Deep Parameter Optimisation

Fan Wu[*], Westley Weimer[+], Mark Harman[*], Yue Jia[*], and Jens Krinke[*]

[*]CREST, UCL, London, UK {fan.wu.12, mark.harman, yue.jia, j.krinke}@ucl.ac.uk

[+]University of Virginia, USA weimer@cs.virginia.edu

ABSTRACT

We introduce a mutation-based approach to automatically discover and expose 'deep' (previously unavailable) parameters that affect a program's runtime costs. These discovered parameters, together with existing ('shallow') parameters, form a search space that we tune using search-based optimisation in a bi-objective formulation that optimises both time and memory consumption. We implemented our approach and evaluated it on four real-world programs. The results show that we can improve execution time by 12% or achieve a 21% memory consumption reduction in the best cases. In three subjects, our deep parameter tuning results in a significant improvement over the baseline of shallow parameter tuning, demonstrating the potential value of our deep parameter extraction approach.

Keywords

Parameter tuning, parameter exposure, memory allocation

1. INTRODUCTION

Many software systems can reap significant performance benefits from workload- or runtime-specific configurations or optimisations. Software developers often expose a set of parameters for users to re-configure such software systems adaptively. However, manual parameter tuning is a demanding challenge because users are usually required not only to have extensive knowledge about the system and the workload, but also to balance many competing objectives, such as memory consumption and execution time.

Many studies have reported on the challenges of automated parameter tuning [1, 6, 15, 17, 20, 29, 30]. Early work focused on finding optimal values with mathmatical approaches [6, 20, 29, 30], while search-based software engineering (SBSE) [10] has been used in more recent research [1, 15, 17] on this problem. Although these approaches can automatically re-configure a system, their improvements are limited to changes to existing, explicit parameters.

Many software systems contain undocumented internal variables or expressions that also affect the performance of the systems. Thus, these elements could also be good candidates for automated parameter tuning. However, many of these elements are 'private', undocumented, or otherwise unexposed. Moreover, some internal values may not even be stored in variables, private or otherwise, but may merely exist as fleeting sub-expression evaluation outcomes. Identifying these variables and expressions is very difficult for general users, as it requires a deep understanding of the source code of the system.

In this paper, we propose an automatic technique to discover internal variables and expressions that normally cannot be accessed directly, but impact non-functional properties of interest. Our goal is to expose new parameters that can directly influence the values of these internal variables and expressions. To distinguish from parameters exposed by software designers (which we call 'shallow parameters'), we call these exposed parameters 'deep parameters' [13]. Modifying shallow parameter values does not necessarily change the internal code elements represented by deep parameters. Therefore, deep parameters provide additional opportunities for subsequent automated parameter tuning.

In previous work, there has been an attempt to automate the exposure of a limited form of 'deep' parameters with the Software Tuning panel for Autonomic Control (STAC) [4]. However, it requires initial human effort to characterise shallow parameters and can only find a subset of deep parameters. Hutter et al. [16], on the other hand, exposed almost all potential deep parameters and required much more computation effort. To overcome these limitations, we apply a mutation-based sensitivity analysis to fully automate the process of locating influencial deep parameters and subsequently apply NSGA-II [7] to search for optimal values for these parameters to balance non-functional properties of interest.

In this paper, we focused on two non-functional properties, memory consumption and execution time, because they are important objectives for many applications and because they are often naturally conflicting, thereby yielding an interesting and rewarding multi-objective solution space. We illustrate the approach by re-configuring a general purpose memory allocator, *dlmalloc*. We choose memory allocators because they are critical to the memory consumption of many programs and can account for up to 60% of the total execution time in some scenarios [33]. As a result, memory optimisation is a widely studied topic [25, 26]. We evaluate our approach using four specimen systems drawn from benchmarks for *dlmalloc* and real world applications. Our approach neither touches the source code of the application itself nor requires any knowledge about the application under optimisation, instead only tuning the parameters for *dlmalloc* library, making it applicable to other C applications with little effort.

The paper presents evidence that deep parameter optimisation targeting *dlmalloc* is an effective approach for improving program's non-functional properties. The experimental results

GECCO '15, July 11 - 15, 2015, Madrid, Spain

© 2015 Copyright held by the owner/author(s). Publication rights licensed to ACM. ISBN 978-1-4503-3472-3/15/07…$15.00

DOI: http://dx.doi.org/10.1145/2739480.2754648

suggest that deep parameter optimisation competes favourably with both shallow optimisation and default configurations. For four subjects, deep parameter optimisation reduces memory consumption by 21% or execution time by 12% in the best cases. The contributions of the paper are summarised as follows:

1. We introduce an automated approach to discover and expose deep parameters. The discovery of these parameters enhances search-based parameter tuning.
2. We report the results of an empirical study comparing the traditional shallow parameter tuning approach with our approach. On four applications totaling over 70,000 lines of code and guarded by over 700 tests, the results show that our approach can reduce memory usage by 21% and execution time by 12%, whereas shallow tuning alone achieves only a 16% and 10% corresponding reduction.
3. Furthermore, we evaluate the offline optimisation-time cost of our approach. For example, in our experiments, deep parameter tuning can improve memory savings by 14%, at the cost of 13% longer offline optimisation time. When deep parameter tuning is not helpful, this extra optimisation-time cost reduces to a mere 0.7%, compared to shallow parameter tuning.

2. MOTIVATING EXAMPLE

We illustrate the idea of deep parameters with an example found by our approach for *dlmalloc* (version 2.8.6) [22].

```
1  static void* sys_alloc(mstate m,size_t nb) {
2  ...
3    if (ss == 0){ //check if first through
4      char* base = (char*)CALL_MORECORE(0);
5    ...
6  }
```

Figure 1: `sys_alloc` function in *dlmalloc*

Figure 1 shows a part of the `sys_alloc` function in *dlmalloc*. We explain its internal operation here to give the reader a feeling for the opportunities for optimisation. Of course, our parameter exposing and search-based tuning are general purpose techniques that have no knowledge of how *dlmalloc* operates. *Dlmalloc* maintains an internal structure to organize the heap for memory reuse. Only when *dlmalloc* cannot find a suitable chunk of memory for a memory request does it call `sys_alloc()` to extend the current heap.

