A Comprehensive Survey of Benchmarks for Improvement of Software's Non-Functional Properties

AYMERIC BLOT, University of Rennes, France, University of the Littoral Opal Coast, France, and University College London, UK
JUSTYNA PETKE, University College London, UK

Despite recent increase in research on improvement of non-functional properties of software, such as energy usage or program size, there is a lack of standard benchmarks for such work. This absence hinders progress in the field, and raises questions about the representativeness of current benchmarks of real-world software.

To address these issues and facilitate further research on improvement of non-functional properties of software, we conducted a comprehensive survey on the benchmarks used in the field thus far. We searched five major online repositories of research work, collecting 5499 publications (4066 unique), and systematically identified relevant papers to construct a rich and diverse corpus of 425 relevant studies.

We find that execution time is the most frequently improved property in research work (63%), while multi-objective improvement is rarely considered (7%). Static approaches for improvement of non-functional software properties are prevalent (51%), with exploratory approaches (18% evolutionary and 15% non-evolutionary) increasingly popular in the last 10 years. Only 39% of the 425 papers describe work that uses benchmark suites, rather than single software, of those SPEC is most popular (63 papers). We also provide recommendations for future work, noting, for instance, lack of benchmarks for non-functional improvement that covers Python, JavaScript, or mobile devices. All the details regarding the 425 identified papers are available on our dedicated webpage: https://bloa.github.io/nfunc_survey.

CCS Concepts: • **General and reference → Surveys and overviews**; • **Software and its engineering → Software post-development issues**; **Extra-functional properties**; **Empirical software validation**.

Additional Key Words and Phrases: software performance, non-functional properties, benchmark

ACM Reference Format:
Aymeric Blot and Justyna Petke. 2025. A Comprehensive Survey of Benchmarks for Improvement of Software's Non-Functional Properties. *ACM Comput. Surv.* 1, 1, Article 1 (January 2025), 35 pages. https://doi.org/10.1145/3711119

1 Introduction

The primary focus of software developers is to write bug-free software. Even then, many issues often arise throughout the development and production cycles, causing significant human resource investment into code maintenance. Poor software quality is costly. For example, Krasner [225] estimated that in 2022 the cost of poor-quality software on the US economy was 2.41 trillion dollars.

In order to deliver better software, many techniques and tools exist to diagnose software's potential flaws, refactor source code, optimise compiled machine code, and even use evolution to automatically derive better software variants. To this purpose, many surveys have been conducted on techniques for functional improvement of software, particularly regarding automated bug fixing [69, 139, 245, 292, 293]. We observed that studies in the field of automated program repair

Authors' Contact Information: Aymeric Blot, University of Rennes, Rennes, France and University of the Littoral Opal Coast, Calais, France and University College London, London, UK, aymeric.blot@univ-rennes.fr; Justyna Petke, University College London, London, UK, j.petke@ucl.ac.uk.

ACM 1557-7341/2025/1-ART1
https://doi.org/10.1145/3711119

often use well-crafted benchmarks[1] (see, e.g., https://program-repair.org/benchmarks.html). It is worth noting that since release of a now famous Defects4J benchmark [202], the publication rate in the field has increased (https://program-repair.org/statistics.html), allowing for faster comparisons.

On the other hand, non-functional properties of software are often relegated to second place, often leading to unneccessary bloat [3]. One type of non-functional properties are performance concerns, such as execution time, memory or energy consumption. Their importance is ever increasing with increased usage of battery-powered mobile smart devices. In fact, studies have shown that 1/3 of instances of Android users abandoning mobile applications and 59% of bad reviews were due to poor performance [186, 258]. However, whilst more and more research is now conducted on improvement of software's non-functional properties (e.g., by automating this process using genetic improvement [313]), the field still lacks standardised benchmarks. These would help drive the field forward by providing a common baseline for newly proposed approaches.

Therefore, we conducted an in-depth literature review, both to better frame what has been done and is currently conducted, as well as to identify the type of software that is most often targeted. First, we conducted a preliminary search, hand-picking 100 research articles to identify the most relevant keywords linked to improvement of software's non-functional properties. We then queried five major online repositories for related work from the past 45 years—ACM Digital Library, IEEE Xplore, Scopus, Google Scholar, and ArXiV—grouping useful keywords into five subsets. Finally, we repeated this search focusing on past four years to complement the main systematic search and ensure relevance to current practices. Out of 5499 results returned we found 4066 unique papers which we systematically checked for empirical work that improves non-functional properties of software, providing potential benchmarks for future work. Ultimately, these three searches resulted in a corpus of 425 unique relevant studies, that we then categorised with regards to the property they target, the type of approach they use, and central to our survey, the benchmark they consider.

With our survey, we aim to answer the following research questions.

RQ1 (State of the Art) How prevalent is empirical work on improvement of non-functional properties of software?
  (a) What type of non-functional property is most often improved?
  (b) When optimised together, which combinations of non-functional properties are considered?
  (c) Which approaches for non-functional improvement are used most often?
  (d) How is software most often modified?

RQ2 (Existing Benchmarks) Which software is used to validate work on improvement of non-functional properties of software?
  (a) How often are existing software benchmarks reused?
  (b) Which software is targeted most often for improvement?

RQ3 (Software Diversity) How representative are the benchmarks used in work on real-world improvement of non-functional properties of software?
  (a) What type of software is targeted most frequently?
  (b) Which programming languages are targeted most frequently?

2 Survey Methodology

We conducted a systematic literature review in order to establish the state of the art in benchmarks used in empirical studies improving non-functional properties of software. We started with a preliminary search to construct a set of relevant keywords. These keywords were then used to conduct a systematic search across five online repositories. Finally, an additional search focusing exclusively on the most recent work ensured relevance to current practices.

---

[1] See Section 3.4 for the definition of a "benchmark".

Table 1. Keywords used in the systematic repository search. Wildcards ("*") are used on digital libraries supporting such queries; otherwise we list the alternative keywords used.

Group	Category	Keywords
Software		code, program, software, application
Improvement		optim* (optimize, optimizing, optimization), improv* (improve, improving, improvement), automat* (automated, automatically), reduc* (reduce, reducing)
Non-Functional Property	Time	time, runtime, speed* (speed, speedup), fast* (fast, faster)
	Memory	memory
	Energy	energy, power
	Quality	performance, effic* (efficient, efficiency), effective* (effective, effectiveness), accura* (accuracy), precis* (precision)
	Other	functional* (functional, functionality), size, slim* (slimming), bloat, debloating

2.1 Preliminary Search

Whilst strongly anchored in the software engineering world, work on improving software's non-functional properties spans many different independent research fields that do not necessarily use a consistent terminology, let alone share a unified one. In order to conduct an adequate literature review, we first needed to make sure relevant keywords were used. To that purpose, we performed a manual search, gathering a small and diverse set of relevant papers on improvement of software's non-functional properties. We used a simple criterion to establish whether a piece of work qualifies as improving a non-functional property of software, namely if the intended semantics of the transformed software is preserved. In that sense work on optimisation of runtime, energy, or memory consumption is deemed relevant, whilst work on bug fixing or software transplantation is not relevant, as, by definition, the input/output behaviour of the transformed software will change. The only exception to this rule is software specialisation, where functionality could be compromised for improvement of a non-functional property. Since the main focus of such work is improvement of non-functional behaviour, we still consider it relevant.

Starting from known related work, we iteratively built a purposely diverse corpus of 100 relevant publications by querying specific research fields (e.g., "genetic improvement", "code refactoring", "compiler tuning"), specific types of non-functional properties (e.g., "reliability", "complexity"), using synonyms (e.g., "software evolution" and "program evolution", "energy consumption" and "energy footprint"), etc. We then extracted, from the title and abstract of every selected publication, every word that could potentially be used as a keyword during the systematic repository search. We calculated how frequently each word occurs in the metadata of selected work. In addition, we investigated the use of wildcards, grouping words sharing similar prefixes, and expressions (e.g., "execution time" or "running time", as opposed to "time" alone). We also tried to singularise words, removing final "s"-es when the pluralised versions of words were found.

