On-the-fly Structure Splitting for Heap Objects

ZHENJIANG WANG and CHENGGANG WU, State Key Laboratory of Computer Architecture, Institute of Computing Technology, Chinese Academy of Sciences
PEN-CHUNG YEW, University of Minnesota at Twin Cities
JIANJUN LI and DI XU, State Key Laboratory of Computer Architecture, Institute of Computing Technology, Chinese Academy of Sciences

With the advent of multicore systems, the gap between processor speed and memory latency has grown worse because of their complex interconnect. Sophisticated techniques are needed more than ever to improve an application's spatial and temporal locality. This paper describes an optimization that aims to improve heap data layout by structure-splitting. It also provides runtime address checking by piggybacking on the existing page protection mechanism to guarantee the correctness of such optimization that has eluded many previous attempts due to safety concerns. The technique can be applied to both sequential and parallel programs at either compile time or runtime. However, we focus primarily on sequential programs (i.e., single-threaded programs) at runtime in this paper. Experimental results show that some benchmarks in SPEC 2000 and 2006 can achieve a speedup of up to 142.8%.

Categories and Subject Descriptors: D.3.3 [**Data types and structures**]

General Terms: Performance

Additional Key Words and Phrases: data layout, runtime, structure splitting, safety

ACM Reference Format:
Wang, Z., Wu, C., Yew, P., Li, J., and Xu, D. 2011. On-the-fly Structure Splitting for Heap Objects. ACM Trans. Architec. Code Optim. 1, 1, Article 1 (January 1111), 20 pages.
DOI = 10.1145/0000000.0000000 http://doi.acm.org/10.1145/0000000.0000000

1. INTRODUCTION

The huge performance gap between modern processors and their memory has long been a main barrier that limits system performance (i.e., so called *memory wall*). The trend shows no sign of relief in the foreseeable future, in particular with the advent of multicore systems because of their complex interconnect and the limited bandwidth to off-chip memory. One of the effective ways to mitigate such problems is to exploit program locality. In this paper, we propose a scheme to optimize *data layout* [Calder et al. 1998; Rabbah and Palem 2003; Rubin et al. 2002; Truong et al. 1998] to enhance such program locality.

Author's addresses: Z. Wang and C. Wu (the corresponding author), State Key Laboratory of Computer Architecture, Institute of Computing Technology, Chinese Academy of Sciences; P. Yew, Department of Computer Science and Engineering, University of Minnesota at Twin Cities; J. Li and D. Xu, State Key Laboratory of Computer Architecture, Institute of Computing Technology, Chinese Academy of Sciences;
This article is supported in part by the National Science and Technology Major Project of China (2009ZX01036-001-002), the National Natural Science Foundation of China (NSFC) grant 60736012, the Innovation Research Group of NSFC grant 60921002, the Chinese National Basic Research grant 2011CB302504, the National High Technology Research and Development Program of China (No. 2007AA01Z110) and NSF CNS-0834599 in USA.

DOI 10.1145/0000000.0000000 http://doi.acm.org/10.1145/0000000.0000000

In many applications, dynamically allocated heap objects occupy a large portion of their working space. Some techniques [Huang et al. 2004; Chilimbi et al. 1999b; Lattner and Adve 2005; Wang et al. 2010; Zhao et al. 2005] improve the inter-object locality by controlling the layout of heap objects, e.g., aggregating closely-related heap objects in memory pools. However, such techniques do not modify the internal layout of these objects (*structures*).

In a structure type, its *fields* usually exhibit different access behavior. Some compiler techniques [Chilimbi et al. 1999a; Kistler and Franz 2000] exploit *affinity*, or *access frequency*, of the fields using profiling or static analysis, and reorder the fields (called *Field Reordering*). Fields with similar *affinity* are placed together so they can fit in the same cache line. Unlike field reordering, *Structure Splitting* [Chilimbi et al. 1999a; Hundt et al. 2006; Hagog and Tice 2005] does not keep the original structure intact. It breaks a structure into two or more pieces according to the *affinity* of its fields. A component of the split structure may need to be accessed by a new pointer. In certain cases (e.g., array of structures), array indices can be used instead of adding new pointers (called *Structure Peeling* [Hagog and Tice 2005]). It can make the layout denser as well as eliminating indirect accesses by pointers.

Such optimizations on the internal layout of a structure will change the definition of the structure. *All* references to its fields must be identified and modified to reflect the change of the layout. Such optimizations are unsafe if any of its references is undetected by the compiler and left unchanged. The modification is relatively simple for *type safe* languages, but is quite challenging for *type unsafe* languages such as C and C++. For example, if an optimized structure is type cast to another type T and is accessed with T's definition, the access must be modified. Another example is passing optimized structure instance/pointer to functions whose source is unavailable at compile time, like library functions. Existing techniques [Hundt et al. 2006; Zhao et al. 2007] use heuristic rules such as *type escape analysis* or *type compatibility analysis* to check whether a structure type is safe to be transformed or not.

Although structure layout optimizations can improve performance, they are beneficial only for their favorite access patterns. Such optimizations are not applied in many widely-used compilers because of the applicability or the safety concerns, so they are rarely used in existing binaries. Therefore, a runtime approach is useful on these binaries. However, safety becomes even more crucial because of the lack of high-level information could prevent precise analysis at runtime.

In this paper, we present a framework to apply structure splitting at runtime on *binary code*. It recognizes the fields of structures and optimizes promising structure instances. Instructions that access the optimized instances are captured and redirected to the new layout. In order to guarantee safety, it employs a protection and checking mechanism to detect uncaptured references to the *old addresses*. Some other layout optimizations like field reordering can also be adopted.

The main contributions in this work include:

— An on-the-fly structure splitting system that enhances the locality of *aggregated* structures at runtime. It uses page protection and address checking to guarantee the safety of optimizations.
— A special pool allocation strategy and a pointer comparison mechanism that can reduce the overhead of address computation after structure splitting.
— A detailed evaluation that shows the effectiveness of the system.

The rest of the paper is organized as follows. In Section 2, we discuss related work. Section 3 presents the system. We describe the problems and solutions to *safety* issues in Section 4, and present some techniques that can reduce the runtime overhead in

Section 5. The experimental results are in Section 6. Section 7 discusses the use of safety mechanism in static compilers, and Section 8 concludes the paper.

2. RELATED WORK

Chilimbi et al [Chilimbi et al. 1999b] proposed a tree reorganizer called *ccmorph*. It used topological information to improve locality by applying clustering and coloring techniques at runtime. It works very well for read-mostly data structures that are built early in a computation and heavily referenced later on. The tool needs programmers to guarantee the safety of the optimizations: the tree must have homogeneous elements and have no external pointers pointing to the inside of the structure.

Ding et al [Ding and Kennedy 1999] proposed a runtime optimization that packs data accessed frequently in short time intervals into the same cache line. The core mechanism that supports such packing is a *runtime data map*. It maps an old location before data packing to its new location after data packing. Static compiler augments each access to a transformed array with an indirection to the corresponding runtime map. It also relies on user directives to determine whether packing should be applied, when and where packing should be performed, and which access sequence should be used to direct packing.

Garbage collectors (GC) can identify all references to an object. Hence, it has the ability to relocate data safely at runtime. [Huang et al. 2004] collected runtime information on data access patterns in object-oriented languages, and presented a new copying algorithm that utilized this information to produce a cache-conscious object layout. However, many programs such as C and C++ do not have GC support. Their unrestricted usage of pointers prevents data relocation unless with the help of programmers. Therefore, many layout optimizations in static compilers do not support relocating data at runtime.

Chilimbi et al [Chilimbi et al. 1999a] showed an implementation of structure splitting in a static compiler. They measured *field-access frequencies* on a per-class basis in type-safe programs using instrumentation, and split classes into hot and cold portions based on *field-access counts* and some heuristics. A static compiler generated optimized native code using some splitting criteria. They also described a tool *bbcache* that could provide field reordering advice to C programs.

Hundt et al [Hundt et al. 2006] used both *hotness* and *closeness* to guide structure splitting for C programs, for both profile- and non-profile-based compilation. Some legality criteria were used to determine whether it is safe to transform a type. They also proposed an advisory tool that combined static analysis with runtime profiling to guide structure layout decisions.