Our approach begins with a form of mutation analysis that evaluates subexpressions in the program to determine their utility as candidate deep parameters. A subexpression is evaluated by mutating it, running the resulting program variant against a test suite, and evaluating the results in terms of functional and non-functional properties. A subexpression that can be profitably mutated to optimise a non-functional property while retaining functional correctness can serve as a deep parameter.

In this example, the mutation analysis finds that mutants generated from mutating Line 4 have a notable affect on the memory consumption and the execution time of *dlmalloc*. We take a close took at Line 4. It calls the `CALL_MORECORE()` function, which takes an integer as input. `CALL_MORECORE()` is a macro wrapping the system call that extends or shrinks the current heap and returns the beginning address of the newly allocated region of heap. Specifically, `CALL_MORECORE(0)` neither extends nor shrinks the heap but simply returns the current address of the heap, which is the original purpose of Line 4 mentioned above.

Changing the input value for `CALL_MORECORE()` in Line 4 allows us to control the amount of memory pre-allocated. However, although *dlmalloc* provides several tuneable parameters to programmers, allowing them to adjust behaviours (see Section 5 for details), none of these shallow parameters can affect the `CALL_MORECORE()` function directly. Our algorithm exposes this as a new deep parameter by transforming Line 4 into the code below, where D is the deep parameter exposed that controls the pre-allocated heap.

```
char * base = (char*)CALL_MORECORE(0 + D);
```

The optimal size of pre-allocated memory depends on the specific program using this tunable memory allocator. Too much pre-allocation may result in waste. On the other hand, too little means that later requests must call `CALL_MORECORE()` again to extend the heap, increasing runtime. By tuning the deep parameter D, an SBSE approach can balance time and space consumption. This is just one example of a potential deep parameter. In our mutation analysis experiments, our tool 'discovers' that by changing the value of this deep parameter, it can achieve a modest (2.5%) time reduction without increasing heap space in one of our subjects.

3. DEEP PARAMETER OPTIMISATION

Figure 2 shows the work flow of our deep parameter optimisation. The approach takes the source code of the program, a set of test data and a set of non-functional properties of interest. It first applies mutation analysis and a non-dominated rank algorithm to discover potential locations for deep parameters, as explained in Section 3.1. It then exposes deep parameters based on the type of expressions found at the locations (Section 3.2). Finally, to tune the program, a multi-objective search algorithm is used to search for optimised values for both shallow and deep parameters (Section 3.3).

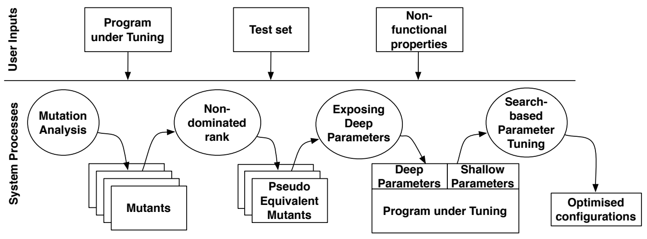


Figure 2: Deep parameter optimisation workflow. Given a program, test suite and non-functional properties, our approach applies mutation analysis, exposes deep parameters, and optimises them.

3.1 Discovering Locations for Deep Parameters

The first step is to identify potential locations at which we could expose deep parameters. In our approach, we represent the input program as an Abstract Syntax Tree (AST) and a potential location L is an expression node of the AST. We want to find a set of locations L_D such that, when we tune the value of the expression at L_D, some non-functional properties of the program could be improved while the program retains identical functional behaviour. We use a suite of regression test data to validate the correctness from the optimisation, following other established Genetic Improvement approaches [11, 21, 24]. We assume the presence of a test suite with each target application.

We use mutation analysis to automate the process of searching for locations L_D. Mutation analysis deliberately makes simple syntactic changes to the input program, to create a set of various versions of a program called mutants, each containing a different syntactic change [19]. A transformation rule that generates a mutant from the input program is known as a mutation operator.

Table 1: Selected mutation operators

Mutation Operators	Changes Between
CRCR – Constant replacement	constants, 0, 1, -1
OAAN – Arithmetic operator	`+`, `-`, `*`, `/`, `%`
OAAA – Arithmetic assignment	`+=`, `-=`, `*=`, `/=`, `%=`
OCNG – Logical context negation	*expr*, `!`*expr*
OIDO – Increment/decrement	`++x`, `--x`, `x++`, `x--`
OLLN – Logical operator	`&&`, `||`
OLNG – Logical negation	x `op` y, x `op` `!`y, `!`x `op` y, `!`(x `op` y)
ORRN – Relational operator	`>`, `>=`, `<`, `<=`, `==`
OBBA – Bitwise assignment	`&=`, `|=`
OBBN – Bitwise operator	`&`, `|`

By carefully choosing mutation operators, we can use mutants to simulate the effect of making changes at those locations Ls from which a potential deep parameter may be exposed. Table 1 lists the operators we used to generate mutants, covering locations of constants, relational, logical and arithmetic expressions. These so-called 'selective' mutation operators have been widely used in mutation analysis experiments [19].

To assess the quality of a mutant, we test each mutant against the input test set and record the values of the non-functional properties. If the functional result of running a mutant is different from the result of running the original program for any test data in the input test set, then the mutant is said to be 'killed', otherwise it is said to have 'survived'. After all mutants are executed, we first filter out the killed mutants which fail to retain the functional behaviour. A mutant is called **pseudo equivalent** with respect to a given test suite T iff. it passes the regression test of T. Thus we only select pseudo equivalent mutants which preserve the functional behaviour of the original program while potentially changing non-functional behaviour.

