Safe and Flexible Adaptation via Alternate Data Structure Representations

Amlan Kusum
University of California, Riverside, USA
akusu001@cs.ucr.edu

Iulian Neamtiu
New Jersey Institute of Technology, USA
ineamtiu@njit.edu

Rajiv Gupta
University of California, Riverside, USA
gupta@cs.ucr.edu

Abstract

The choice of data structures is crucial for achieving high performance. For applications that are long-running and/or operate on large data sets, the best choice for main data structures can change multiple times over the course of a single execution. For example, in a graph-processing application where the graph evolves over time, the best data structure for representing the graph may change as the program executes. Similarly, in a database or a key-value store application, with changes in relative frequencies of different types of queries over time, the most efficient data structure changes as well. We introduce an approach that allows applications to adapt to current conditions (input characteristics, operations on data, state) by switching their data structures on-the-fly with little overhead and without the developer worrying about safety or specifying adaptation points (this is handled by our compiler infrastructure). We use our approach on different classes of problems that are compute- and memory-intensive: graph algorithms, database indexing, and two real-world applications, the Memcached object cache and the Space Tyrant online game server. Our results show that off-the-shelf applications can be transformed into adaptive applications with modest programmer effort; that the adaptive versions outperform the original, fixed-representation versions; and that adaptation can be performed on-the-fly safely and with very little runtime overhead.

Categories and Subject Descriptors D.3.3 [*Programming Languages*]: Language Constructs and Features–Frameworks; D.3.3 [*Programming Languages*]: Processors–Code generation; E.2 [*Data*]: Data storage representations–Composite structures

General Terms Languages, Performance

Keywords adaptation, runtime data structure selection, space-time trade-off, input characteristics

1. Introduction

The performance of data-intensive applications is highly dependent upon the main data structures used. As we demonstrate via a wide range of experiments, using a single data structure representation for an entire program run is problematic in many cases: when *input characteristics* vary (e.g., a graph algorithm analyzing an input graph that evolves over time); or when *task characteristics* vary (e.g., workload profile for a key-value store); or when *state characteristics* vary (e.g., an online game with a variable number of players).

For example, on a read-mostly workload, the performance of the Memcached object cache can double by switching to Cuckoo hashing [8, 20], compared to Memcached's default hashing. However, to achieve this performance gain, the entire implementation has to be switched; we allow this switch to be performed on-the-fly. As a second example, consider running six popular graph algorithms (described in Section 5) on MovieLens, a naturally-evolving graph containing movie reviews which in its final state has 3,979,428 edges and 36,526 vertices. Alternate data structures can store the graphs—adjacency list, adjacency matrix, or shards—each with its own trade-offs. Using a single data structure representation imposes a typical performance overhead of 22% compared to our adaptive version which uses different representations during different execution intervals. Hence data structure representation cannot be selected *a priori* at compile time—rather, it should be selected at runtime, when the choice of the data structure can be adapted to changing input or task characteristics.

We propose a *programmer-assisted approach* to adaptation, where data structures and algorithms change on-the-fly, safely and efficiently. Our approach is based on: (1) programming support for transforming off-the-shelf applications into adaptive applications; (2) compile-time analyses that automatically identify program points at which the application can safely switch between alternative algorithms and data structures, relieving developers from a burdensome and error-prone task; and (3) a runtime component that performs on-the-fly switching, allowing the application to select the right implementation to exploit available resources.

Runtime adaptation poses several challenges: guaranteeing safety of on-the-fly data structure and algorithm changes, imposing low steady-state overhead, reacting quickly to system and input changes, minimizing programmer burden. We address these challenges as follows. Programmers render applications adaptive by indicating the alternate implementations of a certain data structure, the application's main computation loops, and writing conversion functions between the alternate data structures (Section 3). Programmers, however, do not specify *where* adaptation should be performed, as that could jeopardize safety, substantially increase programmer burden, and reduce opportunities for adaptation. Instead, our infrastructure uses a suite of static analyses to find *safe* adaptation points and increase *timeliness*, i.e., react quickly to a mismatch between the current data structure and the input or workload characteristics (Section 4).

In Section 5 we provide an evaluation of our approach along multiple dimensions: ease of use, effectiveness, efficiency. For evaluation we use graph algorithms, database indexing and two real-world applications, the Memcached high-performance key-value store and the Space Tyrant online gaming server. We found that most of the programming effort required to convert off-the-

CC'16, March 17–18, 2016, Barcelona, Spain
© 2016 ACM. 978-1-4503-4241-4/16/03...$15.00
http://dx.doi.org/10.1145/2892208.2892220

Table 1: Worst-case complexity of graph representations.

Data Structure	Space	Edge Lookup Time
Adjacency List (ADJLIST)	$O(\|V\| + \|E\|)$	$O(\|E\|)$
Shards (SHARDS)	$O(\|V\| + k_1\|E\|)$	$O(\|E\|/k_2)$
Adjacency Matrix (ADJMAT)	$O(\|V\|^2)$	$O(1)$

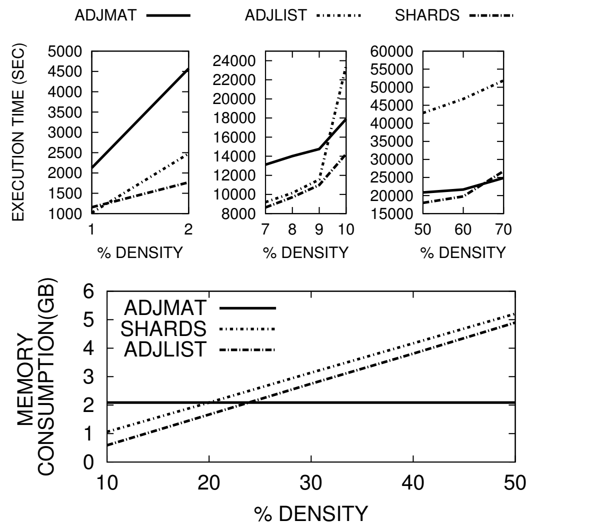


Figure 1: MSSP on Google+ graph with varying density: execution time (top) and memory consumption (bottom).

shelf applications into adaptive applications consists of writing conversion functions between representations. We then measure the benefit provided by our approach by showing that, even when starting with an unfavorable initial data structure choice, applications detect this and quickly switch to using the best representation. We also show that the performance of adaptive applications is close to the best-possible performance on a certain input, i.e., hand-coded applications where the best representation is found in advance via profiling on that input.

2. Motivating Applications

This section has two goals. First, we quantify the impact of input data characteristics, workload characteristics, and data structure choice on memory usage and execution time. Second, we show that, with a limited set of experiments, developers can characterize program behavior to write adaptation policies which will guide data structure selection at run time. We use three data-intensive applications: a graph application that computes the Multiple Source Shortest Path (MSSP); a database application implementing an Indexed Flat File DB (DBMS); and the Space Tyrant online game server. Each application is centered around one main data structure that typically holds a large data set. We explore three alternative representations for the main data structures and study the sensitivity of their relative runtime performance and memory demands to the input characteristics.

2.1 MSSP: A Graph Application

Let $G = (V, E, W)$ be a graph, where V is the set of nodes, E is the set of edges, and W holds the edge weights. Graph *density*, defined as $\frac{2*|E|}{|V|*(|V|-1)}$, is a key characteristic; we use percentages to indicate density, where a low percentage indicates a *sparse* graph and a high percentage a *dense* graph. The main data structure

Table 2: Best graph representation, by density interval.

Criterion	Density			
	0-2%	2-25%	25-67%	68-100%
Space	ADJLIST	ADJLIST	ADJMAT	ADJMAT
Time	ADJLIST	SHARDS	SHARDS	ADJMAT

(the input graph G) can be represented in different ways. We focus on three representations: Adjacency List (ADJLIST), which stores the outgoing edges of each node in a list; a collection of Shards (SHARDS), where each shard contains all edges incident to a distinct subset of nodes in the graph [11, 16]; and Adjacency Matrix (ADJMAT), which stores the edge weights in a matrix. MSSP computes the shortest path from each vertex to every other vertex, and it can be implemented using multiple applications of SSSP, with each vertex as source.

