2014 IEEE 28th International Parallel & Distributed Processing Symposium

Locating Parallelization Potential in Object-Oriented Data Structures

Korbinian Molitorisz, Thomas Karcher, Alexander Bieleš, Walter F. Tichy
Institute for Program Structures and Data Organization (IPD)
Karlsruhe Institute of Technology (KIT)
Am Fasanengarten 5, 767131 Karlsruhe, Germany
Email: molitorisz, thomas.karcher, walter.tichy@kit.edu, alexander.bieles@student.kit.edu

***Abstract*—The free lunch of ever increasing single-processor performance is over. Software engineers have to parallelize software to gain performance improvements. But not every software engineer is a parallel expert and with millions of lines of code that have not been developed with multicore in mind, we have to find ways to assist in identifying parallelization potential.**

This paper makes three contributions: 1) An empirical study of more than 900,000 lines of code reveals five use cases in the runtime profile of object-oriented data structures that carry parallelization potential. 2) The study also points out frequently used data structures in realistic software in which these use cases can be found. 3) We developed DSspy, an automatic dynamic profiler that locates these use cases and makes recommendations on how to parallelize them. Our evaluation shows that DSspy reduces the search space for parallelization by up to 77% and engineers only need to consider 23% of all data structure instances for parallelization.

***Keywords*-Parallelization; Detection; Dynamic Analysis; Tool**

I. Introduction

The multicore era promises growing computing power for those software engineers familiar with parallel programming. But currently the majority of software engineers are not parallel programming experts. Parallelization should neither be seen as a book of seven seals nor as a Pandora's Box. As Vandierendock et al. state in [1] we have to find ways to assist software engineers in parallelization to avert the next software crisis.

In this paper we present an approach that has the potential to lower the burden for many software engineers that have to deal with parallelizing legacy software. Today's automated parallelization techniques have several weaknesses: They are 1) not trusted, 2) not generally applicable and 3) produce results not comparable to an engineer [2]. This paper addresses all three aspects in the following way:

1) **Trust**: As Tournavitis et al. state in [3], it is necessary to let engineers participate in the parallelization process in order to gain their trust. It must be clear what happens and why. Our approach involves engineers and supports program understanding in four ways: DSspy detects relevant locations, provides reasons, gives parallelization recommendations, and visualizes the runtime profiles.

2) **General Applicability**: Many parallelization techniques address very specific use cases. Examples are automatic parallelization without data dependencies or with predefined numbers of iterations ([4], [5], [6]). We focus on coarse structures of object-oriented programs.

3) **Quality**: A parallelization tool should perform well in reducing the search space for the engineer and have high precision and recall rates. We show that we cut down the search space for parallelization by up to 76.92% within several minutes and our recommendations achieve a precision rate of 66.67%. Following them yields an average speedup of 2.13 on an 8-core machine.

II. Data Structure Occurrence and Behavior

We started this research by conducting an empirical study with which we explored whether general parallelization potential can be found in the runtime profile of object-oriented data structures.

Before we looked at the parallel potential we first answered the following two questions and address the aspects *general applicability* and *program understanding*: 1) What data structures are frequently used by software engineers? 2) How can we visualize data structure usage?

In object-oriented programs a data structure not only consists of data containers but also provides operations to manipulate them. Data structures provides these operations, such as `read`, `insert`, or `delete` via a defined interface. We composed a benchmark of 37 realistic programs from the two open source platforms SourceForge [7] and CodePlex [8]. Each program belongs to one of eleven different ap-

Application Domain	#Instances	LOC
File and text search (Srch)	11	1,046
Source code optimization (Opt)	16	2,048
Compression (Comp)	2	4,342
Program visualization (Vis)	57	10,712
Parser	51	17,836
Image algorithm library (Img lib)	60	41,456
Game	315	45,512
Simulation	150	63,548
Graph algorithms library (Graph lib)	184	69,472
Office software	396	151,220
Data structures & algorithms library (DS lib)	718	529,164
$\sum$	1,960	936,356

Table I
Empirical study. Distribution of benchmark programs across domains

1530-2075/14 $31.00 © 2014 IEEE
DOI 10.1109/IPDPS.2014.106

IEEE computer society

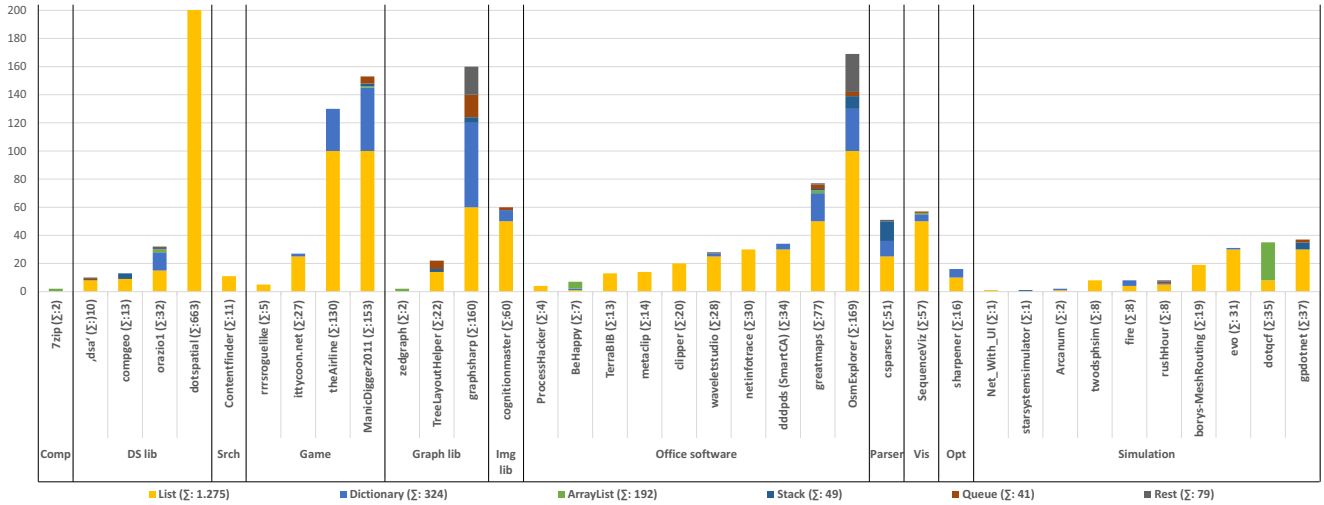

Figure 1. Data structure occurrence. X-axis: Programs with their domains. Y-axis: Number of occurrences by data structure.

plication domains and ranges from 300 to 460,000 lines of code (LOC) written in C#. We observed all data structures in the .NET standard class library and counted the number of data structure instances. Table I outlines all application domains, the number of data structure instances from the standard class library and the number of LOC.