In practice, there is a large number of pseudo equivalent mutants [23, 28, 31] generated. We desire a subset from them that represents the locations that could have the greatest impact on the non-functional properties of interest while also maintaining a diversity of choices. We achieve this by ranking the mutants based on their non-functional properties using the non-dominated sorting approach of the NSGA-II algorithm [7]. Each mutant is assigned a Pareto Level value and a Crowd Distance value, where Pareto Level n means a mutant will be on the Pareto Front after all the mutants with Pareto Level less than n are removed, while Crowd Distance indicates how close a mutant is to its neighbours on the same Pareto Level. For example, a mutant with Pareto Level 1 is on the Pareto Front among all the mutants and has the priority to be considered first. A mutant is better than another in terms of non-dominated sorting if its Pareto Level is smaller or if their Pareto Levels are the same but the former is less crowded (larger Crowd Distance) than the latter. After sorting all the mutants in terms of their non-functional properties, we apply a greedy algorithm to pick the first k locations that could best influence the non-functional properties of the original, where k is the desired number of deep parameters one wants to expose.

3.2 Exposing Deep Parameters

The second step is to expose deep parameters that allow users to modify the value of the expression at selected locations. Based on the type of mutation, we first classify the selected mutants into two sets. Set 1 contains mutants generated from CRCR, OAAN, OAAA and OIDO operators, which simulate locations with non-logical expressions. Set 2 contains mutants generated from the OCNG, OLLN, OLNG and ORRN operators, which simulate locations with logical expressions (Table 1). Given a location L, E_L is the expression at the location L, we use the following transformation rules to rewrite E_L with a new parameter v_L.

$$E_L \rightarrow \begin{cases} (E_L + v_L) & \text{if } L \in \text{Set 1} \\ (E_L)\ xor\ v_L & \text{if } L \in \text{Set 2} \end{cases} \qquad (1)$$

We use addition to affect the value of non-logical expression and `exclusive or` to affect the logical ones. Finally we expose v_L as a 'public' parameter so that users can assign a value to v_L through parameter passing or APIs.

3.3 Search-based Parameter Tuning

Although the exposed deep parameters can provide additional 'knobs' [15] to tune the program, a set of k deep parameters need not necessarily subsume the existing shallow parameters of the program. Thus, in this work, we propose to use both shallow parameters and deep parameters and tune them together using SBSE [10]. Because we are interested in multiple conflicting properties, we consider this as a multi-objective optimisation problem, thus a multi-objective Genetic Algorithm, NSGA-II [7], is applied to search for optimal values for both shallow and deep parameters.

We use an integer vector to represent the tuning parameters. Each integer stores a solution value for one parameter. At each generation, our NSGA-II implementation first applies tournament selection, followed by a uniform crossover and a uniform mutation operation. In our experiments our fitness functions are designed to capture two non-functional properties: execution time and memory consumption, while preserving the functionality by assigning the worst value to both non-functional properties. To measure execution time, *Glibc*'s *wait4* system call is used to calculate the CPU time (mean of 10 evaluations). For memory consumption, we instrumented the program to record the high-water mark of the virtual memory consumption. We chose this instrumentation approach because the physical memory reported by the OS is not always deterministic but depends on the workload and the OS, and because the virtual memory requirement is an upper bound of the physical memory actually needed. For a subject program with configuration c, we measure the execution time $t_i(c)$ and the high-water-mark of memory consumption $m_i(c)$ of each test case i. Then the fitness functions for the configuration c regarding execution time and memory consumption can be formulated as:

$$f_t(c) = \sum_i t_i(c) \qquad f_m(c) = \sum_i m_i(c).$$

After fitness evaluation, a standard NSGA-II non-dominated selection creates the next generation. Finally, all non-dominating solutions in the final population are returned.

4. EXPERIMENTS

To assess the improvement of our Deep Parameter Tuning approach, we compared it with Shallow Parameter Tuning:

RQ1 How much performance improvement, with respect to the unmodified program, can be obtained by Shallow Parameter Turning using random search or NSGA-II?

We consider RQ1 to provide a baseline result against which we compare the results from Deep Parameter Tuning. We used NSGA-II algorithm (described in Section 3.3) and Random algorithm to search for better values for the shallow parameters in *dlmalloc*, then compare the performance with *dlmalloc*'s default configuration.

RQ2 How much additional improvement can be achieved by our Deep Parameter Tuning algorithm compared with Shallow Parameter Tuning alone?

We ask RQ2 to evaluate how useful our approach is at finding better configurations for the given non-functional properties. In these experiments, our Deep Parameter Tuning approach uses a custom mutation analysis to identify the most sensitive parts of the program, followed by an application of NSGA-II to optimise both explicit and implicit parameters for *dlmalloc*.

RQ3 What are the optimisation-time costs for these approaches to find their solutions?

Since our Deep Parameter Tuning approach exposes additional parameters which are then optimised in conjunction with the baseline shallow parameters, it may require extra resources at optimisation time. We thus measure the baseline cost of Shallow Parameter Tuning, as well as the extra computation required by our Deep Parameter Tuning. The user may use this gain/cost ratio to decide whether to employ Shallow or Deep Parameter Tuning.

4.1 Experiment Target

Many memory allocation strategies and managers have been proposed and studied by many researchers to efficiently manage dynamic memory. Among them, Doug Lea's malloc (*dlmalloc*) is "among the fastest, most space-conserving, tunable, and portable general purpose allocators" [22]. A study of Berger et al. [3] shows that many other custom memory allocators do not perform significantly better than *dlmalloc*, and are sometimes worse. We focus on *dlmalloc* as an indicative starting point and optimise its configuration to each of the subject applications.

Dlmalloc is a general memory allocator for C programs. Although it provides a number of configuration parameters, it is usually used with its default values. We call these configurable parameters provided by the original author shallow parameters. In these experiments we consider the nine shallow parameters that are more relevant to the tradeoff between runtime and memory high-water-mark. Table 2 briefly describes these shallow parameters.

4.2 Experiment Setup

For our evaluation, we selected four applications: *espresso*, *gawk*, *flex* and *sed*. *Espresso* is a fast application for simplifying complex digital electronic gate circuits. We use the *espresso* benchmark source code and test cases from the *DieHard* project [2]. *Gawk* is the GNU *awk* implementation for string processing. We collect Version 4.1.0 of this application, as well as its test suite, from the GNU archives. *flex* is a tool for generating scanners, programs which recognizes lexical patterns in text, and *sed* is an editor that automatically modifies files given a set of rules. We obtain these last two programs and corresponding test suites from the SIR repository [8]. Summary data for these subject programs is listed in Table 3.