Space and time trade-off. Table 1 shows the space requirements and edge weight lookup times for the three graph representations. The ADJLIST representation exploits graph sparsity to achieve a compact form, but has the highest worst-case edge weight lookup time which increases with graph density. The SHARDS representation uses additional space and in return has lower worst-case edge lookup time; note that k_1 depends on shard size and graph density, while k_2 depends on graph density. Finally, the ADJMAT representation uses the most space and performs edge weight lookup in constant time, i.e., independent of graph density. Thus, it is expected that input graph density, which is not known at compile time, will affect runtime memory consumption and execution time, and there is an inherent trade-off between space and time among these three representations.

We studied the space and time costs of using the three representations for computing MSSP on a real-world graph, Google+ circles [14]. The graph consists of 23,628 nodes representing users, and 39,242 edges representing links to users in his/her circle. Figure 1 plots the execution time and memory consumption for all three representations and demonstrates the expected space–time trade-off among the three representations. For clarity we only show "interesting" graph density ranges, around crossover points (2%, 9% and 68% for time; 20% and 25% for space), i.e., densities at which one representation starts to outperform another.

Stability across input sizes. To see how the space-time trade-off manifests for representations at different input sizes, we conducted a series of experiments: we varied the graph size from 10,000 to 25,000 nodes, in increments of 5,000; and we varied the density by changing the number of edges. This helps identify crossover points, i.e., threshold densities where one representation starts outperforming another. Table 2 summarizes our findings. The thresholds found in this experiment clarified which representation is a better choice for what density ranges. For example, when the graph density is less than 2%, ADJLIST is the "better" representation as it is both more memory- and time-efficient than ADJMAT. Similarly, in the last interval ($>$ 68%) ADJMAT is the clear winner as it takes less memory and is faster. SHARDS wins the race of time-efficiency between 2% and 25%. Between 25% and 67%, however, ADJMAT is more memory-efficient while SHARDS is more time-efficient, hence the best representation depends on operational constraints, e.g., is memory limited? Is performance critical so it is worth trading off space for time? *With our approach, the best representation is chosen automatically, at runtime, based on density and operational constraints.*

Table 3: Runtime performance of representations for DBMS.

Data Structure	Insert	Search
`BTREE`	Slow	Fast
`AVLTREE`	Slow	Intermediate
`RBTREE`	Fast	Slow

Table 4: Workload characteristic vs. best representation.

Criterion	Percentage of Insert Operations		
	0-37%	37-62%	62-100%
Time	`BTREE`	`AVLTREE`	`RBTREE`

2.2 DBMS: An Indexed Flat File DB

We now illustrate the impact of data structure choice on database operations' performance. For our experiments, the data is stored in a flat file on disk, however the database indexes are stored in memory, in a tree (along with file offsets so data could be accessed in $O(1)$ time after an offset value is fetched from the tree). There are numerous ways to store the indexes; we chose three popular data structures: BTree (`BTREE`) of order 5, AVL Tree (`AVLTREE`), and Red Black Tree (`RBTREE`). The database operations are real-world social network queries, as explained shortly.

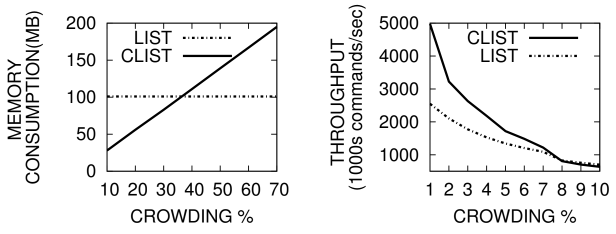
Time requirements. Although the three indexing data structures have the same worst-case time complexity for insert and search operations—$O(log\ n)$, where n is the number of nodes—they have different insertion and search times due to their rebalancing policy (Table 3).

We measured the execution time of insert and search operations on a social network database, with the data and queries from the BG Benchmark [3]. Our workload consisted of two actions, *initiate friend request* (user X sends a friend request to user Y), and *search friendship* (find if X and Y are friends). *Initiate friend request* requires an insert (SQL `INSERT` operation) while *search friendship* requires a search (SQL `SELECT` operation). We populate the initial network with 10,000 users and 100 friend requests for each user. We generated a workload of 5,000,000 operations which varies the insert/search ratios, from 10% inserts–90% searches to 90% inserts–10% searches. Similar to MSSP, we varied workload size from 1,000,000 queries to 5,000,000 queries and found that crossover points are stable across workload sizes. We summarize our findings in Table 4: each representation has an interval where it is the most suitable. *With our approach, the best representation is chosen automatically, at runtime, based on workload.*

2.3 Space Tyrant: An Online Game Server

Space Tyrant is an online multiplayer game server, where players move their ships around a 2D universe. The game state is kept in a *map* divided into *sectors* and information for each sector is stored in a `LIST`, which is backed up periodically on the disk. We found an alternative, compressed method of representing sectors in memory, `CLIST`, in which only the occupied sectors are stored, along with their neighboring sectors. Let $G = \{U, S\}$ be a game where U is the number of users, and S is the number of points in the space represented as sectors. Game *crowding*, defined as $\frac{U}{S}$, is a key characteristic for choosing the map representation.

Space and throughput trade-off. Table 6 shows the worst-case space complexity and the sector information look-up time for each representation. The `CLIST` representation exploits the crowding property of the map to reduce memory consumption, however it causes sector lookups to cost more. The `LIST` representation requires more space but performs the sector lookup in constant time. We studied the throughput and space costs of the two representations during game plays with 1-million sector maps. For each game play, we varied the number of users from 10,000 to 900,000, thus increasing crowding from 1% to 90%. Figure 2 shows that for low *crowding*, i.e., below 8%, `CLIST` is a better representation in terms of both the throughput and the memory consumption; however, above 35%, `LIST` is better in terms of both memory and time.



(a) Memory Consumption **(b) Throughput**

Figure 2: Space Tyrant with different crowding percentages.

Table 5: Best sector representation, by crowding interval.

Criterion	Crowding		
	0-8%	8-35%	35-100%
Space	`CLIST`	`CLIST`	`LIST`
Time	`CLIST`	`LIST`	`LIST`

Table 6: Worst-case complexity of sector representations.

Data Structure	Space	Sector Lookup Time
List (`LIST`)	$O(\|S\|)$	$O(1)$
Compressed List (`CLIST`)	$O(\|U\|)$	$O(\|U\|)$

Stability across game sizes. We studied the stability of the crossover points by varying the game size from 500,000 to 1,000,000. The crossover points remain consistent across different game sizes. We summarize the findings in Table 5.

In summary, these results prove that none of the eight data structure representations yields the best performance in terms of memory consumption and/or execution time for all input data and workload characteristics, and runtime data structure selection is the solution for optimal performance and resource use.

3. Overview

We now present our approach for transforming off-the-shelf applications into adaptive applications that safely and efficiently switch between data structure representations to optimize their operation (higher speed, lower memory usage, etc.) and adapt to changes in input, workload, or state.

