Restructuring Field Layouts for Embedded Memory Systems

Keoncheol Shin
Division of Computer Science
KAIST
Daejeon, Korea

Jungeun Kim*
System Architecture Lab
Samsung Electronics Corp.
Suwon, Korea

Seonggun Kim
Division of Computer Science
KAIST
Daejeon, Korea

Hwansoo Han
Division of Computer Science
KAIST
Daejeon, Korea

Abstract

In many computer systems with large data computations, the delay of memory access is one of the major performance bottlenecks. In this paper, we propose an enhanced field remapping scheme for dynamically allocated structures in order to provide better locality than conventional field layouts. Our proposed scheme reduces cache miss rates drastically by aggregating and grouping fields from multiple instances of the same structure, which implies the performance improvement and power reduction. Our methodology will become more important in the design space exploration, especially as the embedded systems for data oriented application become prevalent. Experimental results show that average L1 and L2 data cache misses are reduced by 23% and 17%, respectively. Due to the enhanced localities, our remapping achieves 13% faster execution time on average than original programs. It also reduces power consumption by 18% for data cache.

1 Introduction

Embedded systems have limited battery, memory and processing power, which is the most contrasting difference between embedded systems and general purpose systems. Early applications of embedded systems are restricted to rather simple tasks. In recent years, however, with increasing design complexity and demand for large data computation like multimedia applications, memory subsystems in embedded systems have become not only the major performance bottleneck but also energy sink.

Panda et al. surveyed various mechanisms related to memory optimizations for embedded systems in [5]. Many improvements through additional hardware, new architectures and compiler optimizations are achieved. Cache memories are frequently found in modern embedded processors in order to reduce the memory access time. Many locality enhancing optimizations are evaluated as beneficial, since improved data reuse and prefetch effect could amortize the optimization overhead. Data remapping is one of the techniques used to improve temporal and spatial locality in structure-type data [6, 4, 2, 8]. While early works often focused on statically allocated data alone, recent researches take account of dynamically allocated data as well, since complex applications tend to use more dynamic data than static data.

Compiler optimizations traditionally played important roles in minimizing the memory footprints of programs. Kistler et al. investigated the effectiveness of field reordering [4]. They used a profiling technique to find out the best order of fields within structures. Chilimbi et al. split Java classes into hot portions and cold portions based on the profiles of field access frequencies [2]. Extra pointers are appended at the end of the hot field areas, linking cold field areas. While they are able to reduce L2-cache misses, the total memory usage may increase due to the extra pointer fields. In addition to that, accessing fields in cold areas requires extra memory references to go through the extra pointer links. Truong et al. proposed another approach to change data layouts through field reorganization and instance interleaving [8]. Identical fields from the multiple instances of the same structure-type are consecutively placed with the consideration of cache-line alignment. Using this scheme, rarely used fields are moved away from frequently used fields. Their placement method, however, is limited in that all fields should be grouped into two parts and the size of each group should be the same.

Rabbah and Palem proposed a vertical field layout that consecutively places the same fields from multiple structure instances by using customized allocation routines and compile-time transformations of field offset calculations [6]. Their method is similar to what Truong et al. proposed in a sense that fields from multiple instances are placed together. The weakest point of their layouts is that they need to insert padding space between fields in order to make constant offsets for all fields. Intuitively, extra padding could increase memory usages and cache misses.

In this paper we present a new field remapping scheme

*This work is done while she was in the division of computer science at KAIST that integrates the benefits from previous works and overcomes the defects of previous works. The main contributions of our remapping scheme are summarized as follows:

- no restriction on field types and field sizes,
- no waste of memory due to padding, and
- no address computation overhead for frequently accessed fields.

The remaining paper is organized as follows: Section 2 discusses the motivation of this research. The data remapping algorithm and our transformation methodology are detailed in Section 3. Our evaluation environments and results are shown in Section 4. Finally, we conclude with summary and discussion.