Lin et al [Lin and Yew 2010] proposed to use groups of objects as candidates, instead of objects of the same type. The objects in the same group are aliased with each other whereas the objects in different groups are not aliased. Their approach could expose more opportunities for structure layout optimization and array flattening.

Curial et al [Curial et al. 2008] combined structure splitting with pool allocation. Their memory pools are designed to generate more efficient code. Instead of allocating the entire structure in one place, only the first field is allocated there. The remaining fields are allocated in other parts of the pool. Their design reduces the number of instructions required to access the data after the structure splitting.

Our scheme uses a runtime address checking and page protection mechanism to guarantee the safety of optimizations. The checking code includes array bound checking that can be optimized [Bodík et al. 2000; Gupta 1990; 1993]. Page protection has been used in other work, such as keeping strong atomicity in transactional memory [Abadi et al. 2009] and detecting violation of locking discipline [Rajamani et al. 2009].

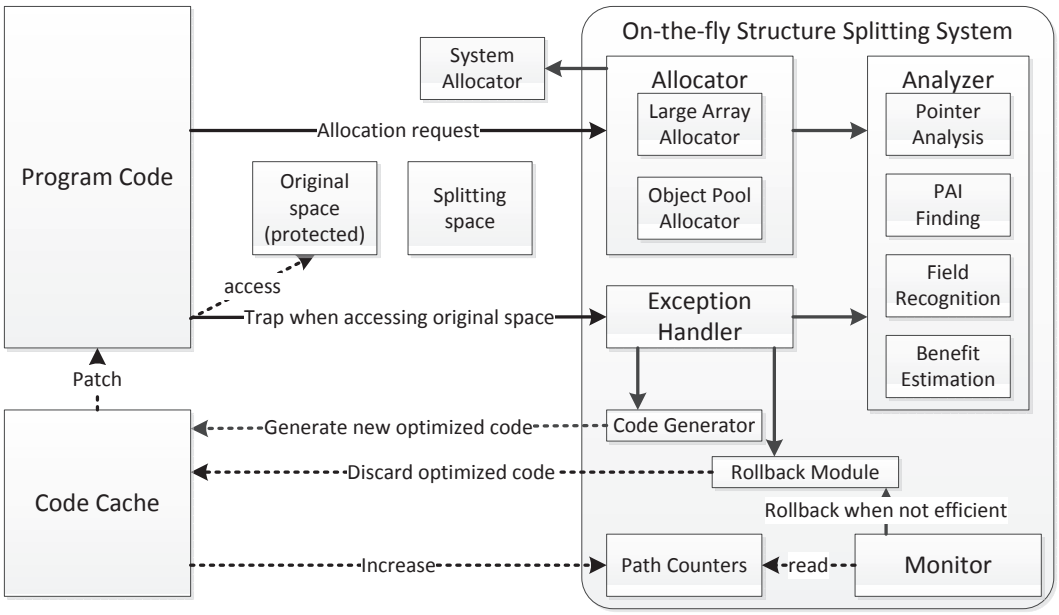

Fig. 1. System Overview

3. SYSTEM OVERVIEW

Our On-the-fly Structure Splitting (OSS) system works on IA-32 platforms running Linux. It has a shared library that is pre-loaded (with the LD_PRELOAD environment variable) into the memory space of an executable. Figure 1 gives an overview of the OSS system. It intercepts allocation requests, detects opportunities for structure splitting, reorganizes data in the memory and redirects their accesses, all at runtime. In order to guarantee safety, the system traps accesses to the old data regions and redirect those accesses to the new locations.

For clarity, we use the term *record* instead of *structure* for the rest of the paper to differentiate it from the generic term of *data structure*. An instruction that may access optimized records is called PAI (*possibly-accessing instruction*).

Two types of heap memory are selected as optimization candidates: 1) large record array, typically larger than the cache size, and 2) records that are aggregated by object pool allocator. They both consist of a large number of records so that the accumulated performance impact can be substantial. Other allocation requests are directed to the default allocator and are served accordingly.

For a request that requires a large allocation of memory blocks, or the number of objects in a request's memory pool reaches a threshold, its optimization potential is analyzed. The analysis includes pointer analysis, PAI finding, field recognition and benefit estimation. If the results show a good potential, the structure splitting optimization will be triggered. Two blocks of memory of the requested size are allocated. They are called *original space* and *splitting space*, respectively. The original space is page protected (marked as non-readable and non-writable) and is returned to the program, while the splitting space is prepared for the split data.

Later, when a PAI tries to access the original copy of the optimized record, an exception will be raised. The exception handler in the optimizer does a similar analysis as in the allocator, but with detailed runtime information at the exception point. OSS generates a new code fragment that accesses the *splitting space* instead of the *original*

space. The new code fragment is placed in a code cache, and is patched to replace the original PAI. The program continues to run, and the patched PAI will not raise exceptions again. In case that any violation prevents generating correct code, OSS uses a rollback mechanism to restore the program to the unoptimized (i.e., the original) state. The program can continue to run correctly.

A monitor in our system periodically checks some path counters that are incremented by the generated code. The monitor analyzes the effectiveness of the optimization, and will undo the optimization if necessary.

The following subsections introduce the implementation details of our system.

3.1. PAI Finding

At allocation time OSS knows little about the allocated memory. More information is needed on how it is formed and how it is used. The information comes from the instructions that access the allocated memory, i.e., PAI.

The address operand of a memory-reference instruction has the $base + index * scale + offset$ (or simplified forms like $base + offset$), where $base$ and $index$ are kept in registers and $scale$ and $offset$ are constants. An instruction is a PAI if $index$ (when $scale$ is 1) or $base$ is a pointer to the memory, or is calculated from a pointer. Therefore, OSS needs to know where $index$ or $base$ points to. However, finding all points-to information is difficult in a static compiler. The problem is even more challenging for a dynamic optimizer because register-indirect addressing mode is widely used.

Existing alias analyses on binary [Thomas and Reps 2004] are unsuitable due to their high runtime overhead. Our prototype uses lightweight flow-sensitive propagation on pointers to the allocated memory, starting from the allocation site. OSS will not propagate across function calls unless the pointer has been stored into a global variable. A memory-referencing instruction is a PAI if its memory address is calculated from the pointer. The analysis does not involve non-candidate objects, so the number of PAIs found is usually fewer than those captured by a sophisticated points-to analysis. Those undiscovered PAIs will be captured by our safety mechanism with the overhead of exception handling.

3.2. Field Recognition

OSS needs field information of the records in the allocated memory. The information includes the size of the record type and offset/size of each field. To make this process simple, we assume that a PAI always accesses a fixed field in the record. This assumption may not be always true. OSS thus provides a double-checking mechanism to detect any violation, which is discussed in Section 4.3.

The access addresses of a PAI in a loop may have a stride when traversing an array. As we assumed that a PAI always accesses a fixed field, the stride must be a multiple of the record size. OSS inspects all such loops and makes the greatest common divisor of their strides be the record size. The calculated record size will be checked later with other field information.

Each PAI indicates the size and the offset of the field it accesses. The information from all PAIs is gathered and checked against each other. The more PAIs are found, the more fields can be recognized. Occasionally, a field may have different sizes in different PAIs, or it may overlap with another field. For example, a PAI reads 4 bytes from offset 0 of a record, while another PAI reads 2 bytes also from offset 0. This kind of conflicts is called *field violation* in this paper. It is usually caused by *type cast* or *union* in C and some other languages. When structure splitting separates fields, a field in a cast type or union may be split into multiple pieces that are against what the programmer wants. OSS does not optimize a record if any field violation is detected.

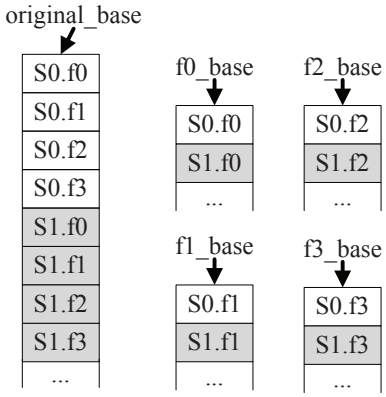

Fig. 2. Example of maximal splitting

The offset and the size of a field can be used to check the previously calculated record size. For example, a field at offset 4 uses 4 bytes, and another field at offset 24 uses 2 bytes. Suppose the previously calculated record size is 20 bytes, the field at offset 24 in fact could be the next array element, also at offset 4. Since OSS cannot determine whether the field is 4 bytes or 2 bytes, it considers this as a field violation. Hence, the array will not be optimized by OSS.