Table 3: Subject applications

Name	Loc	# Tests	Description
espresso	13256	19	Digital circuit simplification
gawk	45241	334	String processing
flex	9597	62	Fast lexical analyzer generator
sed	5720	362	Special file editor

4.3 Experiment Procedures

We first used the shallow parameters only and applied the NSGA-II algorithm with a population of 50 for 300 generations, using 5000 randomly generated chromosomes as seeds. These standard values were chosen after a few trial experiments to have the best performance and ensure convergent result for the algorithms. A random search was also applied with the same computation budget in optimising the shallow parameters.

We used the open source C mutation testing tool MILU [12, 18] to automatically generate mutants from the selective operators shown in Table 1. This mutation based pre-analysis finds the equivalent mutants that are sensitive to the non-functional properties under optimisation. These equivalent mutants are transformed and exposed into 9 deep parameters (the same number as the provided shallow parameters for a fairer comparison) for each subject program separately, as described in Section 3. Combining shallow and deep parameters, we again applied NSGA-II and random search with other identical experimental settings. All experiments were repeated for 20 runs to admit statistical analyses.

All experiments were carried out on desktop machines with a quad-core CPU and 7.7 GB memory runing 64-bit Ubuntu 14.04. We used *dlmalloc* version 2.8.6, which was compiled with gcc 4.8.1 with -O3 option. To capture the execution time and memory consumption precisely, we developed our own performance tool to measure the CPU time and the high-water-mark vitural memory consumption (see Section 3.3). The tool is publicly available at https://github.com/FanWuUCL/memory.

5. RESULTS

We formalise the metrics we use to compare multi-objective optimisation approaches in this section. The results are presented in Section 5.2, and are used to answer the **RQ**s.

5.1 Metrics

To investigate RQ1 and RQ2, we collect the non-dominated set of solutions from each algorithm for 20 runs, and report it in an attainment surface as introduced by Fonseca [9]. To quantitatively compare the quality of each algorithm, we calculate Hypervolume and Contribution indicators to assess the multi-objective Pareto Front.

Hypervolume: The Hypervolume indicator [32] measures the space dominated by the solutions. It is defined as the hypervolume of the union of hypercubes dominated by each solution on the Front. The bigger the Hypervolume is, the larger the area dominated by the Pareto Front in the objective space is, and thus the better the performance is.

Contribution: Since there is no way to know the true Pareto Front, we use the non-dominated set of joint solutions from all experiments to approximate the true Pareto Front, forming a 'reference' front. The Contribution indicator represents the ratio of solutions on the reference front that are found by a given algorithm. A higher ratio indicates a more successful search.

To allow comparison across subject programs, objectives are normalised to the original performance of each subject.

5.2 Answers to RQs

For brevity we use *Sha* to refer to shallow parameters and *All* to refer to all parameters including shallow and deep parameters, followed by *Rand* or *NSGA* to indicate the search method used

Table 2: *dlmalloc* selective shallow parameters made available by the developers and used in our experiments

Name	Default	Range	Type	Description
MALLOC_ALIGNMENT	$2{*}\mathit{sizeof}(\mathit{void}{*})$	$(1-16){*}\mathit{sizeof}(\mathit{void}{*})$	$2^n{*}\mathit{sizeof}(\mathit{void}{*})$	Alignment unit
FOOTERS	*false*	*true* or *false*	boolean	Additional information of each chunk
INSECURE	*false*	*true* or *false*	boolean	Secure check
NO_SEGMENT_TRAVERSAL	*false*	*true* or *false*	boolean	Traversal of chunks before coalescing
MORECORE_CONTIGUOUS	*true*	*true* or *false*	boolean	Contiguous heap extension support
DEFAULT_GRANULARITY	0	4 KB – 512 KB or 0	2^n KB or 0	Unit of heap extension
DEFAULT_TRIM_THRESHOLD	2048 KB	64 KB – 16 MB	2^n KB	Threshold of trimming
DEFAULT_MMAP_THRESHOLD	256 KB	16 KB – 2 MB	integer	Threshold of direct memory mapping
MAX_RELEASE_CHECK_RATE	4095	1000 – 10000	integer	Frequency of coalescing