Programmers need to add a handful of *annotations* to the source code, to indicate alternative representations (data structures & implementations); mark the application's long-running, compute-intensive loop(s); use a progress indicator (variable holding the value of "progress bar"); and write representation conversion functions. This annotated source code is passed through our *static analyzer and compiler*, which: find safe adaptation points in the source code; perform a source-to-source translation to instrument the code to permit adaptation; and add the transition logic. This code is then compiled with a normal C compiler, e.g., gcc, to yield the

```
 1  #pragma __ADAPT_DS(GRAPH)("ADJMAT")
 2  #pragma __ADAPT_DS(GRAPH)("ADJLIST")
 3  #pragma __ADAPT_DS(GRAPH)("SHARD")
 4  #pragma __ADAPT_LONG_RUNNING("LR")
 5  #pragma __ADAPT_LOOP("AL")
 6  typedef struct { //first representation
 7    ...
 8  } ADJMAT;
 9
10  typedef struct { //second representation
11    ...
12  } ADJLIST;
13
14  typedef struct { //third representation
15    ...
16  } SHARD;
17
18  void computeMSTK_ADJMAT( ADJMAT* graph,
19          int* progress)
20  {
21     int totalNodes = graph−>totalNodes;
22     Edge* edge;
23
24     LR:{
25       AL: while(*progress < totalNodes) {
26          check4adapt(progress);
27          edge = findMinimumEdge(graph);
28          check4adapt(progress);
29          markEdgeUsed(graph,edge);
30
31          *progress = addEdgeInTree(graph,edge);
32          check4adapt(progress);
33         }
34     }
35  }
36
37  ...
38
39  callOP1(void* graph, int startDS, int* progress) {...}
```

Figure 3: Excerpt from Minimum Spanning Tree (MST-K); code in green is inserted automatically by our compiler.

adaptive application that will adapt by safely switching among representations.

Running example: Minimum Spanning Tree using Kruskal's algorithm (MST-K). In Figure 3 we show an excerpt from the MST-K application, which computes the minimum weight spanning tree (MST) using Kruskal's algorithm. The input is an undirected graph, while the output is a subgraph whose total weight is less than or equal to every other spanning tree. Three alternate data structure representations are used for holding the large graph in the memory: ADJLIST, ADJMAT and SHARDS, as indicated by the programmer on lines 1–3. The function computeMSTK_ADJMAT, as the name suggests, finds the MST in the ADJMAT representation; similar functions, for ADJLIST and SHARDS representations have been written as well. The progress argument (line 19) holds the value of a "progress bar" for the execution—in this case, the length of the longest tree in the MST. The algorithm's main part is implemented as a long-running loop LR (line 24) whose counter gets incremented each time an edge is successfully added to the subgraph represented in graph data structure. MST-K is a greedy algorithm, trying to add an edge of minimum value to the tree and terminating when the spanning tree has every node connected (an MST has been constructed); findMinimumEdge (line 27) finds the minimum weight edge in the graph which is not yet in the spanning tree and markEdgeUsed (line 29) marks that edge for possible addition. The function addEdgeInTree (line 31) checks if adding the edge to the spanning tree connects two trees and does not form a cycle, and returns the length of the longest spanning tree. The calls to **check4adapt** (lines 26, 28, and 32) and the definition of **callOP1** (line 37) are inserted automatically by our source-to-source compiler after the static analysis.

Programming model. Our approach is designed to minimize programmer's burden. To support adaptation, programmers use just four simple annotations, as shown in the following table, to indicate alternative definitions of data structures, as well as long-running code that should be subject to adaptation. Our compiler will use these annotations to generate code that adapts in a safe and flexible manner. In addition, while programmers need to write functions for converting between representations, they do not need to invoke these functions—for safety and timeliness reasons, these functions are invoked automatically by the runtime system.

Annotation	**Purpose**
progress	progress indicator
__ADAPT_DS(DSname)(ConcreteRep)	mark data type for adaptation
__ADAPT_LONG_RUNNING	long-running block
__ADAPT_LOOP	adaptive loop

First, the programmer should define a variable (named progress in our examples) which tracks execution progress, e.g., the amount of input processed or the amount of result produced. Second, the long-running computation should be prepared to start processing from a certain progress value rather than assuming it starts from scratch (note the compute MSTK_ADJMAT(ADJMAT* graph, **int*** progress) in Figure 3) so the computation can resume at the new representation after a representation switch.

__ADAPT_DS is used to mark a data type for adaptation; the DSname parameter is used as a common name to identify alternate representations, while ConcreteRep indicates the type of the concrete representation. For example, in Figure 3 we use the **#pragma** __ADAPT_DS's on lines 1–3 to indicate that ADJMAT, ADJLIST, and SHARD are alternative implementations of the same conceptual type, GRAPH.

The compute-intensive code, usually the program's main loop, is a lexical scope marked via __ADAPT_LONG_RUNNING (lines 4 and 24). This annotation indicates to our compiler that it should find adaptation opportunities in that scope (or in code transitively called from it, e.g., callees).

Programmers can annotate a loop with __ADAPT_LOOP to tell the compiler that it should find adaptation opportunities inside that loop's body, i.e., break out of the loop upon the first occurrence of **check4adapt**(progress) (**check4adapt**(progress) is actually an **if** (**check4adapt**(progress))**return**;), transfer control to the code associated with the new representation, and begin execution from the same "progress" state. In the Figure 3 example, if the runtime system indicates a switch is needed from `ADJMAT` to `ADJLIST`, we break out of the loop on line 25 and control is transferred from computeMSTK_ADJMAT to computeMSTK_ADJLIST. Loop annotations are particularly useful when programs contain nested loops: programmers can control the granularity of adaptation, i.e., loops marked with __ADAPT_LOOP will permit fine-grained adaptation.

Static analysis and compilation. Our static analysis and compiler do the heavy-lifting of carrying out safe and efficient adaptation. The static analysis, explained in Section 4.1, finds program points where adaptation can be performed while avoiding issues such as type safety violations, inconsistent intermediate results, or long adaptation delays. The source-to-source compiler is responsible for (a) inserting transition code, i.e., the code that will perform the runtime conversion between data structure representations, based on the manual annotations and the results of the static analysis; and (b) inserting potential adaptation points (**check4adapt**). These insertions are guided by pragmas, as explained next. In Figure 3, the long-running loop is marked with LR. After static analysis, a **check4adapt**(progress) adaptation check is inserted at appropriate safe points where the adaptation could be triggered. A custom function **callOP1** is added, which is responsible for converting the in-memory data structures to the new representation. All calls to computeMSTK_ADJMAT are then replaced by **callOP1** and the second parameter of **callOP1**, startDS, represents the current data structure representation. When compute-intensive code executes, at **check4adapt**(progress), it checks whether the switch is necessary and if required, the program switches to another representation and the operation is resumed from the point indicated by progress.

Adaptation policies. To specify adaptation points, programmers define a transition policy file that defines which representation should be used for which interval—the system then monitors the input/workload and triggers adaptation automatically. For example, to reduce processing time, the programmer specifies graph density intervals as shown on the left:

```
/* reduce TIME */ or /* reduce MEMORY */
ADJLIST [0,2) ADJLIST [0,25)
SHARD [2,67) ADJMAT [25,100]
ADJMAT [67,100]

/* HYSTERESIS */
TIME 2
```

To reduce memory, the programmer specifies intervals as shown on the right. We also support a hysteresis value (2 seconds in our example); the system waits for the specified time before switching, to avoid too frequent representation changes due to frequent changes in the input characteristics.

Releasing physical memory: When switching representations, after the conversion is finished, our runtime system releases the memory holding the old representation via malloc_trim (similar to application-directed releases [6]). This is particularly important when adapting in response to memory pressure, e.g., from `ADJMAT` to `ADJLIST`.

4. Static Analysis

We use static analysis for *safety* (automatically finding points where it is safe to switch representations) as well as *timeliness* (responding faster to adaptation requests). The analysis frees the programmer from worrying about adaptation's safety and timeliness—an intractable manual job for any non-trivial program.

4.1 Safety Analysis

The safety analysis prevents the computation code from using mixed data structure representations, as that would be a violation of type safety. We illustrate this on the MST-K example with an excerpt of code from function findMinimumEdge. In a nutshell, the function takes an input graph, sorts its unused edges by calling createSortedEdgeList and returns the first element of the list. We show the function and add a comment to assume we perform a data representation switch from `ADJMAT` to `ADJLIST` at line 51:

```
48  Edge* findMinimumEdge(ADJMAT* graph)
49  {
50      EdgeList* edgeList; // graph in ADJMAT representation
51      // switch ADJMAT to ADJLIST
52      edgeList = createSortedEdgeList(*graph); // type−unsafe!
53
54      return edgeList[0]−>edge; ...
```

Clearly, performing a switch at line 51 would violate type safety: since the current function's activation record (findMinimumEdge's stack layout) is set up to assume ∗graph has type `ADJMAT` and createSortedEdgeList takes an `ADJMAT` argument, performing the switch would invoke createSortedEdgeList with an `ADJLIST` argument, which is a violation of type safety—note that `ADJMAT` and `ADJLIST` differ in size and representation hence have different memory layouts.