In theory, even if there is no field violation, the calculated record size may still be a multiple of real record size. Our optimization is still correct in this case though because address checking at transformed PAIs (Section 4.3) ensures that the size will work.

3.3. Benefit Estimation

As shown later, our optimization introduces more instructions. Therefore, OSS must estimate the potential benefit of structure splitting and choose only those with a large performance benefit.

Currently, we only implemented *Maximal Splitting* for structure splitting. It requires no complex affinity analysis but can achieve the best, or near the best, performance in most cases [Zhao et al. 2007]. An example of maximal splitting is shown in Figure 2. The same fields from records are placed next to each other in split arrays. When a program region accesses a large number of records, but only references several fields of the records, the other fields may cause severe cache pollution. After maximal splitting, only referenced fields are brought into the cache, so the cache and TLB misses can be reduced. As a contrast, if all of the fields are referenced in a program region, maximal splitting will provide no benefit but some overhead.

A loop may have some PAIs in its loop body, accessing several fields of a record. The space these fields occupy is a portion f ($0 < f \leq 1$) of the record's entire space. We define f as the *field coverage* of the loop. It is a quantitative measure on how fully the record's space is used. For PAIs that are not in any loop, OSS builds traces and also calculates their field coverage.

As discussed earlier, the larger the field coverage is, the less benefit OSS can gain. Therefore, we set a threshold for the average field coverage of all loops/traces with PAI. In our current implementation, if the average field coverage is larger than 0.75 (a value from experiments), OSS does not optimize them.

Hardware prefetchers can also eliminate cache misses, thus leaving less potential for our optimization. However, prefetching does not reduce TLB misses. Experiments in Section 6 show that our optimization is useful for reducing TLB misses and the remaining cache misses.

ALGORITHM 1: Code Generated from PAI

Input: Address old_addr the PAI intends to access.
Output: None
Result: The PAI is executed.

```
1  span = old_addr − ORIGINAL_BASE ;
2  if (span < 0) or (span ≥ ORIGINAL_SIZE) then
3      execute PAI accessing old_addr;
4  else
5      index = span/RECORD_SIZE ;
6      offset = span%RECORD_SIZE ;
7      if offset ≠ F_i_OFFSET then
8          Irregularity_Rollback();
9      else
10         new_addr = F_i_BASE + index ∗ F_i_SIZE ;
11         execute PAI accessing new_addr;
12         counter + +;
13     end
14 end
```

3.4. Exception and Code Generation

OSS allocates a candidate array or a memory pool to a page-aligned space, i.e., the *original space* defined earlier. If the original space is not a multiple of page size, OSS adds some padding to prevent other objects sharing the remaining page. The original space is set non-readable and non-writable. OSS allocates another memory block called *splitting space*, which is prepared for split data. There is no user data in the original space at allocation time, so OSS need not copy anything into splitting space initially.

Since the original space is page protected, any PAI accessing the records in the original space will raise an exception. The exception handler can get runtime information at the PAI, helping the analysis for accurate field information.

Suppose a PAI tries to access old_addr in the original space. The original space begins from $ORIGINAL\_BASE$, so the field offset is:

$$offset = (old\_addr - ORIGINAL\_BASE)\ mod\ RECORD\_SIZE \tag{1}$$

The field F_i and its size F_i_SIZE can be determined from the calculated $offset$. The fields of the records in the original space now form a separate array in the splitting space, starting from F_i_BASE. Therefore, the PAI should now access the new location given by:

$$new\_addr = F_i\_BASE + \lfloor \frac{old\_addr - ORIGINAL\_BASE}{RECORD\_SIZE} \rfloor \times F_i\_SIZE \tag{2}$$

The only variable on the right side of the equation is old_addr; the others are constants at the time of optimization. OSS generates code in the code cache and patches the program, replacing the original PAI (or the block/loop/trace that contains the PAI). Patches are 5-byte long, so occasionally OSS has to modify 2 or more instructions when the original instruction is not long enough. For PAIs in a shared library, the in-memory copy of the shared library is modified and it is no longer shared with other processes.

Algorithm 1 describes how the code works. Line 2 checks whether the PAI really accesses the original space. Line 7 ensures that the PAI accesses the correct field. The checking mechanism will be further discussed in Section 4.3. Line 12 increments the counter of the loop or trace. It will be read by the profiling monitor (Section 3.5).

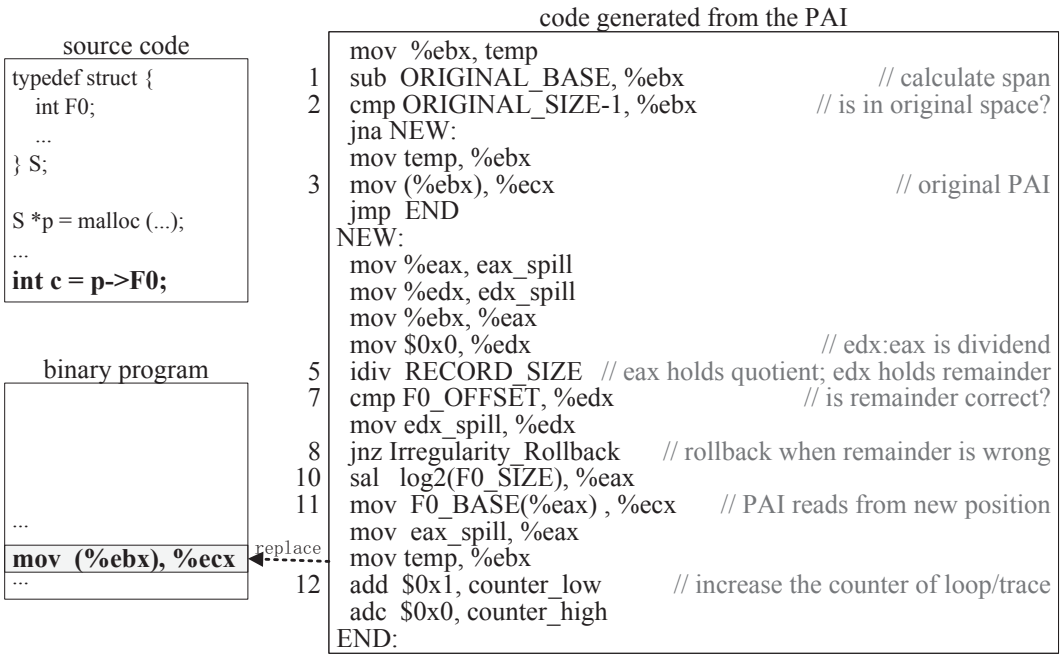

Fig. 3. Example of code generated from PAI

Figure 3 shows a PAI compiled from a given source code and the code fragment OSS generates. The register $\%ebx$ and $\%ecx$ holds the record pointer p and variable c in the source code. 21 instructions are used to replace the original PAI, and the number beside an instruction corresponds to the line number in Algorithm 1. When checking is successful, 18 instructions will be executed to access the split data.

We use several means to reduce the overhead. 1) OSS uses liveness analysis to eliminate unnecessary spill/restore. The instructions executed can be reduced to 12 when $\%eax$, $\%edx$ and $\%ebx$ are dead after the PAI. 2) Several PAIs in the loop/trace can share the address computation and checking code when they use the same pointer. For example, suppose two subsequent PAIs load values from $0x4(\%ebx)$ and $0x8(\%ebx)$, respectively. OSS will generate 21+2 instructions to replace the 3 original PAIs, and execute 18+2 instructions when checking is successful. 3) The *division* operation in the generated code can be optimized to a *shift* operation (*strength reduction*) when record size is a power of two. Some techniques that can further reduce the overhead will be introduced in Section 5.

3.5. Persistent Profiling

After the optimization, OSS keeps profiling the execution counts of the optimized loops and traces. Two instructions in the generated code increase the counter. A monitor in a helper thread periodically reads the counter and checks whether the optimization is effective.