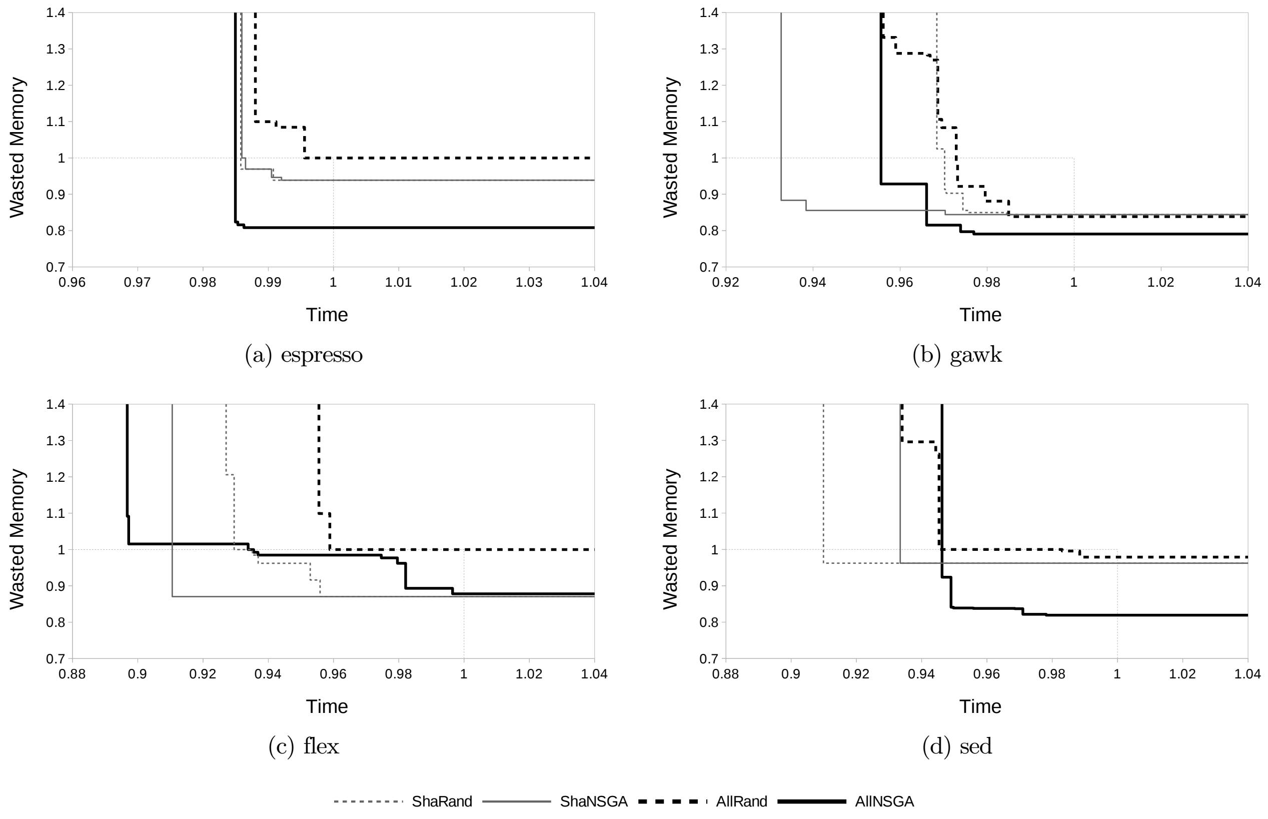


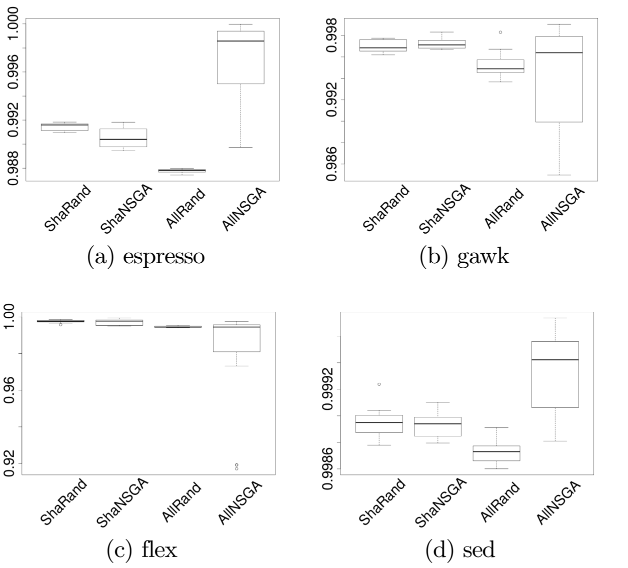
Figure 3: Combined best solutions from the results of *ShaRand*, *ShaNSGA*, *AllRand*, *AllNSGA* over 20 runs for each application. Lower and lefthand solutions dominate high and righthand solutions. 'Wasted' memory is memory that is used but not needed.

(random search or NSGA-II). For example, *ShaNSGA* refers to using NSGA-II to search for better values for shallow parameters.

To answer RQ1 and RQ2, we first report the 0%-attainment surfaces (the 'reference front' that combines best solutions over all runs) of the results of *ShaRand*, *ShaNSGA*, *AllRand* and *AllNSGA* on all subjects in Figure 3. The solutions are plotted according to their execution time and memory usage (at the 'high-water-mark') compared to the original performance. Specially, the original always lies at (1, 1) and is pinpointed by light grey dashed lines. The high-water-mark is our primary target since the remaining non-wasted memory is needed and thus cannot be reduced. The figure shows that all algorithms can reduce time or memory consumption without reducing the other objective, implying that the default configuration of *dlmalloc* is not optimal for any application considered. This finding motivates the use of SBSE for tuning memory allocators. In three subjects (*espresso*, *gawk* and *sed*), *AllNSGA* outperforms the other three on memory objective. In terms of time, no algorithm is strictly better and each has its own strengths on different subjects.

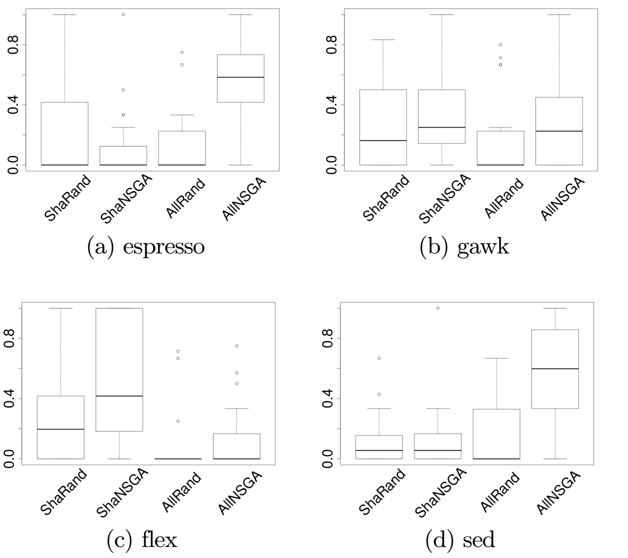
We calculated the Hypervolume and Contribution indicator of each algorithm on every subject, and report them in Figure 4 and 5 respectively for all 20 runs. In Figure 4, all the values are normalised to the hypervolume of the 0% attainment reference front, and the closer the value to 1 is, the better the result is. It is clear that *AllNSGA* outperforms the others on subject *espresso* and *sed* while it performs poorly on subject *flex*, and on subject *gawk* the best value reached by *AllNSGA* is better than that of the others. In terms of Contribution, the performance of all algorithms is similar to that of Hypervolume. In general *AllNSGA* is no worse than other algorithms on all subjects but *flex*, where *ShaNSGA* has the highest Contribution value.

Since *AllNSGA* is good at finding better performance on memory consumption, we report the most memory-saving performance found by each algorithm of each of 20 runs in Figure 6. On subject *espresso* and *sed*, *AllNSGA* finds more memory



(a) espresso (b) gawk (c) flex (d) sed

Figure 4: Hypervolume indicator of *ShaRand*, *ShaNSGA*, *AllRand*, *AllNSGA* on all subjects. Larger values are better.