We solve this problem by enforcing *representation consistency*, a concept originally used to enforce type safety for live program updates [24]. In particular, we use static analysis to annotate each program point with the set Δ of adaptable types used *concretely* in that point's *delimited continuation*[1] and prohibit switching to a new type when the representation assumed by the continuation contains the old type. We now illustrate the analysis by showing the analysis-inferred Δ's in the MST-K example.

```
27  edge = findMinimumEdge(graph);
48  Edge* findMinimumEdge(ADJMAT* graph)
49  {
50      EdgeList* edgeList;
51                    Δ = {ADJMAT, ...}; cannot switch
52      edgeList = createSortedEdgeList(*graph);
53                    Δ = {...}, ADJMAT ∉ Δ; OK to switch
54      return edgeList[0]−>edge;
55  }
```

On line 51, Δ contains `ADJMAT` because the code in the continuation (edgeList = createSortedEdgeList(graph)) assumes the `ADJMAT` representation. The Δ on line 53 does not contain `ADJMAT` as the remaining code in the delimited continuation does not use the graph, hence no representation assumptions are made. To construct Δ's, we have extended the static analysis in [24] to track concrete uses of adaptable data types (as they are marked with an __ADAPT_DS).

Safety condition: We can now provide our formal safety condition: a type-safe switch from representation type τ to τ' can be performed at program point n if:

$$\Delta_n \cap \{\tau\} = \emptyset$$

This check is performed statically. In our example, the condition $\Delta_n \cap \{\texttt{ADJMAT}\} = \emptyset$ is satisfied at line 53, hence our compiler will insert a **check4adapt** call to trigger a representation change if needed.

4.2 Improving Timeliness

After annotating the program with the type-safety analysis results we are left with a set of program points where a switch is safe. However, a safe switching point does not ensure timeliness, as we will illustrate shortly.

We first introduce some terminology. We name "IR" the intermediate result of the computation, e.g., the partially-constructed spanning tree in the MST-K example. We say that the IR is "dirty" if it has been modified and a representation change will require recomputing the changes made to the IR since the last increment—such recomputations are called "killing" the IR.

The key mechanism we introduce for improving timeliness is to use *contextual effects* [19] to figure out if the IR is dirty and should be killed (in other words, if the last computation increment should be discarded, or can be used before waiting for the next computation

[1] The continuation is delimited by the scope of an adaptive loop, as **check4adapt** can break out of the loop, effectively "cutting" the concrete uses in the current iteration or subsequent iterations.

increment). In a nutshell, contextual effects are sets that characterize each function and each program point. For functions, the important part of contextual effects is a set named ε that captures whether that function modifies the IR. In MST-K where the IR is stored in graph, some functions, e.g., addEdgeInTree, do write to the IR, hence we have $\{\text{graph}\} \in \varepsilon_{\text{addEdgeInTree}}$; others, e.g., findMinimumEdge, do not write to the IR, hence $\{\text{graph}\} \notin \varepsilon_{\text{findMinimumEdge}}$. The ε effects are chained together to compute, at each program point, a *prior effect* α, i.e., the effect of code that has executed so far, and a *future effect* ω, i.e., the effect of code that will execute. To explain how contextual effects help improve timeliness, we will again use the MST-K example in Figure 3. For the sake of this example, let us assume that all points in LR's loop body are type-safe so the representation can be switched at any point (of course, in practice only type-safe points will be used to improve timeliness). For each line in LR's body, the next code excerpt indicates the prior contextual effect α (which captures whether the program *has modified* graph) and ω (which captures whether the program *will modify* graph):

```
26                              α = ∅, ω = {graph}
27  edge = findMinimumEdge(graph); // doesn't modify the IR (graph)
28                              α = ∅, ω = {graph}
29  markEdgeUsed(graph,edge);
30                              α = {graph}, ω = {graph}
31  *progress = addEdgeInTree(graph, edge);
32                              α = {graph}, ω = ∅
```

If we inspect the α and ω annotations on lines 26–32 we see that it is OK to perform the representation switch at lines 26 or 28 without "killing" the graph, because the code has not yet written to graph (findMinimumEdge does not change graph). Similarly, it is OK to perform the switch at line 32 without killing the graph, because the code has written to the graph and will not perform any further writes. However, if we perform the switch at line 30, we have to kill the graph since it is dirty—it has changed and it will change. Hence we can perform a static check to determine whether the switch should kill the IR or not; in the MST-K case, $\{\text{graph}\} \notin \alpha_n \cap \omega_n$. For this, we have extended the static analysis in [19] to track writes to the IR.

Timeliness condition: We can now provide our formal timeliness condition: a type-safe switch can be performed at program point n without killing the IR if:

$$\{IR\} \notin \alpha_n \cap \omega_n$$

Safety proofs and analysis infrastructure. Our safety condition is an instance of a property called "con-freeness" while the timeliness condition is an instance of a property called "transactional version consistency". In this paper we just apply these properties — their formal definitions and proofs of correctness can be found elsewhere [19, 24]. The static analyses are inter-procedural, flow-sensitive, though context- and path-insensitive; the pointer analysis is Steensgaard [23]. The analyses and the source-to-source compiler are built on top of the Ginseng infrastructure, which can handle arbitrary C programs [18].

5. Evaluation

We evaluate our approach along multiple dimensions. We show that it is *easy to use* (off-the-shelf applications can be converted to adaptive applications with modest programmer burden), *efficient* (applications adapt quickly to changes in input or system characteristics, and their performance is nearly identical to using the best representation at all times) and imposes minimal *time and memory overhead*.

Applications. We used graph algorithms, database operations, and two real-world applications. The six graph algorithms were: Betweenness Centrality (BC) computing the importance of a node in a network; Breadth First Search (BFS), the classical graph traversal; Boruvka's algorithm (MST-B) finds the minimum spanning tree; Preflow Push (PP) finds the maximum flow in a network starting with each individual node as source; MSSP and MST-K were described in Sections 2 and 3. The alternative data structure representations were `ADJLIST`, `ADJMAT` and `SHARDS`. For database operations, we used the indexed flat file-based DBMS benchmark described in Section 2, with `AVLTREE`, `BTREE` and `RBTREE` as alternative representations. Space Tyrant (ST), an online game server, was described in Section 2.3; the alternate data structures were `LIST` and `CLIST`. Memcached (MEMC) is a high-performance object caching system used widely in the construction of high-traffic websites. The stock Memcached uses hash tables to store the objects in a key-value store; we name this representation (`JH`) after Jenkin's hash; as an alternate representation we used Cuckoo hashing (`CH`), a complete redesign of Memcached by Fan et al. [8, 20] which can deliver more than double throughput compared to stock Memcached on read-mostly workloads ($\geq 95\%$ reads).

Table 7: Application size and programming effort.

Program	Size	Step 3	Step 4				
			Annotations				Other
	(LOC)	(LOC)	DS (input)	DS (IR)	LONG_RUNNING	LOOP	(LOC)
PP	1,066	565	3	0	3	3	14
MSSP	596	”	3	0	3	3	13
BC	629	”	3	1	3	3	15
MST-K	425	”	3	0	3	3	13
BFS	506	”	3	0	3	3	10
MST-B	795	”	3	0	3	3	19
DBMS	2,566	386	3	0	3	3	6
ST	9,027	215	2	0	2	2	0
MEMC	11,722	192	2	0	2	0	216

Table 8: Static analysis results: safe adaptation points discovered (second row) and analysis time (third row).