It is difficult to accurately measure the benefit from a cache or TLB at runtime. Therefore, we use heuristics to estimate the benefit from the cache alone. If a loop or trace executes n times in an interval and accesses n records, it will bring at most $n * RECORD\_SIZE$ bytes of data into the cache before optimization. Suppose the field coverage of the loop/trace is f. In the optimal case, structure splitting will bring $n *$

Table I. Conditions when instruction accesses data

	Transformed Instruction	Original Instruction
Optimized Data	Regularity (optimization specific)	Sufficiency (undiscovered PAIs)
Unoptmized Data	Necessity (PAI may access such data)	Good (original behavior)

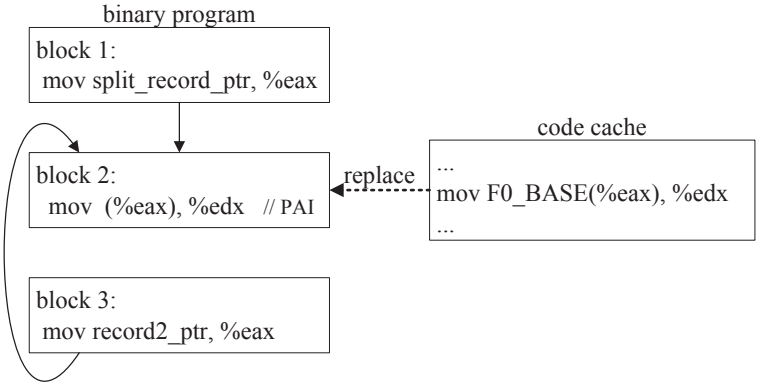

Fig. 4. Scenario of necessity

$RECORD\_SIZE * f$ bytes into cache. Therefore, the cache miss reduction is estimated as $\frac{n*RECORD\_SIZE*(1-f)}{CACHE\_LINE\_SIZE}$.

In the meantime, the loop/trace incurs additional overhead from the extra instructions that calculate and check the new data address. The penalty is estimated by the number and complexity of these instructions. Whether those instructions can be promoted out of the loop or not (Section 5.1) is also considered.

The total penalty cycles in all loops/traces is divided by the total number of reduced cache misses. The ratio shows the tradeoff between the benefit and the overhead of our optimization. If the ratio has been above a threshold for several time intervals, OSS considers the optimization non-beneficial and undoes the optimization.

4. SAFETY GUARANTEE

4.1. Safety Problems

Structure splitting affects both data locations and the instructions that access the data. Table I lists the taxonomy of original/transformed instructions accessing optimized/unoptimized data. There is no problem for original instructions to access unoptimized data, because it is the original behavior of the program. Safety problems may happen in other three cases (*Sufficiency*, *Necessity*, and *Regularity*).

Sufficiency - As mentioned in Section 3.1, OSS cannot guarantee to find *all* PAIs of the optimized data because of the imprecision of the points-to analysis on binary. However, since the original space is page protected, all undetected PAIs will be captured when they execute.

Necessity - Since PAIs are instructions that *may* (not *must*) access the restructured data, they may also access the original data. An example of such a scenario is shown in Figure 4. The record at address $split\_record\_ptr$ was optimized by structure splitting. The PAI in block 2 can access the record when $\%eax$ is defined in block 1. However, when $\%eax$ is defined in block 3 and jumps to block 2, the PAI should now access a non-split record. The generated code in code cache must distinguish between the two cases.

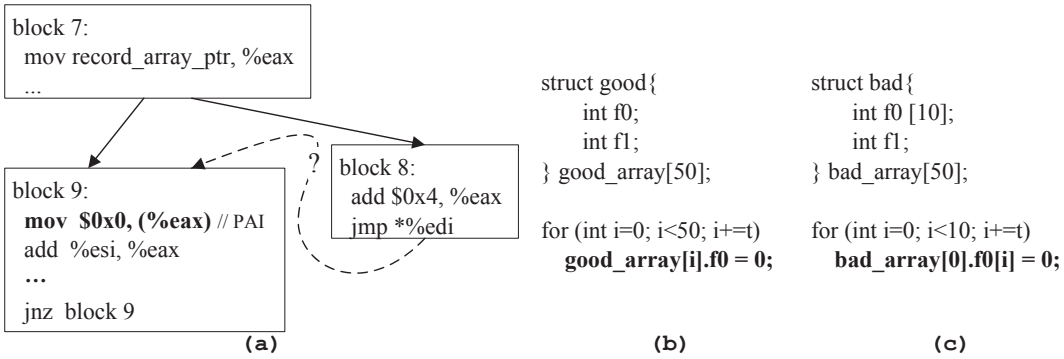

Fig. 5. Example of irregularity

Regularity - In our implementation, we assume that a PAI always accesses the same field in all record instances. Different fields should use different F_i_OFFSET, F_i_SIZE and F_i_BASE in Algorithm 1. However, sometimes this assumption is not valid.

For example, when the pointer of a record is passed to a library function, like *memset()*, *memcpy()* or *fwrite()*, the pointer is type cast to $void*$. These functions will not access the record by its fields.

Another kind of irregularity is shown in Figure 5a. Block 9 can be generated by the loop in Figure 5b. $\%eax$ is the pointer of records in $good\_array$, so the PAI always accesses the first field $f0$. The stride (value in $\%esi$) is $t\times$sizeof(struct $good$) in this case. However, block 9 can also be generated by the source in Figure 5c. The PAI now does not access a fixed field, but an array ($f0[10]$), and the stride is $t\times$ sizeof(int) in this case. OSS cannot transform such PAIs in the current implementation.

Besides, block 8 in Figure 5a changes the value of $\%eax$, and the indirect jump may jump to block 9, so that the PAI in block 9 will access another field. Such irregularity must be checked.

4.2. Page Protection

OSS leverages page protection mechanism to capture all PAIs when they execute. The splitting space cannot reuse the space of the original space, so the space usage is doubled. However, virtual memory is quite large in a 64-bit address space. Even if virtual memory is exhausted in a 32-bit address space, OSS can undo the optimization and reclaim the splitting space.

After raising an exception, an undetected PAI will be captured and fixed, and will not cause any further exceptions. Since the total number of PAIs in an executable is limited, the total amount of overhead in exception handling is also capped. We measured the cost of such exceptions and found it negligible.

4.3. Runtime Address Checking

OSS uses *Runtime Address Checking* to meet the requirements of necessity and regularity. Instructions at the beginning of the generated code check whether the memory address is in the original space or not (line 2 in Algorithm 1). If the address is not in the original space, the unmodified code is executed. The following instructions check whether the PAI is accessing the expected field (line 7 in Algorithm 1). The optimization will be undone if the check fails.

The checking usually succeeds. Hence, the overhead of the two branch instructions is usually very little as their branch prediction is most likely to succeed. The evaluation in Section 6 presents the measured overhead.

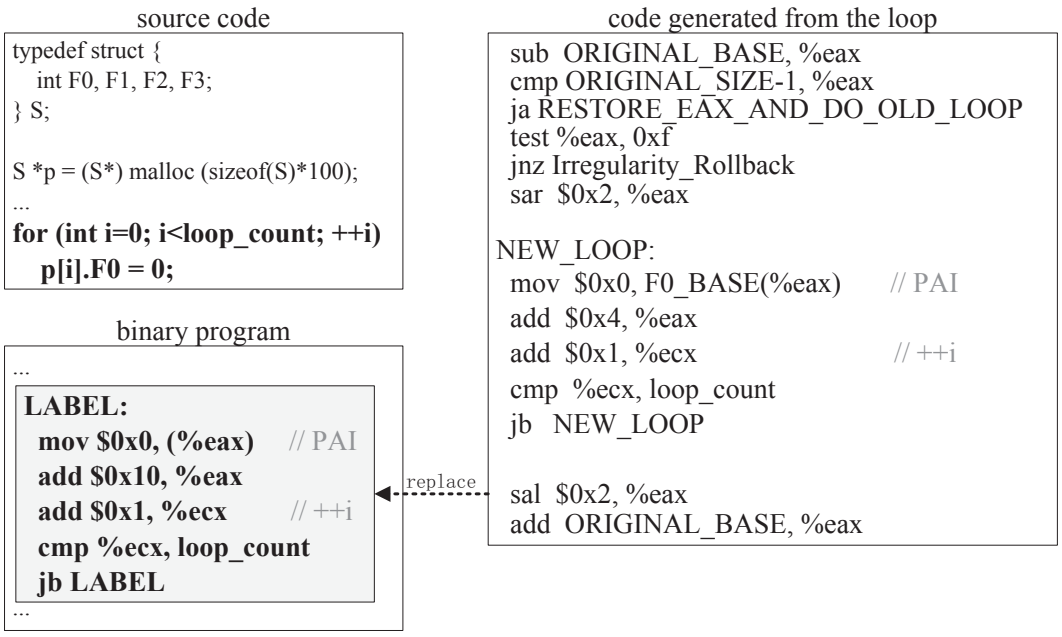

Fig. 6. Example of code promotion

5. OVERHEAD REDUCTION

5.1. Code Promotion

The address computation and the checking code for PAIs can increase code size and incur overhead, especially when PAIs are in loops. However, such code can be promoted from a loop when its accessing stride is a constant. It is a classic compiler optimization (i.e., loop invariant code motion). Figure 6 shows an example loop after optimization. In the example, no extra instruction is added to the loop body.