2 Motivation

Before we introduce our field remapping scheme, we start by describing a field clustering technique. A structure consists of diverse data fields with different sizes. We often encounter programs where reorganizing fields within structures is helpful for their performance. Refer to the example code in Figure 1. The function search() in the example traverses a linked list and returns the data value in the node that has a first matching key. We can find that *key* and *next* fields are accessed every iteration whereas *data* fields are accessed just once when the function returns. Considering the access frequencies, we can categorize *key* and *next* as hot fields, and *data* as a cold field. Generally speaking, it would be beneficial to fetch and store as many hot fields as possible in a cache with one memory access.

To achieve this purpose, Rabbah and Palem's remapping method (DDRemap) is applied to the example in Figure 1 and consecutively locates all the fields from multiple structures as shown in Figure 2(a). Notice that *data* fields are

```
Node {
    int   key;          // 4 bytes
    char  data[6];      // 6 bytes
    Node  ∗next;        // 4 bytes
} ∗T;

char∗  search (int key) {
    Node  ∗cur = T;
    while (cur != NULL) {
        if (cur −>key == key)
            return cur−>data;
        cur = cur−>next;
    }
    return NotFound;
}
```

Figure 1. Motivation example: structure type definition of Node and search function for the list of nodes.

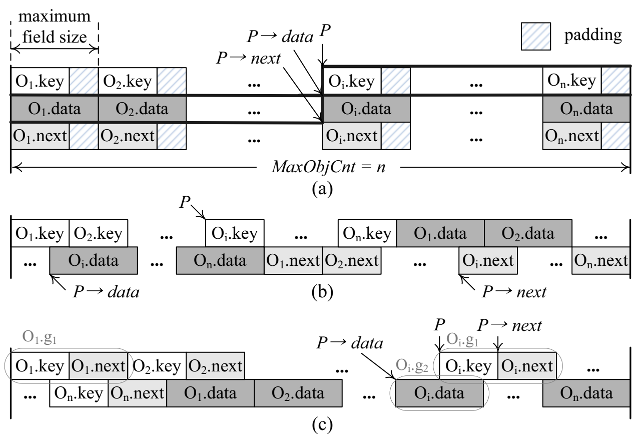


Figure 2. (a) Structure layout after field remapping with Rabbah and Palem's, (b) Structure layout after compacting fields of (a), (c) Structure layout after our field remapping.

put aside from the frequently used fields. To access the fields in structure objects, compilers statically generate field offsets from the base addresses of the structure objects. Note that all the fields of one object are vertically located with a constant interval $MaxFieldSize \times MaxObjCnt$ as marked with thick line in Figure 2(a). $MaxFieldSize$ is the maximum field size among all fields f_i in the structure. $MaxObjCnt$ is the maximum number of the structure objects that can be stored in a memory pool. Since the size of the memory pool is fixed, we can determine the $MaxObjCnt$ value at compile time.

Rabbah and Palem's scheme has extra padding between fields to make the offset calculation of each field a constant. We can improve cache behavior by eliminating the useless padding like Figure 2(b). This compaction intrinsically requires extra instructions to compute address at runtime. However, we can completely eliminate the extra instructions for frequently accessed fields and enhance cache behavior by field grouping technique. Moreover, we can apply several strength reduction optimizations to minimize the access overhead of infrequently used fields. Figure 2(c) shows the layout after our remapping scheme.

3 Restructuring Field Layout

We call our remapping scheme *Compact Group Field Remapping* (CGFRemap). Full details of CGFRemap will be discussed in the following two sections.