Program	MSSP	BC	MST-K	BFS	MST-B	PP	DBMS	ST	MEMC
Safe points	3	4	3	3	4	2	4	4	5
Analysis time (sec.)	0.38	0.39	0.24	0.34	0.41	0.36	0.59	8.1	11

5.1 Effort and Safety of Manual Adaptation

Programming effort. Converting an off-the-shelf application into an adaptive application is a four-step process:

Step 1. Identify alternate representations, beyond the existing (single) representation. These alternate representations may already exist in the source code though turned off by a compiler **#define**, or off-the-shelf (e.g., as in Memcached), or have to be implemented.

Step 2. Run the application with a variety of input/workload characteristics to expose the trade-offs and construct the *Adaptation Policy*.

Step 3. Implement alternate representations' conversion functions. If a data structure has N different representations, there will be N*(N-1) conversion functions.

Step 4. Annotate the source code with pragmas, and add support for incremental computation.

We report this effort in Table 7. We assume the implementation of alternate representations is available (Step 1) hence we only focus on the programming effort for adaptation itself (Steps 3 and 4). The

Table 9: DBMS: throughput of non-adaptive versions and the adaptive version under input-triggered adaptation; values in bold represent the best representation for that workload.

	Phase	1	2	3	4	5	Overall
	Workload Breakup `%INSERT-%SELECT`	20–80	50–50	80–20	50–50	20–80	
Non-Adaptive Throughput	`BTREE` (queries/sec)	**5,847**	725	384	683	**5,151**	860
	`RBTREE` (queries/sec)	4,752	897	**492**	801	4,231	1,033
	`AVLTREE` (queries/sec)	4,315	**920**	422	**867**	3,854	980
	Adaptive Throughput (queries/sec)	5,725	895	465	843	4,977	**1,035**
	Latency (seconds)	1.57	2.12	3.43	5.24		
	Overhead(queries/sec)	122	25	27	24	175	

conversion code size ("Step 3" column in Table 7) depends on the number of alternate representations. For graph applications, the 6 conversion functions for PP amounted to a total of 565 LOC; we were able to reuse that conversion code for the rest of the graph algorithms. The 6 conversion functions for DBMS amounted to 386 LOC, while the 2 conversion functions for ST and MEMC amounted to 215 and 192 LOC, respectively.

"Step 4" code consists of adaptation annotations and support for incrementalization. For annotations (the four grouped columns in Table 7) the input data structure names have to be marked using __ADAPT_DS; indicating the IR is not required when it has the same type as the input— this was the case for 8 out of our 9 programs; for BC only, we used one annotation to indicate the IR (column 5). Identifying the long-running section was straightforward for all these applications: we had one such scope per data structure representation (hence 3 per application for graphs and DBMS, 2 per application for ST and MEMC), which we marked with __ADAPT_LONG_RUNNING (column 6). In each long-running scope for graph application, DBMS and ST, we found one loop which needed to be made adaptive, indicated via __ADAPT_LOOP (column 7).

Incrementalization and other changes. This effort is shown in the last column of Table 7. We modified the compute-intensive functions, so they could be executed in incrementalized fashion (the progress variable from Section 3). Increments are "IR units" to be completed toward the final result. Increments have two benefits: (1) enabling the runtime system to stop and start the execution from a particular state and (2) avoiding recomputation after a transition (note that killing the IR means recomputing that increment). Increments emerge naturally, e.g., in the Multiple Source Shortest Path (MSSP) algorithm which computes SSSP from each vertex to every other vertex, one unit means computing the SSSP from one vertex; similarly, in the Preflow Push (PP) algorithm which computes the maximum flow starting from each vertex, one unit means computing the flow in the network with one vertex as source. Finally, for Memcached, we had to write 216 LOC to create a new cache manager which can use either hashing technique (CH or JH); we believe this effort is acceptable, as we effectively had to merge two off-the-shelf Memcached implementations.

Note that although some of our test applications are sizable, only a very small section of code (the main data structure and the compute-intensive functions) needed to be identified and annotated. This was straightforward even though we were not familiar with the code, and we believe it is even easier for developers already familiar with the code.

Analyses' effectiveness. Finding safe and timely adaptation points manually is impractical for any nontrivial program; reasoning about safety is particularly difficult in the presence of nested loops and aliasing. Our two analyses eliminate this programmer burden: in Table 8 we show the number of safe adaptation points discovered by our analyses. All these points are type-safe; furthermore, the presence of multiple points increases adaptation timeliness (Section 5.5).

Programmer-defined adaptation points and their safety. Our static analyses find program points where representation switching is safe, i.e., type-safe and IR-safe. However, since static analyses must be conservative, we investigated if the programmer could have found better opportunities for adaptation that were missed by our analysis. For this, we manually added adaptation points where we thought it was safe to do so, and then constructed a *dynamic analysis* that traced type and IR accesses to check whether the points were really safe; after the execution, we inspected the trace to find type- and IR-safety violations. We found 1 additional adaptation point in MSSP, BFS and PP; and 2 points in BC that were missed by our static analyses. However, in MST-K (Figure 3), line 30, which we thought was safe, was actually found IR-unsafe by the dynamic analysis; of course, line 30 had already been deemed unsafe by the static analyses.

Analysis time. The "Analysis time" row of Table 8 presents the sum of static analysis and source-to-source compilation times — at most 11 seconds for our examined applications.

5.2 Dataset and System Specification

Real-world datasets. For graph applications we used real-world graphs from the Konect [14] repository. We used snapshots of MovieLens (evolving graph) from 1999 to 2004; the final snapshot has 3,979,428 edges representing reviews from 8,286 users for 28,240 movies (1.7% density). For the DBMS application, we used the data and queries from the BG Benchmark [3] (Section 2). For Memcached we used YCSB [7] to generate key-value queries. For Space Tyrant, we used a large map with various degrees of crowding and a game "controller" which drives game play by adding/removing users and generating commands for each user.

System specification. All experiments were run on a 6-core machine (Intel Xeon CPU X5680) with 24GB RAM. This system ran CentOS 5.11 with kernel version 2.6.18-398.el5. Applications were compiled with GCC 4.1.2.

5.3 Benefits of Adaptation

DBMS. In this scenario we study how adaptive applications respond to the mismatch between the data structure and workload (query) characteristics. We chose a workload size of 5,000,000 queries partitioned into 5 equal sets (execution phases) with different INSERT–SELECT ratios. The first set has 20% INSERTs–80% SELECTs, while the remaining sets have ratios 50–50, 80–20, 50–50, and 20–80, respectively. We start the program with the BTREE representation; as the workload varies, the adaptive version detects the mismatch, and switches to the most appropriate representation.

The results are presented in Table 9. Note that the adaptive version's throughput is close to the best non-adaptive version in each phase, as it adapts to the appropriate version shortly after the beginning of the phase. The overall throughput is computed by dividing the total number of queries processed in all 5 phases over total time taken to complete the phases. The overall throughput of the adaptive version is 1,035 queries/sec, virtually the same as the best performing representation (RBTREE at 1,033) and 20% higher than worst-performing representation (BTREE at 860) for the entire execution. For the last phase (phase 5), Table 9 contains no latency value as there was no representation change at the end of the execution.

To visualize the adaptation, in Figure 4 we show how the throughput varies over time around the interesting (adaptation)

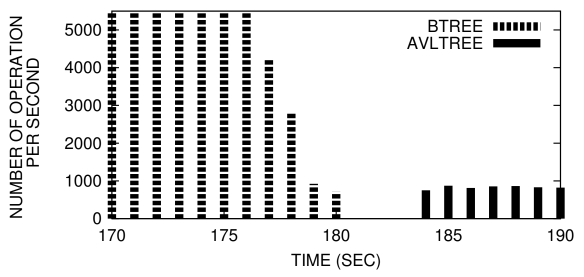


Figure 4: DBMS: query throughput before, during, and after adaptation.