Details of frequency analysis are presented in Figure 1. Note that we excluded prepositions, articles, and other generic words clearly not useful for our literature survey (e.g., "result", "paper", "is", "are"). Surprisingly, word combinations did not result in particularly frequent expressions, appearing significantly less often than their respective individual words. However, prefix wildcards were very effective in providing usable search keywords. Then, a subset of the most frequent words

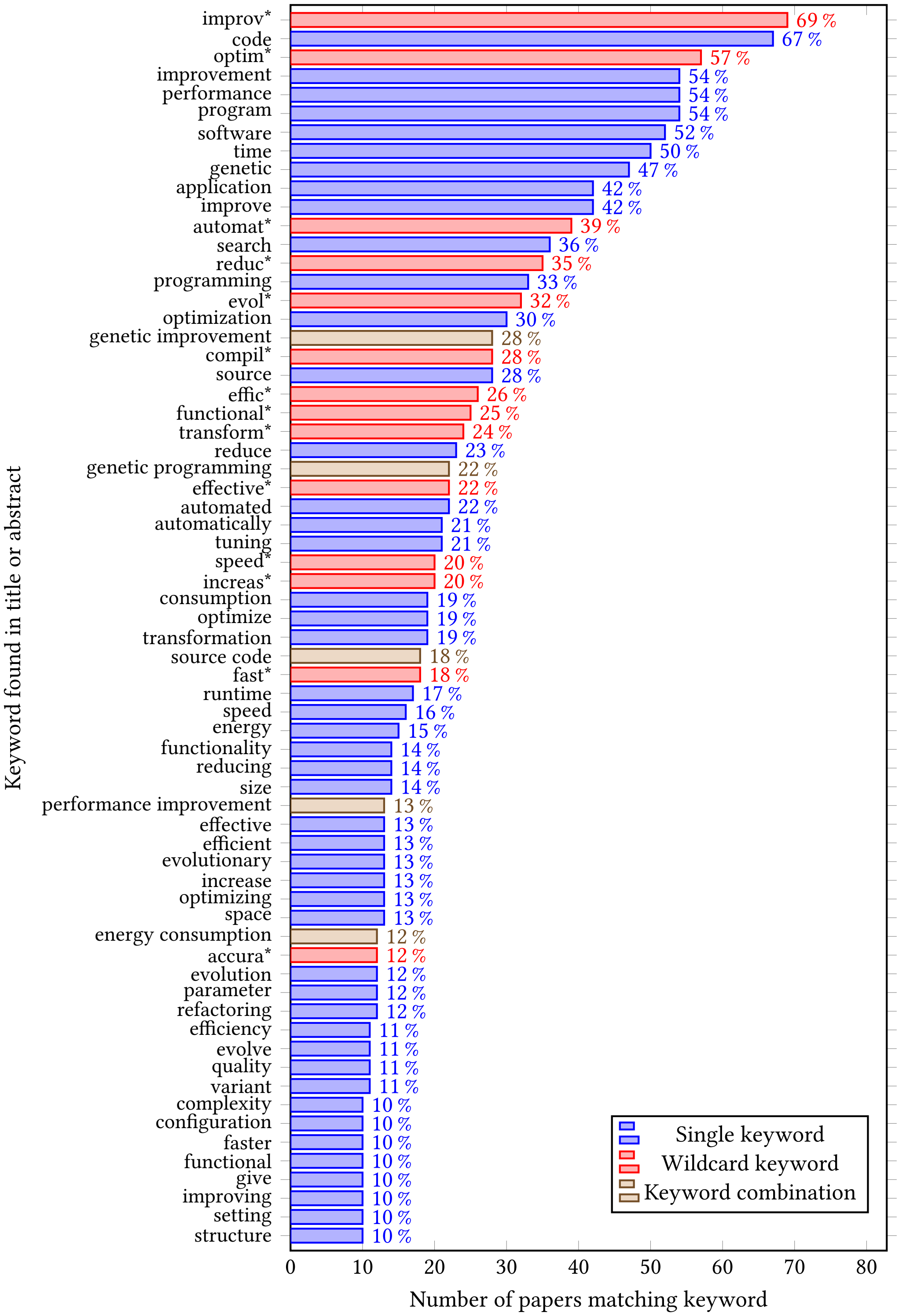

Fig. 1. Preliminary search: frequent words in titles and abstracts. (≥10%)

was derived and further classified into seven classes of potential keywords, as detailed in Table 1. These keywords were used as a basis for the second step of our survey protocol.

2.2 Systematic Repository Search

Based on keywords found in our preliminary search, we performed a systematic search for relevant work, largely inspired by the methodology proposed by Hort et al. [173]. We used three groups of keywords: two that ensure papers relate to software's non-functional property improvement (*Software* and *Improvement* in Table 1) and one group of keywords targeting a specific non-functional property (including *Time*, *Memory*, *Energy*, *Quality*, and *Others*). The addition of the supplementary *Improvement* group of keywords is motivated by the large number of papers otherwise returned by each search. In total, five separate queries are therefore constructed, one for each keyword from the *Non-Functional Property* category. When applied on the preliminary dataset of 100 papers, they collectively achieve 97% coverage, with only one paper missing a *Software* keyword ([295]) and two missing an *Improvement* keyword ([128, 320]).

These five queries were used, to ensure good representation, on five major digital libraries: ACM Digital Library, IEEE Xplore, Scopus, Google Scholar, and ArXiV, for a total of 25 searches. In particular, Springer Link, Science Direct, or JSTOR were not considered due to their inability to handle complex Boolean queries or queries with many keywords. Where filters allowed for it, each search was restricted to the computer science research field (Scopus, ArXiV), as well as restricted to conference proceedings and journal articles (ACM, IEEE, Scopus). We made no further restrictions. Because of the high number of papers returned by the 25 queries, we only focused on the first 200 papers returned by the digital libraries, using the provided default relevance-based sort order.

The specifics of our systematic methodology are as follows.

Inclusion Criteria. A paper is deemed relevant when it fulfils the following four criteria:

(1) it must relate to a quantifiable non-functional property;
(2) it must contain an empirical study which applies a software improvement technique to existing software;
(3) improvement of the targeted non-functional property must be the active focus of the paper and not merely a side-effect;
(4) the approach used must result in a distinct software execution that can thus be compared to the original software.

Selection Process. Publications are then processed according to the following three steps:

Title: first, publications whose titles clearly do not fit the spirit of the survey are discarded without further reading;

Abstract: second, abstracts are inspected and publications are rejected when at least one inclusion criteria clearly does not apply;

Body: only then remaining potentially relevant publications are read in full and included, depending on the relevance of their content.

In total, the 25 queries yielded 5000 results. Papers with identical title were merged together after manual verification, resulting in overall 3749 unique papers (25% redundancy between query terms and online repositories). Note that in some rare cases individual queries resulted in fewer than 200 unique papers; this is, for example, due to some papers having multiple DOIs[2].

Of these 3749 unique papers, we determined 2560 to be irrelevant based on their title alone (e.g., "Memory training and memory improvement in Alzheimer's disease: Rules and exceptions" is

---

[2] E.g., a paper published in OOPSLA 2010 (https://doi.org/10.1145/1869459.1869473) also published in ACM SIGPLAN Notices (https://doi.org/10.1145/1932682.1869473), or a paper published in ICCAD 2006 differently indexed by IEEE (https://doi.org/10.1109/ICCAD.2006.320144) and ACM (https://doi.org/10.1145/1233501.1233551)

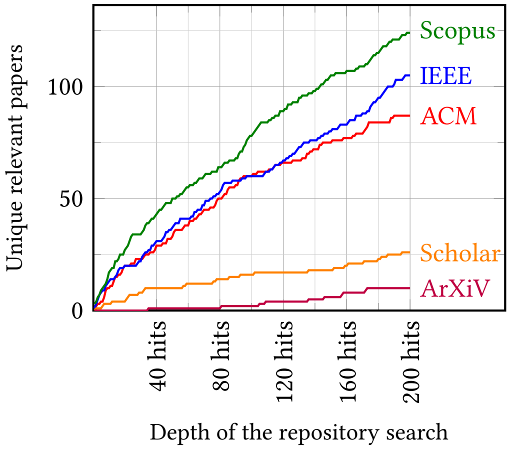

Fig. 2. Cumulative number of unique relevant papers found in each digital library (across all five non-functional property keyword groups).