A. What data structures are frequently used by software engineers?

We used regular expressions to gather the number of data structure instances, their locations, and their types from the Common Type System (CTS, standard class library of .NET). We covered all dynamic data structures from the CTS and `arrays`. One central finding in our study was that `list` was by far the most frequently used dynamic data structure. 1,275 of 1,960 dynamic data structure instances we found were `list` objects (65.05%), followed by `dictionary` with 324 occurrences (16.53%). Additionally to the 1,960 dynamic data structures we found 785 `arrays`. We further looked at the number of `list` instances declared within other data structures and found that every third class contained at least one `list` instance as member. This is seven times more often than `dictionary`. We also found that this ratio is independent of program size but not of application domain. Figure 1 shows the occurrence of dynamic data structures in all 37 benchmark programs. Each application domain is sorted by the absolute number of data structure instances in ascending order. Data structures with a frequency of less than 2% are not shown. These were: `hashSet` (1.94%), `sortedList` (1.02%), `sortedSet` (0.51%), `sortedDictionary` (0.41%), `linkedList` (0.15%) and `hashtable` (0.00%).

During manual code inspections we observed improper data structure usage in several cases: On the one hand we found that `lists` were used although other data structures

```
List<int> list = new List<int>(10);
for (int i=0; i<10; i++)   list.Add(i);
for (int i=9; i>=0; i--)   Debug.Write(list[i]);
```

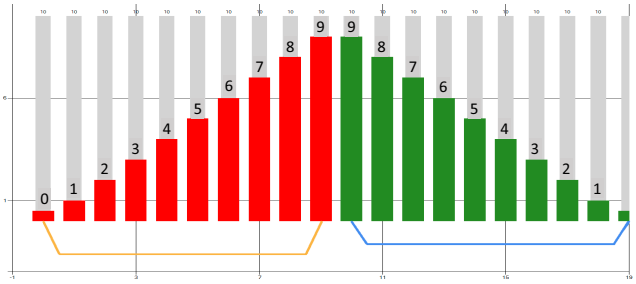

Figure 2. Runtime profile for the data structure `list` for the source code above: Each bar on the chronological x-axis represents an access event. The y-axis on the colored bars specifies the index, the grey bars indicate the overall size of the data structure.

like trees or heaps would have been better suited for the the intended purpose. On the other hand we found that `lists` were decorated to behave like other data structures from the standard library instead of simply using the existing ones. In one case a `list` was used to act like a binary tree, although binary tree implementations are available in the standard library. Both cases showed poor performance and could be optimized using the proper sequential or parallel data structure.

For the question of data structure frequency we conclude that `list` is the most frequently used dynamic data structure from the CTS. We extended our regular expression to also cover `arrays` as static data structures and state that `lists` and `arrays` account for more than 75% of all data structure instances.

B. How can we visualize data structure usage?

Visualizing data structure accesses facilitates their analysis. We use runtime profiles that contain all access events to a data structure instance from initialization to deallocation in chronological order. For a proper visualization we need to be able to capture when an access event occurs, its target location within the data structure instance, and its access type.

We developed a custom data mining tool that helped us to locate recurring access patterns within the runtime profiles. Our tool instruments source code, recompiles and executes it, captures the information, and displays it graphically so we could focus on exploration. Figure 2 shows a screenshot of this tool and the corresponding source code. In this example a list is filled with values from front to end. In a second phase these items are read in reverse order. The number at the top of each bar represents the index of the accessed element. The green bars represent read accesses, the red bars stand for write accesses. The grey bar in the background represents the length of the data structure instance at each point of access. As the list is initialized to a fixed size, the `Add()`-operations insert new elements but do not increase the size of the list. This snippet is meant as an example and it can easily be seen that the runtime profile contains two separate access patterns. Realistic programs may contain a multitude of different patterns.

III. Locating Parallelization Potential

In this section we deal with the question whether we can locate access patterns in runtime profiles and whether they can profit from parallelization. We address the following two questions: 1) What access patterns are predominant? 2) What access patterns offer parallel potential?

Application	Domain	LOC	Recurring Regularities	Parallel Use Cases
TerraBIB	Office	10,309	1	0
rrrsroguelike	Game	659	1	1
fire	Simulation	2,137	1	2
dotqcf	Simulation	27,170	2	0
Contentfinder	Search	1,046	2	2
astrogrep	Computation	846	2	3
borys-MeshRouting	Simulation	6,429	3	3
csparser	Parser	17,836	5	5
dsa	DS lib	4,099	5	0
TreeLayoutHelper	Graph lib	4,673	6	0
ManicDigger2011	Game	24,970	6	6
clipper	Office	3,270	9	5
Net_With_UI	Simulation	1,034	11	2
netinfotrace	Office	7,311	13	5
MidiSheetMusic	Office	4,792	14	7
$\sum$		72,613	81	41

Table II
Access pattern predominance. Recurring regularities on common data structures in 15 programs.

