Layout Transformations for Heap Objects Using Static Access Patterns*

Jinseong Jeon, Keoncheol Shin, and Hwansoo Han

Division of Computer Science
Korea Advanced Institute of Science and Technology (KAIST)
Daejeon 305-701 Republic of Korea
{jinseong.jeon,keoncheol.shin}@arcs.kaist.ac.kr, hshan@cs.kaist.ac.kr

Abstract. As the amount of data used by programs increases due to the growth of hardware storage capacity and computing power, efficient memory usage becomes a key factor for performance. Since modern applications heavily use structures allocated in the heap, this paper proposes an efficient structure layout based on static analyses. Unlike most of the previous work, our approach is an entirely static transformation of programs. We extract access patterns from source programs and represent them with *regular expressions*. Repetitive accesses are usually important pieces of information for locality optimizations. The expressive power of regular expressions is appropriate to represent those repetitive accesses along with various access patterns according to the control flow of programs. By interpreting statically obtained access patterns, we choose suitable structures for pool allocation and reorganize field layouts of the chosen structures. To verify the effect of our static optimization, we implement our analyses and optimizations with the CIL compiler. Our experiments with the Olden benchmarks demonstrate that layout transformations for heap objects based on our static access pattern analysis improve cache locality by 38% and performance by 24%.

1 Introduction

Efficient memory usage is getting more important as more programs try to deal with large and complex data sets. Researchers investigated many ways to improve the efficiency of memory management, including additional hardware, new architectures, and compiler optimizations. Compiler optimizations are more attractive than other methods, since compilers can transform application codes to have more memory-friendly behaviors without any additional costs but a longer compile time spent in static analyses. Several compiler optimizations for memory management attempt to attain better locality by modifying application codes. Segregating the heap according to the lifetime of objects [1] or the pointing shapes of data structures [2], for instance, is studied. Region-based memory management [3], array regrouping [4, 5], and field layout restructuring [6–8] are

* This work was supported by grant No. R01-2006-000-11196-0 from the Basic Research Program of the Korea Science & Engineering Foundation.

S. Krishnamurthi and M. Odersky (Eds.): CC 2007, LNCS 4420, pp. 187–201, 2007.

other optimizations investigated by researchers. Some techniques use compile-time evaluated properties of programs by applying data structure analysis [2] or region inference [9]. Other techniques, on the other hand, rely on profiling for necessary information.

Previous studies on optimizations using memory access patterns are usually profile-based, since it is difficult to analyze memory access behaviors at compile-time. Profiling, however, is sensitive to inputs and execution environments sometimes. Hence profile-based optimizations can be limited in their usages. On the contrary, our goal is to predict memory access patterns through static analysis. In this paper, we propose a novel method to represent memory access patterns as *regular expressions*. This method is a completely compile-time process. Once we obtain memory access patterns in the forms of regular expressions, we use those pieces of information to guide heap layout transformations based on *pool allocation* and *field layout restructuring*.

The common essence of pool allocation and field layout restructuring is to enhance data locality by modifying the heap layout of programs. To achieve better data locality, both techniques find data that are simultaneously referred and collocate them with one another. Granularity is the only difference; the former deals with instances of structures, but the latter focuses on fields within structures. Our pool allocation scheme is similar to the pool allocation by Lattner and Adve [2] in that both use custom memory allocation routines for certain data structures. The difference is how to choose target structures for pool allocation; they use an expensive pointer analysis to find close relationships among structures, while we use an inexpensive pattern analysis to find heavily accessed ones. Our earlier work [8] used profiling to find memory access sequences of programs. In this paper, we propose a regular expression technique to make the whole optimizing procedure static.

Regular expressions are simple yet expressive enough to capture important access patterns for locality optimizations. Affinity relations among fields or objects are mostly determined by frequently executed portions of programs such as loops. The *Kleene Closure*[1] [10] in regular expression is intuitively appropriate to represent repetition. Considering other features commonly found in C-like imperative languages, regular expressions are indeed adequate for denoting the memory access patterns of programs. Not only can we abstract repetitive accesses with *closure* but also consecutive instructions with *concatenation* and conditional branches with *alternation*. Moreover, interpreting regular expressions is straightforward, thanks to their conciseness.

This paper makes the following contributions:

- We propose a novel method to represent the memory access patterns of programs with regular expressions.
- We present new analyses to select structures for pool allocation and to estimate their field affinity relations by interpreting memory access patterns represented as regular expressions.