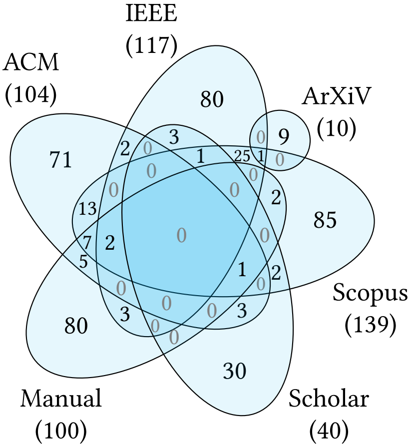

Fig. 3. Venn diagram of all 425 corpus papers according to origin. Missing intersections (e.g., between Google Scholar and ArXiV) are all empty.

clearly not relevant), 510 to be irrelevant based on their abstract, and finally 372 to be irrelevant based on their actual content. Overall, this second step of the survey yielded 307 unique relevant papers (8.19%). This manual step was conducted over the span of two months (FTE).

We observed that Scopus yielded by far the highest number of relevant papers, almost four times the number from Google Scholar and ArXiV combined. All types of queries yielded similar numbers of papers, with the exception of the "Time" keyword group, although Section 3 will show that execution time is by far the most common non-functional property optimised in the literature.

As means to validate the threshold of 200 papers considered for every query, we investigated the rate at which relevant work appears throughout the systematic repository search. Figure 2 shows, for each of the five digital libraries, how many would have been returned had a smaller threshold been used. Very surprisingly, the rate according to which relevant work is found is almost constant, meaning that a considerable amount of relevant work can be expected to be found even after our threshold of 200 papers per query. Similar rates are also observed when controlling for each of the keywords categories, i.e., there is a lot more related work for all types of non-functional properties.

Finally, to complement the main systematic search—performed in July 2021—and ensure coverage of more recent work, we repeated the same methodology in July 2024 with two main differences. We restricted publication date to 2021–2024 to avoid overlaps, and we only considered the first 20 hits to avoid overfits. In total, the 25 queries yielded 499 results, resulting in 332 unique papers. Out of these, we determined 155 papers to be irrelevant based on their title alone, 64 based on their abstract, and 69 based on their actual content, resulting in 44 relevant unique papers.

Details of both steps of the systematic repository search are presented in Table 2 with a total of 345 unique papers (which excludes work found via our preliminary manual search). In particular, for each class of query and each online digital library we present the total number of hits, and the numbers of papers rejected at each step of the selection process, or ultimately classified as relevant.

2.3 Corpus

In preliminary search we identified 100 relevant papers. The main systematic repository search then yielded 307 unique relevant papers. Finally, The complementary systematic repository search

Table 2. Systematic repository search. For each of the 50 queries we detail how many papers were found, how many unique papers were selected, and how many papers were considered after each of the three steps of our selection process. We indicate for every step how many unique papers are found, both across all five digital libraries (in the last column) as well as across all five types of queries (last rows).

		Main search (first 200 results, −2021)					Complementary search (first 20 results, 2021−2024)					
	Step	ACM	IEEE	Scopus	Scholar	ArXiV	ACM	IEEE	Scopus	Scholar	ArXiV	Unique
Time	Total hits	26K	216K	375K	6.3M	1268	13K	62K	124K	6.3M	158	–
	Unique work	200	199	200	200	200	20	20	19	20	20	1091
	Selected (Title)	60	84	16	55	31	12	16	13	11	6	299
	Selected (Abstract)	39	52	3	18	13	7	14	12	9	1	164
	Selected (Body)	16	29	1	7	2	3	7	3	4	0	72
Memory	Total hits	6326	30K	57K	5.2M	273	2851	8000	18K	667K	19	–
	Unique work	200	199	198	200	199	20	20	19	20	19	1030
	Selected (Title)	78	89	84	66	56	11	14	16	13	6	396
	Selected (Abstract)	42	62	62	33	28	7	7	11	10	2	236
	Selected (Body)	18	38	32	14	5	5	1	5	4	0	105
Energy	Total hits	7907	140K	154K	5.1M	727	4416	42K	59K	1.5M	161	–
	Unique work	199	200	198	200	200	20	20	20	20	20	1049
	Selected (Title)	93	80	111	47	41	7	10	17	6	6	386
	Selected (Abstract)	56	46	84	8	18	3	6	14	4	1	215
	Selected (Body)	16	27	41	1	2	1	2	5	3	0	82
Quality	Total hits	44K	351K	591K	5.7M	1439	18K	94K	190K	1.6M	139	–
	Unique work	200	200	199	200	200	20	20	20	20	20	1071
	Selected (Title)	97	79	94	43	36	13	18	16	6	6	389
	Selected (Abstract)	55	50	71	15	14	8	16	10	3	1	229
	Selected (Body)	26	34	31	7	2	3	13	3	1	0	111
Others	Total hits	8166	63K	116K	6.1M	826	17K	74K	145K	1.2M	240	–
	Unique work	199	198	192	200	199	20	20	19	20	20	1025
	Selected (Title)	93	75	99	39	35	15	17	13	11	6	360
	Selected (Abstract)	67	47	82	14	15	11	15	11	7	1	231
	Selected (Body)	36	32	56	6	2	8	11	9	2	0	129
Overall	Unique work	952	806	962	934	603	77	74	78	90	48	4066
	Selected (Title)	405	315	391	246	122	41	51	60	42	13	1356
	Selected (Abstract)	249	194	286	103	52	26	36	47	29	3	785
	Selected (Body)	104	117	139	40	10	15	17	15	13	0	345

yielded 44 unique papers. Combined, they resulted in 425 unique relevant papers on the topic of improvement of non-functional properties of software, as shown in Table 3.

Figure 3 presents the source distribution of the 425 corpus papers. The set of papers originating from ArXiV (10 papers) is very small and almost disconnected from the other sets of work. In fact, the vast majority of publications (84%) is only found once, and with the exception of Scopus

Table 3. Summary of the 425 unique corpus papers, according to the main types of non-functional property targeted, the types of multi-objective focuses, the types of search approaches, and the types of modifications applied to the original software. An interactive and more comprehensive artefact for this data is available both in the supplementary materials and also live at https://bloa.github.io/nfunc_survey whilst raw data can be accessed at https://github.com/bloa/nfunc_survey.