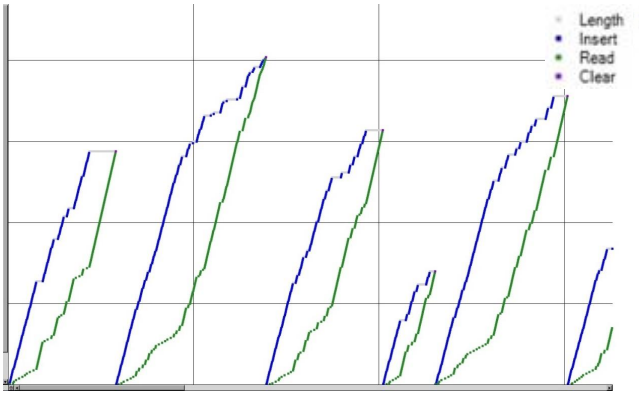

Figure 3. Visualization of a data structure access pattern that experiences index-sequential inserts and reads. The x-axis shows the temporal order and the y-axis the target index.

A. What access patterns are predominant?

In section II we described how we captured and visualized runtime profiles. As we performed manual code inspections for each access profile we only took a subset of 15 of 37 benchmark programs with a total of 72,613 LOC. At first we looked for recurring access patterns and marked the runtime profile with "contains regularity" or "contains no regularity". After that we reopened all profiles with regularities and looked at the corresponding source code to understand how the data structure instance was used. In a third step we tried to find similarities among all runtime profiles from the 15 programs and classified them manually.

Figure 3 shows regularities in the runtime profile of a `list`: The x-axis shows the temporal order of the access events and the y-axis their target position within the list. The blue line represents an insertion operation that repeatedly adds elements. The read operations are marked in green and always occur in ascending order from front to end. Along the whole runtime profile there is also a grey line that shows the current length of the list. It can hardly be seen because it is overlapped by the insertion operations. This means that new elements are always appended to the end of the list. Every time the read index reaches the last element the list instance is cleared. This **access pattern** occurs very frequently and obviously exhibits regularities. We identified 81 locations with recurring regularities like the one presented in Figure 3 and derived the following eight access pattern types. Table II shows the distribution of access patterns across the 15 programs together with the number of parallel use cases that result from them. They are introduced in the next section.

- `Read-Forward`: Read adjacent elements; access positions increases in time.
- `Write-Forward`: Write adjacent elements; access positions increases in time.
- `Read-Backward`: Read adjacent elements; access

positions decreases in time.
- `Write-Backward`: Write adjacent elements; access positions decreases in time.
- `Insert-Front`: Adjacent insert operations; always start at the front.
- `Insert-Back`: Adjacent insert operations; always start from the end.
- `Delete-Front`: Adjacent delete operations; always start at the front.
- `Delete-Back`: Adjacent delete operations; always start from the end.

B. What access patterns offer parallel potential?

After we identified 81 locations with recurring patterns and defined eight access pattern types we manually looked through all of them to explore their parallel potential. For each location we tried to figure out whether parallelization could be used successfully so that the code remained correct and yielded a speedup. This led us to eight generic **use cases**, i. e. a statement on how the data structure is used together with a recommendation on how to improve it. These eight use cases serve as an advice for the engineer and five of which deal with parallel potential. Each one contains a combination of access patterns, defines threshold values, and a recommended action.

Coming back to Figure 3 we found that the list contains two distinct access patterns and both have direct implications on parallelization: This runtime profile illustrates the access patterns `Insert-Back` marked in blue and `Read-Forward` marked in green several hundreds times. This leads to the two use cases `Long-Insert` and `Frequent-Long-Read`. Hence, the `insert` operation should be parallelized and it should be checked whether the operation that causes `Frequent-Long-Read` contains a loop that iterates over the data structure. In this case, this operation is most likely a search operation and should be parallelized.

We evaluated the five use cases with parallel potential on a subset of 23 of our 37 benchmark programs. The results are listed in Table III. We took nine programs from the same sample that we used to detect recurring patterns and added 14 programs that had not been analyzed by us before. We also used these 23 programs to tune the threshold values to yield the best detection quality. In total we detected 66 use cases. For each category we defined a **recommended action** for parallelization. The use cases, recommended actions and threshold values are listed below. Drawing the software engineer's attention to these locations and following the recommended actions not only reduces the search space for parallelization but also yields speedup as we will show in section V.

- `Long-Insert` (LI, 49 instances in 21 programs). This use case is defined by an insertion pattern from either end of a linear data structure that inserts more than one element and applies to runtime profiles which contain frequent insertion phases (>30% of runtime). An insertion phase is classified as long, if it consists of at least 100 consecutive access events.
  Recommended action: Parallelize the insert operation.
- `Implement-Queue` (IQ, three instances in three programs). A data structure is used like a queue but is implemented as a list. This use case applies to situations in which a high amount of read and write accesses (>60% in sum) affect two different ends of the data structure.
  Recommended action: Employ a parallel queue as data container.
- `Sort-After-Insert` (SAI, one instance in one program). A data structure is sorted after a long insertion phase (>30% of runtime, >100 consecutive access events). When a sort pattern follows an insertion pattern, then the insertion order is obviously not important.
  Recommended action: Parallelize both insert and search phases.
- `Frequent-Search` (FQ, three instances in two programs). The program often searches for a specific element within a linear data structure (>1000 search operations). We define search operations as frequent when at least 2% of all access events are `Read-Forward` or `Read-Backward` patterns. As we operate on lists it might be useful to change the data structure to one that is optimized for searches. Binary trees might be better suited.
  Recommended action: Either employ a parallel data structure that is optimized for searches or parallelize

Application	# LI	# IQ	# SAI	# FS	# FLR	∑
QIT	6	1			1	8
ManicDigger2011	3	1	1		1	6
csparser	5					5
clipper	4			1		5
gpdotnet	4				1	5
netlinwhetcpu	3				2	5
Mandelbrot	3					3
quickgraph	3					3
astrogrep				2	1	3
borys-MeshRouting		1			2	3
Contentfinder	2					2
DambachMulti	2					2
LinearAlgebra	2					2
MathNetIridium	2					2
Net_With_UI	2					2
fire	1				1	2
DesktopSuche	1					1
FIPL	1					1
FreeFlowSPH	1					1
networkminer	1					1
rrrsroguelike	1					1
WordWheelSolver	1					1
wordSorter	1					1
Algorithmia					1	1
∑	49	3	1	3	10	66

Table III
LISTING OF 66 USE CASES IN 23 PROGRAMS BY USE CASE CATEGORY.

the search operation in a way that splits the list into smaller chunks and search them in parallel.