[1] We use the term *closure* for the rest of this paper.

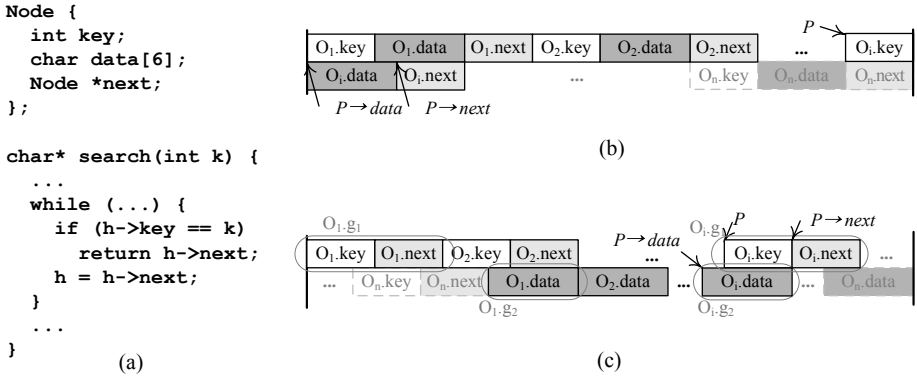

Fig. 1. (a) Motivating example, (b) structure layout with pool allocation, (c) structure layout after field layout restructuring [8]

– We evaluate the performance impact of our static scheme with the compiler implementation of our analyses and heap layout transformations.

The rest of this paper is organized as follows. Section 2 briefly introduces pool allocation and field layout restructuring. Calculating memory access patterns with regular expressions is detailed in Sect. 3. Selecting structures for pool allocation and estimating field affinity relations are discussed in Sect. 4. Section 5 shows our experimental environments and evaluations. Finally, Sect. 6 contrasts our work with prior studies and Sect. 7 concludes this paper.

2 Heap Layout Transformations

Before we discuss how to extract access patterns at compile-time, this section describes two heap layout transformations. Pool allocation [2] and field layout restructuring [8] 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 Sect. 4.

2.1 Pool Allocation

When objects are individually managed by `malloc` and `free`, compilers cannot predict exact addresses of dynamically allocated objects. This lack of layout information causes many compiler techniques (*e.g.*, field layout restructuring, software prefetching, *etc.*) to be either less effective or not exploitable [2]. Pool allocation [2, 8] is an effective technique that provides compilers with layout information and leads to better data locality as well. Figure 1(b) shows a structure layout when objects are allocated in a pool.

Collocating closely related objects with one another improves data locality by the effect of prefetching in the cache memory. In addition to that, 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. They reserve chunks of memory beforehand and assign a fraction of the chunks for each object allocation in a simple and uniform way. In the event of memory releases, custom free routine just restore 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 Fig. 1(a), 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 Fig. 1(c) has an advantage over the original pool layout as in Fig. 1(b) in terms of data locality and performance. Based on this observation, we proposed a field layout restructuring scheme in our previous work [8]. 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 Fig. 1(c) are extra run-time 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 run-time calculations. Nevertheless, compiler optimization and pool alignment are able to make the overhead lower.

3 Regular Expressions for Access Patterns

Our specific goal 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 to estimate adequate field layouts for those structures. Then the framework finally transforms the heap layout into the locality-enhanced layout as shown in Fig. 1(c). In order to design the compile-time framework, we have 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 the 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* can naturally represent repetitions with *closures*, which make regular expressions suitable for the abstraction of memory access patterns. Besides the repetition (*closure*), regular expressions can capture the access patterns in sequential instructions with *concatenation* and conditional branches with *alternation*.