3.1 Field Compaction

Our first objective is to get rid of wasted padding space. Figure 2(b) shows the layout in which all fields are compactly packed. If we generalize the offset calculation of compactly packed layout, the address offset of the i_{th} field(f_i) of the object pointed by P is:

$$\begin{aligned}
&Offset(P \rightarrow f_i) \\
&\quad= FieldSize(f_1) \times (MaxObjCnt - rank) \\
&\quad+ \sum_{k=2}^{i-1} FieldSize(f_k) \times MaxObjCnt \\
&\quad+ FieldSize(f_i) \times rank \\
&\quad= \sum_{k=1}^{i-1} FieldSize(f_k) \times MaxObjCnt \\
&\quad+ (FieldSize(f_i) - FieldSize(f_1)) \times rank \qquad (1)
\end{aligned}$$

When the field offset is computed, we need to know the relative position i of the object in the memory pool. Let $rank$ be $i - 1$ to represent the relative position. In Equation 1, $FieldSize(f_i)$ for all fields are compile-time constants. $MaxObjCnt$ is determined at compile time, since it depends on the sizes of one memory pool and one structure instance. The only variable that needs to be determined at run time is the value of $rank$. Since we cannot know the relative positions of structure objects in a memory pool at compile time, they need to be calculated at runtime. In the later section we describe how to reduce the runtime overhead involved in offset calculations.

3.2 Field Grouping

In general many applications frequently use two or three fields together among several fields within a structure. Refer to our motivation example in Figure 1. The $search()$ function in the code intends to access key and $next$ fields together. If we layout fields on the memory as in Figure 2(c), the probability of fetching the two fields together would increase. Consequently, our scheme has to decide which fields are grouped together. We can easily get the field access sequences by profiling, since embedded systems usually use a few typical programs. From these access sequences, we can build temporal relationship graph(TRG) [3] modified to allow self edges. We can determine whether to combine the nodes connected by largest weighted edge in the TRG through field affinity relations.

As shown in Figure 2(c), in order to access the $next$ field of O_i, we have to know the offset of the field group to which the field $next$ belongs. We then find the offset within the group. By adding the group offset and intra-group offset, we can calculate the exact offset of $next$ from P. If we generalize this observation, we can write the following equation for the field offset in CGFRemap:

$$\begin{aligned}
&CGFRemap(P \rightarrow f_i) = CGFRemap(P \rightarrow g_{u,v}) = \\
&CGFRemap_{group}(P \rightarrow g_u) + \sum_{k=1}^{v-1} FieldSize(f_k \; in \; u_{th} \; group) \\
&CGFRemap_{group}(P \rightarrow g_u) = \sum_{k=1}^{u-1} FieldSize(g_k) \times MaxObjCnt \\
&\qquad + (FieldSize(g_u) - FieldSize(g_1)) \times rank \qquad (2)
\end{aligned}$$

When we want to access f_i by denoting $P \rightarrow f_i$, we have to find an equivalent expression $P \rightarrow g_{u,v}$. The field notation $g_{u,v}$ means the v_{th} field within the u_{th} field group. The way in which we get the group offset is virtually the same as Equation 1. We just substitute the group name for the individual field name as in Equation 2.

When we decide the order among field groups, we place the most frequently accessed group at the first place. Since the group offset of the first group is zero, only the intra-group offset is needed for the field offsets in the first group. In addition to that, intra-group offsets are all compile-time constants, once we decide which fields belong to which groups. As a result, frequently used fields should be placed at the first group and their field offsets will be compile-time constants. For the rest of field groups we need to perform extra instructions at runtime to calculate the field offsets. The following subsections describe how we can reduce the overhead of runtime offset calculations.

3.2.1 $rank$ calculation using pool alignment

CGFRemap needs to calculate $rank$ at runtime to find a relative position of the structure object in the pool. If we know the start address of the pool to which the object pointed by P belongs, we can easily calculate $rank$ from the address of P. Assume that the value of P is 0x4b329030 and the base address of pool is 0x4b329000 as shown in the figure 3(a). Suppose that the size of the first field group is four bytes, the $rank$ of the object pointed by P is (0x4b329030 - 0x4b329000)/4 = 12. A simple thought would record the base addresses of all memory pools in a table and look up the table whenever necessary. This is, however, very expensive to calculate the $rank$, since this scheme would require extra memory references. To reduce the overhead involved in the calculation of $rank$, we align pools on 4K boundaries as shown in the Figure 3(b). Assuming we have a pointer P whose value is 0x4b329030, we can extract the base address of the pool using a bit operation (0x4b329030 & 0xfffff000 = 0x4b329000).

