Abstracting Access Patterns of Dynamic Memory Using Regular Expressions

JINSEONG JEON

Agency for Defense Development, Korea

KEONCHEOL SHIN

Korea Advanced Institute of Science
and Technology (KAIST)

and

HWANSOO HAN

Sungkyunkwan University

18

Unless the speed gap between CPU and memory disappears, efficient memory usage remains a decisive factor for performance. To optimize data usage of programs in the presence of the memory hierarchy, we are particularly interested in two compiler techniques: *pool allocation* and *field layout restructuring*. Since foreseeing runtime behaviors of programs at compile time is difficult, most of the previous work relied on profiling. On the contrary, our goal is to develop a fully automatic compiler that statically transforms input codes to use memory efficiently. Noticing that *regular expressions*, which denote repetition explicitly, are sufficient for memory access patterns, we describe how to extract memory access patterns as regular expressions in detail. Based on static patterns presented in regular expressions, we apply pool allocation to repeatedly accessed structures and exploit field layout restructuring according to field affinity relations of chosen structures. To make a scalable framework, we devise and apply new abstraction techniques, which build and interpret access patterns for the whole programs in a bottom-up fashion. We implement our analyses and transformations with the CIL compiler. To verify the effect and scalability of our scheme, we examine 17 benchmarks including 2 SPECINT 2000 benchmarks whose source lines of code are larger than 10,000. Our experiments demonstrate that the static layout transformations for dynamic memory can reduce L1D cache misses by 16% and execution times by 14% on average.

This research was supported by the Ministry of Knowledge Economy, Korea, under the Information Technology Research Center support program supervised by the Institute of Information Technology Advancement (IITA-2008-C1090-0801-0020).

This work was done while J. Jeon was in the department of computer science at KAIST.

This is extended from “Layout Transformations for Heap Objects Using Static Access Patterns” published in *CC ’07: International Conference on Compiler Construction*.

Corresponding author’s address: Hwansoo Han, Department of Computer Engineering, Sungkyunkwan University, 300 Cheoncheon-dong, Jangan-gu, Suwon, Gyeonggi-do 440-746 Republic of Korea; email: hhan@skku.edu.

Permission to make digital or hard copies part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies show this notice on the first page or initial screen of a display along with the full citation. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, to redistribute to lists, or to use any component of this work in other works requires prior specific permission and/or a fee. Permissions may be requested from the Publications Dept., ACM, Inc., 2 Penn Plaza, Suite 701, New York, NY 10121-0701 USA, fax +1 (212) 869-0481, or permissions@acm.org.
© 2009 ACM 1544-3566/2009/03-ART18 \$5.00 DOI 10.1145/1498690.1498693 <http://doi.acm.org/10.1145/1498690.1498693>

ACM Transactions on Architecture and Code Optimization, Vol. 5, No. 4, Article 18, Publication date: March 2009.

Categories and Subject Descriptors: D.3.4 [**Processors**]: Compilers, Memory Management, Optimization

General Terms: Algorithms, Experimentation, Performance

Additional Key Words and Phrases: Access patterns, field affinity, layout transformation, pool allocation, regular expressions

ACM Reference Format:

Jeon, J., Shin, K., and Han, H. 2009. Abstracting access patterns of dynamic memory using regular expressions. *ACM Trans. Architect. Code Optim.* 5, 4, Article 18 (March 2009), 28 pages. DOI = 10.1145/1498690.1498693 <http://doi.acm.org/10.1145/1498690.1498693>

1. INTRODUCTION

The more heavily programs use data, the more important their efficient memory usage becomes. As hardware storage capacity and computing power grow, so does the amount of data used by programs. According to this tendency, the role of compilers or runtime systems for effective memory management becomes important to performance. Researchers have studied many ways so far to enhance the efficiency of memory management, including additional hardware, new architectures, and compiler optimizations. Compiler optimizations are easier to exploit than other methods, since compilers are able to transform program codes to have more memory-friendly behaviors without any hardware support. The only drawback of compiler optimizations is a longer compile time due to static analyses or profile runs. As a result of recent studies to reduce the overhead, compiler techniques are worth enduring additional costs.

Several compiler optimizations for memory management attempt to improve performance by obtaining higher data locality. In order to strengthen data locality, changing the memory layouts is most extensively studied. Memory layouts can be converted by using either explicit code transformations or implicit modifications of memory management routines. Segregating the heap according to the lifetime of objects [Seidl and Zorn 1998] or the pointing shapes of data structures [Lattner and Adve 2005], for instance, is studied. Region-based memory management [Cherem and Rugina 2004], array regrouping [Shen et al. 2005; Zhong et al. 2004], and field layout restructuring [Truong et al. 1998; Chilimbi et al. 1999; Rabbah and Palem 2003; Shin et al. 2006; Hundt et al. 2006] are other advanced techniques explored by researchers. The common essence of these optimizations is to transform memory layouts to be suitable for the behaviors of programs. The methods to extract properties of programs are only different. Some techniques use static (compile-time) methods, such as data structure analysis [Lattner and Adve 2005], region inference [Tofte and Birkedal 1998] and static reference count estimation [Hundt et al. 2006], for the required information. Other techniques, on the other hand, rely on dynamic (runtime) methods, such as profiling.

Our goal is to develop a static compiler optimization that transforms input codes to have more memory-efficient behaviors, especially based on *pool allocation* [Lattner and Adve 2005] and *field layout restructuring* [Shin et al. 2006]. In order to achieve our goal, understanding the memory reference behaviors of programs is necessary. One way to predict behaviors of programs

is dynamic methods, such as profiling. Profiling, however, requires optimizing frameworks to equip and cooperate with a profiler. Selecting representative inputs for profiling is also a burden for ordinary users. The other way to infer behaviors of programs is static methods. It is often difficult to predict runtime behaviors at compile-time. In addition, compile-time predictions frequently bring the trade-off between accuracy and efficiency. Since static methods rarely provide enough accuracy in efficient ways, recent studies on layout transformations use simple static approaches. For these reasons, we propose a simple yet efficient static method to explore the capability of static approach in optimizing memory access behaviors.

There is a practical observation that repetitive small pieces of programs dominate most of memory usage. From this observation, we notice that representing the repetition of programs is important for static optimizations, such as layout transformations. Also, we observe that *regular expressions* are indeed adequate to denote memory access patterns because of the following evidences. The *Kleene Closure*¹ [Hopcroft et al. 2001] in regular expression is intuitively appropriate to represent repetition. Considering other program control structures commonly found in C-like imperative languages, we can abstract consecutive instructions with *concatenation* and conditional branches with *alternation* as well. In this regard, regular expressions are simple yet expressive enough to capture important memory access patterns for locality optimizations. Moreover, interpreting regular expressions is straightforward, thanks to their conciseness.

Once we obtain memory access patterns in the forms of regular expressions, we use these pieces of information to guide heap layout transformations based on pool allocation and field layout restructuring. The goal of our pool allocation technique is similar to the pool allocation by Lattner and Adve [2005] in that both use custom memory allocation routines for certain data structures. The difference is how to select target data structures for pool allocation. They use a pointer analysis to find close relationships among structures, while we regard the structures that are found in closures of a pattern as candidates for pool allocation. Some of previous field layout restructuring techniques [Rabbah and Palem 2003; Shin et al. 2006] used profiling to obtain memory access sequences of programs. On the contrary, we propose a regular expression technique to make the whole optimizing procedure static. An optimizing framework developed by Hundt et al. [2006] includes field layout restructuring. Their approach is similar to ours in that both utilize static analysis based on field affinity relations. The difference is how to catch field affinity relations. They focus on the fields accessed in the same statements and loops, while we analyze the access pattern expressed in a regular expression. Thus, our method can catch the field affinities across the boundaries of loops and procedure calls.