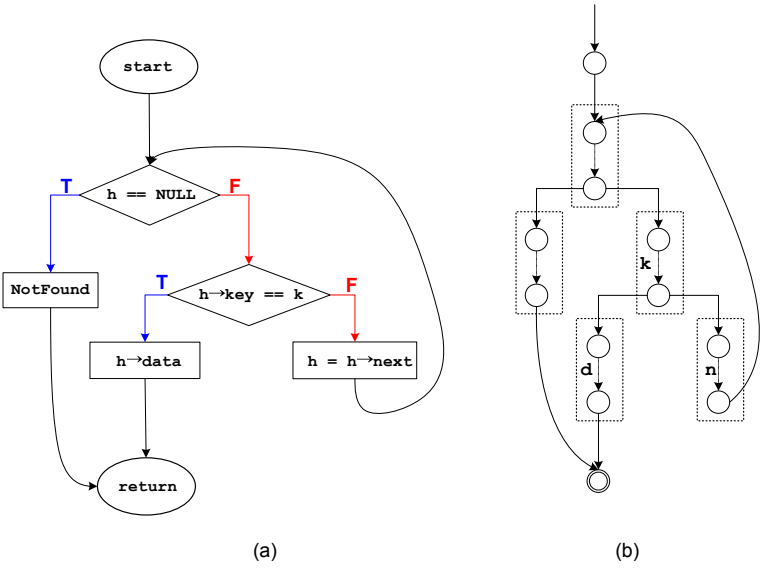

Fig. 2. (a) Control-flow graph of the motivating example, (b) converted automaton

3.1 Conversion of CFGs into Automata

Access patterns of programs are determined by their control flow and data access instructions. When we want to obtain field access patterns, we can use the sequence of referred field names. The sequence, however, can be too long and we need to statically abstract the sequence somehow. The control-flow information obtained from the *control-flow graph* (CFG) plays a critical role in reducing the sequence of field names. Observing that automata and regular expressions are equivalent, we find a novel method to capture access patterns with regular expressions. By converting CFGs into automata with access sequences labeled on edges, we can express access patterns with automata. We then exploit an automata reduction technique to summarize access patterns as regular expressions.

Figure 2(a) depicts the CFG of the motivating example and Fig. 2(b) depicts the automaton[2] converted from Fig. 2(a). 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 of a function to a corresponding start point, and end points to the accepting state of the function, labeling the edges with empty strings too. Consequent automaton as shown in Fig. 2(b) encompasses all the possible behaviors of a function, since it mimics the CFG of the function without loss of control-flow information.

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

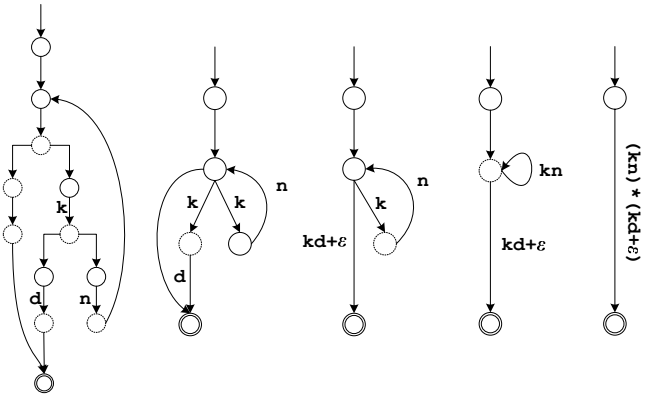

Fig. 3. Automata reduction

3.2 Access Pattern Extraction from Automata

Extracting regular expressions from automata is an instance of path problems [11, 12]. Regular expressions for access patterns are simply obtained by using a state elimination technique (Chap. 3.2.2 of [10]). Figure 3 shows the progress of automata reduction. The order of state elimination is crucial for compilers to extract understandable patterns from automata. First of all, we remove the states which have outgoing edges labeled with empty strings and no incoming back-edges. Because these states represent the instructions unrelated with field accesses 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 [13]. To make closures correctly enclose loops, we need to postpone the elimination of the states that have incoming back-edges, since those 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 [13]. 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 excluded states from the criteria mentioned above are erased in topological order.

The second automaton in Fig. 3 depicts the status after removing all the states which 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.

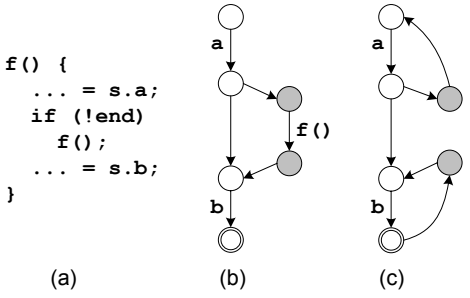

Fig. 4. (a) Example code for self recursive function, (b) automaton after intra-procedural pattern analysis, (c) automaton after inter-procedural pattern analysis

3.3 Extension to Inter-procedural Patterns

Since the CFG in Fig. 2(a) has just intra-procedural information, the access pattern extracted from the corresponding automaton includes the reference behavior of the function body only. To gain accurate field affinity relations over the entire execution, 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 inter-procedural access patterns is straightforward. Unfortunately, we cannot handle mutual recursion yet. The following description only deals with normal and self recursive calls.