5.2. Aligned Pool Allocation

Records in the memory pool are placed next to each other as if they are in an array. However, the reference addresses of those records in adjacent loop iterations may not have a constant stride. In this case, the address computation and checking code cannot be promoted.

We implemented an allocation strategy in memory pools, called *Aligned Pool Allocation*. Algorithm 2 describes the initialization of the memory pool. The original memory pool is divided into several $chunks$. The number of chunks depends on the record size. The size of each chunk is pre-defined (must be a power of two). 16MB is our current implementation. The starting address of each chunk must be a multiple of the chunk size. The splitting space is initialized the same way as arrays.

The allocation algorithm is presented in Algorithm 3. We use the example in Figure 7 to explain the algorithm. $pool.curr\_span$ and $pool.curr\_chunk\_id$ are both 0, so the first record $s0$ is allocated at the beginning of the first chunk, and its fields are assigned to the beginning of each array in splitting space. The second record $s1$ is not allocated after $s0$, shown as the dashed blocks in Figure 7a. Instead, it is allocated in the next chunk ($pool.curr\_chunk\_id$) and 4 bytes ($pool.curr\_span$) from the beginning of the chunk, shown as the dark blocks in the original space. Its fields are placed at the position right after $s0$'s fields. We can clearly see that the offset of $s1$ from the begin-

ALGORITHM 2: Initialization of the Memory Pool for Aligned Pool Allocation

Input: The memory pool $pool$, size $record\_size$ and field information $field\_info$ of record type
Output: None
Result: The memory pool is initialized for APA.

```
1  pool.chunk_number = (record_size+3)/4 ;
2  pool.chunk_size = 0x1000000 ;
3  pool.start_addr = memalign(pool.chunk_size, pool.chunk_size*pool.chunk_number) ;
4  for each chunk i in pool do
5      pool.chunk_addr[i] = pool.start_addr + pool.chunk_size * i ;
6  end
7  pool.curr_chunk_id = 0 ;
8  pool.curr_span = 0 ;
9  pool.free_list = NULL ;
10 Splitting_space_init(pool, record_size, field_info) ;
```

ALGORITHM 3: Allocation in Aligned Pool

Input: The memory pool $pool$ from which the record is allocated.
Output: The allocated record's pointer p

```
1  if pool.free_list ≠ NULL then
2      p = Allocate_from_free_list(pool) ;
3  else
4      p = pool.chunk_addr[pool.curr_chunk_id] + pool.curr_span ;
5      pool.curr_span = pool.curr_span + 4 ;
6      pool.curr_chunk_id ++ ;
7      if pool.curr_chunk_id == pool.chunk_number then
8          pool.curr_chunk_id = 0 ;
9      end
10 end
11 return p ;
```

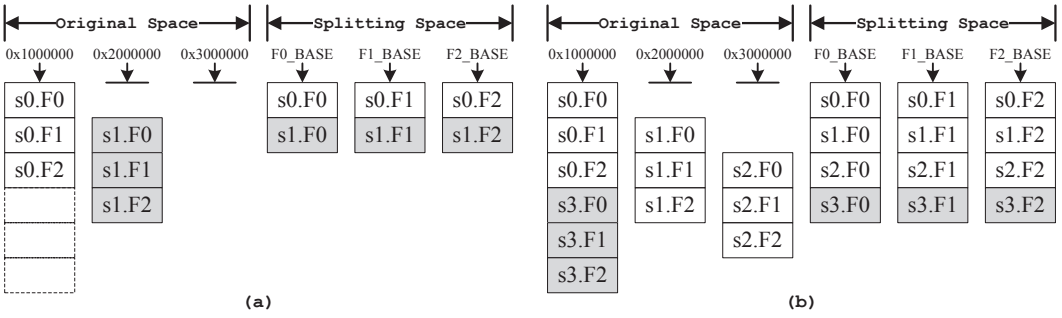

Fig. 7. Example of aligned pool allocation

ning of the chunk is 4, and the offsets of its fields in split arrays are also 4, so they are "*aligned*". Similarly, $s2$ is allocated in the next chunk with the offset of 8; $s3$ rotates back to the first chunk and has an offset of 12, right after $s0$, as Figure 7b shows.

The offset of each record can be calculated easily with a bitwise *AND* operation. The offset is then used to address its fields in the split arrays. Figure 8 shows an example of a PAI, and the generated code using ordinary pool (Figure 8c) and our aligned pool allocation scheme (Figure 8d). The address checking code is not shown in this figure.

```
d = p->F0;
```
(a) source code

```
// %eax is p, %edx is d
mov (%eax), %edx
```
(b) PAI in binary program

```
mov %eax, temp
mov $0x0, %edx
idiv  RECORD_SIZE
shl  log2(F0_SIZE)
mov F0_BASE(%eax), %edx
mov temp, %eax
```
(c) ordinary pool

```
mov %eax, temp
and  CHUNK_SIZE-1, %eax
mov F0_BASE(%eax), %edx
mov  temp, %eax
```
(d) aligned pool

Fig. 8. Example of generated code fragment (no address checking)

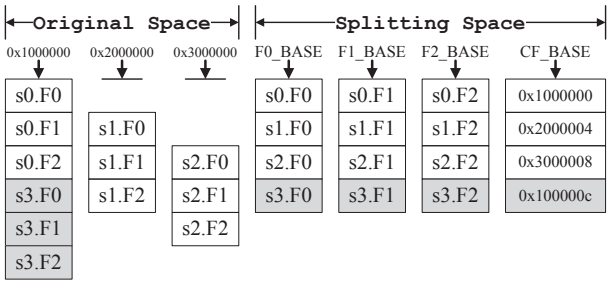

Fig. 9. Example of pointer comparison

The example is an ideal (but not rare) record type because all its fields are 4 bytes. If some field has a different size (e.g., 8 bytes for a double precision floating-point), the offset needs to be shifted before accessing the field. In general, the field address f_i_addr is calculated from the record pointer $record\_ptr$ by:

$$f_i\_addr = F_i\_BASE + \frac{record\_ptr\&(CHUNK\_SIZE - 1)}{4} \times F_i\_SIZE \quad (3)$$

On the right side of the formula, all values except $record\_ptr$ are constants. We convert a time consuming integer division instruction in Figure 8c into a logic *AND* operation in Figure 8d (a so-called *strength reduction* optimization).

5.3. Pointer Comparison

As discussed in Section 4.3, the address checking code checks whether a PAI accesses a fixed field of an optimized record. For records in a memory pool, it is equivalent to check whether the given $record\_ptr$ actually points to the beginning of a record. Using an aligned pool allocation scheme, the checking requires a division operation in most cases (when the number of chunks is not a power of 2). We propose a *Pointer Comparison* mechanism to eliminate the division operation.

An extra field is added to each split record, called *comparison field*. The comparison field contains the original address of the record, shown in the rightmost column in Figure 9. When checking the pointer $record\_ptr$, OSS gets the record's original address from its comparison field. If the address is different from $record\_ptr$, it proves that $record\_ptr$ is not the record's pointer. OSS could then undo the optimization. The comparison can also check whether the record is in the original space. In general, it replaces 1 division and 2 branches with 1 memory read and 1 branch.