This article makes the following contributions.

- We propose a novel way to represent the memory access patterns of programs with regular expressions. We detail our own automata reduction algorithm

¹This is also called as star or closure; we use the term *closure* for the rest of this article.

Fig. 1. Overview of our framework for static heap layout transformation.

to obtain more concise patterns, and experimentally compare our algorithm with a traditional one.

- We also propose efficient static methods to choose structures for pool allocation and to estimate their field affinity relations by interpreting static patterns in a bottom-up fashion.
- We experimentally evaluate the impact of our static optimizing scheme on performance with the compiler implementation and various benchmarks.

Figure 1 shows the overview of our static framework. First, *access pattern analysis* (Section 3) predicts the memory access pattern of a program. Based on the static pattern, *structure selection analysis* (Section 4) identifies profitable data structures for pool allocation. Next, *field affinity analysis* (Section 5) estimates field affinity relations and finds an adequate field layout for each chosen structure. Finally, *layout transformer* modifies the input code to have the consequent layout.

Section 2 introduces pool allocation and field layout restructuring briefly. Predicting memory access patterns with regular expressions is detailed in Section 3. Selecting structures for pool allocation and estimating field affinity relations are discussed in Section 4 and Section 5, respectively. Section 6 shows our experimental environments and evaluations. Finally, Section 7 contrasts our work with prior studies, and Section 8 concludes this article.

2. HEAP LAYOUT TRANSFORMATIONS

Before we discuss how to extract access patterns at compile-time, this section describes two heap layout transformations. Pool allocation [Lattner and Adve 2005] and field layout restructuring [Rabbah and Palem 2003; Shin et al. 2006] are main transformations, which use our static access patterns. Detailed descriptions on how to use our static access patterns for two transformations will be presented later in Sections 4 and 5.

2.1 Pool Allocation

When objects are individually managed by `malloc` and `free` as shown in Figure 2(b), compilers are not able to predict their accurate addresses. This lack of layout information causes many compiler techniques (e.g., field layout

Fig. 2. (a) Structure types, (b) crude heap layout, (c) heap layout with pool allocation.

restructuring, software prefetching, etc.) to be either less effective or not exploitable [Lattner and Adve 2005]. Pool allocation [Lattner and Adve 2005; Rabbah and Palem 2003; Shin et al. 2006] is an effective technique that provides compilers with layout information and leads to better data locality as well. Figure 2(c) shows a heap layout when objects are allocated in pools.

Collocating closely related objects with one another improves data locality by the effect of prefetching in the cache memory. Besides, pool allocation can improve performance due to a simpler scheme for memory management. The general memory allocation routines in the standard C library consume lots of execution cycles due to complex free-list management. For every allocation, they try to find an available fraction of memory searching the free-list. On the contrary, custom memory management routines for pool allocation are quite simple. A custom allocating routine reserves chunks of memory beforehand and assigns a fraction of the chunks for each object allocation in a simple and uniform way. In the event of memory releases, a custom free routine just restores given fractions to corresponding pools. Thus, pool allocation schemes execute less number of instructions, resulting in the performance increase.

2.2 Field Layout Restructuring

Considering the example code shown, in Figure 3(a), which is an extended example used in Rabbah and Palem [2003], we notice that *key* and *next* fields are referred every loop iteration, whereas *data* field is referred just once when the function *search* finds the node whose key matches with the argument *k*. According to this reference behavior, it is expected that grouping *key* and *next* fields, as shown in Figure 3(c), has an advantage over the original pool layout, as in Figure 3(b), in terms of data locality and performance. The previous field layout restructuring schemes were based on this observation [Rabbah and Palem 2003; Shin et al. 2006]. Because the *key* and *next* fields are frequently accessed in the loop, they are collocated together in a group. The *data* fields are shaped into another group and placed apart from the *key* and *next* group. The drawbacks of field layout restructuring in Figure 3(c) are extra runtime instructions to compute correct field offsets from the base pointers of objects. Although extra instructions are not necessary for the first group, the rest of groups require extra

Fig. 3. (a) Motivating example for field layout restructuring, (b) structure layout with pool allocation, (c) structure layout after field layout restructuring.

runtime calculations. Nevertheless, compiler optimization and pool alignment are able to make the overhead lower.

3. REGULAR EXPRESSIONS FOR ACCESS PATTERNS

The goal of our work is to establish a fully automatic compile-time framework for field layout restructuring with pool allocation. Such framework needs to find structures whose instances are intensively used and the field layouts adequate for chosen structures. Then the framework finally transforms the heap layout into the locality-enhanced layout, as shown in Figure 3(c). In order to design a compile-time scheme that chooses candidates for pool allocation and estimates field affinity relations, we need to obtain memory access patterns from the semantics of programs. Moreover, the memory access patterns should imply both repetitive accesses and field affinity relations. Considering the empirical knowledge that the repetitive small parts of programs dominate most of data usage, we notice that field affinity relations will be heavily affected by frequently executed parts of programs, such as loops. *Regular expressions* are suitable for the abstraction of memory access patterns, since they are naturally able to represent repetitions with *closures*. Besides the repetition (*closure*), regular expressions can represent sequential instructions with *concatenation* and conditional branches with *alternation*.

3.1 Conversion of CFGs into Automata

The access patterns of programs are determined by their control flow information and data access instructions. When we want to obtain access patterns for structures or fields, we can use sequences of referred structure names or field names. The sequences, however, are possibly too long to handle and we need to statically abstract them somehow. The control-flow information obtained from *control-flow graph* (CFG) [Aho et al. 1986] plays a critical role in reducing the sequences. Observing that automata and regular expressions are equivalent, we find a novel method to capture access patterns as regular

Fig. 4. (a) Control-flow graph of Figure 3(a), (b) converted automaton.

expressions. By converting CFGs into automata with access sequences labeled on edges, we can express access patterns through automata. We then exploit an automata reduction technique to summarize access patterns as regular expressions.

Figure 4(a) depicts the CFG of the motivating example and Figure 4(b) depicts the automaton² converted from the CFG. Each instruction is converted to its own start state and end state. An edge is added between the two states and labeled with the access pattern of the instruction. In order to preserve control-flow information, we connect the end state to the start states of its successors, labeling the edges with empty strings. Finally, we link the start state for a function to a corresponding start point, and end points to the accepting state of the function, labeling the edges with empty strings too.

Converted automaton, as shown in Figure 4(b), is similar to *control-flow automaton* (CFA) [Henzinger et al. 2002] in that both label edges with meaningful operations. Our converting procedure, however, is so simple yet intuitive that implementation is a straightforward matter. Moreover, we can guarantee the soundness of the converting process due to the reuse of crude CFGs. In other words, the converted automaton encompasses all the possible behaviors of a function, since it mimics the CFG of the function without loss of control-flow information.

²For convenience, we use an initial letter of each field for the rest of this article.

Fig. 5. (a) DFA converted from ε -NFA, (b) our automata reduction based on WTO.