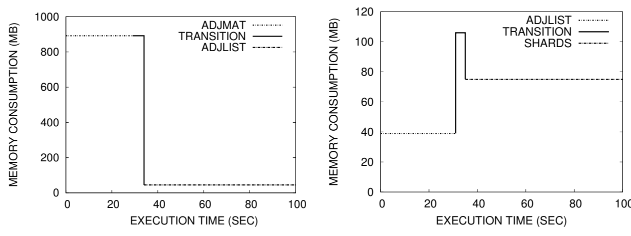
Table 10: Non-adaptive and adaptive execution times under input-triggered adaptations for MovieLens graph.

Application	**Max Density** (%)	**Non-adaptive Execution time**					
		`ADJMAT` (sec)	`ADJLIST` (sec)	`SHARDS` (sec)	**Adaptive** (sec)	**Latency** (sec)	**Overhead** (sec)
MSSP	2	1,270	**800**	910	837	43	37
	3.4	2,508	1,639	**1,415**	1,445	35	30
	2	2,103	**1,344**	1,654	1,380		36
Overall		5,881	3,783	3,980	**3,668**		103
BC	2	1,197	**691**	803	727	32	36
	3.4	2,399	1,471	**1,278**	1,311	28	33
	2	2,044	**1,204**	1,419	1,241		37
Overall		5,639	3,366	3,500	**3,286**		106
MSTK	2	430	**261**	304	298	14	37
	3.4	844	638	**511**	546	19	35
	2	771	**473**	544	504		31
Overall		2,044	1,372	1,358	**1,354**		104
BFS	2	1,394	**929**	1,041	961	22	32
	3.4	2,718	1,814	**1,695**	1,733	17	38
	2	2,323	**1,678**	1,839	1,718		40
Overall		6,436	4,421	4,575	**4,421**		110
MSTB	2	1,203	**739**	844	778	15	39
	3.4	2,375	1,534	**1,370**	1,404	18	34
	2	2,060	**1,245**	1,548	1,277		32
Overall		5,637	3,518	3,762	**3,466**		105
PP	2	72	**26**	36	35	7	9
	3.4	138	62	**58**	64	1	6
	2	186	**119**	163	125		6
Overall		395	**207**	256	230		21

region between phase 1 and phase 2, i.e., while changing to a more `INSERT`-heavy workload. Before the transition, as workload characteristics change, the throughput drops due to the mismatch between the characteristics and the current representation. During the transition, the throughput briefly drops to 0, and then after the transition to `AVLTREE`, the throughput stabilizes. The figure reveals that (1) the mismatch is detected early on during the change in workload characteristics, and (2) the transition time is low, relative to the total execution time.

Graph Applications. We use MovieLens as the evolving input graph. The final (year 2004) snapshot has 1.7% density, thus making `ADJLIST` the best representation (lowest memory consumption and execution time). However, at the end of 2002, the density was 3.4%, thus making `SHARDS` the most time-efficient representation. As initial density is less than 2%, the execution starts with `ADJLIST` in the first phase. Then, during execution, the density increases to 3.4% in the second phase, and decreases back to 1.7% in the third phase.

The result of this experiment is summarized in Table 10: for each algorithm, we show the maximum density; completion time, in seconds, for the non-adaptive and adaptive versions; the transition latency between phases (again, no transition after phase 2); and the overhead, computed as the difference between phase completion time for the adaptive version and the non-adaptive version at the same representation. As we can see, during each phase, the performance of the adaptive version is close to the best performing representation in that phase as our system selects the most appropriate version. The overall execution time for non-adaptive versions is calculated as the sum of execution times for each phase. For the adaptive version, the overall execution time is the sum of execution times for each phase, plus the time required for transitions. The overall time of the adaptive version is less than the execution times of the `ADJMAT`, `ADJLIST` and `SHARDS` by an average of 38%, 2%, and 5% respectively. For the PP application, the execution time of the adaptive version is 11% more than `ADJLIST`, and 41% less than `ADJMAT`, the worst choice. Hence, the adaptive version proves to be a better choice than using a single data structure for the entire execution. In addition, we observe that the maximum transition latency is just 43 seconds, which represents 1.1% of the total execution time for MSSP.



(a) `ADJMAT` → `ADJLIST`. **(b)** `ADJLIST` → `SHARDS`.

Figure 5: Graph applications: memory consumption before, during, and after adaptation for BC.

To visualize the adaptation, in Figure 5 we show how the memory consumption of graph applications varies over time. We consider two scenarios. First, using the 1999 MovieLens graph, we start in the `ADJMAT` representation. Since its density is low, a mismatch is detected and the representation is switched to `ADJLIST`. Second, we use a MovieLens snapshot from 2002 when it had 3.4% density. Under this scenario we start with `ADJLIST`; our system then switches to `SHARDS`, the most time-efficient representation. In both scenarios, we observe that the mismatch is detected at a very early stage and the system switches to the most efficient representation.

Space Tyrant. We study how the adaptive version quickly rectifies the mismatch between the current crowding level (Section 2.3) and the current data structure. In a real-world gameplay users join, play and leave the game. We used a controller which emulates this scenario and controlled the number of players in the game, thus maintaining the crowding value. We started the game with 1% crowding and `CLIST` representation; after 10 seconds the controller increases the crowding to 10%; after 20 seconds the controller removes players to bring crowding back to 1%. The results of this experiment are summarized in Table 11: for each time interval, we show the crowding range; total number of commands executed; the transition latency between intervals (no latency from 1 to 10 seconds as there was no transition); and the overhead, computed as the difference between the average throughput for the adaptive version and the best non-adaptive version. We can see that, in two out of

Table 11: Space Tyrant: throughput under input-triggered adaptation; values in bold represent the best representation for that phase; units for throughput and overhead are thousand commands/sec.

Time		1-10	10-20	20-30	Overall
Crowding		1%	1-10%	10-1%	
Non-Adaptive Throughput (Kcmd/s)	`LIST`	2,551	1,388	1,278	1,739
	`CLIST`	**4,975**	1,958	1,849	2,927
Adaptive Throughput (Kcmd/s)		4,970	**2,241**	**1,860**	**3,024**
Latency (seconds)			0.132	0.206	
Overhead (Kcmd/s)		5			

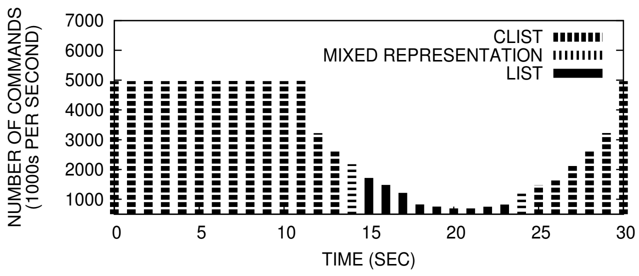


Figure 6: Space Tyrant: throughput of the adaptive version.

three time periods, the performance of the adaptive version is better than the non-adaptive version, since there is a mismatch in that time period. In the first period, from 0 to 10 seconds, there was no transition required, since the controller maintained 1% crowding. The low overhead incurred during this time period, (5,000 commands/second, i.e., 0.01%), indicates the efficiency of the adaptive version running with the same representation as the best non-adaptive version. In the second and third time periods, the adaptive version has better performance, since there was a mismatch between the crowding value and its corresponding best representation, hence the overhead is subsumed by increased performance due to switching. The overall throughput is computed by dividing the total number of commands executed in all 3 time periods by 30 seconds. Overall, the throughput of the adaptive version is 3.18% better than `CLIST` and 42% better than `LIST`.

To visualize the adaptation, in Figure 6 we show how the throughput varies during game play. Before the first transition, the throughput is decreasing as crowding is increasing from 1% to 10%. When it crosses 8%, the adaptation logic quickly detects the mismatch, and triggers the transition from `CLIST` to `LIST`. Similarly, between seconds 23 and 24, the adaptation logic detects the change in crowding, and triggers the change from `LIST` to `CLIST`.