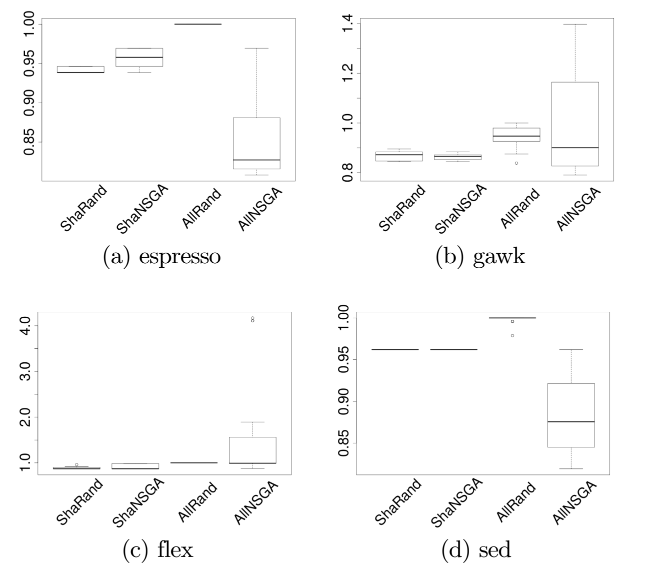
(a) espresso (b) gawk (c) flex (d) sed

Figure 5: Contribution indicator of *ShaRand*, *ShaNSGA*, *AllRand*, *AllNSGA* on all subjects. Larger values are better.

reduction than the other approaches. On *gawk*, it does not perform as consistently, but can also find more memory reduction than other approaches in the best case.

Inferential statistical tests were applied to the Hypervolume, Contribution and Best-Memory-Reduction results over all subjects. We used the Mann-Whitney-Wilcoxon U-test since we make no assumptions about results distributions and apply a Bonferroni Correction (catering for 16 total statistical tests) to draw conservative conclusions with no risk of Type 1 error. For those p-values less than $0.05/16 = 0.003125$, we apply the Vargha-Delaney ($\hat{A}_{12}$) effect size measure (see Table 4). The effect sizes are all large (either above 0.79 or below 0.21).

In all experiments involving *All** we generated and evaluated invalid configurations (i.e., those that that cause the program to crash). However, this issue is not specific to our deep-parameter



(a) espresso (b) gawk (c) flex (d) sed

Figure 6: The least memory consumption found by each algorithm. Smaller numbers are better.

approach: surprisingly, even by just tuning the programmer-specified shallow parameters (*ShaRand* and *ShaNSGA* optimisations) we *also* encounter (and discard) some configurations that crash the program. This suggests that SBSE memory allocator tuning can be used as a search based testing technique [14]. Without any guidance, *AllRand* finds valid configurations less often than *ShaRand*, and thus requires more optimisation time than *ShaRand*. Holding the searches to the same budget means that *AllNSGA*, which must explore a higher search space, will exhibit a higher variance. Despite this more challenging search space, exposing and optimising deep parameters still allows *AllNSGA* to find better configurations than *ShaNSGA*.

To enable a more quantitative look at maximal time and memory savings, we examine the extreme performance observed in our experiments. We report those that have the best performance on one objective, even at the cost of reducing performance on the other objective, found by each algorithm on each subject and summarise them in Table 5. Some of these results are significant departures from the original and are thus not plotted in Figure 3.

To answer RQ3, we provide the average optimisation computation time for each of the apporaches in Table 6. Recall that *AllRand* generates and evaluates numerous invalid configurations. However, since crashing or incorrect mutants can be discarded immediately, the computation time of *AllRand* is the lowest among all approaches (given a fixed budget in terms of mutants considered). Similarly, *AllNSGA* generates invalid configurations more often than *ShaNSGA*, so it costs less computation time than *ShaNSGA*. Taking the deep parameter discovery time into account, *AllNSGA* requires slightly more time than *ShaNSGA* does, and the percentage of the extra computation time is reported in the last column of Table 6. Ultimately, *AllNSGA* requires at most 18% more computation time than *ShaNSGA* (on *espresso*), but requires only 0.7% more computation time on *flex*, on which *AllNSGA* does not perform as good as *ShaNSGA*. Overall, since this optimisation step is a compile-time rather than run-time cost and can be done before deployment, we view the benefits of deep parameter optimisation as significantly outweighing their slight additional optimisation time cost.

Table 4: Vargha-Delaney effect sizes of Hypervolume, Contribution and Best Memory Reduction for any two of the approaches on all subjects. Only the effect sizes of tests with p-value less than $5\%/16 = 0.3125\%$ are reported.

Comparing Approachs		Hypervolume				Contribution				Best Memory Reduction			
		espresso	*gawk*	*flex*	*sed*	*espresso*	*gawk*	*flex*	*sed*	*espresso*	*gawk*	*flex*	*sed*
AllNSGA	*AllRand*	1.000	–	–	0.975	0.859	–	–	0.835	0.000	–	–	0.000
	ShaNSGA	0.935	–	0.105	0.808	0.868	–	0.191	0.868	0.063	–	0.950	0.050
	ShaRand	0.900	–	0.035	0.785	0.814	–	–	0.875	0.100	–	0.979	0.050
AllRand	*ShaNSGA*	0.000	0.053	0.038	0.045	–	–	0.144	–	1.000	0.940	1.000	1.000
	ShaRand	0.000	0.070	0.000	0.040	–	–	–	–	1.000	0.928	1.000	1.000
ShaNSGA	*ShaRand*	0.198	–	–	–	–	–	–	–	0.800	–	–	–

Table 5: Best reduction of time or memory (separately) found by each algorithm

Subject	Time Original (s)	Time Reduction (%)				Memory Original (Peak/Wasted KB)	Wasted Memory Reduction (%)			
		ShaRand	*ShaNSGA*	*AllRand*	*AllNSGA*		*ShaRand*	*ShaNSGA*	*AllRand*	*AllNSGA*
espresso	7.24	1.4	1.4	1.5	1.5	3500/521	6.1	6.1	0	19.2
gawk	3.43	3.2	6.7	4.4	4.4	29680/3552	15.6	15.6	16.2	20.9
flex	0.13	7.9	10.0	6.2	11.6	10816/525	13.0	13.0	0	12.2
sed	0.25	9.4	7.0	7.0	5.4	7048/948	3.8	3.8	2.1	17.9