6. EVALUATION

In this section, we show the performance of our structure splitting optimization. The experimental setup is in Table II. We use C benchmarks in SPEC CPU 2000 and 2006

Table II. Experimental platform

CPU Family	Intel Northwood
Cores	1
Frequency	2.4GHz
L1I Cache Size	32KB
L1D Cache Size	32KB
L2 Cache Size	512KB
Memory Size	2GB
OS	Linux 2.6.27

Table III. Runtimes of C benchmarks in SPEC 2000 and 2006

benchmark	#work loads	baseline runtime	runtime with OSS	#analysis triggered	analysis time cost	#splitting	#exception	exception time cost
179.art	2	1129s	465s	2	2746ms	2	62	176ms
181.mcf	1	380s	353s	3	1243ms	1	42	140ms
429.mcf	1	1851s	1846s	3	1232ms	1	14	56ms
462.libquantum	1	3104s	1762s	1	1249ms	1	25	89ms
472.moldyn	1	1965s	1972s	1	1272ms	1	21	97ms
176.gcc	5	134s	138s	0	0ms	0	0	0ms
186.crafty	1	135s	134s	0	0ms	0	0	0ms
254.gap	1	140s	140s	0	0ms	0	0	0ms
255.vortex	3	212s	213s	0	0ms	0	0	0ms
300.twolf	1	563s	514s	0	0ms	0	0	0ms
188.ammp	1	588s	579s	0	0ms	0	0	0ms
400.perlbench	3	1277s	1302s	0	0ms	0	0	0ms
456.hmmer	2	1657s	1529s	0	0ms	0	0	0ms
464.h264ref	3	2057s	2060s	0	0ms	0	0	0ms
164.gzip	5	219s	219s	5	1ms	0	0	0ms
177.mesa	1	235s	237s	4	1ms	0	0	0ms
470.lbm	1	2757s	2675s	1	1ms	0	0	0ms
482.sphinx3	1	2590s	2578s	2	2ms	0	0	0ms
401.bzip2	6	2655s	2641s	24	6ms	0	0	0ms
183.equake	1	143s	134s	5	7ms	0	0	0ms
197.parser	1	279s	281s	1	19ms	0	0	0ms
175.vpr	2	334s	331s	3	22ms	0	0	0ms
458.sjeng	1	1715s	1702s	4	25ms	0	0	0ms
256.bzip2	3	279s	280s	18	48ms	0	0	0ms
433.milc	1	1932s	1948s	30	174ms	0	0	0ms
253.perlbmk	7	207s	215s	12	242ms	0	0	0ms
445.gobmk	5	1581s	1582s	20	351ms	0	0	0ms
403.gcc	9	1596s	1598s	13	4043ms	0	0	0ms

generated by GCC 4.3.2 at -O3. All timing results are the median of 3 executions with the reference input set.

Table III lists the results on the benchmarks we used. Our system is based on a simplified pool allocation [Wang et al. 2010] framework. Some benchmarks have performance improvement even if structure splitting is not done. The number of requests that trigger an analysis for structure splitting is shown in the fifth column. The sixth column is time cost. Some benchmarks have multiple workloads, so the number/time is accumulated. Only a few allocation requests triggered the analysis in allocator, because most requests are not large enough to be analyzed as large arrays, and not frequent enough to build a large memory pool. The cost is relatively high in some benchmarks because the allocated memory pointer is saved in a global variable. It enables the finding of PAIs in all functions instead of only within the function that allocates the memory. The seventh column shows how many of these requests triggered the structure splitting optimization after analysis. The optimization could raise exceptions, and the exception handler does accurate local analysis, code generation and so on. The number of exceptions and their time overhead are shown in the last two columns.

Our system finds opportunities to apply structure splitting in four benchmarks, 179.art, 181.mcf, 429.mcf and 472.moldyn. An array in 462.libquantum cannot automatically trigger optimization, because the lightweight pointer analysis in our system does not uncover all PAIs. We checked the source code and found it could be beneficial, so we forced it to be optimized. Figure 10 shows the speedup compared to the baseline runtime (original program without any optimization) on the 5 benchmarks. The basic pool allocation optimization of our system has no obvious impact on the benchmarks except 179.art, as the first bar shows. The second bar is the runtime with on-the-fly structure splitting but without persistent profiling (OSS w/o profiling). The last bar is OSS with profiling. OSS improves 181.mcf, 179.art and 462.libquantum by 7.6%, 142.8% and 76.2%, respectively. 429.mcf and 472.moldyn have little improvement because the persistent profiling module undid the optimization. The rollback happens at 27 seconds after 429.mcf starts and takes about 9.8 seconds to restore the 874MB array. Rollback in 472.moldyn takes place at 22 seconds after program starts, and costs 0.17 second to restore the 62MB array. Records in the two arrays are mostly accessed by their own pointers rather than by the array pointer and strides, so the computation and checking code cannot be promoted. They incur large overhead. The two benchmarks could slow down by 47.6% and 21.3% if the profiling was disabled.

Figure 11 shows L2 cache miss ratios after optimizations compared to the baseline. OSS greatly reduces cache misses, 23% for 181.mcf, 57% for 179.art and 74% for 462.libquantum. 472.moldyn also has fewer cache misses if OSS does not profile and roll back, but the extra overhead finally overwhelms the benefit. 429.mcf has more cache misses when profiling is disabled. It is because the non-sequential memory accesses incur more misses. Although the memory footprint is reduced when accessing only a few fields, it is still too large for cache. The non-sequential accesses make it difficult to reuse cache lines before their eviction.

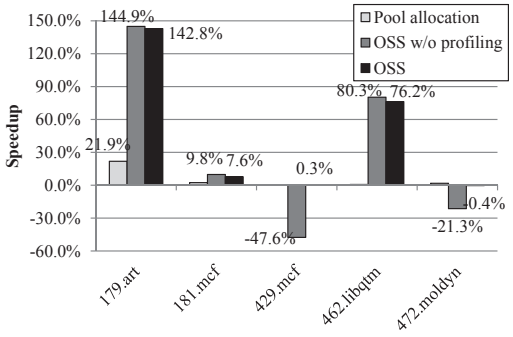

Fig. 10. Performance compared to baseline

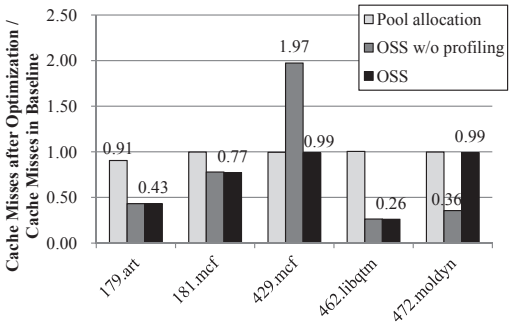

Fig. 11. Cache misses after optimization / cache misses in baseline

Figure 12 shows execution count and field coverage of optimized loops/traces in the benchmarks. We get the data with the profiling module enabled, but do not roll back even if the optimization is not effective. Loops that can promote their address computation and checking code out of the loop body are on the left side ("promoted" on the x-axis), and the other loops and traces are on the right side ("not promoted" on the x-axis). In 179.art and 462.libquantum, the most frequently executed loops belong to the "promoted" type and they have small field coverage values. This explains why they have significant performance improvements. 181.mcf, 429.mcf and 472.moldyn all have higher counts on "not promoted" loops and traces. Although the low field coverage in 472.moldyn reduces many cache misses, it still cannot achieve a good speedup.