Memcached. This application is a high-performance object cache used by sites such as YouTube, Facebook, Twitter, and Wikipedia [17]. We considered two different hashing techniques as alternate representations: Jenkin's hash (`JH`) used in stock Memcached, and Cuckoo hashing (`CH`) used in its MemC3 variant [8, 20]. Atikoglu et al. [2] have analyzed Memcached use at Facebook, and found that the distribution of request type ratios (`GET`:`SET`) range from 30:1 to 8:37. `JH` has faster `SET`s and slower `GET`s than `CH` [8, 20]. We studied the behavior of Memcached using both representations on workloads with fixed size (10 million) and found that `JH` is the better representation when the `GET` percentage is below 44% while `CH` is the better representation above 44%.

To realize the benefits of adaptation, we used YCSB [7] to generate 300 million queries, in three phases of 100 million. The `SET`–`GET` split is 70%–30% in the first phase, 30%–70% in the second phase, and 70%–30% in the third phase. Table 12 summarizes the results. For each phase, we show the workload characteristics; throughput for non-adaptive and adaptive versions; the transition latency after phases 1 and 2 (no latency for phase 3 since there was no transition). The table shows that during each phase the performance of the non-adaptive version is close to the best-performing representation in the phase, as our system quickly detects the workload characteristics and switches to the best-performing representation. The overall throughput is computed by dividing the total number of queries processed in all 3 phases over total completion time. The overall throughput of the adaptive version is 2.58% higher than `JH` and 6.45% higher than `CH`. The overhead is subsumed by the benefits of adapting to the best hashing technique in each phase; these findings clearly show the benefit of adaptation. Figure 7 shows the throughput of the adaptive version: before, during, and after the transition between phase 1 and phase 2. Before the transition the throughput drops due to representation mismatch. During the transition, the throughput drops to around 30,000 queries per second, as the transfer of key-values from one hash to another takes place. After the transition to `CH`, the throughput increases again. We conclude the following from this figure: first, the mismatch is detected early during the change in workload characteristics; second, although the throughput decreases during the transition, transition time is low compared to the total execution time; third, there is significant performance improvement after the transition.

Table 12: Memcached: throughput under input-triggered adaptation; values in bold represent the best representation for that phase; units for throughput and overhead are million queries/sec.

Phase		1	2	3	Overall
Workload Breakup (`%GET-%SET`)		30–70	70–30	30–70	
Non-Adaptive Throughput (Mqueries/s)	`JH`	**1.65**	1.27	**1.67**	1.51
	`CH`	0.95	**1.86**	0.95	1.45
Adaptive Throughput (Mqueries/s)		1.45	1.59	1.64	**1.55**
Latency (seconds)		0.001	0.001		
Overhead (Mqueries/s)		0.25	0.27	0.03	

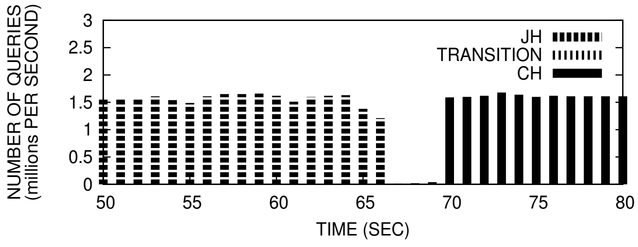


Figure 7: Memcached: throughput before, during and after transition.

5.4 Overhead of our Approach

The superior performance of the adaptive version is in part achieved because the overhead of our approach is low. The overhead of adaptation has two components: the overhead imposed by the runtime system; and the overhead of switching between the data structure representations.

Runtime overhead. We measured the overhead of the runtime system as follows. For graph applications, we computed the difference in execution time between the adaptive version and the non-adaptive version for the same data structure in the respective execution phase; the sum of the overheads in each phase gives the execution overhead

Table 13: Conversion overhead.

Application	Conversion time overhead (seconds)		Memory overhead (MB)	
	min	max	min	max
Graph	3.65	5.84	0.35	30.12
DBMS	4.25	6.54	7.62	7.62
Space Tyrant	2.84	2.84	0	0
Memcached	3.16	3.82	0.07	0.07

Table 14: Response time without/with timeliness analysis.

Analysis		Response time (sec.)						
		MSSP	BC	MST-K	BFS	MST-B	PP	Others
Safety Only	avg.	6.32	2.17	3.62	3.57	4.52	2.18	<0.001
	max	13.87	11.43	8.35	7.42	43.17	4.21	<0.001
Safety+ Timeliness	avg.	4.51	1.23	2.17	2.01	1.13	0.57	<0.001
	max	5.11	5.72	3.56	2.77	6.41	1.02	<0.001

of the runtime system for a particular graph application. For DBMS, ST, and MEMC, we found the difference in throughput between the adaptive and non-adaptive versions executing with the same representation in the corresponding phase; the average of the overhead in each phase gives the execution overhead. We note from Table 9 (last row) that the DBMS overhead imposed by our system is on average 4% of the throughput in each phase. Similarly, Table 10 (last column) shows that for all graph applications, the execution time overhead imposed by our runtime system is on average 5.2% of the total execution time. Memcached has higher overhead (15%) than other applications, as the mismatch check is performed at the end of every query. The memory overhead of the runtime system itself ranges between 9 to 23 KB, which is negligible compared to the memory used by the rest of the application.

Conversion overhead. We measured the time and memory overheads incurred during representation changes. For graph applications, we used BC running on MovieLens graph's first snapshot (1.7% density). For DBMS, we used a workload of 5,000,000 queries with 50% `INSERT`s and 50% `SELECT`s with 1,000 initial records in the database. For MEMC we used 100 million queries, 30% `GET`s and 70% `SET`s. For ST, we used the same game play as for measuring the benefits of adaptation. The memory overhead is the additional memory required to carry out the transition between data structures. The time overhead is the duration of the conversion. In Table 13 we show the minimum and maximum time/memory overheads across all adaptations. The results show that conversion time is approximately 1% of the total execution time. We also infer that although there is a memory spike (at most 30.12 MB) for some conversion scenarios, it lasts for a short time (less than 2% of total execution time). The additional memory required to handle transition for Space Tyrant is zero, as there was no additional data structure required to carry out the transition.

5.5 Timeliness Improvements due to Static Analysis

To quantify the benefits of the timeliness-improving static analysis, we ran all the benchmarks as follows: all start in the wrong representation, so an adaptation will be triggered when deemed safe. We measured *response time* as the difference between the time the input monitor has signaled an adaptation and the time when the program reaches a safe adaptation point. We performed this experiment in two settings: first, the program with the safety analysis enabled and a single adaptation point in the main loop; second, using our normal compilation scheme, with both the safety and timeliness analyses enabled, hence the compiler could discover additional adaptation points. We present the average and maximum (worst-case) response times in Table 14. On average, the timeliness analysis reduces response time by 45%. For Space Tyrant, Memcached and DBMS, (the "Others" column) this benefit is less apparent since adaptation can occur at the end of each query. For graph algorithms, however, where a long-running block consists of several compute-intensive statements, the benefit is clear, e.g., in one MST-B scenario, the response time *without the timeliness analysis was 43.17 seconds* while *with the analysis it was just 0.57 seconds*, out of a total execution time of 874 seconds.

6. Related Work

Our prior work used manual switching between two data structures on small-scale graph algorithms [15]. Therefore it lacked static analysis and compilation, which imposes high programming burden (programmers must control adaptation timing and data transformations), does not guarantee safety, and does not attempt to increase timeliness.

Brainy [9] profiles applications using alternative C++ STL data structures, to find the best data structure implementation for a certain input on a certain architecture. Chameleon [22] collects JVM profiling information on Java collections use, and uses a set of rules to find optimization opportunities (e.g., use `ArrayList` instead of `LinkedList` when `get()`s are frequent). Based on profiling results, Chameleon can recommend and implement solutions in the next run. These approaches are offline, hence do not support adaptation when input characteristics change across runs, or when available resources change during a run.

CoCo [28] allows Java container implementations to be switched at runtime depending on predefined conditions (usually container size). Programmers must identify abstraction and concretization operations; the compiler generates objects to profile and replace containers; the runtime system (JVM) effects the on-the-fly transfer between containers. We aimed to avoid manual identification of abstraction and concretization operations, because it is tedious, error-prone and discourages programmers from using off-the-shelf code [24]. Second, CoCo can replace containers that have a standard, albeit narrow interface: *ADD* or *GET*, while we allow general replacement of data structures and their associated implementation, with no interface constraints. Third, CoCo does not consider adaptation for space or the time-space trade-off.