- `Frequent-Long-Read` (FLR, ten instances in eight programs). >10 sequential read patterns occur repeatedly. This is similar to `Frequent-Search`: The program might be looking for an element, but more disguised, i. e. without an explicit search operation, because the existence of frequent read operations over a majority of the list elements indicates a search. In our case 50% of all access types have to be `Read` or `Search` and each pattern has to read at least 50% of the data structure in order to be classified as frequent and long. The software engineer possibly developed a search algorithm on his own, and we are unable to detect this because the source call comes from a different data structures instance. This use case might reveal a situation which contradicts our definition of a data structure that canalizes all operations via a defined interface.
  Recommended action: Check the origin of this access. In case it contains a program loop that looks for a specific element the program might profit from transforming this operation into a parallel search operation.

Besides these five use cases with parallel potential we identified three additional use cases in our study. They do not specifically address parallelization but rather serve as sequential optimizations.

- `Insert/Delete-Front` (IDF). For arrays insert and delete operations result in a relative high copy overhead at runtime, because arrays are fixed size data structures. Resizing them means that an array of the new size is allocated and all elements are copied to the new locations.
  Recommended action: If insert and delete patterns often occur in combination or alternate each other, a dynamic data structure like `list` might be better suited.
- `Stack-Implementation` (SI). When insert and delete operations always access a common end of a list, we identify this as the implementation of a stack.
  Recommended action: Analyze the data structure and think about using a stack implementation.
- `Write-Without-Read` (WWR). Our analysis of runtime profiles showed that they often end with write patterns and the results of these write accesses are never read. This is often the case when objects are cleaned up at the end of their life cycle. For example, all entries might be set to NULL. This functionality resembles garbage collection or deallocation and should be left to these mechanisms.
  Recommended action: Check if the write accesses in the corresponding profiles are necessary.

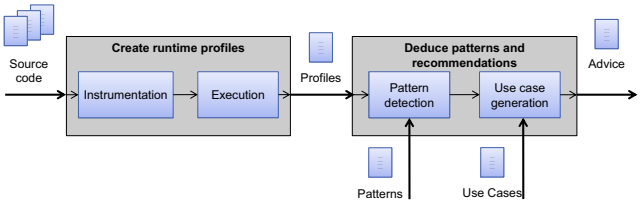

Figure 4. DSspy: Automatic deduction of use cases and recommended actions

IV. Automatic Detection of Parallelization Potential in Object-Oriented Data Structures

We implemented all findings from our empirical study in DSspy, an automatic tool that locates parallel potential in the runtime profile of object-oriented data structures and derives use cases and recommended actions on how to parallelize them. As we showed in section II, lists are the most frequent dynamic data structures from the standard class library. Apart from that is is well known that arrays are widely used. To support general applicability we implemented our tool for lists and arrays. DSspy uses static and dynamic analyses to collect the runtime profiles, to find recurring access patterns and use cases, and to deduce recommended actions. The results are then charted and presented to the engineer. Figure 4 illustrates DSspy.

Creation of runtime profiles: We process the source code of software projects using the compiler framework Roslyn [9]. At first we execute a static analysis to identify all `list` instances and `arrays` and add instrumentation statements to the beginning and the end of all interface methods. In the second step DSspy compiles the instrumented program, executes it, and starts the dynamic analysis module. This module is triggered any time an instrumentation statement is executed during program execution. We keep the execution slowdown low by only recording the access events at runtime and analyzing them post-mortem. DSspy executes the dynamic analysis module in a separate process which receives the runtime information via asynchronous intra-process communication. This design lets us bypass the typical disadvantages of file-based or in-memory log files: I/O is time consuming and for in-memory the log size can be a limiting factor. DSspy instruments and executes a full source code copy that is cleaned up after data collection so the slowdown only occurs once during the analysis.

Object-oriented data structures incorporate both, a container to store data elements and the algorithms to manipulate them. Any data interaction happens via interface methods. We classify all interactions as **access events** of different **access types** (Insert, Read, Clear ...) and bind them to their target object instances in temporal order to compose the **runtime profile** for each data structure instance. We locate recurring **access patterns** as sequences of specific access events. As already mentioned, Figure 3 visualizes

Source Code			Dynamic Analysis			Access Patterns & Use Cases			Recommendation
Name	LOC	Domain	Runtime	Profiling	Slowdown	Data Structures	Use Cases	Search Space Reduction	Total Speedup
Algorithmia	2,800	Library	0.50	2.40	4.80	16	2 of 4	75.00%	1.83
Astrogrep	4,800	File Search	4.80	5.80	1.21	21	1 of 2	90.48%	2.90
Contentfinder	290	File Search	1.80	5.20	2.89	11	2 of 2	81.82%	1.56
CPU Benchmarks	400	Benchmark	0.01	0.55	55.00	7	4 of 5	28.57%	1.20
Gpdotnet	7,000	Simulation	0.36	78.00	216.67	37	2 of 5	86.49%	2.93
Mandelbrot	150	Solver	0.11	1.20	10.91	7	4 of 4	42.86%	3.00
WordWheelSolver	110	Solver	0.04	1.50	38.46	5	1 of 2	60.00%	1.50
Total	15,550				47.13	104	16 of 24	76.92%	2.13

Table IV
EVALUATION OF DSSPY. ANALYSIS SLOWDOWN, SEARCH SPACE REDUCTION, PRECISION, AND SPEEDUP.

the access profile of a list that consists insertion and read sequences. Both overlap and occur repeatedly. DSspy gathers the following information for each access event:

- Time stamp: When did the event occur?
- Read/Write: Did the event read or write to the data structure?
- Position: What location of the data structure was accessed?
- Size: What was the size of the structure in the moment of access?
- Thread-ID: What thread raised the access event?