Criteria	Relevant papers
Property	**time (268)**: [1, 5, 8–12, 14, 15, 17–20, 22–24, 26, 27, 29, 32–35, 38, 43–45, 47–49, 52, 53, 59, 60, 64, 65, 67, 70, 72–74, 76–79, 81–83, 85–87, 90–96, 98–100, 105, 107–109, 111, 112, 114–119, 121–123, 125, 126, 130, 134, 136, 138, 141, 143, 146, 147, 149, 150, 152, 155, 156, 160, 162–165, 167, 169–171, 174, 176–179, 183, 184, 187, 188, 190–192, 194, 197–200, 203, 204, 206–208, 210, 211, 214–219, 222, 223, 227–231, 234–236, 238–241, 243, 246–248, 250–252, 256, 257, 259–261, 263–268, 270, 275, 276, 276, 277, 279–282, 284, 286–288, 290, 291, 294, 298, 299, 302, 305–309, 314–318, 322, 326, 327, 329, 330, 332–337, 340–342, 345–353, 355, 356, 359, 361, 363–366, 369, 371, 373, 375, 376, 379, 381–383, 385–387, 390–394, 396–408, 411, 414, 415, 419, 420, 423–432, 435], **energy (77)**: [2, 16, 22, 23, 28, 40, 46, 52, 54, 58, 61, 62, 66, 68, 71, 81, 84, 91, 92, 96, 97, 100, 102–104, 110, 114, 124, 126, 132, 133, 135, 137, 137, 153, 154, 172, 181, 182, 189, 205, 217, 220, 226, 250, 253–255, 272, 273, 275, 300, 301, 303, 304, 310–312, 317, 326, 327, 338, 339, 341–344, 352, 354, 362, 364, 366, 377, 384, 387, 416, 436], **size (71)**: [4, 7, 21, 25, 41, 50, 51, 55, 56, 63, 75, 80, 90, 92, 101, 106, 113, 116, 127, 137, 137, 140, 143, 157, 166–168, 170, 194–196, 201, 224, 230, 246, 247, 249, 252, 278, 281–283, 289, 296, 319, 321, 323–325, 328, 331, 352, 357, 358, 367, 368, 370, 374, 380, 388, 390, 402, 409, 410, 412, 413, 418, 421, 422, 433, 434], **other (61)**: [4, 6, 13, 28, 31, 36, 37, 42, 43, 46, 57–59, 64, 88, 89, 103, 120, 124, 128, 129, 131, 142, 144, 151, 158, 161, 168, 175, 193, 213, 219, 221, 224, 232, 237, 242, 244, 262, 264, 265, 267–269, 271, 288, 295–297, 299, 308, 320, 321, 358, 360, 378, 395, 404, 405, 412, 413], **memory (37)**: [12, 26, 32, 33, 39, 40, 43, 47, 77, 96, 141, 145, 148, 159, 164, 180, 185, 194, 200, 208, 209, 211, 212, 217, 226, 230, 233, 274, 284, 285, 308, 372, 389, 414, 417, 423, 435]
Multi-objective	**focus (58)**: [12, 13, 26, 32, 40, 52, 64, 77, 90–92, 96, 100, 103, 116, 137, 137, 141, 143, 164, 167, 168, 194, 200, 208, 211, 217, 219, 224, 226, 230, 246, 247, 250, 252, 268, 275, 281, 282, 284, 288, 296, 299, 308, 320, 321, 327, 341, 342, 352, 358, 364, 366, 390, 412–414, 423], **report (56)**: [5, 7, 25, 27, 53, 56, 89, 98, 110, 113, 128, 134, 142, 147, 148, 154, 155, 158, 159, 172, 177, 181, 185, 189, 195, 196, 199, 203, 214, 215, 228, 236, 248, 251, 262, 272–274, 285, 301, 304, 312, 319, 324, 337, 351, 359–361, 372, 373, 376, 384, 389, 395, 432], **search (30)**: [4, 22, 23, 28, 33, 43, 46, 47, 58, 59, 81, 86, 87, 114, 124, 126, 170, 174, 213, 264, 265, 267, 269, 317, 326, 387, 402, 404, 405, 435]
Search	**static (217)**: [1, 5–7, 11–14, 17, 19, 21, 24, 26, 29, 32, 34, 38, 39, 49–51, 53, 56, 57, 63, 64, 71, 75, 77–79, 89, 90, 97, 98, 101, 104, 107, 109, 116, 117, 121, 123, 127, 129–131, 134, 135, 137, 137, 140–147, 151, 152, 155–158, 165–167, 169, 172, 175, 177, 182, 183, 187, 189–193, 196–201, 203, 204, 206–208, 212, 216, 217, 221–224, 226, 228–231, 244, 247–249, 251, 253, 255, 257, 260–262, 266, 272–276, 276, 278, 281, 282, 286, 287, 290, 291, 294, 296, 298–305, 307, 309, 310, 312, 318–325, 328, 329, 331–334, 336–340, 343, 348–351, 355, 357–363, 366–370, 372–375, 378–381, 384–386, 388–390, 392, 395–397, 400, 401, 406–410, 414, 415, 417, 418, 420, 423, 426, 429, 430, 432–434, 436], **evolutionary (77)**: [4, 9, 22, 23, 28, 31, 33, 37, 42, 44–47, 58, 61, 62, 66, 67, 80, 81, 84, 86, 87, 93–95, 99, 105, 106, 108, 112, 113, 124, 126, 138, 150, 161, 162, 174, 195, 213, 214, 219, 227, 232, 234–243, 263–265, 267–270, 280, 283, 297, 314–316, 345, 346, 352, 387, 404, 421, 422, 428, 435], **exploratory (52)**: [15, 20, 41, 43, 45, 48, 59, 60, 65, 68, 70, 74, 85, 88, 92, 96, 111, 113–115, 118–120, 132, 133, 149, 153, 168, 179, 195, 233, 252, 254, 289, 306, 317, 326, 330, 335, 346, 365, 376, 382, 393, 394, 402, 403, 412, 413, 425, 428, 431], **sampling (48)**: [8, 18, 35, 52, 59, 73, 74, 76, 83, 91, 100, 110, 122, 125, 148, 154, 160, 163, 170, 171, 176, 178, 180, 194, 209, 215, 218, 246, 256, 259, 277, 279, 288, 295, 317, 341, 356, 371, 391, 393, 394, 405, 411, 416, 419, 424, 427, 428], **manual (46)**: [2, 10, 16, 25, 27, 36, 40, 54, 55, 72, 82, 102, 103, 128, 136, 156, 159, 164, 181, 183–185, 188, 189, 205, 210, 211, 220, 250, 271, 284, 285, 308, 311, 327, 342, 344, 347, 353, 354, 364, 377, 383, 398–400]
Change	**semantic (218)**: [2, 4–7, 10–13, 16–19, 21, 24–29, 35–37, 39, 41, 48, 50, 51, 53–58, 63, 64, 71, 75, 78, 82, 89, 90, 92, 97, 98, 101, 102, 104, 107, 108, 116, 117, 121, 123, 127–133, 135–137, 137, 140–144, 146, 155–159, 164, 166–168, 170–172, 180, 182–184, 187–190, 192, 193, 196, 197, 199, 201, 205, 206, 210–213, 220–223, 226, 228–231, 244, 246–253, 255, 260, 262, 266, 271, 272, 275, 276, 276–278, 281, 284–287, 289–291, 296, 297, 299–303, 307–310, 312, 319–321, 323–325, 328, 329, 331, 333, 334, 336, 338, 339, 342, 343, 347, 349, 351, 353–355, 357–359, 361–364, 368–370, 372–375, 377–381, 383, 385, 386, 388–390, 392, 395, 398, 399, 403, 405–407, 410, 412–415, 418, 426, 429–432], **loops (99)**: [1, 7, 11, 17, 22, 26, 32, 34, 37, 49, 72, 73, 77, 91, 92, 109, 111, 121, 122, 134, 145, 147, 151, 152, 154, 156, 165, 171, 182, 187, 191, 198, 200, 203–205, 207–211, 214–217, 219, 226, 230, 250, 257, 270, 272–274, 276, 276, 277, 279, 280, 286, 294, 298, 305, 317, 318, 322, 326, 328, 332, 334–337, 340, 348, 350, 353, 356, 358, 360, 364, 366, 384, 391, 401, 403, 407–409, 415, 419, 420, 423–425, 430, 433, 434, 436], **destructive (85)**: [10, 14, 15, 31, 33, 38, 40, 42, 44–47, 58, 60–62, 65, 66, 84, 93–95, 100, 103, 112, 113, 120, 123, 124, 136, 149, 150, 153, 161, 162, 172, 175, 177, 210, 212, 219, 232–243, 254, 257, 261, 263–265, 267–269, 282, 283, 300, 304, 311, 314–316, 344, 352, 353, 361, 367, 371, 376, 384, 396, 400, 404, 416, 417, 421, 422], **configuration (83)**: [2, 8, 9, 20, 23, 40, 43, 47, 48, 52, 58, 59, 67, 68, 70, 73, 74, 76, 79–81, 83, 85–88, 96, 99, 105, 106, 110, 111, 114, 115, 118, 119, 125, 126, 136, 138, 148, 160, 163, 169, 172, 174, 176, 178, 179, 181, 185, 194, 195, 212, 218, 224, 227, 233, 256, 259, 272, 288, 295, 306, 317, 324, 327, 330, 341, 345, 346, 365, 382, 387, 393, 394, 397, 402, 406, 411, 427, 428, 435]

covering a non-negligible number of publications from ACM (23) and IEEE (29). Despite this, the corpus contains three papers simultaneously returned from four different sources [330, 358, 413]. We note that despite a theoretical 97% coverage, only 20 of the 100 papers from the preliminary search were actually rediscovered during the two systematic repository searches. This can be explained by the large number of hits returned by every query (see Table 2) and the consistent rate at which relevant work is found (see Figure 2). Recall that in the systematic search we only considered relevant work located in the first 200 results of the 25 queries of the repository search. On the one hand, this means that only 20% of our hand-picked papers appear within that threshold, once again corroborating the idea that many more relevant pieces of work exist in the literature. However, on the other hand, it also means that most of the relevant work identified in the systematic search is new, reducing potential unconscious bias from the preliminary search.

