second level, and 14400 Bytes for the first level. If the block sizes were powers-of-two, the block sizes increases to 1536 Bytes for the third level, 12288 Bytes for the second level, and 49152 Bytes for the first level, which exceeds the available LM size when multiple array variables use these block sizes. By allowing flexible

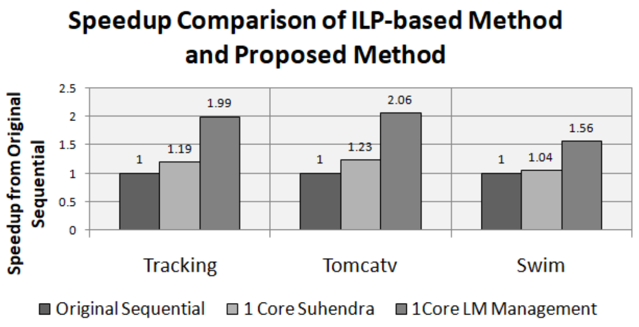


Fig. 19 Speedups of the proposed method against and an ILP-based method.

block sizes for LM mapping, the proposed method generated block sizes that matches the target application and fits onto the available LM.

To compare the proposed method with other studies, Fig. 19 shows the speedup of the LM management method against the speedup obtained applying one of the ILP-based SPM optimization approaches by Suhendra et al. [5] on three benchmark applications: Tracking, Tomcatv, and Swim. Both methods with single core executions are compared with the original sequential versions of the applications.

The speedups of the ILP-based method were 1.19, 1.23, and 1.04 for the three benchmarks. In contrast, the speedups of the three applications by the proposed LM management method were 1.99, 2.06 and 1.56. Since all applications have large variables, the ILP-based method has difficulty allocating them onto LM, forcing some variables to be allocated onto the off-chip shared memory. However, the proposed method decomposes large data by the working set size and successfully allocates them onto LM, minimizing off-chip memory accesses.

8. Conclusions

This paper proposes an integrated compiler and Local Memory (LM) management method that automatically decomposes data and assigns them to LM on multicore processors for applications that focus on predictability and performance. The presented Data Layout scheme safely decomposes large multi-dimensional arrays into smaller subarrays that fit onto limited sized LM. Moreover, the proposed Data Mapping scheme avoids memory fragmentation through application specific flexible memory blocks called Adjustable Blocks and preserves the original indices of array codes through multi-dimensional structures called Template Arrays. The method is implemented on the OSCAR source-to-source automatic parallelizing compiler and evaluated on 6 benchmark applications executed on the RP2 8-core processor. The method obtained a maximum speed up of 20.44 against single threaded versions with off-chip shared memory.

References

[1] J. Sjödin and C. von Platen, "Storage allocation for embedded processors," Proc. 2001 International Conference on Compilers, Architecture, and Synthesis for Embedded Systems, pp.15–23, ACM, 2001.

[2] O. Ozturk, M. Kandemir, M.J. Irwin, and S. Tosun, "Multi-level on-chip memory hierarchy design for embedded chip multiprocessors," 12th International Conference on Parallel and Distributed Systems, ICPADS '06, 2006.

[3] M. Kandemir, J. Ramanujam, and A. Choudhary, "Exploiting shared scratch pad memory space in embedded multiprocessor systems," Proc. 39th Annual Design Automation Conference, DAC '02, New York, NY, USA, pp.219–224, ACM, 2002.

[4] I. Issenin, E. Brockmeyer, B. Durinck, and N. Dutt, "Multiprocessor system-on-chip data reuse analysis for exploring customized memory hierarchies," 43rd ACM/IEEE Design Automation Conference, pp.49–52, July 2006.

[5] V. Suhendra, C. Raghavan, and T. Mitra, "Integrated scratchpad memory optimization and task scheduling for MPSoC architectures," Proc. 2006 International Conference on Compilers, Architecture and Synthesis for Embedded Systems, pp.401–410, ACM, 2006.