3.2 Access Pattern Extraction from Automata

Extracting regular expressions from automata is an instance of path problems Tarjan [1981a, 1981b]. The path problem is to find, for a given graph and a certain vertex, a regular expression, which represents all paths in the graph from the source vertex to the target vertex. Automata for functions, start states, and accepting states correspond to graphs, source vertices, and target vertices, respectively. Figure 5(a) shows a conventional way to extract a regular expression from an automaton. We can obtain the first automaton by exploiting the subset construction technique with ε -closures (Chapter 2.5.5 of Hopcroft et al. [2001]). The rest of automata depict all cases of producing the automaton for each accepting state. As shown in the figure, one or more accepting states are prone to make the corresponding regular expression untidy. The more complex the behavior of a function is, the more likely plenty of accepting states appear. In order to get tidier access patterns, we devise a new method. Figure 6 describes our automata reduction algorithm. Section 6.3 will show experimental evaluations for the difference of ours from the traditional method in terms of analysis times and memory usage.

Our algorithm is based on the state elimination technique (Chapter 3.2.2. of Hopcroft et al. [2001]). We only concentrate on the order of state elimination, which is crucial for compilers to extract an understandable pattern from automata. First of all, we remove the states, which have outgoing edges labeled with empty strings and no incoming back-edges. Since these states represent


```

state_compare (state  $s_1$ , state  $s_2$ )
     $b1 \leftarrow \text{whether } \exists s'. (s' \rightarrow s_1, s_1.dfn \leq s'.dfn)$     // true or false
     $b2 \leftarrow \text{whether } \exists s'. (s' \rightarrow s_2, s_2.dfn \leq s'.dfn)$     // true or false
    if  $b1$  and not  $b2$  then
        1    //  $s_1 > s_2$ 
    else if not  $b1$  and  $b2$  then
        -1   //  $s_1 < s_2$ 
    else if  $b1$  and  $b2$  then
         $\text{compare}^1(s_2.dfn, s_1.dfn)$     //  $dfn$  stands for Depth First Numbering
    else    // neither  $b1$  nor  $b2$ 
         $\text{compare}^1(s_1.dfn, s_2.dfn)$ 
    end if

 $worklist \leftarrow \emptyset$     // a sorted list based on state_compare
workhorse (state  $s$ )
    if  $s \neq \text{start state}$  and  $s \neq \text{end state}$  then
        for all  $s' \in s.\text{successor}$  do
            delete  $s'$  from  $worklist$ 
        end for
        eliminate( $s$ )    // elimination may change the order among states
        for all  $s' \in s.\text{successor}$  do
            push  $s'$  into  $worklist$ 
        end for
    end if

reduce ()
     $E \leftarrow \{s \in S \mid \exists s'. s \xrightarrow{e} s'\}$     //  $S$  is a set of states
     $R \leftarrow \{s \in E \mid \nexists s'. (s' \rightarrow s, s.dfn \leq s'.dfn)\}$     // excluding loop heads
    for all  $s \in R$  do
        workhorse( $s$ )
    end for
     $worklist \leftarrow S \setminus R$     // working for remaining states
    while  $worklist \neq \emptyset$  do
        workhorse( $\text{pop}(worklist)$ )    // pop takes the smallest state
    end while
    
```

¹ **compare** is what we usually exploit for integer comparison; it returns -1, 0, or 1 if the first argument is less than, equal to, or greater than the second one, respectively.

Fig. 6. Our automata reduction algorithm.

meaningless instructions or straightforward control-flow information, removing them first helps automata more concise. The remaining steps follow the *weak topological order* (WTO) that combines hierarchical ordering and topological ordering [Bourdoncle 1993]. To make closures correctly enclose loops, we need to postpone the eliminations of the states that have incoming back-edges, since these states are the heads of components (usually the heads of loops). The elimination order among the heads of components follows the *recursive strategy* that is also introduced in Bourdoncle [1993]. Not to prematurely evaluate outer components before the analyses of inner components stabilize, the heads of components should be eliminated from the inner-most one to the outer-most one. The states excluded from the criteria mentioned above are erased in topological order. We can follow this state elimination order using a worklist

algorithm, where the worklist is arranged on the custom comparison routine (**state.compare** in Figure 6). A subtle trouble with a worklist algorithm is that eliminating a state can change the order among the states in the worklist. To compensate these modifications, we have to take what will be changed out of the worklist, and then put them again in the worklist after the elimination. The **workhorse** in Figure 6 fulfills this requirement well.

Figure 5(b) shows the progress of automata reduction according to Figure 6. The second automaton depicts the status after removing all the states that have outgoing edges labeled with empty strings and no incoming back-edges. The third and fourth automata show progressive changes, eliminating the rest of states except for the one that has an incoming back-edge. In the last automaton, the field access pattern of the motivating example is abstracted as $(kn)^*(kd + \varepsilon)$. This pattern implies all the possible behaviors of the function search as follows.

- $(kn)^*kd$: the function successfully finds the specific key.
- $(kn)^*$: the search of the matching key failed, or the first *while* condition check fails due to the null-valued head of the linked-list.

3.3 Extension to Interprocedural Patterns

Because the CFG in Figure 4(a) has just intraprocedural information, the access pattern extracted from the corresponding automaton includes the reference behavior of the function body only. To gain accurate relations over entire executions, access patterns should cover the semantics of the whole programs as well. Thus, function call relations are also important. Unless programs have mutually recursive calls, extending our scheme to interprocedural access patterns is straightforward. Unfortunately, we cannot handle mutual recursion yet. The following description only deals with normal calls and self-recursive calls.

We first utilize one-level flow pointer analysis [Das 2000], including function pointers to handle virtual calls. Then we build a call graph of a program using call relations. In order to obtain interprocedural patterns, we visit functions in reverse topological order of the call graph. When we meet a call site while building an automaton for a function, we label the corresponding edge with the name of the callee. We can guarantee that access patterns for normal call sites are already completed, since we are visiting in reverse topological order of the call graph. For such cases, we just replace function names with access patterns of callees. As for self-recursions, consider the example code in Figure 7(a). The function *f* has a recursive call to itself. Once we calculate its intra-procedural access pattern, we will have the automaton shown in Figure 7(b) that has the function name on the edge representing the call site. Obviously, we do not have the access patterns for the function. For such recursive call sites, we connect the state before a call site to the start state of a function and connect the accepting state of the function to the state after the call site. Then we eliminate the edge representing the call site. The consequent automaton is shown in Figure 7(c).

Although the method described above is able to resolve self-recursions well, obtained patterns through that solution are not perfect. Let the access pattern of

Fig. 7. (a) Example code for self-recursive function, (b) automaton after intraprocedural pattern analysis, (c) automaton after interprocedural pattern analysis.

the function in Figure 7(a) be F . The precise access pattern, F can be described with the following grammar.

$$F \rightarrow ab \mid aFb$$

This grammar is summarized as $a^i b^i$ ($i > 0$). This pattern is one of the typical examples that cannot be expressed by regular expressions. In other words, an exact way to represent the access patterns for recursive cases requires *context-free grammar*. Nevertheless, regular expressions have enough evidences to understand the reference behaviors of programs. For example, the automaton in Figure 7(c) implies the regular expression a^*abb^* . We can, however, infer a very helpful knowledge that a and b are accessed frequently but separately. We only lose the information that a and b are accessed at the same number of times as $a^i b^i$ can imply. This may not be an important fact for our optimization.