Figure 4 is our customized memory allocation function for CGFRemap. This function preserves a large amount of memory called $pool$. Whenever the customized memory allocation is called, it returns the address within a previously preserved pool as long as the pool can hold the structure. When the function $getPool\_4Kaligned()$ in the code is first called to get a new pool, it allocates a large amount of memory called $bank$ which can hold multiple pools inside. It then returns the address of 4K pool aligned on the 4K boundaries. For subsequent requests of pools, it returns a pool from the previously allocated bank. One more thing to note is that there is a leading space in a bank. As the start address of $bank$ given by $malloc()$ is arbitrary in reality, we skip a leading memory space in order to start the first pool from the 4K aligned address. In our experiment the bank

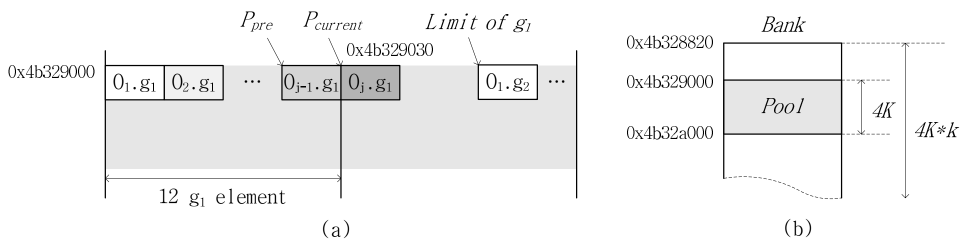


Figure 3. (a) A memory $Pool$ and description of allocation mechanism, (b) Aligning a memory pool on 4K boundaries.

sizes are adapted from 40KB to 4000KB according to the data sizes of applications.

```
Input: record type ($R$),
      max number of objects in a pool ($MaxObjCnt$).
Output: valid heap (base) address where $R$ is allocated.

01. Static Initialize ($Base = Limit = 0$);
02. if $Base == Limit$ then
      // get a 4KB memory pool
03.   $Base \leftarrow getPool\_4Kaligned(4096)$;
04.   $Limit \leftarrow Base + MaxObjCnt \times FieldSize(R.g_1)$;
05. end if
06. $Address \leftarrow Base$;
07. $Base \leftarrow Base + FieldSize(R.g_1)$;
08. Return $Address$;
```

Figure 4. Customized memory allocation algorithm for our field remapping scheme.

3.2.2 Strength reduction in offset calculations

We used several strength reduction techniques to minimize the runtime overhead of address computations. The optimized equation for $rank$ is as follows.

$$\begin{aligned} rank(P) = \quad & (P \ \&\ BitMask)\ /\ FieldSize(g_1) \\ \Rightarrow \quad & (P \ \&\ BitMask) \gg Bits(g_1) \qquad (if\ |g_1| = 2^k) \end{aligned} \tag{3}$$

First, we can substitute subtraction by bitwise AND operation due to the pool alignment. $BitMask$ is used to get the difference between the starting address of the memory pool and the address value of the pointer P. If we use 4KB memory pools, the difference would be the last 12 bits from the pointer P. $BitMask$ (0x00000fff) zeroes out all higher bits except for the lower 12 bits of P.

Second, we can replace the division operation with the shift operation if the bit size of the first group is in the form of power of two. As mentioned previous section, frequently used fields are put in the first group. Then we do not have to provide rank calculations for them. If the size of the first group is not power of two, we had better find another field that has high frequency of uses and makes the size power of two. Now we can replace $rank$ in equation 2 with equation 3, then rewrite of the equation is as follows.

$$\begin{aligned} CGFRemap_{group}(P \rightarrow g_u) = & \sum_{k=1}^{u-1} FieldSize(g_k) \times MaxObjCnt + \\ & (FieldSize(g_u) - FieldSize(g_1)) \times (P\&BitMask) \gg Bits(g_1) \end{aligned} \tag{4}$$