[6] L. Li, L. Gao, and J. Xue, "Memory coloring: A compiler approach for scratchpad memory management," 14th International Conference on Parallel Architectures and Compilation Techniques, PACT 2005, pp.329–338, IEEE, 2005.

[7] P.R. Panda, N.D. Dutt, and A. Nicolau, "Efficient utilization of scratch-pad memory in embedded processor applications," Proc. 1997 European Conference on Design and Test, EDTC '97, Washington, DC, USA, pp.7–11, IEEE, 1997.

[8] O. Avissar, R. Barua, and D. Stewart, "An optimal memory allocation scheme for scratch-pad-based embedded systems," ACM Trans. Embed. Comput. Syst., vol.1, no.1, pp.6–26, Nov. 2002.

[9] J.D. Hiser and J.W. Davidson, "EMBARC: An efficient memory bank assignment algorithm for retargetable compilers," ACM SIGPLAN Notices, vol.39, no.7, pp.182–191, 2004.

[10] S. Steinke, L. Wehmeyer, B.-S. Lee, and P. Marwedel, "Assigning program and data objects to scratchpad for energy reduction," Proc. 2002 Design, Automation and Test in Europe Conference and Exhibition, pp.409–415, 2002.

[11] W. Che, A. Panda, and K.S. Chatha, "Compilation of stream programs for multicore processors that incorporate scratchpad memories," Proc. 2010 Design, Automation & Test in Europe, DATE '10, 3001 Leuven, Belgium, Belgium, pp.1118–1123, European Design and Automation Association, 2010.

[12] Y. Guo, Q. Zhuge, J. Hu, J. Yi, M. Qiu, and E.H.-M. Sha, "Data placement and duplication for embedded multicore systems with scratch pad memory," IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst., vol.32, no.6, pp.809–817, June 2013.

[13] S. Meftali, F. Gharsalli, F. Rousseau, and A.A. Jerraya, "An optimal memory allocation for application-specific multiprocessor system-on-chip," Proc. 14th International Symposium on System Synthesis, pp.19–24, ACM, 2001.

[14] F. Angiolini, L. Benini, and A. Caprara, "Polynomial-time algorithm for on-chip scratchpad memory partitioning," Proc. 2003 International Conference on Compilers, Architecture and Synthesis for Embedded Systems, pp.318–326, ACM, 2003.

[15] M. Verma, L. Wehmeyer, and P. Marwedel, "Dynamic overlay of scratchpad memory for energy minimization," Proc. 2nd IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis, CODES+ISSS '04, New York, NY, USA, pp.104–109, ACM, 2004.

[16] A. Yoshida, K. Koshizuka, and H. Kasahara, "Data-localization for Fortran macro-dataflow computation using partial static task assignment," Proc. 10th International Conference on Supercomputing,

pp.61–68, ACM, 1996.

[17] K. Yamamoto, T. Shirakawa, Y. Oki, A. Yoshida, K. Kimura, and H. Kasahara, "Automatic local memory management for multicores having global address space," International Workshop on Languages and Compilers for Parallel Computing, LCPC 2016, Lecture Notes in Computer Science, vol.10136, pp.282–296, Springer, 2016.

[18] 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 SMP servers," International Workshop on Languages and Compilers for Parallel Computing, LCPC 2009, Lecture Notes in Computer Science, vol.5898, pp.188–202, Springer, 2010.

[19] M. Ito, T. Hattori, Y. Yoshida, K. Hayase, T. Hayashi, O. Nishii, Y. Yasu, A. Hasegawa, M. Takada, M. Ito, H. Mizuno, K. Uchiyama, T. Odaka, J. Shirako, M. Mase, K. Kimura, and H. Kasahara, "An 8640 MIPS SoC with independent power-off control of 8 CPUs and 8 RAMs by an automatic parallelizing compiler," 2008 IEEE International Solid-State Circuits Conference, ISSCC 2008, Digest of Technical Papers, pp.90–598, IEEE, 2008.