PetaBricks [1] combines language, compiler and runtime support to specify HPC computations as sequences of rules (code regions and their input/output). The compiler generates legal rule compositions while the runtime system permits sequential or parallel tasks. An autotuner finds optimal algorithms and decompositions on a certain platform and for a certain input size. PetaBricks offers more principled support for region decomposition and region dependencies, but existing implementations have to undergo significant effort to be converted to PetaBricks programs, whereas we target off-the-shelf programs. Our approach offers algorithmic and data structure choice: the programmer just specifies alternate implementations while decisions regarding data structure selection are made on-the-fly (autotuning is not required). PetaBricks checks consistency dynamically by observing the output, whereas we enforce consistency statically.

Other compiler-enabled adaptations include altering an application's contentiousness [25, 26] so it can be colocated with other applications without interfering with their performance; and data spreading, a self-adaptation that permits application speedups by using more cache when available [10]. Runtime-enabled adaptations have mainly focused on co-locating applications that have complementary resource needs so they can run together with minimal interference [27]; or managing contention, guided by last level shared cache and lock contention [4, 5, 12, 13, 21, 29].

7. Conclusion

We show that, for data-intensive applications, the best choice of main data structure depends on input characteristics, memory availability, and operations being performed on the data structure. Since a fixed, compile-time choice of data structure is not appropriate, we introduce an approach for developing adaptive applications that dynamically switch between data structures to suit the running conditions. Our experience with a variety of applications shows that static analysis is highly effective at reducing the manual burden for locating *safe* and *timely* adaptation points. Experiments demonstrate that our approach allows applications to adapt and switch to the best representation quickly, with little runtime and memory overhead.

Acknowledgments

This work is supported by NSF Grants CCF-1318103, CCF-1524852, and CCF-1149632 to UC Riverside.

References

[1] J. Ansel, C. Chan, Y. L. Wong, M. Olszewski, Q. Zhao, A. Edelman, and S. Amarasinghe. Petabricks: A language and compiler for algorithmic choice. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '09, pages 38–49, 2009.

[2] B. Atikoglu, Y. Xu, E. Frachtenberg, S. Jiang, and M. Paleczny. Workload analysis of a large-scale key-value store. In *Proceedings of the 12th ACM SIGMETRICS/PERFORMANCE Joint International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '12, pages 53–64, 2012.

[3] S. Barahmand and S. Ghandeharizadeh. Bg: A benchmark to evaluate interactive social networking actions. In *CIDR'13*.

[4] M. Bhadauria and S. A. McKee. An approach to resource-aware co-scheduling for cmps. In *Proceedings of the 24th ACM International Conference on Supercomputing*, ICS '10, pages 189–199, 2010. ISBN 978-1-4503-0018-6.

[5] S. Blagodurov, S. Zhuravlev, M. Dashti, and A. Fedorova. A case for numa-aware contention management on multicore systems. In *Proceedings of the 2011 USENIX Conference on USENIX Annual Technical Conference*, USENIXATC'11, 2011.

[6] A. D. Brown and T. C. Mowry. Taming the memory hogs: Using compiler-inserted releases to manage physical memory intelligently. In *Proceedings of the 4th Conference on Symposium on Operating System Design & Implementation - Volume 4*, OSDI'00, pages 3–3, 2000.

[7] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking cloud serving systems with ycsb. In *Proceedings of the 1st ACM Symposium on Cloud Computing*, SoCC '10, pages 143–154, 2010.

[8] B. Fan, D. G. Andersen, and M. Kaminsky. Memc3: Compact and concurrent memcache with dumber caching and smarter hashing. In *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation*, nsdi'13, pages 371–384, 2013.

[9] C. Jung, S. Rus, B. P. Railing, N. Clark, and S. Pande. Brainy: Effective selection of data structures. In *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '11, pages 86–97, 2011.

[10] M. Kamruzzaman, S. Swanson, and D. M. Tullsen. Software data spreading: Leveraging distributed caches to improve single thread performance. In *Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '10, pages 460–470, 2010.

[11] F. Khorasani, K. Vora, R. Gupta, and L. N. Bhuyan. CuSha: Vertex-centric graph processing on gpus. HPDC '14, pages 239–252.

[12] R. Knauerhase, P. Brett, B. Hohlt, T. Li, and S. Hahn. Using os observations to improve performance in multicore systems. *Micro, IEEE*, 28(3):54–66, May 2008.

[13] K. Kumar Pusukuri, R. Gupta, and L. N. Bhuyan. Adapt: A framework for coscheduling multithreaded programs. *ACM Trans. Archit. Code Optim.*, 9(4):45:1–45:24, 2013. ISSN 1544-3566.

[14] J. Kunegis. Konect: The koblenz network collection. In *Proceedings of the 22Nd International Conference on World Wide Web*, WWW '13 Companion, pages 1343–1350, 2013.

[15] A. Kusum, I. Neamtiu, and R. Gupta. Adapting graph application performance via alternate data structure representation. In *ADAPT'15*.

[16] A. Kyrola, G. Blelloch, and C. Guestrin. GraphChi: Large-scale graph computation on just a pc. OSDI'12, pages 31–46.

[17] Memcached. Memcached. `http://memcached.org/`.

[18] I. Neamtiu, M. Hicks, G. Stoyle, and M. Oriol. Practical dynamic software updating for c. In *Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '06, pages 72–83, 2006.

[19] I. Neamtiu, M. Hicks, J. S. Foster, and P. Pratikakis. Contextual effects for version-consistent dynamic software updating and safe concurrent programming. In *Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '08, pages 37–49, 2008. ISBN 978-1-59593-689-9.

[20] R. Pagh and F. F. Rodler. Cuckoo hashing. *Journal Of Algorithms*, 51 (2), 2004.

[21] K. Pusukuri, R. Gupta, and L. Bhuyan. No more backstabbing... a faithful scheduling policy for multithreaded programs. In *Parallel Architectures and Compilation Techniques (PACT), 2011 International Conference on*, pages 12–21, Oct 2011.

[22] O. Shacham, M. Vechev, and E. Yahav. Chameleon: Adaptive selection of collections. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '09, pages 408–418, 2009.

[23] B. Steensgaard. Points-to analysis in almost linear time. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '96, pages 32–41, 1996.

[24] G. Stoyle, M. Hicks, G. Bierman, P. Sewell, and I. Neamtiu. Mutatis mutandis: Safe and predictable dynamic software updating. *ACM Trans. Program. Lang. Syst.*, 29(4), Aug. 2007. ISSN 0164-0925.

[25] L. Tang, J. Mars, and M. L. Soffa. Compiling for niceness: Mitigating contention for qos in warehouse scale computers. In *Proceedings of the Tenth International Symposium on Code Generation and Optimization*, CGO '12, pages 1–12, 2012.

[26] L. Tang, J. Mars, W. Wang, T. Dey, and M. L. Soffa. Reqos: Reactive static/dynamic compilation for qos in warehouse scale computers. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, ASPLOS '13, pages 89–100, 2013.

[27] W. Wang, T. Dey, J. Mars, L. Tang, J. Davidson, and M. Soffa. Performance analysis of thread mappings with a holistic view of the hardware resources. In *Performance Analysis of Systems and Software (ISPASS), 2012 IEEE International Symposium on*, pages 156–167, April 2012. doi: 10.1109/ISPASS.2012.6189222.

[28] G. Xu. Coco: Sound and adaptive replacement of java collections. In *Proceedings of the 27th European Conference on Object-Oriented Programming*, ECOOP'13, pages 1–26, 2013.

[29] S. Zhuravlev, S. Blagodurov, and A. Fedorova. Addressing shared resource contention in multicore processors via scheduling. In *Proceedings of the Fifteenth Edition of ASPLOS on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XV, pages 129–142.