Deduction of access patterns, use cases, and recommended actions: After the execution of the instrumented program DSspy executes the phase detection on the access profiles as third step. All access events are assigned to their instantiation location and the access types are derived. We derive the trivial access types `Read` and `Write` and define the compound access types `Insert`, `Search`, `Delete`, `Clear`, `Copy`, `Reverse`, `Sort` and `ForAll` for each access event. The y-axis in Figure 3 symbolizes the position of the access, so the two patterns `Insert-Back` and `Read-Forward` can be obtained programmatically. We want to be able to support single- and multithreaded code so we are aware of access events that occur in parallel. In order to detect successive access events we also capture the thread id and bind it to each access event. DSspy loads the patterns specified in section III and maps them onto each runtime profile. In the final step the specified use cases and parameters are loaded and applied to the access patterns. As a result DSspy presents the access profiles, the use cases and the recommended actions to the engineer.

We decided to directly manipulate the source code and add instrumentation statements. This provides a second way of using DSspy besides the fully automatic approach that detects parallelization potential in `lists` and `arrays`: An engineer can use DSspy as a selective profiler that only analyzes instances that he manually instrumented before. Furthermore we implemented the dynamic profiler using the proxy design pattern so that it is easily extensible to runtime profiles of other data structures or use cases.

V. EVALUATION

This paper presents an approach to automatically detect parallelization potential in runtime profiles of object-oriented data structures and to derive recommendations on how to parallelize them. We assess its detection and recommendation quality by evaluating these three questions: 1) What is the search space reduction for the software engineer? How many instances can be sorted out? 2) What is the slowdown when executing the dynamic analysis? 3) How many true-positives are in the result set and 4) What is the speedup when following the recommended action? Table IV summarizes all benchmarked programs and quantitative numbers.

Experimental setup: For the evaluation we assembled a benchmark of seven programs with a total of 15,550 LOC written in C#. We took two programs from our empirical study in section II, the two established benchmarks *Linpack* and *Whetstone*, and four additional programs that had not been analyzed by us previously. With *gpdotnet* and *mandelbrot* we had two programs that contained a sequential and a manually parallelized version. This is particularly interesting because it allows us to compare the results and speedup gains from DSspy with a parallel version from a parallel software engineer. Our test system was an 8-core AMD FX 8120h with 3,1 Ghz each and 8 GB RAM.

Search space reduction: As mentioned we implemented our analysis for arrays and lists. In order to quantify the search space reduction we manually counted the number of instantiations of both data structures and set this in relation to the number of data structures that our use cases referenced. For *Algorithmia*, an engineer now only has to look through 4 specific access patterns instead of 16 data structure instances in the original program (Reduction: 75.00%). Across all seven benchmark programs DSspy decreases the number of instances for the software engineer from 104 down to 24 (Reduction: 76.92%). Apart from the net reduction the recommended action provides the additional advantage that the engineer does not only see at what location to parallelize but also why it should be transformed.

Slowdown during data collection: To measure the slowdown we wrote a tool that runs all instrumented versions ten times and computes their average execution times. Across all

benchmark programs the average slowdown factor is 47.13. *Gpdotnet* can be seen as an outlier with a slowdown of 216.67. We looked into it and found that this program makes extensive use of access patterns that take relatively long to capture in DSspy, such as `Clear`: For this access pattern all information stated in section III are gathered. Without *gpdotnet*, the average slowdown is 18.88. As the slowdown only accounts once during data capturing, this is a minor aspect.

Precision: The search space reduction quantifies the decrease of elements in the result set but does not make a qualitative statement on how good the result set in fact is. The *recall rate* states, how many of the correct use cases are in fact part of the DSspy result set. We are unable to provide this information with certainty, because we did not evaluate how many of the data structures that were not part of the result in fact yielded a speedup. But we manually looked through all 104 data structure instances and did not recognize parallel potential in them. As the main focus of this research was the feasibility and not a high accuracy we did not further investigate this information. The *precision rate* in another qualitytive measure and states, how many use cases in the DSspy result set are in fact correct. We manually looked through all 24 use cases and followed the recommended actions. We executed the parallel programs on different program inputs, calculated average speedups and classified the use cases in true and false positives. The number of true positives is shown in column "Use Cases" in Table IV. Across all parallelizations we measured an average speedup gain of 2.13. With *gpdotnet* and *mandelbrot* our benchmark contains two programs with a previous parallelization that yielded speedups of 2.88 and 2.50.

Algorithmia is a data structures and algorithms library that comes with hand-written unit tests. We selected 16 unit tests that are built to simulate typical data structure use cases. We used them as input for DSspy and received four results. The average speedup was 1.83.

- Use case one addresses the initialization of a list with random values. The recommended action tells us to parallelize the method call because the pattern `Long-Insert` was detected. The parallelization yielded a speedup of 1.35 for this location, but as the initialization is only executed once this might hardly be noticeable during program execution.
- Use case two is a `Frequent-Long-Read` and recommends to check whether this method is meant to be a search operation. If so, the search operation should be parallelized. In fact, this data structure implemented a priority queue using a list object. Each search for an element with the highest priority causes linear overhead because each element has to be traversed in sequence. We parallelized the search operation and obtained a speedup of 2.30 for a list with 100.000 elements.