The sum of field group sizes from g_1 to g_{u-1}, $MaxObjCnt$, and the size difference between the first field group and the u_{th} field group can be obtained at compile time. Even with those constants, we need four extra run-time instructions to calculate the offset: one add, one multiply, one bitwise AND, and one shift. It is possible to further simplify Equation 4. When the u_{th} field group size is the same as the first field group size, the result of $(FieldSize(g_u) - FieldSize(g_1))$ becomes zero making the whole second term zero. In this case we do not require any extra run-time instructions for the offset calculation.

Finally, multiply can be replaced with a shift operation if the result of $(FieldSize(g_u) - FieldSize(g_1))$ is in the form of power of two. In this case we can also combine two shift operator and reduce the calculation to one shift operation. Figure 5(c) shows the assembly code as a result of strength reductions. Register $\%eax$ has the value of pointer cur which points to the instance of Node structure in Figure 1. As described so far, there are several opportunities to reduce the overhead of the offset calculation. Even though most of these optimizations are conditional, we could apply these optimizations to the most of the field references in the experiment.

```
cur->key = 1;        | movl  $1, (%eax)    |  movl  $1, (%eax)
cur->next = 0;       | movl  $0, 12(%eax)  |  movl  $0, 4(%eax)
cur->data[0] = 'a';  | movb  $97, 4(%eax)  |  movl  %eax, %edx
                     |                     |  andl  %4095, %edx
                     |                     |  sarl  $2, %edx
                     |                     |  subl  %edx, %eax
                     |                     |  movb  $97, 2336(%eax)

        (a)                   (b)                    (c)
```

Figure 5. (a) original source code, (b) assembly code for original layout, (c) assembly code for our layout.

4 Experimental Evaluation

Four applications from the Olden benchmark version 1.01 [1] are used to substantiate our claims on locality improvement. Tsp is a famous travelling salesman problem solver, which uses a balanced binary-tree. $Health$ simulates the Columbian health care, which heavily uses double-linked lists. Mst is a benchmark to calculate the minimum spanning trees from the input graphs. *Perimeter* is an application to calculate perimeters of the regions that are represented with Quad-tree. For our experiment, we manually modified the benchmark codes to access fields and allocate structures with our macros.

We measured execution times on a RedHat 9.0 Linux PC equipped with a 2.4GHz Pentium4 processor which has 8KB L1 data cache (64byte cache line, 4-way set associative), 512KB L2 cache (64byte cache line, 8-way set associative) and 1GB main memory. All benchmarks are compiled with gcc (Ver. 3.2.2) -O3. For simulating the behavior of cache memory, we used Cachegrind in the Valgrind's Tool suite (Ver. 3.0.1) [9]. Sim-Panalyzer (Ver. 2.0.2) [7] is used for the power estimation of the cache memory.

4.1 Impact on cache memory

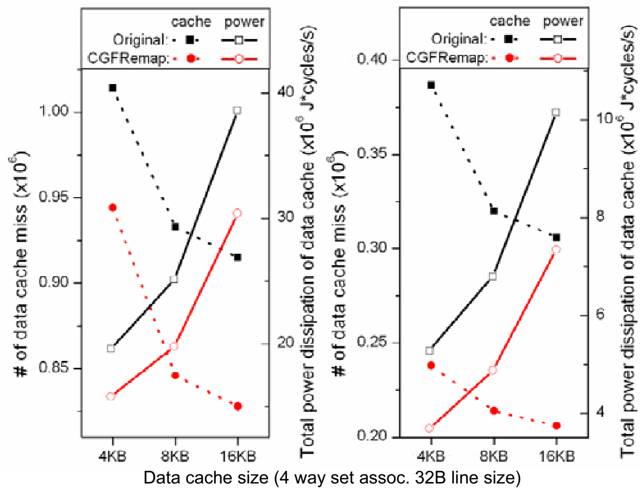
In order to evaluate memory performance, we measured the numbers of cache misses. Table 1 and Table 2 show L1 and L2 cache misses respectively. We used the same cache configuration as the machine on which we measured execution times. The column labeled *Original* represents the base results from unoptimized layout. The results using Rabbah and Palem's method [6] are shown in the column labeled *DDRemap*. The results under *CGFRemap* represents the measurements from our layout method. Compared to original layouts, reductions of CGFRemap are 23.1% for L1 and 17.4% for L2. DDRemap achieves fairly good reductions in L1 and L2 cache misses, but less than CGFRemap on average. Average numbers for DDRemap look bad due to the pathological cases in Tsp, where structures consist of fields with different sizes.