Table 6: Computation Cost in Time

Subject	Optimisation Time (h)				Exposing Time (h)	Extra Time Needed for **NSGA* (%)
	ShaRand	*ShaNSGA*	*AllRand*	*AllNSGA*		
espresso	39.7	46.4	9.0	39.3	12.5	18.5
gawk	22.7	18.4	13.9	16.4	5.4	11.7
flex	7.7	6.3	5.3	5.0	1.3	0.7
sed	9.4	7.6	5.9	6.6	1.9	12.6

5.3 Threats to Validity

Internal Validity When exposing deep parameters, we used a mutation-based sensitivity analysis because of its advantages in terms of efficiency and automation. Whether it is the best way to expose deep parameters remains to be proven. In addition, we have not formally investigated the relative merits of the Mutation Operators used. Intuitively, our Mutation Operators change a constant or an operator in an expression, and thus are likely to change the values of expressions to different degrees, allowing us to capture the sensitivity of that program's non-functional behaviour to the value of that expression. Any lack of efficacy of these Mutation Operators at capturing sensitivity information introduces a threat to the effectiveness of our approach. A formal evaluation of mutation operators for deep parameter tuning remains as future work.

Another threat to the internal validity is that the execution time measured may depend on the workload of the machine. We mitigate this threat by averaging the execution time of 10 trials on an otherwise-unloaded machine.

External Validity Our choice of benchmark programs and their associated test suites influences the generality of our results. Even a good test suite that achieves high branch coverage, for example, could still differ from real world inputs, in which case the optimised configuration over this test suite may neither achieve the best performance nor retain required functionality. We attempt to mitigate this threat by including two subjects (*flex* and *sed*) from the SIR repository [8]. These subjects come with sets of high quality test suites, which achieve multiple adequacy testing criteria.

Another aspect of generality is whether these results hold on other applications. We attempt to mitigate this threat by selecting subject applications from different fields, but our results may not generalize beyond these benchmarks.

6. RELATED WORK

State-of-the-art dynamic memory managers (DMM) usually combine several different allocation strategies to serve memory requests with different sizes. Risco-Martín et al. [27, 5] search for the best allocation strategy for different sizes as well searching for the best range of sizes on which each strategy should be applied. Their work requires human effort to implement the allocation strategies. In our approach, changing the parameters not only (indirectly) changes the separators of size ranges and allocation strategy applied on each range, but also influences strategy behaviour.

ParamILS [17] is an automatic framework proposed by Hutter et al., which automatically configures an algorithm's parameters to optimise performance on a given test suite. While targeting parameter tuning as well, our approach focuses on standard library code, based on the assumption that the general-purpose memory allocators may not be optimal for each specific application. In addition, *ParamILS* can only optimise existing (shallow) parameters, while our approach exposes additional parameters and adjusts their values to gain more improvement.

Hoffmann et al. [15] proposed *PowerDial*, a system which dynamically adjusts an application's behaviour to make it adaptable to fluctuating workloads. Whenever *PowerDial* detects a resource shortage it sacrifices output quality by changing the values of variables, allowing the application to 'survive the crisis'. One limitation of this work is that the search space of configuration variables must be small enough to admit an exhaustive search. In our work the search space is too large to use such an exhaustive search, and thus we applied search-based techniques. *PowerDial* only operates on existing (shallow) parameters.

Hutter et al. [16] have tuned the parameters of a SAT solver, SPEAR, by adjusting not only the explicit parameters but also many implicit parameters. They expose almost all possibly tunable variables and thereby a much larger search space than we do. To reduce the computation effort by limiting the search space, we use a mutation-based technique to find the most influential parts of the code and focus the search on just them. This sensitivity analysis effectively reduces the space, admitting practical searches.

In previous work, the Software Tuning panel for Autonomic Control (STAC) [4] automated the expsoure of a limited form of 'deep' parameters. Although STAC can discover some deep parameters effectively, it suffers from two limitations. First, STAC requires initial human effort to characterise shallow parameters. Second, STAC can only find a subset of deep parameters, those that have similar data transition patterns to the known shallow parameters. To overcome these limitations, we apply a mutation-based sensitivity analysis to fully automate the process of locating potential deep parameters and subsequently apply NSGA-II to search for optimal values for these parameters to balance non-functional properties of interest.

7. CONCLUSIONS

In this paper we propose an automatic algorithm for discovering and optimising deep parameters to tune programs with respect to non-functional properties. In particular, we focus on tuning *dlmalloc*, a memory allocator, to reduce the time and memory high-water-mark requirements of off-the-shelf programs. Our approach combines mutation analysis to discover sensitive deep parameters as well as an SBSE approach which subsequently optimises these parameters, while retaining the functionality expressed in a test suite. In a series of experiments involving over 70,000 lines of code and 700 test cases we found that our deep parameter approach outperformed baseline optimisations (which use only the programmer-provided shallow parameters), ultimately improving execution time by 12% and memory consumption by 21% in the best cases. In addition, despite the larger search space considered, the additional optimisation time cost of our approach is acceptably low. Overall, we feel that deep parameter tuning approaches show much promise for the automated improvement of software with respect to non-functional properties.

8. REFERENCES

[1] A. Arcuri and G. Fraser. On parameter tuning in search based software engineering. In *Search Based Software Engineering.* 2011.

[2] E. D. Berger and B. G. Zorn. Diehard: Probabilistic memory safety for unsafe languages. In *Programming Language Design and Implementation*, PLDI '06, 2006.

[3] E. D. Berger, B. G. Zorn, and K. S. McKinley. Reconsidering custom memory allocation. In *Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '02, 2002.

[4] N. Brake, J. R. Cordy, E. Dan Y, M. Litoiu, and V. Popes U. Automating discovery of software tuning parameters. In *Workshop on Software Engineering for Adaptive and Self-managing Systems*, SEAMS '08, 2008.