By combining all three searches we thus construct a very rich and diverse corpus of relevant publications, that is, by construction, both relevant in terms of coverage of the different aspects of non-functional improvement, as well as in terms of statistical representativity, as much as this is possible using digital library searches.

3 Empirical Work on Non-Functional Properties of Software

Section 2 described the literature review process that resulted in a corpus of 425 papers related to improvement of non-functional improvement of software. In this section, we examine these 425 papers in more detail, focusing on extracting information according to the following four criteria:

Research landscape. For each paper we note the publication or release date, and the name and the type of venue in which it appeared.

Non-functional property. [RQ1a&b] We note the non-functional properties targeted, and in the cases where multiple properties are reported, whether they have been actually used to produce improved software variants or simply measured at the end of the experiments. We note how each of the non-functional properties was measured.

Search Approaches. [RQ1c&d] We note both the type of approach used to generate software variants and the type of modifications applied to the original software.

Benchmark. [RQ2&3] We note the number and names of software used in the empirical evaluation. We note if it was selected from an existing benchmark, as well as its size, programming language it was written in, its origin, and the platform on which it was run.

Unless explicitly stated otherwise—i.e., Figure 10 and Figure 11, that focus on the 345 papers yielded by the systematic search—all analyses in this section use the full corpus of 425 corpus papers (i.e., including preliminary manual search results).

3.1 Research Landscape

Figure 4 shows the publication year distribution of all 425 corpus papers; almost all papers appeared after 1995, with a clear upward trend. While it can be an artefact of the relevance sort of the online repositories, there is no doubt that work on software's non-functional properties is increasingly widespread. Finally, most related work is published in conferences, although we note a fair number of workshop papers and a very high number of journal articles in 2021.

Figure 5 details the origin (i.e., preliminary search or digital library), by publication year, of all 425 corpus papers. With the exception of two papers published in 2019, all relevant work obtained in the main systematic search through Google Scholar appeared before 2011. Conversely, ArXiV only yielded papers from 2019 onwards and ACM Digital Library from 2008 onwards. However, both IEEE and Scopus yielded papers for every year since 1998.

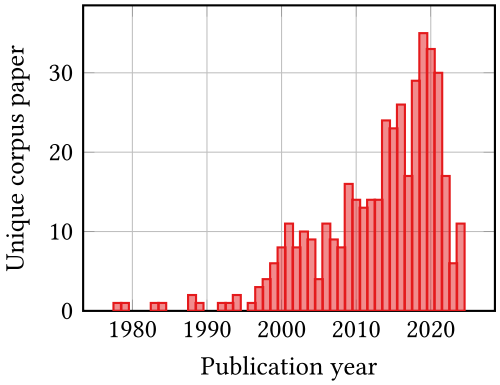

Fig. 4. Publication year distribution of all 425 unique corpus papers.

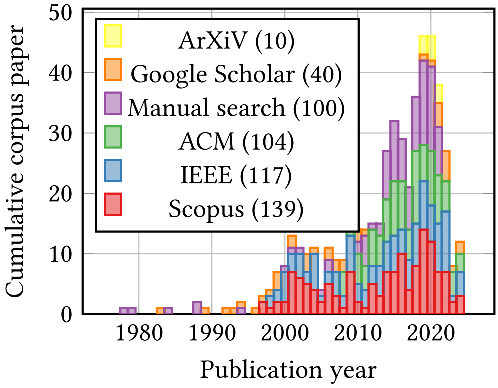

Fig. 5. Publication year distribution of all 425 unique corpus papers, according to origin (preliminary manual search or digital library). Papers found multiple times are counted multiple times.

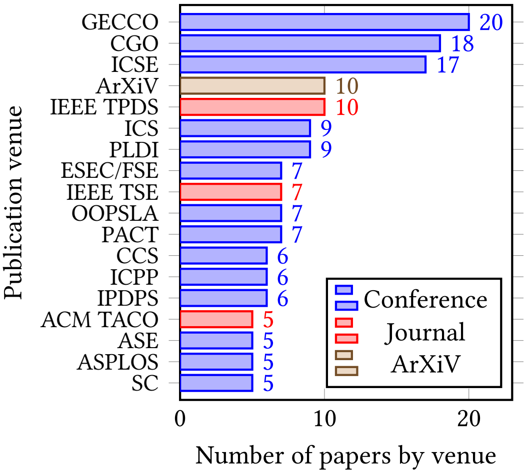

Fig. 6. Most frequent publication venues (≥5) across all 425 unique corpus papers.

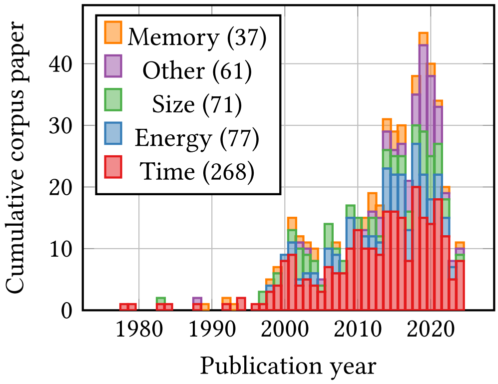

Fig. 7. Publication year distribution of all 425 unique corpus papers, according to the types of non-functional software property targeted. Papers targeting multiple property types are counted multiple times.

Figure 6 presents the venues in which the papers appear most frequently in (i.e., at least five papers published in a given venue). With the notable exception of GECCO, an evolutionary computation conference with the highest figure of 20 related papers, venues are thematically split between applied computing (with CGO, IEEE TPDS, ICS, PLDI, OOPSLA, PACT, CCS, ICPP, IPDPS, ACM TACO, ASPLOS, and SC), for a total of 93 papers, and general software engineering (with ICSE, ESEC/FSE, IEEE TSE, and ASE), for a total of 36 papers.

3.2 Non-functional Properties

Figure 7 shows the distribution of all 425 corpus papers over the years, according to the types of non-functional software property they target. In order to increase readability, properties have

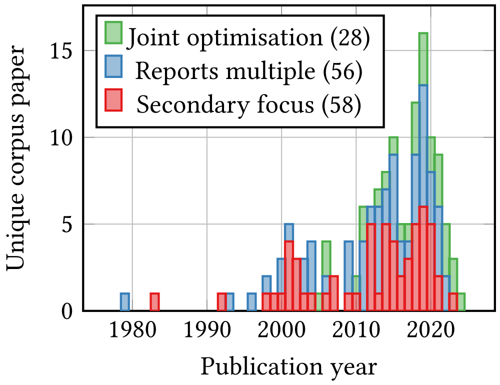

Fig. 8. Publication year of all 142 corpus papers that consider more than one non-functional property, according to their multi-objective philosophy.