To obtain inter-procedural patterns, we visit functions in reverse topological order of a call graph. When we meet a call site while building an automaton for a function, we label the corresponding edge with the name of 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 Fig. 4(a). The function `f` has a recursive call to itself. Once we calculate the intra-procedural access pattern of the code, we will have the automaton shown in Fig. 4(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 the call site to the start state of the 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. Figure 4(c) depicts the consequent automaton.

Although the method described above can resolve self recursive functions well, obtained patterns through that solution are not precise enough to express the access patterns of self recursive functions. Let the access pattern of the function in Fig. 4(a) be F. The precise access pattern, F can be described with the following grammar:

$$F \rightarrow \quad ab \mid aFb$$

This grammar is represented 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 Fig. 4(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^ib^i can imply. This may not be an important fact for our optimization.

4 Interpretation of Regular Expressions

This section explains how we interpret regular expressions for access patterns. We use regular expressions to identify beneficial structures for pool allocation and to estimate affinity relations among the fields of chosen structures. In the following subsections, we introduce previous work and a profiled-based method, and then describe our static methods.

4.1 Structure Selection for Pool Allocation

Lattner and Adve [2] 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 [2]. 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.

A structure access pattern is obtained by substituting field names in a field access pattern with the structure names to which the fields belong. As for the motivating example in Fig. 1(a), structure access pattern $(N \cdot N)^*(N \cdot N + \varepsilon)$ (N denotes the structure `Node`) is obtained from the corresponding field access pattern $(kn)^*(kd+\varepsilon)$. Since the structure `Node` is the only structure that appears in the closure, 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 in closures as frequently allocated ones.

4.2 Field Affinity Estimation

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) [7]. Figure 5 depicts the progress of profile-based affinity estimation. *Temporal relationship graph* (TRG) [14] is a weighted graph where its nodes represent fields and the weights of its edges represent

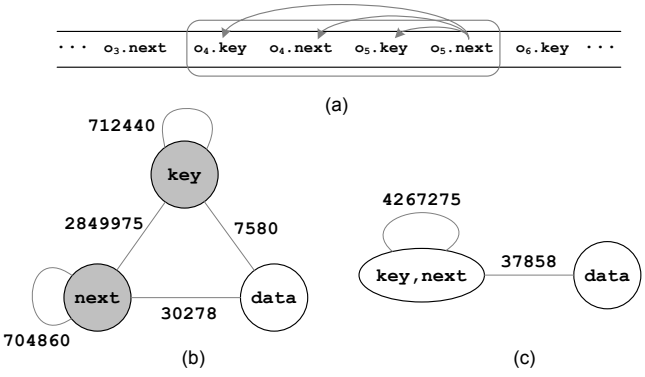

Fig. 5. (a) Profiled memory access sequence and a sliding window to calculate NAP, (b) initial STRG, (c) STRG after grouping *key* and *next* fields

NAPs between fields. Since one field can be accessed consecutively, we extend TRG to have self-edges and name it STRG. Figure 5(a) shows the concept of NAP calculation using a sliding window over a profiled field access sequence. An initial STRG after profiling the motivating example is shown in Fig. 5(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 Fig. 5(c). The edges are merged and the weights are modified to encompass the previous relationships. Until the STRG does not change, we repeat the following procedure: finding a beneficial grouping and merging fields. Each node in the final STRG becomes a group in a field layout restructuring scheme. The groups in final STRG are placed in decreasing order of the weights of self-edges. In Fig. 5(c), we cannot find a profitable grouping any more. As a result, $\{key, next\}$ and $\{data\}$ are placed in the heap as shown in Fig. 1(c).