4. STRUCTURE SELECTION FOR POOL ALLOCATION

This section explains how we identify beneficial structures for pool allocation. In the following subsections, we introduce an earlier study and our intuitive *top-down* approach to interpreting access patterns. We then describe our actual *bottom-up* implementation for scalability.

4.1 Structure Detection in Closures

Lattner and Adve [2005] proposed a structure selecting algorithm for their automatic pool allocation framework. They find data structures whose instances have distinct behaviors and then segregate the instances into separate memory pools. According to their experimental results, most pools are used in a type-consistent style [Lattner and Adve 2005]. From this observation, our pool allocation uses a “one structure per pool” policy. We simply focus on how to choose structures that are intensively used in programs. Those structures are easily identified by investigating regular expressions for structure access patterns. The structures in closures of regular expressions are what we want to identify as intensively used ones.

To obtain structure access patterns, structure names are used to represent the semantics of instructions. In other words, we just need to label edges of an

Fig. 8. (a) Call graph, (b) building progress of the structure access pattern, (c) entries added to a PCmap type map.

automaton with accessed structure names, while making the automaton for a function. Figure 8(a) is part of an example code, and Figure 8(b) shows a building progress of the structure access pattern for the code. The pattern is built from bar2 to main. Since the structure s_3 is the only structure that appears within a closure of the final pattern, it becomes a candidate for pool allocation. Lastly, we accept the only structures that are frequently allocated with dynamic memory allocation routines. We can obtain allocation patterns for candidate structures by labeling automata with their allocation sites. As we did in structure selection, we regard the structures within closures as frequently allocated ones. There is an exceptional case in our structure selection algorithm for pool allocation. If a program already allocates a pool memory for a structure type using `calloc`, our structure selection algorithm cannot select it as a target structure for pool allocation, which means that it is not automatically a target for field layout restructuring. So, we extend our structure selection algorithm to select pool-allocated structures by interpreting `calloc` as `malloc` in a closure. Then, only a field layout restructuring technique is applied to those structures.

4.2 Promotion of Candidate Structures

When we try to find target structures based on a top-down manner, we have to keep searching closures in case of structures within deeper closures. The size of patterns matters under this method, since we may exhaustively revisit what we have built. In order to make our scheme scalable, we decide to interpret access patterns in a bottom-up manner; we first interpret every intraprocedural pattern and propagate it up the call graph to make an entire access pattern.

In this analysis, we are concerned about which structures are heavily used. Thus, for each function, we record which structures occur within closures instead of the pattern. We define a map (PCmap) to save the abstracted information

as follows.

$\text{PCmap} := \text{VAR} \mapsto 2^{\mathbb{S}} \times 2^{\mathbb{S}}$ (promoted set, candidate set)
 (where VAR and $\mathbb{S}$ are the set of function and structure names,
 respectively)

In addition to structures within closures (promoted set), we record normally accessed structures (candidate set) as well. This is because a closure of an empty string and that of a nonempty string sharply differ in that the former remains as it was, but the latter becomes a true closure. In other words, normally accessed structures are possibly regarded as being within closures by some callers, if the callers repeatedly invoke the function to which they belong.

Once we obtain the structure access pattern for a function, we examine which structures are accessed normally or within closures. If some structures are within closures, we add them to the promoted set of the function. If some structures are accessed but not within closures, we add them to the candidate set of the function. When we meet a call site while building the automaton for the function, we just label the corresponding edge with the name of the callee. Then, at the interpretation stage, we use abstracted information of the callee. If the call site is within a closure, we add the elements in both the promoted set and the candidate set of the callee to the promoted set of the caller. Otherwise, we just move the elements from the sets of the callee to the corresponding sets of the caller.

Figure 8(a) depicts a call graph, and Figure 8(c) shows entries added to a PCMap type map. Functions are still analyzed in reverse topological order of the call graph. The function `bar2` accesses the structure s_3 normally. A promotion only happens to the function `bar1`. Since it calls `bar2` in the `while` loop, we judge that the structure s_3 is repeatedly accessed. As a result, the structure s_3 is promoted. The remaining function invocations are simple. Finally, we conclude that only the structure s_3 is heavily used and suitable for pool allocation.

5. FIELD AFFINITY ESTIMATION

Once we choose target structures for pool allocation, we need to determine appropriate field layouts of candidates. This section explains how we estimate affinity relations among the fields of chosen structures. In a similar way, we introduce a profile-based method and our symbolic *top-down* approach. We then describe our actual *bottom-up* implementation for scalability.

5.1 Symbolic Representation

One way to analyze field affinity relations is counting co-occurrences within a window sliding over a field access sequence. The counted number is called neighbor affinity probability (NAP) [Rabbah and Palem 2003]. Figure 9 depicts the progress of profile-based field affinity estimation. Temporal relationship graph (TRG) [Gloy and Smith 1999] is a weighted graph where its nodes denote fields and the weights of its edges represent NAPs between fields. Since one field can be accessed consecutively, we extend TRG to have self-edges and name it STRG. Figure 9(a) shows the concept of NAP calculation using a sliding

Fig. 9. (a) Profiled memory access sequence and a sliding window to calculate NAP, (b) initial STRG, (c) STRG after grouping *key* and *next* fields.

Fig. 10. (a) STRG with closure signs, (b) STRG with a closure variable.

window over a profiled field access sequence. An initial STRG after profiling the motivating example is shown in Figure 9(b). Since the NAP between *key* and *next* fields is larger than the sum of their own self-affinities, we choose two fields as a group. After grouping, the resulting STRG is shown in Figure 9(c). The edges are merged and the weights are modified to encompass the previous relationships. Until the STRG is not changed, we repeat the procedure that finds a beneficial grouping and merges fields. Each node in the final STRG becomes a group in a field layout reconstruction scheme. The groups in final STRG are placed in decreasing order of the weights of self-edges. In Figure 9(c), we cannot find a profitable grouping any more. As a result, $\{key, next\}$ and $\{data\}$ are placed in the heap as shown in Figure 3(c).

Statically obtained field access patterns imply abstract relationships between fields, but not presented with numerical values. To overcome the gap between realistic values and abstract relationships, we devise a symbolic method. Instead of NAP, we label edges of STRG with closure signs to indicate how often two fields are accessed together. Consider the example in Figure 10, assuming

a program that performs list generation, parity check, and random search in turn. The regular expression that represents the field access pattern of the program is shown in the top of the figure. Based on the pattern, we construct symbolic STRGs, as in Figure 10(a), where the weights of edges are denoted with closure signs.

Note that the access patterns, which reside within doubly nested closures, are denoted with double closure signs to distinguish nested levels. For example, imagine that the search function is invoked repeatedly. The pattern for this case is $((kn)^*(kd + \epsilon))^*$. We get this by enclosing the pattern of the function with an outer closure. From that pattern, we label the edge between *key* and *next* fields with one double closure and another single closure. The former represents the presence of two fields in the inner-most closure. The latter represents that the *next* field appears at the end of the inner-most closure and meets the *key* field at the very following access. In a similar way, we label the edge between *key* and *data* fields with one single closure.

If more than two fields are concatenated within a closure (e.g., $(kdn)^*$), we label with closure signs all the edges of all possible combinations of two consecutive fields within the closure (as if we see $(kd)^*(dn)^*(nk)^*$). After building symbolic STRGs, we regard all closure signs as the same variable, as shown in Figure 10(b). Since it is next to impossible to predict the number of loop iteration (function invocation) at compile-time, we assume loops (functions) are iterated (invoked) at the same number of times. Finally, we evaluate the affine equations by assigning the fairly large value³ (100) to the closure variable. The rest of estimating procedure is the same as profile-based estimation depicted in Figure 9.