Use Case 1	
Class:	`GPdotNet.Engine.GPModelGLobals`
Method:	`GenerateTerminalSet`
Position:	`120`
Data structure:	`Array<System.Double>`
Use Case:	`Frequent-Long-Read`
Use Case 2	
Class:	`GPdotNet.Engine.CHPopulation`
Method:	`.ctor`
Position:	`14`
Data structure:	`List<GPdotNET.Core.IChromosome>`
Use Case:	`Frequent-Long-Read`
Use Case 3	
Class:	`GPdotNet.Engine.CHPopulation`
Method:	`.ctor`
Position:	`14`
Data structure:	`List<GPdotNET.Core.IChromosome>`
Use Case:	`Long-Insert`
Use Case 4	
Class:	`GPdotNet.Engine.CHPopulation`
Method:	`FitnessProportionateSelection`
Position:	`68`
Data structure:	`Array<System.Double>`
Use Case:	`Frequent-Long-Read`
Use Case 5	
Class:	`GPdotNet.Engine.CHPopulation`
Method:	`FitnessProportionateSelection`
Position:	`68`
Data structure:	`Array<System.Double>`
Use Case:	`Long-Insert`

Table V
EXAMPLE DSSPY USE CASE FOR GPODOTNET

- The other two use cases were initializations without speedup.

Gpdotnet uses genetic optimization algorithms for discrete time series analyses. The five use cases comprised `Frequent-Long-Read`s (3x) and `Long-Insert`s (2x). The output of DSspy for *Gpdotnet* is shown in Table V.

- Use case one identified a `Frequent-Long-Read` in a program loop that iterates over a data structure to compute an aggregate value. The length of the data structure in this case was too short for parallelization to yield a speedup. However, for longer lists a parallelization would be well-suited.
- Two use cases referred to the same data structure but for two different reasons: It is often read and new values are often inserted consecutively. It is the main data structure for the genetic algorithm which often updates the list entries. Each value has to be found first and is then changed to the new value. The two recommendations are to parallelize both methods because the first one might be a search operation (which it in fact is) and the second one might profit from using a parallel insertion. Both patterns refer to the same data structure that has been parallelized in the manual parallelization.
- The use cases four and five also involve both access patterns and refer to a data structure that is used to generate new populations. By following the recommendation the runtime of this location could be lowered from 164ms

to 50ms (speedup 3.28), but it is executed rarely. With the computations of 100 generations we achieved a small performance increase from 2.57 minutes to 2.40 minutes (speedup 1.07).

Mandelbrot calculates the well-known fractal and displays it to the user as image. We used the resolution of 1,858 x 1,028 pixels for the result image and identified seven use cases in the program run. Four yielded a speedup and three of them referred to parallel locations that were also parallelized in the corresponding manual parallelization.

- Use case one leads the engineer to a location that had been parallelized manually with a runtime improvement from 490ms to 170ms (speedup 2.90).
- Use cases two and three both refer to a program loop that initializes an array of floating point values. They recommend to execute the initialization in parallel. This had in fact also been done in the parallel version by the use of a compiler switch. This optimization decreased the initialization time from 60ms to 34ms (speedup 1.77).
- Use case four identifies `Long-Inserts` in the image that holds the resulting fractal. The recommended action is to parallelize the operation that is responsible for the long insertions, which is the operation to create the final image. This operation yielded a speedup of 1.40.

CPU Benchmarks is a typical benchmark suite for CPU computations and combines the two commonly know benchmarks *Linpack* and *Whetstone*. This program provides a user interface to execute them. With DSspy only a relative moderate speedup of 1.20 could be achieved. To further investigate this, we manually assessed the parallel potential for *CPU Benchmarks*. We analyzed the original program and determined, what parts need to be executed sequentially and what parts might profit from parallelization. After this we determined the runtime share of both parts. This allowed us to assess the parallel potential because the lower the sequential fraction, the higher obviously the parallel potential. We performed this procedure for three other benchmark programs that yielded better speedups than *CPU Benchmarks* and compared the sequential fractions with the achieved speedups. The results are shown in Table VI. We can show that the sequential fraction in *CPU Benchmarks* is significantly higher than in the other three cases. This tells us that the moderate speedup of 1.20 is caused by the low parallel potential of this program.

Name	Sequential Runtime	Parallelizable Runtime	Sequential Fraction
CPU Benchmarks	7,600ms	460ms	94.29%
Gpodotnet	7,000ms	173,000ms	3.89%
Mandelbrot	50ms	500ms	9.09%
WordWheelSolver	55ms	140ms	28.21%

Table VI
COMPARISON OF SEQUENTIAL AND PARALLEL RUNTIME FRACTIONS IN MILLISECONDS

VI. Related Work

This paper describes a method to detect parallel potential on the level of data structure runtime profiles. To achieve this goal we used principles from the disciplines programming assistance systems, software visualization, data layout optimization, memory access analysis and automatic parallelization. The specific strengths of each discipline are outlined in Table VII.

Parallel Libraries ([10], [11], [12]): The *Task Parallel Library* (TPL), the *Pattern Parallel Library* (PPL), and Threading Building Blocks (TBB) are three examples of parallel data structure libraries and provide two different use cases for parallelization: 1) They ensure that a data structure can safely be used in a parallel program without causing parallel errors. To achieve thread safe execution, they implicitly employ a locking mechanism. This aspect addresses parallel correctness but not performance improvement. 2) They bring parallelism into the program by implementing parallel data structure operations. These operations are provided via defined interfaces that differ from former sequential versions. An engineer has to identify the relevant parts of a data structure and perform the code transformation by himself without any assistance in identifying and utilizing parallelization potential.

Automatic Parallelization ([4], [5]): The highest abstraction layer for parallelization are fully automatic approaches. Here, static and dynamic analyses are used to identify parallelizable regions. Static analyses only rely on source code statements and have no information about runtime distributions or call frequencies, so they tend to over approximate. The information we need can only be detected dynamically. In [4] Tournavitis et al. employ a dynamic analysis to derive parallel potential. The results are meant to assist engineers in code analysis and identification of parallelization potential, but the decision how to interpret and how to parallelize them are still left to the engineer.