Table 1. The numbers of misses in L1 data cache ($\times 10^3$).

Benchmarks	Input	Original	DDRemap	CGFRemap
Tsp	4096	187.4	250.5	92.0
	16384	808.7	1028.6	423.7
Health	7,20	510.5	500.2	483.5
	9,20	8165.7	8054.7	7763.2
Mst	2048	25207.4	23494.5	21330.7
	3000	53499.3	50473.2	45847.6
Perimeter	11	85.7	72.4	66.8
	12	167.0	138.9	126.2
Avg.% improved		-	-1.59%	+23.11%
excluding Tsp		-	+8.02%	+14.39%

Table 2. The numbers of misses in L2 cache ($\times 10^3$).

Benchmarks	Input	Original	DDRemap	CGFRemap
Tsp	4096	6.6	9.0	6.2
	16384	92.4	86.5	68.5
Health	7,20	476.8	382.5	416.4
	9,20	7681.8	6273.5	6791.1
Mst	2048	20659.0	20820.3	18279.0
	3000	51827.6	48788.4	43771.3
Perimeter	11	64.8	49.1	47.5
	12	143.0	106.3	103.5
Avg.% improved		-	+8.07%	+17.36%
excluding Tsp		-	+15.54%	+17.60%



(a) Health with input :7,20 (b) Perimeter with input: 11

Figure 6. Tradeoff between cache miss and power dissipation as cache size increases.

Figure 6 shows the changes of cache misses and power consumption as we vary the size of the cache. Our scheme reduces the average power consumption of the data cache by 17.6% on the same cache size. In addition to that, we can see that our scheme preserves the application's performance with less than half the cache size. This result can play a significant role in optimizing the memory subsystem design in embedded systems. Design space exploration can make full use of our scheme to reduce cache sizes without loss of performance, which means better designs in power, cost, and size. In summary, CGFRemap is able to enhance the memory performance by benefiting from field compacting and hot field grouping. It can also reduce the power consumption of the data cache by reducing switching activities.

4.2 Impact on execution times

Table 3 shows the execution times of each benchmark with various input values. The execution times under the Original column show the base performance with unoptimized layouts. The following columns show execution times and the percentages of improvement for corresponding layout optimizations. When we use small size inputs, the effect of each optimization is similar since the execution time is too short. To intensify the impact of improved locality on performance, we run the experiment with large size inputs. DDRemap works relatively well except for Tsp. Average improvement in execution times excluding Tsp is 8.5%, which is comparable to our field remapping. With a large input ($2 * 10^7$), however, unnecessary padding in

Table 3. Comparison of execution times for different field layouts.

Bench-marks	Input	Original (sec)	DDRemap (sec/% improved)		CGFRemap (sec/%improved)	
Tsp	$5 * 10^5$	2.96	2.76	+6.8%	2.75	+7.1%
	$5 * 10^6$	50.57	46.85	+7.4%	45.11	+10.8%
	$2 * 10^7$	881.93	1421.95	-61.2%	637.59	+27.7%
Health	10,20	3.97	3.73	+6.1%	3.63	+8.6%
	11,20	15.36	14.49	+5.7%	13.78	+10.2%
Mst	3000	6.70	7.00	-4.5%	6.55	+2.2%
	10000	731.25	716.49	+2.0%	694.33	+5.0%
Perimeter	11	2.57	1.99	+22.6%	2.011	+21.8%
	12	765.20	618.82	+19.1%	612.17	+20.0%
Avg.% improved		-	-	+0.43%	-	+12.61%
excluding Tsp		-	-	+8.49%	-	+11.32%

DDRemap takes toll on the performance, slowing the execution about 60%. On the contrary, CGFRemap does not have pathological cases as in DDRemap. Average performance improvement is 12.6%, which is better than DDRemap.