[5] J. M. Colmenar, J. L. Risco-Martín, D. Atienza, and J. I. Hidalgo. Multi-objective optimization of dynamic memory managers using grammatical evolution. In *Conference on Genetic and Evolutionary Computation*, GECCO '11, 2011.

[6] C. Ţăpuş, I.-H. Chung, and J. K. Hollingsworth. Active harmony: Towards automated performance tuning. In *Conference on Supercomputing*, Supercomputing '02, 2002.

[7] K. Deb, A. Pratap, S. Agarwal, and T. Meyarivan. A fast and elitist multiobjective genetic algorithm: NSGA-II. *Evolutionary Computation, IEEE Transactions on*, 6(2), 2002.

[8] H. Do, S. Elbaum, and G. Rothermel. Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact. *Empirical Software Engineering*, 10(4), 2005.

[9] C. Fonseca and P. Fleming. On the performance assessment and comparison of stochastic multiobjective optimizers. In *PPSN*. Springer Berlin Heidelberg, 1996.

[10] M. Harman. The current state and future of search based software engineering. In *2007 Future of Software Engineering*, FOSE '07, 2007.

[11] M. Harman, Y. Jia, and W. Langdon. Babel pidgin: Sbse can grow and graft entirely new functionality into a real world system. In *Search-Based Software Engineering*, volume 8636 of *Lecture Notes in Computer Science*, pages 247–252. Springer International Publishing, 2014.

[12] M. Harman, Y. Jia, and W. B. Langdon. Strong higher order mutation-based test data generation. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, ESEC/FSE '11, pages 212–222, 2011.

[13] M. Harman, Y. Jia, W. B. Langdon, J. Petke, I. H. Moghadam, S. Yoo, and F. Wu. Genetic improvement for adaptive software engineering (keynote). In *Symposium on Software Engineering for Adaptive and Self-Managing Systems*, SEAMS, 2014.

[14] M. Harman, Y. Jia, and Y. Zhang. Achievements, open problems and challenges for search based software testing (keynote). In 8^{th} *IEEE International Conference on Software Testing, Verification and Validation (ICST 2014)*, Graz, Austria, April 2015.

[15] H. Hoffmann, S. Sidiroglou, M. Carbin, S. Misailovic, A. Agarwal, and M. Rinard. Dynamic knobs for responsive power-aware computing. In *Architectural Support for Programming Languages and Operating Systems*, 2011.

[16] F. Hutter, D. Babic, H. H. Hoos, and A. Hu. Boosting verification by automatic tuning of decision procedures. In *Formal Methods in Computer Aided Design, 2007. FMCAD '07*, Nov 2007.

[17] F. Hutter, H. H. Hoos, K. Leyton-Brown, and T. Stützle. ParamILS: an automatic algorithm configuration framework. *Journal of Artificial Intelligence Research*, 36(1), 2009.

[18] Y. Jia and M. Harman. MILU: A Customizable, Runtime-Optimized Higher Order Mutation Testing Tool for the Full C Language. In *Proceedings of the TAIC PART'08*, pages 94–98, Windsor, UK, 29-31 August 2008.

[19] Y. Jia and M. Harman. An analysis and survey of the development of mutation testing. *Software Engineering, IEEE Transactions on*, 37(5), 2011.

[20] T. Katagiri, K. Kise, H. Honda, and T. Yuba. Fiber: A generalized framework for auto-tuning software. In *High Performance Computing.* 2003.

[21] W. B. Langdon, M. Modat, J. Petke, and M. Harman. Improving 3d medical image registration cuda software with genetic programming. In *Conference on Genetic and Evolutionary Computation*, GECCO '14, 2014.

[22] D. Lea and W. Gloger. A memory allocator, 2000. http://g.oswego.edu/dl/html/malloc.html.

[23] M. Papadakis, Y. Jia, M. Harman, and Y. L. Traon. Trivial Compiler Equivalence: A Large Scale Empirical Study of a Simple Fast and Effective Equivalent Mutant Detection Technique. In *Proceedings of the 37th International Conference on Software Engineering*, 15. To appear.

[24] J. Petke, M. Harman, W. Langdon, and W. Weimer. Using genetic improvement and code transplants to specialise a c++ program to a problem class. In *Genetic Programming.* 2014.

[25] J. L. Risco-Martín, D. Atienza, J. M. Colmenar, and O. Garnica. A parallel evolutionary algorithm to optimize dynamic memory managers in embedded systems. *Parallel Computing*, 36(10 - 11), 2010. Parallel Architectures and Bioinspired Algorithms.

[26] J. L. Risco-Martín, D. Atienza, R. Gonzalo, and J. I. Hidalgo. Optimization of dynamic memory managers for embedded systems using grammatical evolution. In *Conference on Genetic and Evolutionary Computation*, GECCO '09, 2009.

[27] J. L. Risco-Martín, J. M. Colmenar, J. I. Hidalgo, J. Lanchares, and J. Díaz. A methodology to automatically optimize dynamic memory managers applying grammatical evolution. *Journal of Systems and Software*, 91(0), 2014.

[28] E. Schulte, Z. Fry, E. Fast, W. Weimer, and S. Forrest. Software mutational robustness. *Genetic Programming and Evolvable Machines*, 15(3), 2014.

[29] R. Vuduc, J. W. Demmel, and J. Bilmes. Statistical models for automatic performance tuning. In *International Conference on Computational Science (ICCS)*, 2001.

[30] R. C. Whaley and J. J. Dongarra. Automatically tuned linear algebra software. In *Conference on Supercomputing*, Supercomputing '98, 1998.

[31] X. Yao, M. Harman, and Y. Jia. A study of equivalent and stubborn mutation operators using human analysis of equivalence. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, pages 919–930, 2014.

[32] E. Zitzler and L. Thiele. Multiobjective evolutionary algorithms: a comparative case study and the strength pareto approach. *IEEE Transactions on Evolutionary Computation*, 3(4), Nov 1999.

[33] B. Zorn and D. Grunwald. Empirical measurements of six allocation-intensive C programs. *SIGPLAN Not.*, 27(12), Dec. 1992.