5.2 Generation of Affinity Relation

As the structure selection analysis suffered, the size of patterns likewise matters to this field affinity analysis. We similarly change the frame of the analysis. In this analysis, we need to abstract field affinity relations within closures; we record which fields are appeared together within closures in terms of the depth of closures ($\mathbb{D} := \mathbb{N}$) and the number of occurrences ($\mathbb{C} := \mathbb{N}$). We also record first and last accessed fields for each function to link them with the context of callers. The types for this scheme are defined as follows.

$$\begin{aligned} \text{AR} &:= \mathbb{F} \times \mathbb{F} \mapsto 2^{\mathbb{D} \times \mathbb{C}} && (\text{where } \mathbb{F} \text{ is the set of fields}) \\ \text{ARmap} &:= \mathbb{VAR} \mapsto 2^{\mathbb{F}} \times 2^{\mathbb{F}} \times \text{AR} && (\text{where } \mathbb{VAR} \text{ is the set of function names}) \end{aligned}$$

Affinity relations (AR) type is a map from a pair of fields to a set of two natural numbers. The set representation allows us to log whole counts per depth of closures. Since the number of first and last accessed fields can be greater than one due to conditional branches, we record them as a set too. For each function, these data are summarized pairwise and recorded in reverse topological order of a call graph.

³Assigning large values (larger than 10) to the closure variable makes the same field layouts in our evaluations.

Fig. 11. Call graph and entries added to an ARmap type map.

Once we obtain the field access pattern for a function, we examine the pattern to estimate field affinity relations for the function. During the estimation, we keep recently accessed fields. Looking around the pattern in sequence, we repeat the procedure of recording the relations between recently accessed fields and newly accessed fields and updating the recent ones as the new ones until the end of the pattern. Relations recorded here are composed of the depth of the closure where two fields are and the accumulated number of their occurrences. We especially treat sequential accesses as a depth 0 closure for the compatibility of depth. If a function that has sequential accesses is invoked in a loop, the sequential accesses should be regarded as being in a closure. For the case of such promotion, we need to record them as a depth 0 closure. Besides, first accessed fields are necessary to keep relations of sequential accesses, since fields accessed before a call site are related to the fields first accessed in a callee. For the same reason, last accessed fields in the callee are related to the fields accessed after the call site. When we meet a call site while building the automaton for the function, we just label the corresponding edge with the name of the callee. Then, at the interpretation stage, we use the affinity relations of the callee. If the call site is within a closure, we copy them after increasing depth information. Otherwise, we just bring them in the raw.

Figure 11 depicts a call graph and entries added to a ARmap type map. For convenience, the figure shows components of the entries separately. The function bar2 accesses the field f₁ and f₂ in sequence. First and last accessed field are, thus, f₁ and f₂. They are also recorded as having a relation in a depth 0 closure. A promotion only happens to the function bar1. It calls bar2 in the while loop and accesses the field f₃ before the loop and f₁ after the loop. Imagining that we embed the body of bar2 in the call site, we notice following two facts. First, the field f₁ and f₂ are frequently accessed within the loop. In consequence, all the depth information of bar2 is promoted so that the field f₁ and f₂ are regarded as having a relation in a depth 1 closure. Second, the fields accessed before the loop are related to the fields first accessed in the loop and the fields accessed after the loop are related to the fields last accessed in the loop. In the case of bar1, f₃

and f_1 (f_1 and f_2) corresponds to this observation. The new relation between f_1 and f_3 , which are sequentially accessed at the start of the loop, is added to bar_1 . After adding the information that the field f_1 and f_2 are sequentially accessed at the end of the loop, their relations increase to two, one for a depth 1 closure, the other for a depth 0 closure. The remaining function invocations are simple. We just bring the raw information of the callees and add local information to them. Finally, we conclude that f_1 and f_2 are accessed in a depth 1 closure once.

6. EXPERIMENTAL EVALUATION

6.1 Implementation

Based on the CIL framework [Necula et al. 2002], we implement our framework: access pattern analysis, structure selection analysis, field affinity analysis, and layout transformation. Type-safety assumption is actually necessary to guarantee the correctness of our technique. We use C programs for experiment and C language is not a strongly type language. Thus, only the programs which use structures in a type-safe fashion can be safely transformed using our technique. Under this assumption, we only transform explicit field names into field references on modified field layouts. For some field references, we add extra instructions to calculate exact field offsets as described in Shin et al. [2006]. Unlike field references, structure references using pointer arithmetic, such as $p[i]$ or $p + i$, are correctly transformed into $(\text{int})p + i \times (\text{size of first group})$. Memory management routines such as `malloc` and `free` calls are transformed into custom memory management routines using pool allocation [Lattner and Adve 2005; Shin et al. 2006]. We do not transform `calloc` calls but reuse them when our field affinity analysis finds better field layouts of the corresponding pool-managed structures.

One limitation in our current framework is we cannot handle mutual recursions correctly yet. In the later implementation, we will extend our framework to include mutual recursions too. Another limitation in our framework is that it cannot recognize custom memory management routines already used by original programs. In addition, it handles the only structures that are allocated in a type-aware fashion. If our framework cannot recognize dynamic allocations for certain structures due to lack of type information, the structures will be discarded by the structure selection analysis. *Health* in the Olden suite [Rogers et al. 1995] has its own allocators, which lose type information and cause both the structure selection analysis and the transformer not to identify beneficial structures. For such case, we feed the structure selection analysis with user-given hints, which consist of target structures and corresponding custom allocators. The CIL is extended to accept user-given hints for our experiments.

6.2 Experimental Environment

Our evaluations are performed on a Redhat 9.0 Linux PC equipped with a 2.6-GHz Pentium 4 processor. This machine contains 8 KB L1D cache (64-byte

Table I. Comparison of Analysis and Transformation Time to Original GCC Compilation Time (sec)

Benchmark	Program	SLOC	Analysis and Transformation Time				Original GCC Time
			Structure Selection (1)	Field Affinity (2)	Code Transform (3)	Total (1)+(2)+(3)	
SPECINT 2000	175.vpr	11,301	0.271	0.324	0.107	0.702	3.708
	181.mcf	1494	0.016	0.013	0.007	0.036	0.404
	300.twolf	17,821	0.658	3.455	1.126	5.239	6.560
FreeBench	analyzer	763	0.013	0.027	0.012	0.052	0.376
McGill	chomp	378	0.006	0.006	0.003	0.015	0.212
	misr	181	0.003	0.002	0.001	0.006	0.172
Olden	bisort	597	0.003	0.003	0.002	0.008	0.178
	health	474	0.003	0.012	0.010	0.025	0.202
	mst	408	0.004	0.004	0.002	0.010	0.195
	perimeter	345	0.005	0.012	0.001	0.018	0.197
	treeadd	154	0.001	0.000	0.000	0.001	0.120
	tsp	433	0.005	0.004	0.002	0.011	0.201
	voronoi	975	0.007	0.004	0.003	0.014	0.295
Ptrdist	anagram	355	0.005	0.003	0.001	0.009	0.229
	bc	4,303	0.116	0.634	0.193	0.943	1.208
	ft	926	0.008	0.014	0.010	0.032	0.298
	ks	551	0.008	0.012	0.020	0.040	0.237

cache line, 4-way set associative), 512 KB L2 cache (64-byte cache line, 8-way set associative), and 1.7-GB main memory. All the benchmarks are compiled with GCC 3.2.2 at -O3 optimization level. We use the Cachegrind from the Valgrind's Tool suite (ver. 3.1.0) [cac] to simulate cache behaviors and to measure cache misses using the same cache configuration as the machine on which we evaluate execution times. We measure the number of cache misses at both levels of cache to estimate locality improvements in the cache memory hierarchy. We measure execution times to evaluate the effect of layout transformations on performance by using the UNIX time command. All the reported execution times are the minimum elapsed time out of 10 runs. To confirm the influence of our static mechanism, we examine programs with two different size inputs.