Since our method uses runtime offset calculations for some fields, the benefits from improved locality can be reduced by the runtime overhead. Table 4 shows the increased static code sizes due to address computations. Even with these increased overhead, we could achieve better performance by enhancing locality further.

Table 4. Static code size for Original and CGFRemap.

Benchmarks	Tsp	Health	Mst	Perimeter
Original	18758B	18774B	18232B	17297B
CGFRemap	20315B	19362B	18583B	17602B
Increased%	+8.3%	+3.1%	+1.9%	+1.8%

5 Conclusion and Future Work

We present a new field remapping method for structure-type data, CGFRemap. CGFRemap places bundles of the same fields on consecutive memory in turn, instead of mapping a structure to the memory as a whole. CGFRemap eliminates unused padding introduced in the previous work, but with the runtime overhead to calculate field offsets. For the hot fields, however, CGFRemap does not require runtime calculations for their field offsets. Thus, grouping makes CGFRemap further reduce cache misses with less overhead. Compared to original layouts, CGFRemap reduces L1 cache misses and L2 cache misses by 23% and 17% respectively. As a result, execution times are reduced by 13% and power consumptions are reduced by 18% in the data cache. Our remapping scheme can be utilized to find good alternate solutions to meet the specified constraints in the design space exploration.

In the experiment, we translated structure related statements in the benchmarks to the predefined macros by processing at the source level. Thus, it would be more efficient to generate the assembly codes for address computations automatically in the compiler. If we complete the automation with compiler, we can expect more reduction in execution cycles needed for runtime offset calculations.

6 Acknowledgement

This research was supported by the MIC(Ministry of Information and Communication), Korea, under the ITRC(Inofrmation Technology Research Center) support program supervised by the IITA(Institute of Information Technology Assessment) (IITA-2005-C1090-0502-0031)

References

[1] M. Carlisle and A. Rogers. Olden benchmark. http://www.cs.princeton.edu/ mcc/olden.html.

[2] T. Chilimbi, B. Davison, and J. Larus. Cache-conscious structure definition. In *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementations (PLDI'99)*, pages 13–24, May 1999.

[3] N. Gloy, T. Blackwell, M. Smith, and B. Calder. Procedure placement using temporal ordering information. In *Proceedings of IEEE/ACM International Symposium on Microarchitecture (MICRO'97)*, pages 303–313, December 1997.

[4] T. Kistler and M. Franz. Automated data-member layout of heap objects to improve memory-hierarchy performance. *ACM Transactions on Programming Languages and Systems*, 22(3):490–505, May 2000.

[5] P. Panda, F. Catthoor, N. Dutt, K. Danckaert, E. Brockmeyer, C. Kulkrani, A. Vandercappelle, and P. Kjeldsberg. Data and memory optimization techniques for embedded systems. *ACM Transactions on Design Automation of Electronic Systems*, 7(2):149–206, April 2001.

[6] R. M. Rabbah and K. V. Palem. Data remapping for design space optimization of embedded memory systems. *ACM Transactions on Embedded Computing Systems*, 2(2):186–218, May 2003.

[7] Sim-panalyzer. http://www.eecs.umich.edu/ panalyzer.

[8] D. N. Truong, F. Bodin, and A. Seznecs. Improving cache behavior of dynamically allocated data structures. In *Proceedings of International Conference on Parallel Architectures and Compilation Techniques (PACT'98)*, page 322, Octobor 1998.

[9] Valgrind. http://www.valgrind.org/.