Statically obtained field access patterns imply the abstract relationship between fields, but not presented with numerical values. To overcome the gap between realistic values and abstract relationships, we devise a symbolic approach. Instead of NAP, we label edges of STRG with closure signs to indicate how often two fields are accessed together. Consider the example in Fig. 6, assuming a program that performs list generation, parity check, and random search in turn. The regular expression that represents the access pattern of the program is shown in the top of the figure. Based on this regular expression, we construct a symbolic STRG as shown in Fig. 6(a) where the weights of the 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 $((kn)^*(kd + \varepsilon))^*$. 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*

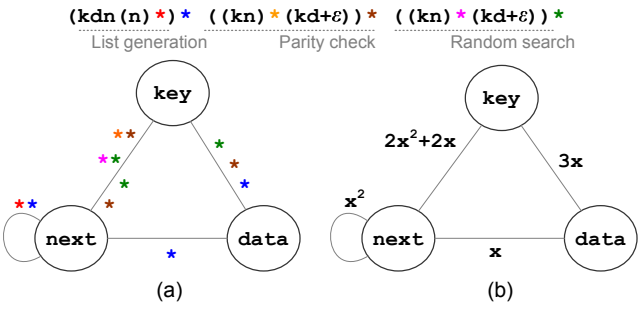

Fig. 6. (a) STRG with closure signs, (b) STRG with a closure variable

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. Similarly, 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 Fig. 6(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 value (100) to the closure variable. 10, 100, and 1000 make the same field layouts in our evaluations. The rest of estimating procedure is the same as profile-based estimation depicted in Fig. 5.

5 Experimental Evaluation

5.1 Implementation

We implement our framework based on the CIL framework [15], which includes access pattern analysis, structure selection analysis, field affinity analysis, and layout transformation. We assume that most programs access fields by explicit field names, since users cannot ensure the memory layouts generated by compilers. Under this assumption, we transform explicit field names to field references on modified field layouts. For some field references we add extra instructions to calculate field offset as described in [8]. Memory management routines such as `malloc` and `free` calls are transformed into custom memory management routines using pool allocation [2, 8]. Programs used in our evaluations do not have mutually recursive calls. Therefore, our framework can obtain inter-procedural patterns without any effort to handle such cases.

Table 1. Times spent in analysis and compilation

Program	SLOC	Structure Selection	Field Affinity	Code Transform	Total	GCC
chomp	378	0.021	0.006	0.003	0.030	0.212
ft	926	0.050	0.014	0.010	0.074	0.298
health	474	0.024	0.004	0.002	0.030	0.202
mst	408	0.031	0.004	0.002	0.037	0.195
perimeter	345	0.012	0.012	0.001	0.025	0.197
treeadd	154	0.002	0.000	0.000	0.002	0.120
tsp	433	0.011	0.004	0.002	0.017	0.201
voronoi	975	0.048	0.004	0.003	0.055	0.295

One limitation in our framework is that we 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 [16] 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 cases, 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.

5.2 Experimental Environment

Our evaluations are performed on a Redhat 9.0 Linux PC equipped with a 2.6GHz Pentium4 processor. This machine contains 8KB L1D cache (64byte cache line, 4-way set associative), 512KB L2 cache (64byte cache line, 8-way set associative), and 1.7GB 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) [17] 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 in order 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 ten runs. To confirm the effect of our static mechanism, we examine programs with two different size inputs.

Some of the Olden suite, "chomp" from the McGill benchmark suite [18], and "ft" from the Ptrdist suite [19] 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. Those benchmarks are excluded from our experiments. Table 1 shows source lines of code (SLOC) [20] and analysis times for each program. As shown in the table, additional times spent in our analyses and conversion are small enough for all cases.

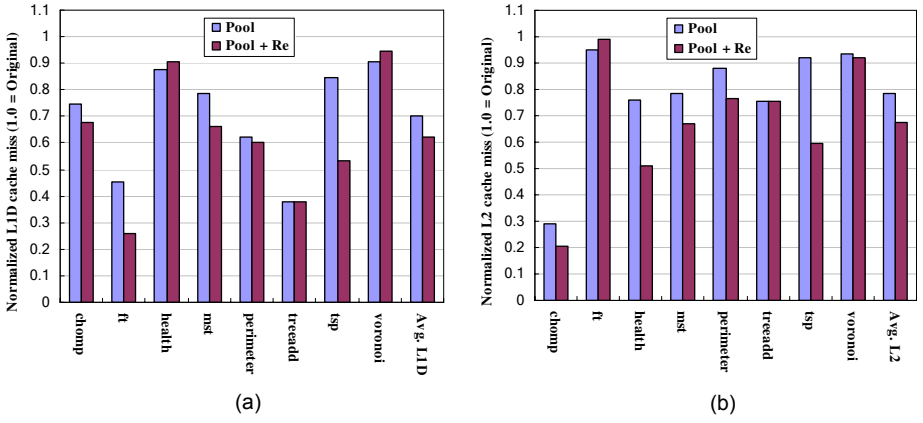

Fig. 7. Normalized numbers of misses in (a) L1D and (b) L2 caches

5.3 Improvements in Cache Locality

Figure 7 shows normalized cache misses in L1D and L2. The numbers are averages of two different size inputs. *Pool* and *Pool + Re* denote the effect of pool allocation alone and field layout restructuring with pool allocation, respectively.