Programming Assistance ([6], [13], [14], [15]): Parallelization can be done on different levels of abstraction. One way are libraries like *STAPL* [13] which is a collection of parallel data structures and algorithms. *PetaBricks* [6] enables specifying different parallelization strategies and algorithms, so that the runtime environment can decide which implementation fits best for the current system load. *XJava* [14] is a language and runtime extension for Java which makes it specifically easy to develop object-oriented stream programs. *MapReduce* [15] is a very specific parallel pattern that enables software engineers to implement a parallel framework in a fast and less error prone way. One common property of these research activities and main difference to this work is that engineers have to identify themselves where

	Parallel Libraries	Programming Assistance	Software Visualization	Data Layout Optimization	Memory Access Analysis	Data Structure Optimization	Automatic Parallelization	**This work**
Chronological order of data	+	-	+	o	+	-	-	o
Collection of data accesses	-	-	o	+	-	-	-	+
Detection of parallel potential	-	-	-	-	-	+	+	+
Deduction of use cases	-	-	-	-	-	-	-	+

Table VII
COMPARISON OF RELATED WORK

to parallelize. Also the source code has to be transformed manually to match the parallel constructs.

Data Layout Optimization ([16]): In this paper Zhang et al. analyze the target locations of array accesses in order to optimize cache efficiency of multithreaded programs. The authors propose a technique called array restructuring which aligns the arrays to cache lines. They monitor all arrays accesses and rearrange the arrays at runtime. For this they use hyperplanes that enable local and temporal array segmentation. We have in common the analysis of array accesses but for Zhang et al. it is sufficient to capture access targets within a certain time frame in order to restructure the in-memory representation of the array. A distinction of different access types is not necessary. For us, the evolution of access types over the whole life cycle and the interaction of different access types are inevitable information.

Memory Access Analysis ([17]): Like the former work Rane et al. also aims at performance optimization but here this is achieved by monitoring cache performance at runtime. The authors propose to collect histories of cache access times, cache-hit-ratio or other cache-related information. The engineer can identify bottlenecks from the analysis and track them back to source code locations, because the source code had been instrumented before. The main difference to our work is that Rane et al. present aggregated values to the engineer that have to be analyzed manually. We evaluate whole profiles and automatically derive recommended actions without any user interaction.

Data Structure Optimization ([18]): Jung et al. introduce the tool *brainy* which is similar to DSspy because it measures the execution times of different data structures and tries to derive optimization potential. *Brainy* follows a different approach because it executes each data structure with different inputs and uses the results to train an evolutionary algorithm. A dynamic analysis is used to generate different program inputs. *Brainy* represents a model that projects all gathered performance data onto a predefined set of implementation alternatives for a data structure. Two differences to our work are that we focus on parallelization and do not rely on the presence of certain hardware counters. We retrieve all necessary information from the runtime profiles.

Software Visualization ([19], [20]): Research activities in this field mainly deal with the proper presentation of static information or dynamic runtime data like pointer arithmetic or program execution histograms. In [20] Mukherjea et al. analyze the number of instances per class, capture what methods are invoked across the life cycle of an instance and propose an enriched call graph as form of expression. In [19] Walker et al. employ graphic animations which enable zooming into the call graph to support a deep program understanding. The main difference to our work is that here the information is only visualized but not processed any further to retrieve parallel potential.

VII. Caveats and Criticism

In this section we want to discuss potential weaknesses of our approach and threats to validity.

- **Empirical Study**: In the study we took freely available open source projects and based all of our work on them. Although a considerable amount of software is open source, it is an open question whether source code quality is as high as in closed source projects. Also, it is not clear whether the average programming skills of an open source software engineer is equivalent to the skills of a closed source engineer. If so, the data structure frequency, the number of use cases, and the recommended actions would probably be different in closed source software projects, but this does not affect our concept. Furthermore, we are sure that `lists` and `arrays` are also frequently used in closed source projects and are therefore also interesting for parallelization purpose.
  A second shortcoming of our study is that we did all our research in C#, but we do not see any reason why we would get a different picture if we conducted our study in other object-oriented environments like C++ or Java. Both, STL and JFC contain equivalent object-oriented data structures as the CTS.
- **Derivation of Recommended Actions**: For the evaluation we counted how many data structure instances were in the benchmark programs and how many in the result set. We parallelized the latter and checked which yielded a speedup and which did not. What we did not check was if the instances that were not part of our result set also yielded a speedup. We therefore cannot make a statement on the recall rate of DSspy. The motivation for this work was to research the parallel potential in the runtime profile of data structure instances and not to develop a detection mechanism with high accuracy. With our results we showed that recurring patterns and appropriate recommended actions

can be identified automatically. We will now work on improving the detection accuracy.

A second shortcoming deals with the speedup gains. We evaluated whether a manual parallelization of the recommended action yielded a speedup, so DSspy can successfully be used to parallelize legacy software. As we did not have parallelized versions for most of our benchmark programs as reference value, we cannot assess the quality of our speedups. Hence, we added two programs to our benchmark that contained a sequential and a manual parallelized version. The sequential versions were analyzed by DSspy and we compared the results with the parallel version. For both programs DSspy found all locations that had also been parallelized in the parallel version. This is not a significant statement, but a good indication.

A third shortcoming deals with the frequency of the five use cases. As Table III shows the use case frequency varies and some occur more often than others. This accounts for the empirical study and the evaluation benchmark. In 23 programs `Queue-Implementation`, `Sort-After-Insert`, and `Frequent-Search` were only found seven times in total. As we have shown in section V, the main speedup in all benchmark programs come from the remaining two use cases `Frequent-Long-Read` and `Long-Insert`. So there is reason to believe that our study revealed only special cases and not patterns of general applicability. This will be investigated in future work. But even if we identified rather special cases: They can be found in realistic programs across different application domains and they effectively lead to speedups.

VIII. Conclusion

In this paper we presented an empirical study which reveals parallel potential in the runtime profiles of object-oriented data structures. For the study we composed a benchmark of 37 programs from eleven applications domains with a total of 936,356 lines of code (LOC). The study unveils the distribution of data structures from the .NET standard class library in these 37 programs and points out that `list` is used 3.94 times more often as the second most frequent data type `dictionary`, making it by far the most frequently used data structure in open source projects.