Some of the SPECINT 2000 benchmarks, the FreeBench suite [Rundberg and Warg], the McGill benchmark suite [mcg], the Olden suite, and the Ptrdist suite [Austin et al. 1994] are used in our evaluations. Some benchmarks in those suites do not use dynamic structures at all and some are not compiled with the CIL framework. Those benchmarks are excluded from our experiments. Especially, *164.zip* uses primitive type pools so that it results in the same heap layouts after our analysis. There is another special case. *181.mcf* accesses some structures as if they are array elements. As mentioned in Section 4.1, our framework catches the pool-managed structures and applies a field layout restructuring technique to them. In this regard, the spaces for *Pool* of *181.mcf* in experimental results are equal to the baseline. Table I shows source lines of code (SLOC) [Wheeler] and analysis and transformation times for each program. Additional times incurred by our analyses and conversion are fairly small for most programs and reasonably tolerable even for large programs.

Table II. Traditional Algorithm vs. WTO Based Algorithm

Program	SLOC	ε -NFA $\rightarrow$ DFA $\rightarrow$ R.E.			ε -NFA $\rightarrow$ R.E. (WTO)		
		Time (sec)	Peak (Mb)	Total (Mb)	Time (sec)	Peak (Mb)	Total (Mb)
175.vpr	11,301	N.A.	N.A.	N.A.	0.154	15.97	178.68
181.mcf	1494	0.053	1.72	18.75	0.040	1.47	15.07
300.twolf	17,821	N.A.	N.A.	N.A.	0.360	27.03	313.14
analyzer	763	0.022	1.47	16.59	0.007	1.23	11.97
chomp	378	0.003	0.74	5.01	0.003	0.74	4.96
misr	181	0.003	0.49	2.87	0.003	0.49	2.69
bisort	597	0.002	0.74	4.79	0.002	0.74	4.66
health	474	0.004	0.74	5.90	0.002	0.74	5.47
mst	408	0.003	0.74	5.92	0.002	0.74	5.51
perimeter	345	0.003	0.74	4.52	0.002	0.74	4.19
treeadd	154	0.000	0.49	1.52	0.000	0.49	1.51
tsp	433	0.004	0.74	5.31	0.002	0.74	4.94
voronoi	975	0.005	1.72	14.64	0.003	1.72	14.28
anagram	355	0.002	0.74	5.33	0.002	0.74	5.32
bc	4,303	572.897	612.93	4,379.97	0.059	9.34	114.09
ft	926	0.006	0.98	9.07	0.004	0.98	8.67
ks	551	0.008	0.98	9.37	0.004	0.98	7.92

6.3 Efficiency of Automata Reduction Algorithm

In order to estimate the effect of our automata reduction algorithm, we implement a traditional one and compare our algorithm with it. We fairly apply the same order of state elimination based on WTO to both algorithms. The only difference is empty string manipulation; our algorithm treats all states without discrimination, even the states that have outgoing edges labeled with empty strings. On the other hand, the traditional algorithm eliminates those states by using a subset construction technique with ε -closures.

Table II shows the estimated results in terms of analysis time, memory peak, and total memory usage. This table especially deals with the results for intraprocedural patterns, which are heavily used by our framework. The results for interprocedural patterns, nevertheless, rarely differ from the intraprocedural ones due to the memory reuse of OCaml, the programming language we use. For most benchmarks, except for complex cases (*175.vpr*, *300.twolf*, and *bc*), two algorithms look as if they have the same effect. However, for the extreme three cases, the gap between them appears obviously. We cannot even get results for two SPECINT 2000 benchmarks in 10 hours. From this experience, we believe that our automata reduction algorithm is suitable for reducing the overheads of both time and memory.

6.4 Improvements in Cache Locality

Figure 12 shows normalized cache misses in L1D and L2. The numbers are averages of two different size inputs except *175.vpr*. The result for *175.vpr* reflects a small case only, because Cachegrind did not work at a large input. *Pool* and *Pool + Re* denote the effect of pool allocation alone and field layout restructuring with pool allocation, respectively.

Fig. 12. Normalized numbers of misses in (a) L1D and (b) L2 caches.

In our evaluations using data intensive benchmarks, pool allocation is significantly effective. Compared with original programs, the miss reductions of pool allocation are at most 62% for L1D and 71% for L2. These miss reductions are due to better locality by gathering the instances of certain structures in the same pools. There are some noticeable cases (*300.twolf*, *misr*, *bc*, and *ks*) where their miss reductions are negative in some cache levels. Since they use connected structures simultaneously, assigning each structure to an individual

Fig. 13. Dynamic instruction counts.

pool makes data locality worse. However, the absolute values of their original cache misses are very small particularly in *misr* and *bc*. In other words, the performance loss due to the miss increases are very marginal, though their cache behaviors look relatively inefficient.

Under pool allocation, field layout restructuring can be an auxiliary method to reduce cache misses more. Compared with original programs, the miss reductions of field layout restructuring are at most 73% for L1D and 80% for L2. In six cases (*181.mcf*, *analyzer*, *chomp*, *mst*, *perimeter*, and *tsp*), field layout restructuring is beneficial to miss reductions in both cache levels over pool allocation alone. For the particular case (*bisort*), the field affinity analysis determined entire structures as a group, that is, the very same layouts of pool allocation resulting in the same cache misses. For some cases (*treeadd* and *ks*), the field affinity analysis choose inefficient layouts, which make more cache misses. But, the miss increases are very marginal. In the rest of the benchmarks, cache misses in either L1D or L2 are reduced more than pool allocation alone.

For some cases where cache misses in either L1D or L2 increase, the miss increases in one cache are usually cancelled out by the reductions in the other level cache. For *health* and *ft*, miss reductions in one cache are influential enough to eliminate the effect of increased cache misses in the other level. For *300.twolf* and *voronoi*, however, the miss increases in one cache are not cancelled out due to relatively small improvements of the other level cache.

6.5 Improvements in Performance

Improvements in performance are due to not only the reductions of cache misses, but also the reductions in the number of instructions executed in custom memory management routines using pools. As shown in Figure 13, dynamic instruction counts of pool allocation alone and field layout restructuring with

pool allocation are reduced by 6% and 1% on average, respectively. These results are due to the replacement of many `malloc` and `free` calls by the simpler routines for pool management. In many benchmarks, field layout restructuring increases dynamic instruction counts due to the overheads caused by runtime field address calculations.