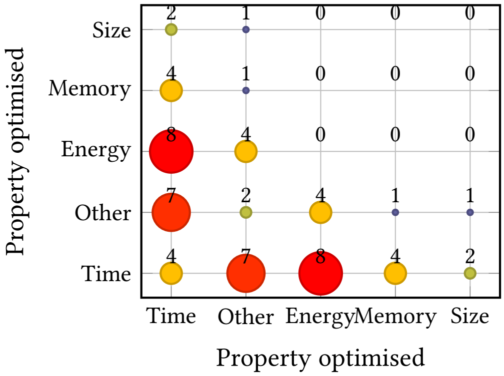

Fig. 9. Correlation between non-functional properties considered in multi-objective work.

been grouped thematically: e.g., "Time" includes mostly execution time (63% of all papers), but also test or compilation time (≈1%), "Memory" encompasses both optimisation of the total amount of resource used, but also its actual usage during execution (e.g., minimising cache misses), and "Size" groups binary size on disk as well as source code size (e.g., in terms of lines of code).

By far the most frequent non-functional property targeted for improvement is time efficiency. Time is optimised in 268 of the 425 corpus papers, followed by energy usage (77 papers), software size (71 papers) and memory usage (37 papers). 61 papers also targeted other types of properties, including mostly the quality of the output, the software's attack surface, or the overall writing quality of source code. The remaining papers targeted other properties, such as copyright or security issues, code complexity, or source code obfuscation.

Answer to RQ1 (a): Empirical work on optimisation of non-functional properties of software most frequently focuses on improvement of execution time. We found 268 out of 425 papers that describe such work. Code size follows with 71 papers (20 more as a secondary objective), while energy usage is targeted in 77 papers (and only 5 more as a secondary objective). Memory usage is considered in 37 papers, and in further 16 as a secondary objective. Work on improvement of other non-functional properties of software is more rare, with the next most frequently targeted property, "output quality", being improved in 28 papers, while "attack surface" improved in 12.

Furthermore, as shown in Figure 8, the authors of 142 papers considered more than one non-functional property. We can distinguish 28 papers that proposed work that actively optimised multiple non-functional properties of software simultaneously, 58 papers that had a secondary focus on at least another property, and finally 56 other papers where authors simply reported on more than one property. Figure 9 shows the pairs of property types considered in work simultaneously optimising at least two non-functional properties during search. As expected, execution time is also the most popular non-functional property considered. We also note a few papers where both software's energy consumption and output solution quality was targeted. Overall, whilst the number of such "multi-objective papers" is undeniably growing, work toward proper multi-objective improvement of non-functional properties is still quite rare.

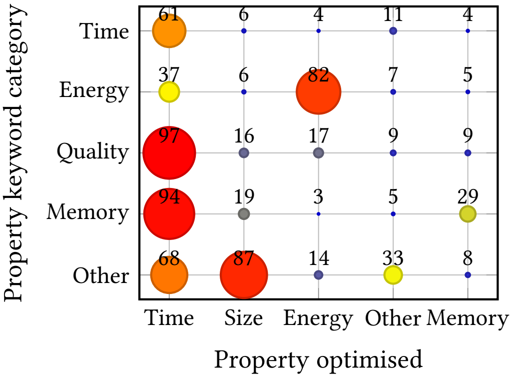

Fig. 10. Correlation between non-functional property keyword category and category of actual targeted property (345 repository papers).

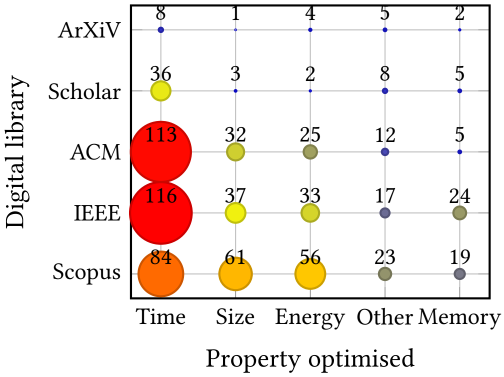

Fig. 11. Correlation between digital library of origin and category of targeted property (305 repository papers).

Answer to RQ1 (b): There is little work on multi-objective improvement of non-functional properties of software (only 28 relevant papers found). In addition to being the most frequent objective (found 25 times), in most cases execution time is one of the targeted properties, often used as a trade-off to energy usage (eight times) or solution quality (seven times).

Figure 10 shows the correlation between the non-functional property keyword categories used in the systematic repository search and the actual primary property types targeted in the relevant papers. Surprisingly, only "energy" and "other" keywords —the latter including bloat- and size-related terms— are effective in yielding thematically relevant papers. Furthermore, "quality" and "memory" keywords are far more effective in yielding papers optimising software speed than their expected properties, even more than "time" keywords.

Conversely, Figure 11 shows the correlation between the repositories and the primary targeted non-functional property. First, as already pointed out, search through ArXiV and Google Scholar really wasn't effective. Then, whilst yielding many more relevant results, both the ACM and the IEEE digital libraries show a bias towards work targeting running time improvements, the most frequent non-functional property overall. Finally, surprisingly, Scopus library does not seem to exhibit that strong of a bias, yielding high numbers of papers targeting size and energy concerns.

3.3 Search Approaches

Five main types of search approaches are distinguished. (1) *static* approaches in which decisions about software modifications are taken without remeasuring the non-functional property; typically a single software variant is generated and compared to the original software. (2) *sampling* approaches, in which a given number of variants are generated and evaluated, the best variant being only determined at the end of the procedure. We include in that category both random and systematic sampling, as well as exhaustive enumeration. (3) *exploratory* (non-evolutionary) approaches, in which multiple software variants are iteratively generated and evaluated in order to produce a final software variant. Exploratory approaches differ from sampling approaches in that they are trajectory-based, intermediary variants being used to guide the search. This category includes, for example, local search algorithms and greedy approaches, specifically excluding evolutionary

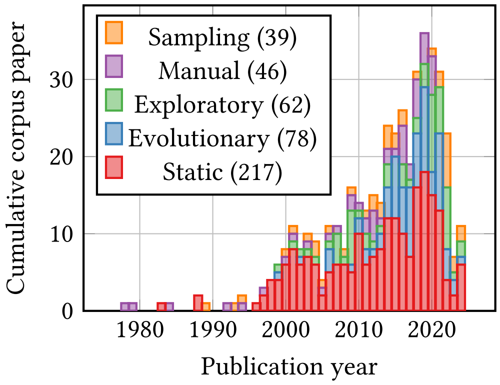

Fig. 12. Publication year distribution of all 425 unique corpus papers, according to the types of search approaches. Papers using multiple search approaches are counted multiple times.

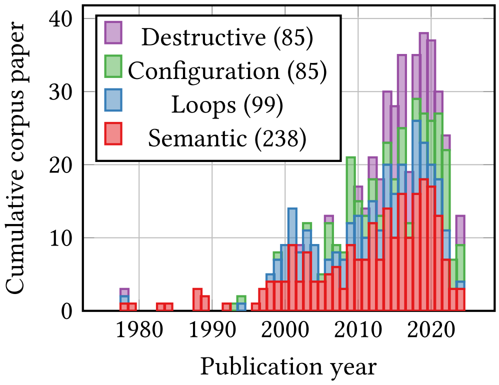

Fig. 13. Publication year distribution of all 425 unique corpus papers, according to the types of software modifications. Papers considering multiple types of modifications are counted multiple times.

approaches. (4) *evolutionary* approaches, that expand on exploratory approaches by using biology-inspired procedures such as, for example, genetic algorithms or genetic programming. Finally, (5) *manual* approaches in which the software is modified by hand.

Figure 12 shows the distribution of types of search approaches by year of publication. Static approaches are the most frequent (217 papers), and constitute with manual (46 papers) and sampling (39 papers) approaches almost all publications until around 2005. Both exploratory (62 papers) and evolutionary (78 papers) approaches start to appear around 2000, being more prevalent after 2005. Finally, most evolutionary approaches appear after 2014.

Answer to RQ1 (c): Most of the work on improvement of non-functional properties of software use static approaches (51%). However, for the past ten years evolutionary (18%) and exploratory (15%) approaches have been increasingly popular.