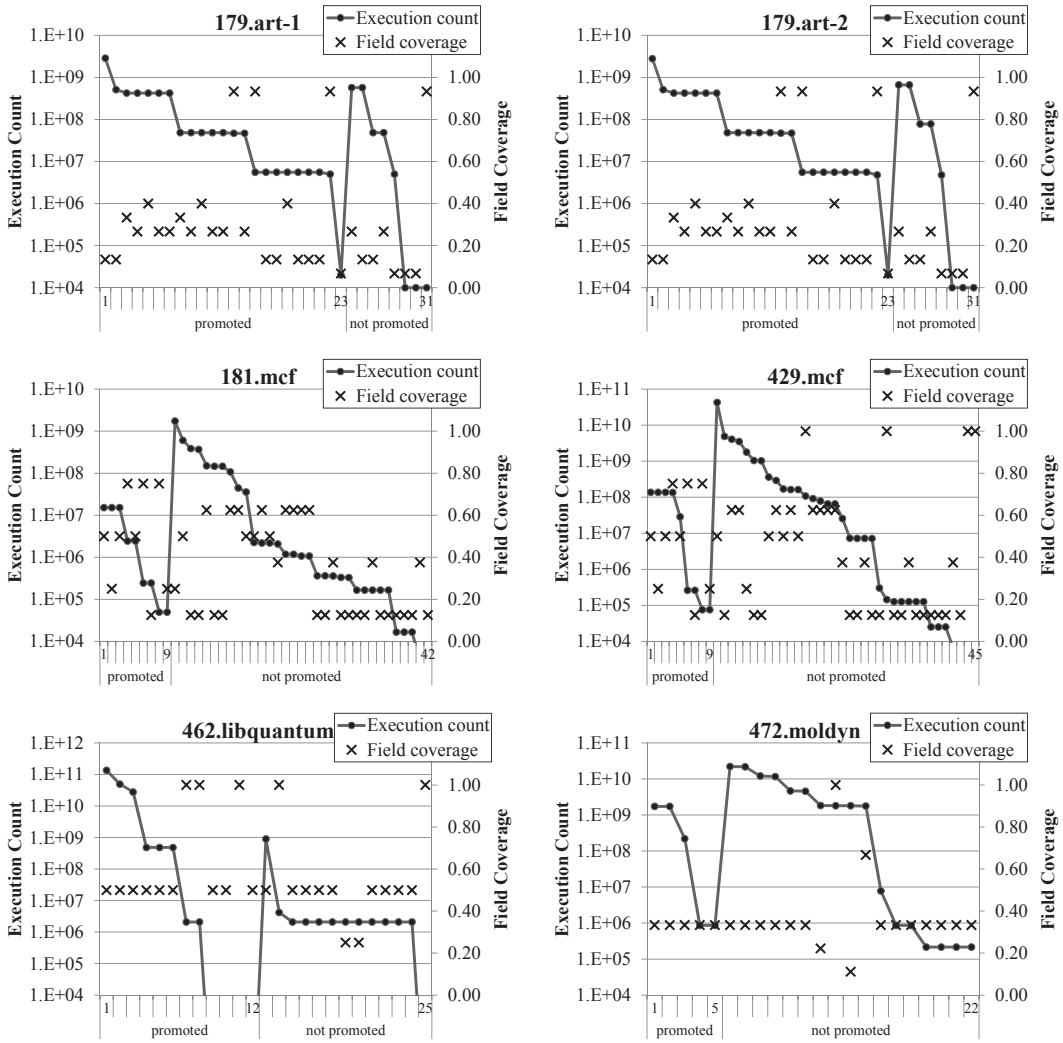

Fig. 12. Execution count and field coverage of optimized loops/traces

Our system can eliminate many cache misses in 179.art, 462.libquantum and 472.moldyn. In the presence of hardware prefetchers, the performance of our system could be affected. We tested the three benchmarks on another machine with configurable hardware prefetchers, and the results are shown in Table IV. The machine has two Intel E5520 CPUs (4-core and 8MB L3 cache) and 8GB memory running Linux 2.6.39. Hardware prefetchers can reduce cache misses significantly (except in 179.art, which has few misses), but cannot reduce TLB misses. Our system eliminates many of the remaining cache misses as well as TLB misses.

Figure 13 shows the effectiveness ratio (introduced in Section 3.5). 179.art and 462.libquantum have a much smaller ratio than the other benchmarks. A larger ratio means more overhead and less benefit. The heuristics will undo the optimization when the ratio rises above the threshold (set at 30) for at least 20 seconds.

Table IV. Runtimes with and without hardware prefetchers

benchmark	hardware prefetcher	optimization	cache misses	TLB misses	runtime
179.art	on	no opt.	<1M	372M	86s
		OSS	18M	56M	79s
	off	no opt.	<1M	374M	86s
		OSS	16M	55M	79s
462.libquantum	on	no opt.	4006M	937M	810s
		OSS	1491M	500M	704s
	off	no opt.	53969M	944M	1697s
		OSS	27104M	500M	1011s
472.moldyn	on	no opt.	3782M	469M	586s
		OSS w/o profiling	2062M	232M	859s
	off	no opt.	13687M	482M	765s
		OSS w/o profiling	8194M	238M	885s

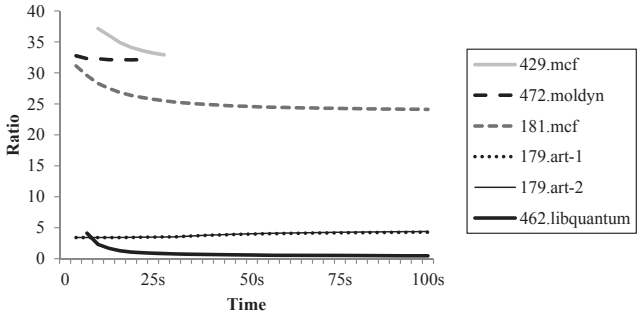

Fig. 13. Effectiveness ratio calculated by monitor

```
struct record {int f0, f1, ⋯ f7;};

record *ptr_array[ELEM_NUM]
for i = 1 to ELEM_NUM
   ptr_array[i] = new record
endfor

for j = 1 to MAX_TIMES
   for i = 1 to ELEM_NUM
      ptr_array[i]->f0 += ⋯
   endfor
endfor
```

Fig. 14. Core loop of micro benchmarks

Our approach uses more virtual memory because the original space and the splitting space are separated. However, the extra memory will not cause more page swaps because the original space is not heavily accessed. The timing overhead is partially from initialization, analysis, and profiling. It is small compared to the program execution time. The address checking also incurs overhead, especially when they cannot be promoted out of the loop. Several techniques have been introduced to reduce the overhead. We tested their effect with some micro-benchmarks. The benchmarks use small data sets so they will cause few cache misses. The core loop of the micro-benchmarks is shown in Figure 14. We used several micro-benchmarks with the field coverage ranging from 1/8, 1/4 and 1/2.

Figure 15 shows the runtime ratios of these micro benchmarks with different computation and checking code styles. The computation code can use division or shift for records whose size is a power of two, as *div+br* and *shift+br* bars show. We can see that code with shift operations has obviously less overhead than that with division operations. We also remove the checking code (unsafe) to evaluate its performance impact (*div* bar and *shift* bar) Their runtimes are not far from those with branches. The reason is that branch instructions always exhibit the same outcome in these benchmarks. Branch prediction can thus have an excellent success rate. The aligned pool allocation and pointer comparison can replace the division with bitwise *AND* operation, thus can reduce the overhead significantly (*and* bar). Although it is not as good as *shift*, it can be applied even if record size is not a power of two. We tested code promotion (the

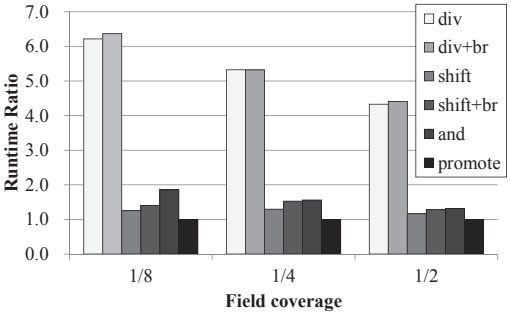

Fig. 15. Runtime overhead for different field coverage

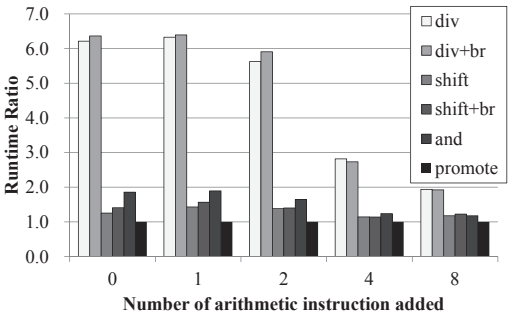

Fig. 16. Runtime overhead after adding more arithmetic instructions

last bar) with similar benchmarks working on arrays. Its runtime is nearly identical to that of the baseline.