Yoshitake Oki is a research assistant at Waseda University, Tokyo, Japan. He received his B.S. in 2013 and M.S. in 2015 in Natural Science from International Christian University, Tokyo, Japan. He is also a member of the IEEE Eta Kappa Nu Mu Tau Chapter. His research concentrates on local memory optimizations and task schedulings for multicore processors.

Yuto Abe is a 2nd year graduate student in Waseda University, Tokyo, Japan. He received his B.E. (2018) in Computer Science and Engineering from Waseda University. Since 2018, he has been working on local memory management of applications using sparse matrix.

Kazuki Yamamoto is a first year master student at Waseda University, Tokyo, Japan. He received his B.E. in Computer Science and Engineering from Waseda University. His research interests include compilers and local memory management in machine learning programs.

Kohei Yamamoto is a graduate student at Waseda University, Tokyo, Japan. He received his B.E. and M.E. in Computer Science and Engineering from Waseda University. His research interests include compilers, parallelization, and local memory managements.

Tomoya Shirakawa is a graduate student at Waseda University, Tokyo, Japan. He received his B.E. and M.E. in Computer Science and Engineering from Waseda University. He has been engaged in parallelzing compilers and local memory optimizations.

Akimasa Yoshida received his B.E., M.E. and Dr.Eng. degrees from Waseda University in 1991, 1993 and 1996, respectively. He became a JSPS research fellow (DC1) in 1993, a research associate at Waseda University in 1995, an assistant professor at Toho University in 1997 and an associate professor in 2004. He became an associate professor at Meiji University in 2013 and has been engaged as a professor at Meiji University since 2016. He has joined the green computing system research organization as a visiting professor at Waseda University since 2014. His research interests include parallel computing, parallelizing compiler and data-locality optimization. He is a member of the IPSJ, IEICE, IEEJ, IEEE and ACM.

Keiji Kimura received the B.S., M.S. and Ph.D. degrees in electrical engineering from Waseda University in 1996, 1998, 2001, respectively. He was an assistant professor in 2004, associate professor of Department of Computer Science and Engineering in 2005, and professor in 2012 at Waseda University. He was also a department head of CSE from 2015 to 2016 and an assistant dean of FSE from 2016 to 2017. He is a recipient of 2014 MEXT (Ministry of Education, Culture, Sports, Science and Technology in Japan) award. His research interest includes microprocessor architecture, multiprocessor architecture, multicore processor architecture, and compilers. He is a member of IPSJ, ACM and IEEE. He has served on program committee of conferences such as ICCD, ICPADS, ICPP, LCPC, IISWC, ICS, IPDPS, and PACT.

Hironori Kasahara is IEEE Computer Society (CS) 2018 President and a senior executive vice president of Waseda University. He received a BS in 1980, a MSEE in 1982, a PhD in 1985 from Waseda University, Tokyo, joined its faculty in 1986, and has been a professor of computer science since 1997 and a director of the Advanced Multicore Research Institute since 2004. He was a visiting scholar at University of California, Berkeley, in 1985 and the University of Illinois at Urbana-Champaign's Center for Supercomputing R&D from 1989 to 1990. Kasahara received the IEEE Fellow in 2017, CS Golden Core Member Award in 2010, IEEE Eta Kappa Nu Professional Member in 2017, IFAC World Congress Young Author Prize in 1987, IPSJ Fellow in 2017 and Sakai Special Research Award in 1997, and a Science and Technology Prize of the Japanese Minister of Education, Culture, Sports, Science and Technology in 2014. He led Japanese national projects on parallelizing compilers and embedded multicores, and has presented 216 papers, 176 invited talks, and 48 international patents. His research on multicore architectures and software has appeared in 608 newspaper and Web articles. He has served as a chair or member of 258 society and government committees, including the CS Board of Governors; Executive Committee; Planning Committee; chair of CS Multicore STC and CS Japan chapter; associate editor of IEEE Transactions on Computers; vice PC chair of the 1996 ENIAC 50th Anniversary International Conference on Supercomputing; general chair of LCPC 2012; PC member of SC, PACT, and ASPLOS; board member of IEEE Tokyo section; and member of the Earth Simulator committee.