We mined on lists and arrays for recurring **access patterns** in the runtime profile and used a subset of 15 programs with a total of 72,613 LOC. This mining lead us to eight **use cases**. Five of them permit general **recommendations** on how to parallelize accesses to the particular data structures. For the remaining three use cases the recommendations cover general code restructuring but did not involve parallelization.

For the five use cases, we automated the process in DSspy. DSspy collects runtime profiles on `lists` and `arrays` for each instance, localizes access patterns, derives use cases, provides recommendations on how to parallelize each use case, and visualizes the results to the software engineer. DSspy aims at assisting the engineer in understanding the runtime profile, where parallel potential is located and for what reason parallelization might yield a performance improvement.

We applied DSspy to 7 open source programs written in C#. DSspy revealed 24 use cases in 104 data structure instances within those programs and reduced the search space by 76.92 %. 16 of 24 use cases yielded an average speedup of 2.13 on an 8-core machine. DSspy found them within minutes with a precision of 66.67%. For now, each recommendation needs to be implemented manually; however automated transformation is possible if the recommended action is clearly specified [21]. We intend to integrate the dynamic analysis into the parallelization process published in [22].

Acknowledgment

We thank Siemens Corporate Technology for their financial support within the Shared Research Group APART. We also appreciate the support of the Initiative for Excellence at the Karlsruhe Institute of Technology.

References

[1] H. Vandierendonck and T. Mens, "Averting the next software crisis," *Computer*, vol. 44, no. 4, pp. 88–90, Apr. 2011.

[2] A. J. Ko and B. A. Myers, "Designing the whyline: a debugging interface for asking questions about program behavior," in *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, ser. CHI '04, 2004, pp. 151–158.

[3] G. Tournavitis, Z. Wang, B. Franke, and M. F. O'Boyle, "Towards a holistic approach to auto-parallelization: integrating profile-driven parallelism detection and machine-learning based mapping," in *Proceedings of the 2009 ACM SIGPLAN conference on Programming language design and implementation*, ser. PLDI '09, 2009, pp. 177–187.

[4] G. Tournavitis and B. Franke, "Semi-automatic extraction and exploitation of hierarchical pipeline parallelism using profiling information," in *Proceedings of the 19th international conference on Parallel architectures and compilation techniques*, ser. PACT '10, 2010, pp. 377–388. [Online]. Available: http://doi.acm.org/10.1145/1854273.1854321

[5] S. Rul, H. Vandierendonck, and K. De Bosschere, "A profile-based tool for finding pipeline parallelism in sequential programs," *Parallel Comput.*, vol. 36, no. 9, pp. 531–551, Sep. 2010.

[6] 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 2009 ACM SIGPLAN conference on Programming language design and implementation*, ser. PLDI '09, 2009, pp. 38–49.

[7] (2013) SourceForge. [Online]. Available: http://sourceforge.net/

[8] (2013) Microsoft Corporation, CodePlex. [Online]. Available: http://www.codeplex.com/

[9] (2013) Microsoft Corporation, Roslyn "CTP". [Online]. Available: http://msdn.microsoft.com/en-us/vstudio/roslyn.aspx

[10] (2013) Microsoft Corporation, Task Parallel Library (TPL). [Online]. Available: http://msdn.microsoft.com/de-de/library/dd460717(v=vs.110).aspx

[11] (2013) Microsoft Corporation, Parallel Patterns Library (PPL). [Online]. Available: http://msdn.microsoft.com/de-de/library/dd492418.aspx

[12] (2013) Intel Corporation: Threading Building Blocks (TBB). [Online]. Available: http://software.intel.com/en-us/intel-tbb

[13] G. Tanase, A. Buss, A. Fidel, H. Harshvardhan, I. Papadopoulos, O. Pearce, T. Smith, N. Thomas, X. Xu, N. Mourad, J. Vu, M. Bianco, N. M. Amato, and L. Rauchwerger, "The stapl parallel container framework," in *Proceedings of the 16th ACM symposium on Principles and practice of parallel programming*, ser. PPoPP '11, 2011, pp. 235–246.

[14] F. Otto, V. Pankratius, and W. F. Tichy, "Xjava: Exploiting parallelism with object-oriented stream programming," in *Proceedings of the 15th International Euro-Par Conference on Parallel Processing*, ser. Euro-Par '09, 2009, pp. 875–886.

[15] Y. Liu, Z. Hu, and K. Matsuzaki, "Towards systematic parallel programming over mapreduce," in *Proceedings of the 17th international conference on Parallel processing - Volume Part II*, ser. Euro-Par'11, 2011, pp. 39–50.

[16] Y. Zhang, W. Ding, J. Liu, and M. Kandemir, "Optimizing data layouts for parallel computation on multicores," in *Parallel Architectures and Compilation Techniques (PACT), 2011 International Conference on*, 2011, pp. 143–154.

[17] A. Rane and J. Browne, "Performance optimization of data structures using memory access characterization," in *Cluster Computing (CLUSTER), 2011 IEEE International Conference on*, 2011, pp. 570–574.

[18] 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*, ser. PLDI '11, 2011, pp. 86–97.

[19] R. J. Walker, G. C. Murphy, B. Freeman-benson, D. Wright, D. Swanson, and J. Isaak, "Visualizing dynamic software system information through high-level models," 1998.

[20] S. Mukherjea and J. T. Stasko, "Applying algorithm animation techniques for program tracing, debugging, and understanding," in *Proceedings of the 15th international conference on Software Engineering*, ser. ICSE '93, 1993, pp. 456–465.

[21] K. Molitorisz, J. Schimmel, and F. Otto, "Automatic parallelization using autofutures," in *International Conference on Multicore Software Engineering, Performance, and Tools (MSEPT'12)*, May 2012.

[22] K. Molitorisz, "Pattern-based refactoring process of sequential source code," in *Proceedings of the 2013 17th European Conference on Software Maintenance and Reengineering*, ser. CSMR '13, 2013, pp. 357–360.