Similarly, we distinguished between four types of software modifications. (1) *Loop transformations*, encompassing, for example, loop merging, splitting, unrolling, polyhedral transformations of nested loops, but also many other modifications specifically targeting loops in source code. (2) *Semantic*-based modifications, with, for example, template-based code generation or refactorings, which are meant to guarantee semantics preservation. (3) Potentially *destructive* modifications that do not guarantee semantics preservation; they can preserve semantics, but do not come with such guarantees, typically leaving change acceptance to a code review process. (4) *Configuration*-based modifications, in which parameter values at known decisions points are changed.

Figure 13 shows the distribution of types of software modifications by year of publication. The vast majority of work applies semantic modifications to the software at hand (238 papers). Loop transformations (99 papers) are especially prevalent in compiler work. Configuration (85 papers) has been regularly tackled from around 2005, while destructive modifications (85 papers) have been very popular in the last ten years. Unsurprisingly, the use of software destructive modifications can be linked to the increased use of evolutionary search in software engineering and the introduction of genetic improvement [313].

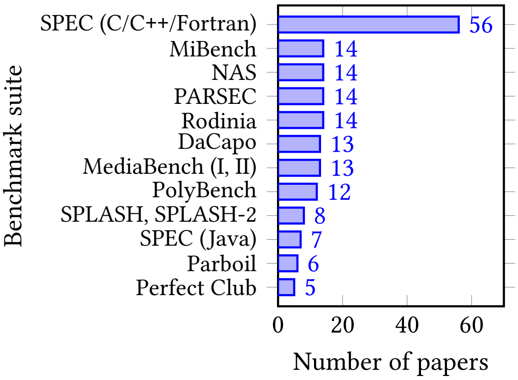

Fig. 14. Most frequent software benchmark sets (≥ 5) across all 425 unique corpus papers.

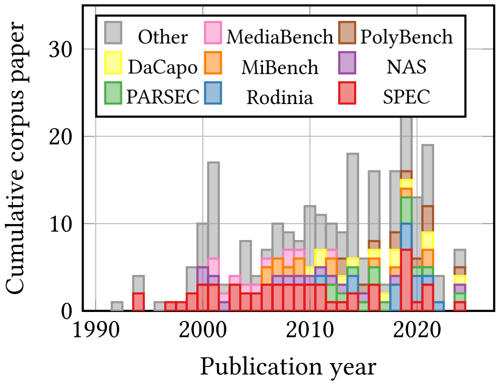

Fig. 15. Publication year of all 167 corpus papers that reuse an existing benchmark set.

Answer to RQ1 (d): Most of the work on non-functional properties of software use semantic modifications (56%). However, similarly to RQ1 (c), destructive source code changes (20%) have been increasingly popular in the past ten years. Other types of modifications are loop transformations (23%) and configuration tuning (20%), steadily appearing in the literature for the past twenty years.

3.4 Benchmarks[3]

Empirical studies on improvement of non-functional program properties are evaluated on particular sets of instances to measure performance improvement. Although almost all contain the name of the software that was improved, many lack details regarding the dataset they were evaluated on.

In what follows, we denote by *benchmark* any pair of both a given software code to execute and the necessary input data required to make execution reproducible, so that performance across different environments can be fairly and reliably compared. A *benchmark suite* is a collection of benchmarks; whilst theoretically designed to be executed as a whole, many studies only consider subsets of one or more benchmark suites to solely focus on specifically chosen benchmarks.

In 167 of the 425 papers on empirical work on improvement of non-functional properties of software authors reused existing benchmark suites (39%). Figure 14 shows the most commonly used benchmark suites. For SPEC, we distinguished Java benchmark suites (SPECjbb, SPECjvm) from the C/C++/Fortran benchmark suites (mostly SPEC CPU when specified). One issue we encountered was that many times authors simply mention “SPEC” without specifying which version of that benchmark suite was used, which specific software was used, or even which programming language was targeted. Similarly, we grouped together the original and revised versions of both MediaBench (MediaBench II) and SPLASH (SPLASH-2).

SPEC is the most frequently used benchmark suite in work on non-functional property improvement of software. This is partly due to the longevity of its various benchmark suites that are regularly updated (e.g., SPEC CPU has been revised five times since its inception). Figure 15 shows that despite a great number of benchmark suites being reused, and an increasing number of reuse taking place over the years, no particular suite appears to be prevalent. One reason might be the increasing difficulty of compiling older software on newer systems, making benchmark suites quickly outdated, issue that SPEC may avoid by updating their benchmarks suites more frequently.

---

[3] Details on all benchmarks, including URLs and references, are available at https://bloa.github.io/nfunc_survey/benchmarks

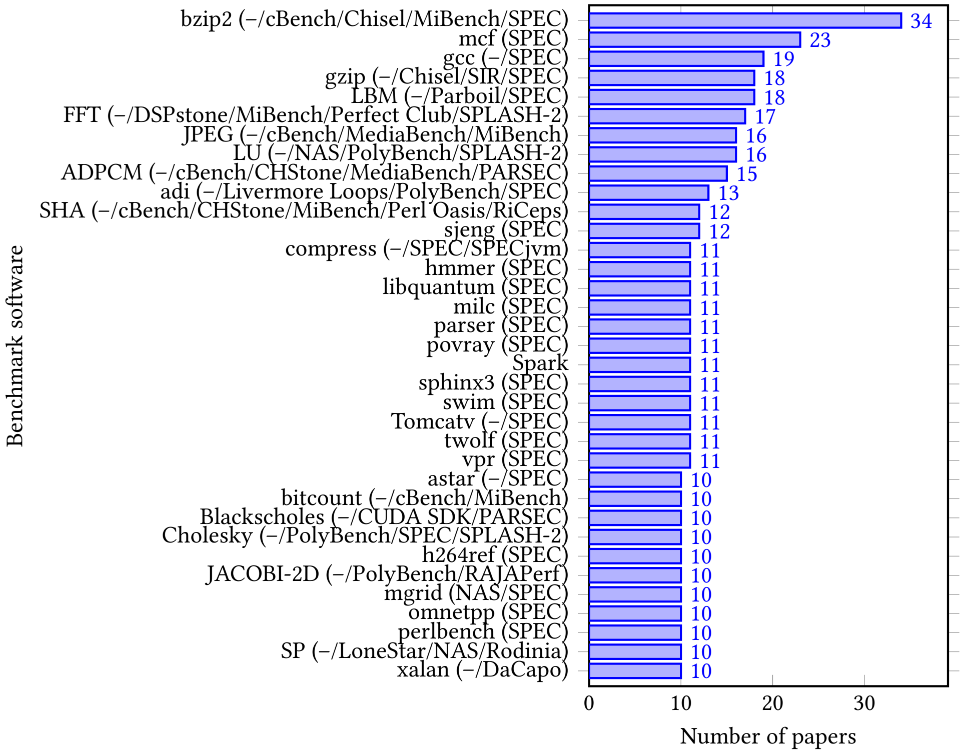

Fig. 16. Software most often targeted (≥ 10 papers) accross all 425 unique corpus papers.

Answer to RQ2 (a): Whilst many software benchmark suites have been proposed, they are used in less than half (39%) of the papers surveyed. SPEC, with its multiple types of benchmarks, is by far the most commonly reused benchmark suite. However, it was originally proposed for raw performance evaluation of hardware systems and may not be suitable for all non-functional property improvement purposes.

Figure 16 presents software most frequently targeted in the 425 papers, as well as when relevant, the benchmark suite it was explicitly taken from (we use "–" to indicate that no benchmark suite was specified in the paper). Unsurprisingly, the most frequently targeted software come from the SPEC benchmark suites (23 of the 35 software targeted ten times or more). With the notable exception of Spark (found eleven times), frequently targeted in the context of big-data software parameter configuration, software not being part of benchmark suites are not frequently reused, meaning that most studies consider new benchmark and do not directly compare to previous work.

Answer to RQ2 (b): Unsurprisingly the most often targeted software are those originating from the SPEC benchmarks, including, for example, `bzip2`, `mcf`, `gcc`, or `gzip`.