Figure 15 also shows that benchmarks with larger field coverages have smaller ratios. The reason is that multiple PAIs in the loop body can share the same computation and checking code, so the overhead is amortized. Besides, the latency of computation and checking code can be partially hidden by other instructions in the loop body. We add some extra arithmetic instructions in the core loop whose field coverage is 1/8, and their runtime ratios are shown in Figure 16. The x-axis shows the number of extra arithmetic instructions in the loop body. The figure also shows the same trend as in Figure 15. Unlike extreme memory-intensive examples in the micro-benchmarks, loops in real-world applications have far more operations, so the overhead in practice will not be as large as shown in the micro-benchmarks. It is still profitable to apply the optimization when benefit can outweigh such overhead.

7. DISCUSSION

Type-based structure splitting techniques [Hundt et al. 2006; Zhao et al. 2007] in static compilers guarantee optimization safety through analyses on structure types, such as *type escape analysis* and *type compatibility analysis*. However, such analyses tend to be conservative because any violation (e.g., a type cast) will prevent all instances of the same type from applying the optimization.

Alias group-wise safety analysis [Lin and Yew 2010] can identify a finer-grained splitting candidate. The violation from one instance will only affect the alias group it belongs to, not the entire type, so more opportunities can be exposed. However, such an analysis relies on accurate pointer analysis, which is difficult for many programs. Static compilers are quite often forced to give up many optimization opportunities when potential aliases exist.

The proposed page protection mechanism can detect potential aliases, so that safety of structure splitting as well as other layout optimizations can be guaranteed. The mechanism can also be used in static compilers for more aggressive optimizations. Static compilers can apply more aggressive optimizations even if it may be aliased to other pointers, and a co-operative runtime system with our page protection mechanism can capture the undetected aliases at runtime.

8. CONCLUSION AND FUTURE WORK

Heap data occupies a large portion of the workload in many applications. Proper structure layout is an effective way to improve data locality. Although some latest compilers have adopted such techniques, there still exist a large number of codes without such optimizations. Most of them are legacy codes and binaries.

This paper presents an optimization system that can apply structure splitting on binary code at runtime. We leverage page protection mechanism and runtime address checking to guarantee the safety of optimizations. It selects promising optimization candidates and forms better splitting layouts in a new memory space. It then redirects all accessing instructions to the new location. We also present some compiler techniques to reduce runtime overhead, including code promotion, aligned pool allocation and pointer comparison. Experimental results show that some benchmarks in SPEC 2000 and 2006 can substantially reduce their cache misses, and achieve a speedup of up to 142.8%.

Our proposed safety guaranty mechanism can also be used to support similar optimizations in static compilers. With such a runtime support, a static compiler can apply optimizations aggressively, and still guarantee PAI's safety (such PAIs no longer need checking code). The runtime system can capture the unexpected PAIs when they occur.

REFERENCES

ABADI, M., HARRIS, T., AND MEHRARA, M. 2009. Transactional memory with strong atomicity using off-the-shelf memory protection hardware. In *PPoPP '09: Proceedings of the 14th ACM SIGPLAN symposium on Principles and practice of parallel programming*. New York, NY, USA, 185–196.

BODÍK, R., GUPTA, R., AND SARKAR, V. 2000. Abcd: eliminating array bounds checks on demand. In *Proceedings of the ACM SIGPLAN 2000 conference on Programming language design and implementation*. PLDI '00. ACM, New York, NY, USA, 321–333.

CALDER, B., KRINTZ, C., JOHN, S., AND AUSTIN, T. 1998. Cache-conscious data placement. In *ASPLOS-VIII: Proceedings of the eighth international conference on Architectural support for programming languages and operating systems*. New York, NY, USA, 139–149.

CHILIMBI, T. M., DAVIDSON, B., AND LARUS, J. R. 1999a. Cache-conscious structure definition. In *PLDI '99: Proceedings of the ACM SIGPLAN 1999 conference on Programming language design and implementation*. New York, NY, USA, 13–24.

CHILIMBI, T. M., HILL, M. D., AND LARUS, J. R. 1999b. Cache-conscious structure layout. In *PLDI '99: Proceedings of the ACM SIGPLAN 1999 conference on Programming language design and implementation*. New York, NY, USA, 1–12.

CURIAL, S., ZHAO, P., AMARAL, J. N., GAO, Y., CUI, S., SILVERA, R., AND ARCHAMBAULT, R. 2008. Mpads: memory-pooling-assisted data splitting. In *ISMM '08: Proceedings of the 7th international symposium on Memory management*. New York, NY, USA, 101–110.

DING, C. AND KENNEDY, K. 1999. Improving cache performance in dynamic applications through data and computation reorganization at run time. In *PLDI '99: Proceedings of the ACM SIGPLAN 1999 conference on Programming language design and implementation*. New York, NY, USA, 229–241.

GUPTA, R. 1990. A fresh look at optimizing array bound checking. In *Proceedings of the ACM SIGPLAN 1990 conference on Programming language design and implementation*. PLDI '90. ACM, New York, NY, USA, 272–282.

GUPTA, R. 1993. Optimizing array bound checks using flow analysis. *ACM Lett. Program. Lang. Syst. 2*, 135–150.

HAGOG, M. AND TICE, C. 2005. Cache aware data layout reorganization optimization in gcc. In *GCC Summit*.

HUANG, X., BLACKBURN, S. M., MCKINLEY, K. S., MOSS, J. E. B., WANG, Z., AND CHENG, P. 2004. The garbage collection advantage: improving program locality. In *OOPSLA '04: Proceedings of the 19th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*. New York, NY, USA, 69–80.

HUNDT, R., MANNARSWAMY, S., AND CHAKRABARTI, D. 2006. Practical structure layout optimization and advice. In *CGO '06: Proceedings of the International Symposium on Code Generation and Optimization*. Washington, DC, USA, 233–244.

KISTLER, T. AND FRANZ, M. 2000. Automated data-member layout of heap objects to improve memory-hierarchy performance. *ACM Trans. Program. Lang. Syst. 22,* 3, 490–505.

LATTNER, C. AND ADVE, V. 2005. Automatic pool allocation: improving performance by controlling data structure layout in the heap. In *PLDI '05: Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*. New York, NY, USA, 129–142.

LIN, J. AND YEW, P.-C. 2010. A compiler framework for general memory layout optimizations targeting structures. In *INTERACT-14: Proceedings of the 2010 Workshop on Interaction between Compilers and Computer Architecture*. New York, NY, USA, 1–8.

RABBAH, R. M. AND PALEM, K. V. 2003. Data remapping for design space optimization of embedded memory systems. *ACM Trans. Embed. Comput. Syst. 2,* 2, 186–218.

RAJAMANI, S., RAMALINGAM, G., RANGANATH, V. P., AND VASWANI, K. 2009. Isolator: dynamically ensuring isolation in comcurrent programs. In *ASPLOS '09: Proceeding of the 14th international conference on Architectural support for programming languages and operating systems*. New York, NY, USA, 181–192.

RUBIN, S., BODÍK, R., AND CHILIMBI, T. 2002. An efficient profile-analysis framework for data-layout optimizations. In *POPL '02: Proceedings of the 29th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. New York, NY, USA, 140–153.

THOMAS, G. B. AND REPS, T. 2004. Analyzing memory accesses in x86 binary executables. In *Compiler Construction*. 5–23.

TRUONG, D. N., BODIN, F., AND SEZNEC, A. 1998. Improving cache behavior of dynamically allocated data structures. In *PACT '98: Proceedings of the 1998 International Conference on Parallel Architectures and Compilation Techniques*. Washington, DC, USA, 322.

WANG, Z., WU, C., AND YEW, P.-C. 2010. On improving heap memory layout by dynamic pool allocation. In *CGO '10: Proceedings of the 8th annual IEEE/ACM international symposium on Code generation and optimization*. New York, NY, USA, 92–100.

ZHAO, P., CUI, S., GAO, Y., SILVERA, R., AND AMARAL, J. N. 2007. Forma: A framework for safe automatic array reshaping. *ACM Trans. Program. Lang. Syst. 30,* 1, 2.

ZHAO, Q., RABBAH, R., AND WONG, W.-F. 2005. Dynamic memory optimization using pool allocation and prefetching. *SIGARCH Comput. Archit. News 33,* 5, 27–32.

Received xxxx; revised xxxx; accepted xxxx