In our evaluations using data intensive benchmarks, pool allocation is significantly effective. Compared with original programs, its miss reductions are roughly 30% for L1D and 22% for L2 on average. These miss reductions are due to better locality by gathering instances of certain structures in the same pools.

Under pool allocation, field layout restructuring can be an auxiliary method to reduce cache misses more. Compared with original programs, its miss reductions are 38% for L1D and 32% for L2 on average. In four cases (*chomp*, *mst*, *perimeter*, and *tsp*), it is quite beneficial to miss reductions in both cache levels. For the particular case (*treeadd*), the field affinity analysis choose an inefficient layout, which makes more cache misses. But, the miss increases are very marginal. In the remaining three cases, cache misses in either L1D or L2 are reduced more than pool allocation alone.

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

5.4 Improvements in Performance

Table 2 shows execution times and dynamic instruction counts of benchmarks. The column labeled with *Original* provides base results from original programs. *Pool* and *Pool + Re* columns show the impact of pool allocation alone and

Table 2. Execution times and dynamic instruction counts

Program	Input Parameters	Original		Normalized (1.0 = Original)			
		exec (sec)	#instr ($\times 10^9$)	Pool		Pool + Re	
				exec	#instr	exec	#instr
chomp	7,8	7.44	2.28	0.59	1.34	0.47	1.34
	6,10	18.05	3.77	0.55	1.43	0.50	1.43
ft	$10^3, 2 \times 10^5$	8.25	1.78	0.83	0.97	0.73	0.97
	$10^3, 3 \times 10^5$	19.40	3.13	0.87	0.97	0.79	0.97
health	11, 50, 1, 1	56.24	40.59	0.78	0.75	0.70	0.81
	11, 60, 1, 1	86.05	49.69	0.71	0.76	0.63	0.81
mst	5000	19.23	19.12	0.84	0.82	0.83	0.83
	9000	65.73	62.08	0.82	0.82	0.82	0.84
perimeter	12	7.18	10.85	0.78	0.77	0.84	0.81
	13	17.11	25.59	0.76	0.74	0.84	0.78
treeadd	24	2.52	4.60	0.48	0.44	0.55	0.44
	26	10.17	18.39	0.48	0.44	0.55	0.44
tsp	10^6	9.92	15.33	0.96	0.99	0.97	1.02
	2×10^6	20.44	31.68	0.96	0.99	0.97	1.02
voronoi	10^6	5.22	7.53	0.98	0.99	0.99	1.00
	2×10^6	11.03	15.81	0.99	0.99	0.99	1.00
Avg. % improved				22.64%	11.24%	24.04%	9.41%

field layout restructuring with pool allocation, respectively. The results of layout transformations are normalized with original programs.

As a result of the enhancement of cache locality, the performance of programs also improves. Compared with original programs, execution times of pool allocation are reduced by 23% on average. This substantial improvements in performance are due to not only the remarkable miss reductions of the caches, but also the reductions in the number of instructions executed in custom memory management routines using pools. As shown in the table, dynamic instruction counts of pool allocation are reduced by 11% on average. These results are due to simpler internal structures for memory management and removal of many custom `malloc` and `free` calls.

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

6 Related Work

Rabbah and Palem [7] 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 the fields in vertically aligned layouts. Their layouts have no overhead of run-time field offset calculation, however, require extra padding space to be inserted between fields to make constant offsets for all fields. These useless padding sometimes incurs the waste of memory usage and causes more cache misses.

Our previous work [8] proposes a field layout restructuring scheme that combines the benefits of previous works and relieves the problems of Rabbah and Palem's scheme [7]. We compact fields by eliminating useless padding. This condensation demands extra run-time instructions for some fields accesses. Due to pool alignment and field grouping, however, we can eliminate or reduce the overhead of run-time offset calculations.