In order to analyze how much proportion of performance improvement are affected by instruction counts and cache behaviors, we estimate execution time based on the number of dynamic instructions and cache misses. Figure 14(a) shows the estimated relative ratios of the time spent in instruction execution, L1D cache misses, and L2 cache misses for all benchmarks. Average estimation of two different size inputs is used and instruction per cycle (IPC) is assumed as 1, so we can consider dynamic instruction counts as computation cycles. Runtime cycles spent in cache misses are calculated by multiplication of the number of cache misses and cache miss latency. We assume L1 cache miss latency and L2 cache miss latency as 17 cycles and 160 cycles, respectively. There are three bars for each benchmark at Figure 14(a). First bar shows a ratio of the original layout and its runtime is normalized. Second and third bar mean the ratios under pool allocation alone and field layout restructuring with pool allocation, respectively. This graph nicely shows how heap layouts have an impact on the ratio of instructions and cache behavior.

To justify our estimation is reliable, normalized graph of execution times based on Table III are given in Figure 14(b). For some programs, our estimation shows a little gap with actual execution time, but we can see that our estimation has very similar flow to the actual execution time.

Table III shows execution times for each transformation and ratios compared to original executions. The column labeled with *Original* provides the base results from original programs. *Pool* and *Pool + Re* columns show the impact of pool allocation alone and field layout restructuring with pool allocation, respectively. The results of layout transformations are normalized with original programs.

As a result of locality enhancement and dynamic instruction count reduction, the performance of programs also improves. Compared with original programs, execution times of pool allocation and field layout restructuring with pool allocation are reduced at 13.4% and 13.5% on average, respectively.

As shown in the *Pool + Re* column, the performance of transformed programs with field layout restructuring improves less than the corresponding cache performance. This result is caused by the overhead of runtime address calculations, which is not negligible for some benchmarks. Although we can have no doubt that our field affinity analysis is beneficial to enhance cache behaviors, field affinity relations are not a dominant factor to determine ideal field layouts for real executions. We guess that the overhead of field offset calculations should have been considered as importantly as field affinity relations. Taking the runtime overhead into field layout selection is another direction of future work. Nevertheless, there are four cases (*181.mcf*, *chomp*, *health*, and *ft*) where performance improvements are quite sizable. These results are occurred when the benefits gained from enhancing cache locality overwhelm the overhead of runtime address calculations.

(a) performance estimations

(b) execution times

Fig. 14. Normalized numbers of (a) performance estimations and (b) execution times.

7. RELATED WORK

Our primary goal is to make the whole process of layout transformation completely static, since most of the previous work usually relied on profiling. We achieve this goal using static memory access patterns. This section summarizes previous studies on layout transformation using profiling, static analysis, and dynamic analysis.

Table III. Execution Times

Program	Input Parameters	Original (sec)	Normalized (1.0 = Original)	
			Pool	Pool + Re
175.vpr	test	10.96	1.01	1.01
	ref	3857.29	1.01	1.01
181.mcf	test	0.192	1.00	0.99
	ref	270.175	1.00	0.79
300.twolf	test	0.19	1.00	1.12
	ref	435.24	0.98	0.99
analyzer	test	0.13	0.89	0.86
	ref	66.64	0.41	0.45
chomp	7,8	7.44	0.59	0.47
	6,10	18.05	0.55	0.50
misr	10	1.37	0.98	1.01
	50	31.39	0.99	1.01
bisort	10^6	2.89	0.99	0.99
	5×10^6	24.29	0.99	0.99
health	11, 50, 1, 1	56.24	0.78	0.70
	11, 60, 1, 1	86.05	0.71	0.63
mst	5000	19.23	0.84	0.83
	9000	65.73	0.82	0.82
perimeter	12	7.18	0.78	0.84
	13	17.106	0.76	0.84
treeadd	24	2.52	0.48	0.55
	26	10.172	0.48	0.55
tsp	10^6	9.92	0.96	0.97
	2×10^6	20.44	0.96	0.97
voronoi	10^6	5.22	0.98	0.99
	2×10^6	11.03	0.99	0.99
anagram	300	1.53	0.99	1.11
	900	4.568	1.00	1.13
bc	sqrt	0.07	0.96	0.96
	prime	1.95	0.82	0.81
ft	$10^3, 2 \times 10^5$	8.25	0.83	0.73
	$10^3, 3 \times 10^5$	19.34	0.87	0.79
ks	KL-4	2.06	1.04	1.02
	KL-5	7.46	1.03	1.03
AVERAGE			0.866	0.865

Chilimbi et al. [1999] describe a class-splitting algorithm that separates a cold portion (rarely referred) from a hot portion (frequently referred) of Java classes based on profiled field access statistics. For a certain class, cold fields of the class are divided by making the definition of a new class. The original class needs an auxiliary pointer field to link the new cold class with itself. Although their approach contributes to reducing L2 cache misses, the additional pointer fields make field accesses to cold classes suffer additional memory references resulting in the increase of total memory usage.

Rabbah and Palem [2003] suggest a field clustering technique that consecutively puts the same fields from numerous structure instances by employing customized allocation routines. After clustering the instances, they place all

fields in vertically aligned layouts. Their layouts have no overhead of runtime field offset calculations, but, require extra padding spaces to be inserted between fields to make constant offsets for all fields. These useless padding spaces incur waste of memory usage and cause more cache misses sometimes.

Zhong et al. [2004] exploit array regrouping and structure-splitting schemes using their formal model of reference affinity. Their concept represents the togetherness of data with reuse distances at the trace level. They introduce a k-distance analysis, so-called distance-based affinity analysis, to reorganize data sets. They obtain the most profitable data layout among all methods they tried, though reuse-distance profiling requires such a high overhead.

Our previous work [Shin et al. 2006] proposes a field layout restructuring scheme that combines the benefits of previous studies and relieves the problem of Rabbah and Palem's scheme [Rabbah and Palem 2003]. We compact fields by eliminating useless padding. This condensation demands extra runtime instructions for some field accesses. Due to pool alignment and field grouping, however, we are able to eliminate or reduce the overhead of runtime offset calculations.

Shen et al. [2005] suggest a frequency-based affinity analysis for array regrouping. Their approach is similar to the work of Chilimbi et al. [1999], in that both are based on data access frequencies. They enrich their analysis by designing a context-sensitive interprocedural analysis. They implement both static estimation and light-weight profiling of the execution frequency, and compare them with each other. According to their experiments, it is fairly safe to assume that all the counts of loops and function calls are the same.

Hundt et al. [2006] develop a framework that analyzes profitability of structure layout transformation with or without profile information. Their framework is capable of structure splitting, structure peeling, dead field removal, and field layout restructuring. Their optimizations are very similar to ours in that both exploit an interprocedural static analysis based on field affinity relations. They estimate field reference counts in tightly executed modules like loops, while we infer field affinities via memory access patterns represented in regular expressions. Our method obtains concise yet accurate patterns so that we are able to handle the field affinity relations across the boundaries of repeated regions, such as functions or loops.

Lattner and Adve [2005] devise an automatic pool allocation, which segregates pointer-based data structure instances in C and C++ programs into separate memory pools. Based on a context-sensitive pointer analysis and the escape-property for data structures, they determine which structures are beneficial to pool allocation. As shown in our experimental results, pool allocation improves program performance due to locality enhancement.

Java programming language enables researchers to apply dynamic analyses, since it is inherently performed on the runtime system, Java virtual machine (JVM). Dynamic analyses can obtain very accurate information in that they take runtime behaviors into account. This advantage leads runtime optimizations to better performance if their overheads are sufficiently relieved.

Guyer and McKinley [2004] introduce dynamic object colocation. Their basis is the same as our pool allocation in terms of interobject locality and custom

memory allocation; they estimate static object connectivity and modify connected objects to use the same allocators. The difference is that their allocation is not deterministic; they allocate objects in some regions according to dynamic object connectivity, while we statically assign each structure to its own pool.

Huang et al. [2004] propose online object reordering. Their key idea is to exploit the object reordering during the copying phase of generational copying garbage collection. Cooperating with a virtual machine, they update the information on hot fields. When the copying phase occurs, their garbage collector copies and enqueues the hot fields first. Their reordering scheme is similar to our layout transformation in that both attempt to enhance intra- and interobject locality and to use memory access behaviors. We devise a static method to obtain memory access patterns, while they rely on runtime supports.

In our work, we combine the benefits of pool allocation [Lattner and Adve 2005] and field layout restructuring [Shin et al. 2006] based on our own static analysis for memory access patterns.

8. CONCLUSION

We propose a novel method to represent memory access patterns with regular expressions. Regular expressions are suitable for explicitly denoting repetition and variety of memory reference behaviors. Access patterns are simply obtained by converting CFGs into automata. Closures of regular expressions imply not only the repetition of program executions, but also the affinity relations among the fields listed within them. Using statically obtained access patterns, we select structures for pool allocation and estimate field affinity relations for field layout restructuring. These analyses are integrated into our static framework based on the CIL.

In this article, we deal especially more with scalability problems of our previous work. To make our framework scalable, we improve the clarity of access patterns and investigate new approaches to interpreting patterns. We devise an automata reduction algorithm to extract more concise regular expressions from automata. We show the clarity and efficiency of our algorithm by implementing a traditional one and comparing our algorithm with it. Since our previous top-down approaches exhaustively revisit internal components of patterns, they prevent our framework from being a scalable scheme. We clear the inefficiency by devising novel bottom-up methods for both pool allocation and field layout restructuring. As a consequence, we are able to handle large benchmarks such as SPECINT 2000.

We implement the layout transformations, pool allocation, and field layout restructuring. Our evaluations show that both transformations dramatically reduce cache misses by 16%. As a result, we can improve the performance as well, reducing execution times by 14%. However, due to the overhead of field offset calculations, there are few cases that field layout restructuring is effective. Besides field affinity relations, runtime overhead of address calculations should be considered significantly. Nevertheless, statically analyzed access patterns are useful not only for layout transformations, but also for compiler techniques that attempt to optimize memory management of data intensive programs. We

believe our access pattern analysis will find profitable usages in many compiler optimizations.

REFERENCES

- AHO, A. V., SETHI, R., AND ULLMAN, J. D. 1986. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, New York.
- AUSTIN, T. M., BREACH, S. E., AND SOHI, G. S. 1994. Efficient detection of all pointer and array access errors. *ACM SIGPLAN Notices* 29, 6, 290–301.
- BOURDONCLE, F. 1993. Efficient chaotic iteration strategies with widenings. In *Proceedings of the International Conference on Formal Methods in Programming and Their Applications (FMPA'93)*. ACM, New York, 128–141.
- CHEREM, S. AND RUGINA, R. 2004. Region analysis and transformation for java programs. In *Proceedings of the 4th International Symposium on Memory Management (ISMM'04)*. ACM, New York, 85–96.
- CHILIMBI, T. M., DAVIDSON, B., AND LARUS, J. R. 1999. Cache-conscious structure definition. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'99)*. ACM, New York, 13–24.
- DAS, M. 2000. Unification-based pointer analysis with directional assignments. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'00)*. ACM, New York, 35–46.
- GLOY, N. AND SMITH, M. D. 1999. Procedure placement using temporal-ordering information. *ACM Trans. Program. Lang. Syst.* 21, 5, 977–1027.
- GUYER, S. Z. AND MCKINLEY, K. S. 2004. Finding your cronies: static analysis for dynamic object colocation. In *Proceedings of the 19th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'04)*. ACM, New York, 237–250.
- HENZINGER, T. A., JHALA, R., MAJUMDAR, R., AND SUTRE, G. 2002. Lazy abstraction. In *Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'02)*. ACM, New York, 58–70.
- HOPCROFT, J. E., MOTWANI, R., AND ULLMAN, J. D. 2001. *Introduction to Automata Theory, Languages, and Computation* 2nd Ed. Addison-Wesley, New York.
- 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 *Proceedings of the 19th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'04)*. ACM, New York, 69–80.
- HUNDT, R., MANNARSWAMY, S., AND CHAKRABARTI, D. 2006. Practical structure layout optimization and advice. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO'06)*. ACM, New York, 233–244.
- LATTNER, C. AND ADVE, V. 2005. Automatic pool allocation: improving performance by controlling data structure layout in the heap. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'05)*. ACM, New York, 129–142.
- MCGILL BENCHMARK SUITE. <http://llvm.org/>.
- NECULA, G. C., MCPK, S., RAHUL, S. P., AND WEIMER, W. 2002. CIL: Intermediate language and tools for analysis and transformation of c programs. In *Proceedings of the International Conference on Compiler Construction (CC'02)*. ACM, New York, 213–228.
- 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.
- ROGERS, A., CARLISLE, M. C., REPPY, J. H., AND HENDREN, L. J. 1995. Supporting dynamic data structures on distributed-memory machines. *ACM Trans. Program. Lang. Syst.* 17, 2, 233–263.
- RUNDBERG, P. AND WARG, F. Freebench v1.0 benchmark suite. <http://www.freebench.org/>.
- SEIDL, M. L. AND ZORN, B. G. 1998. Segregating heap objects by reference behavior and lifetime. In *Proceedings of the 8th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VIII)*. ACM, New York, 12–23.
- SHEN, X., GAO, Y., DING, C., AND ARCHAMBAULT, R. 2005. Lightweight reference affinity analysis. In *Proceedings of the 19th Annual International Conference on Supercomputing (ICS'05)*. ACM, New York, 131–140.

- SHIN, K., KIM, J., KIM, S., AND HAN, H. 2006. Restructuring field layouts for embedded memory systems. In *Proceedings of the Conference on Design, Automation and Test in Europe (DATE'06)*. IEEE, Los Alamitos, CA, 937–942.
- TARJAN, R. E. 1981a. Fast algorithms for solving path problems. *J. ACM* 28, 3, 594–614.
- TARJAN, R. E. 1981b. A unified approach to path problems. *J. ACM* 28, 3, 577–593.
- TOFTE, M. AND BIRKEDAL, L. 1998. A region inference algorithm. *ACM Trans. Program. Lang. Syst.* 20, 4, 724–767.
- TRUONG, D. N., BODIN, F., AND SEZNEC, A. 1998. Improving cache behavior of dynamically allocated data structures. In *IEEE International Conference on Parallel Architectures and Compilation Techniques (PACT'98)*. IEEE, Los Alamitos, CA, 322–331.
- VALGRIND. <http://valgrind.org>.
- WHEELER, D. A. SLOccount. <http://www.dwheeler.com/sloccount/>.
- ZHONG, Y., ORLOVICH, M., SHEN, X., AND DING, C. 2004. Array regrouping and structure splitting using whole-program reference affinity. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'04)*. ACM, New York, 255–266.

Received January 2007; revised September 2007; accepted July 2008