Shen *et al.* [4] develop a frequency-based affinity analysis for array regrouping. Their approach is similar to the work of Chilimbi *et al.* [6] in that both are based on data access frequencies. They enrich their analysis by designing a context-sensitive inter-procedural analysis. They implement both static estimation and lightweight profiling of the execution frequency, and compare them with each other. According to their experiments, it is a fairly safe to assume that all the counts of loops and function calls are the same.

Lattner and Adve [2] 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 enables researchers to apply dynamic analyses [21, 22], since it is inherently performed on a run-time system. Dynamic analyses can obtain very accurate information in that they take run-time behaviors into account. This advantage leads run-time optimizations to better performance if their overheads are sufficiently relieved.

7 Conclusion

We present a novel method to represent memory access patterns with regular expressions. Using statically obtained access patterns, we select structures for pool allocation and estimate field affinity relations for field layout restructuring. These analyses and both layout transformations are integrated into our framework based on the CIL. Our evaluation shows that the cache misses for L1D and L2 are reduced by 38% and 32%, respectively. As a result, performance improvement is 24% on average. Statically analyzed access patterns are useful not only for layout transformation but also for the 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

1. Seidl, M.L., Zorn, B.G.: Segregating heap objects by reference behavior and lifetime. In: ASPLOS-VIII. (1998) 12–23
2. Lattner, C., Adve, V.: Automatic pool allocation: improving performance by controlling data structure layout in the heap. In: PLDI '05. (2005) 129–142
3. Cherem, S., Rugina, R.: Region analysis and transformation for java programs. In: ISMM '04: International Symposium on Memory Management. (2004) 85–96
4. Shen, X., Gao, Y., Ding, C., Archambault, R.: Lightweight reference affinity analysis. In: ICS '05: International Conference on Supercomputing. (2005) 131–140
5. Zhong, Y., Orlovich, M., Shen, X., Ding, C.: Array regrouping and structure splitting using whole-program reference affinity. In: PLDI '04. (2004) 255–266
6. Chilimbi, T.M., Davidson, B., Larus, J.R.: Cache-conscious structure definition. In: PLDI '99. (1999) 13–24
7. Rabbah, R.M., Palem, K.V.: Data remapping for design space optimization of embedded memory systems. ACM TECS **2**(2) (2003) 186–218
8. Shin, K., Kim, J., Kim, S., Han, H.: Restructuring field layouts for embedded memory systems. In: DATE '06: Design, Automation and Test in Europe. (2006) 937–942
9. Tofte, M., Birkedal, L.: A region inference algorithm. ACM TOPLAS **20**(4) (1998) 724–767
10. Hopcroft, J.E., Motwani, R., Ullman, J.D.: Introduction to automata theory, languages, and computation, 2nd edition. Addison-Wesley (2001)
11. Tarjan, R.E.: A unified approach to path problems. J. ACM **28**(3) (1981) 577–593
12. Tarjan, R.E.: Fast algorithms for solving path problems. J. ACM **28**(3) (1981) 594–614
13. Bourdoncle, F.: Efficient chaotic iteration strategies with widenings. In: FMPA '93: Formal Methods in Programming and their Applications. (1993) 128–141
14. Gloy, N., Smith, M.D.: Procedure placement using temporal-ordering information. ACM TOPLAS **21**(5) (1999) 977–1027
15. Necula, G.C., McPeak, S., Rahul, S.P., Weimer, W.: CIL: Intermediate language and tools for analysis and transformation of c programs. In: CC '02. (2002) 213–228
16. Rogers, A., Carlisle, M.C., Reppy, J.H., Hendren, L.J.: Supporting dynamic data structures on distributed-memory machines. ACM TOPLAS **17**(2) (1995) 233–263
17. : Valgrind. (http://valgrind.org/)
18. : McGill benchmark suite. (http://llvm.org/)
19. Austin, T.M., Breach, S.E., Sohi, G.S.: Efficient detection of all pointer and array access errors. ACM SIGPLAN Notices **29**(6) (1994) 290–301
20. Wheeler, D.A.: SLOCcount. (http://www.dwheeler.com/sloccount/)
21. Guyer, S.Z., McKinley, K.S.: Finding your cronies: static analysis for dynamic object colocation. In: OOPSLA '04. (2004) 237–250
22. Huang, X., Blackburn, S.M., McKinley, K.S., Moss, J.E.B., Wang, Z., Cheng, P.: The garbage collection advantage: improving program locality. In: OOPSLA '04. (2004) 69–80