Figure 17 shows the distribution of the programming language of the targeted software relative to the publication year. Targeted software are most frequently written in C or C++, with 51% of all papers. An additional 8% of papers target GPU software (e.g., CUDA) essentially also written in C/C++. Java follows with 16% of papers, then Fortran (5%, appearing in only two papers since 2010), Scala (3%), and Javascript (2%). Other languages (including Python, Erlang, Haskell...) only appear

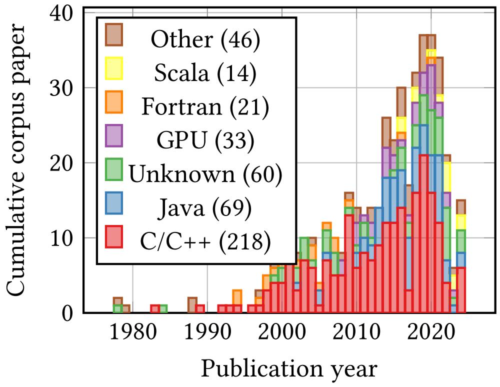

Fig. 17. Publication year distribution of all 425 corpus papers, according to the programming language of the software targeted. Papers with more than one programming language are counted multiple times.

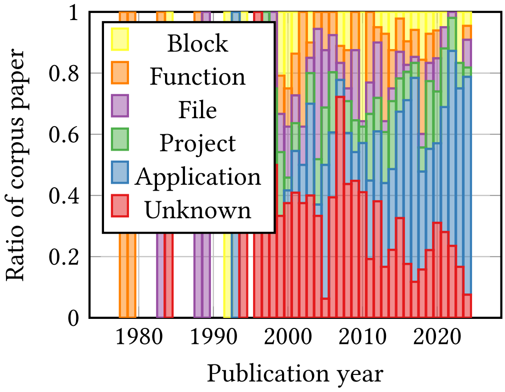

Fig. 18. Publication year distribution of all 425 unique corpus papers, according to the ratio of size of software targeted.

in 1% or fewer papers. Surprisingly, in 14% of papers the programming language of the targeted software is not explicitly stated.

As for the type of the targeted software, 59% of the papers include real-world software, usually freely available online, while 50% considered toy examples specifically written for research purposes; in 10% of the papers the origin of software was unclear. Regarding papers with real-world software, which seem to be increasingly favoured, we find that 56% is academic in origin, 49% come from the open-source community, 7% come from industry; 16% is unknown.

For each paper we noted the approximate size of the software targeted within, with categories such as "block" (a few lines of code; 11% of papers), "function" (a small number of functions; 15% of papers), "file" (typically one or two files; 17% of papers), and "project" (26%) and finally "application" (44%) for larger software with more files and a much larger code base. In 35% of papers the size of the targeted software was neither specified nor obvious. Figure 18 details proportions by publication year. Whilst results appear relatively constant, it can still be noted that the proportion of both small software (i.e., "block", "function", and "file") and software of unknown size seem to be slowly decreasing, in favour of large software.

Finally, we looked at the environment in which software was executed. Unfortunately, in 51% of the papers this was not specified and could not be inferred. In 38% of papers software was run on a Unix or Linux machine, 3% on Windows, and 2% on Mac specifically. Additionally, 7% of the papers considered software running on a mobile device (Android: 6%, iPhone: 1%).

Answer to RQ3 (a): The most frequently considered software for improvement of its non-functional behaviour is, statistically, a large real-world Linux application written in C or C++; this might be a direct consequence of the prevalence of work based on SPEC.

There are obvious discrepancies between real-world software engineering and the papers identified in our survey. In many aspects, our survey fails to reflect that richness of software development.

First, despite 59% of the papers targeting real-world software, C/C++ software is vastly over-represented (58%), as is Unix/Linux software (40%). According to the TIOBE index[4]—an indicator of online popularity—Python has recently become the most popular programming language. In

---

[4] https://www.tiobe.com/tiobe-index

contrast, only six papers in our survey (1%) consider Python software, with five also coincidentally considering C, C++, or Java software. Whilst real-world software systems often integrate multiple languages, such as Python software often relying on C/C++ libraries for performance-critical operations, our survey found no example of work specifically targeting such systems. Moreover, we posit that automating the improvement of polyglot software presents a significant challenge, requiring holistic strategies that go beyond the isolated optimisation of individual software components.

The 2024 Stack Overflow's annual developer survey[5] provides more practical insight regarding developers demographics. In particular, JavaScript (69% of professional respondents), SQL (51%), Python (41%, "most wanted language" for the fifth year), NodeJS/TypeScript (36% both), C# (30%), or Bash/Shell (28%) are all technologies very popular in industry that are apparently completely missing from academic research, as is Rust ("most loved language" for the sixth year). Likewise, most professional developers use Windows (41%, whilst 30% use MacOS and 25% use a Linux-based distribution), which again is not reflected in our own survey.

Finally, only 7% of the identified relevant work target software in mobile devices. In the meantime, in 2024 the number of active Android devices is reported to be as high as 3.9 billions (2.2 billions for iOS devices). For these mobile devices, non-functional properties such as memory usage and energy consumption are absolutely critical [186, 258].

Answer to RQ3 (b): Benchmarks revealed in our survey are not representative of real-world software as a whole, with, for example, Java (69 papers, 16%), Javascript (nine papers, 2%), or Python (six papers, 1%) software being extremely underrepresented in view of their actual popularity.

4 Recommendations

Our survey shows that there is yet no standard benchmark specifically tailored to the improvement of non-functional properties of software. Lack of such a benchmark hinders reproducibility and reliable comparisons with state of the art, hindering fast progress.

Whilst SPEC is the closest thing, it was proposed with hardware comparison and compiler optimisation in mind, and is not particularly well suited for potentially destructive evolutionary approaches. Indeed, SPEC is designed for applications in which semantic changes are not expected. As such, running the optimised software on few known inputs is sufficient to adequately compare performance. However, as soon as potential semantic changes are introduced (e.g., through parameterisation, or destructive source code changes), it becomes essential to control for correctness and generalisation. Hence, benchmarks crafted for general improvement of non-functional properties of software should ideally consider software providing a comprehensive test suite or at least software for which a large amount of input data is available.

In terms of experimental protocol, one should generally follow the example of machine learning research, that provides a variety of strong procedures to ensure that the performance improvements are generalisable and reproducible. Whilst the simplest isolated holdout method —in which training and parameterisation should be performed on data disjoint from the data used for actual performance comparison— may seem reasonable, especially for stochastic methods and methods introducing semantic changes we strongly advocate cross-validation.

Representation in terms of the choice of target software is critical. We highlight five software characteristics that should be considered when considering how techniques that improve non-functional properties of software might generalise. Whilst some opportunities for improvement *might* be universal, e.g., bloat removal, it is sensible to think that some can only be detected and specially taken care of when considering a large panel of diverse software.

---

[5] https://survey.stackoverflow.co/2024/

Programming language. Programming languages are driven by different coding paradigms and development best practices, have access to entirely different libraries, and follow different syntax. The vast majority of published research target C/C++ or Java software, ignoring the popularity of languages such as Python, JavaScript, PHP, or SQL. Work is also almost entirely centred on imperative programming features, neglecting functional functionalities essential, e.g., for languages such as Haskell or Scala.

Software size. Software size is the second most indisputable critical characteristic. Small programs can be maintained much more rigorously, expose fewer and harder inefficiencies, and overall provide much less material for repairs, which may impede some types of approaches [30]. On the other hand, large software with hundred or more files may be difficult to improve for the complete opposite reasons, as inefficiencies may be more numerous and potentially simpler to deal with, but also be much harder to locate.

Architecture/Application. There are as many types of software as there are types of applications, and each may expose different specific types of inefficiencies. It is important that research is conducted on all types of software, including for example GUI and terminal-based/command-line software, single- and multi-purpose software, model-view-controller and monolithic software, single-core and message-passing software, desktop/mobile/embedded software, or general libraries/APIs and specific applications.

Application. Similarly, the application domain the selected software targets may strongly impact the types of inefficiencies it might expose. We can cite for example system software (e.g., kernels, compilers, drivers, general utility tools), media-related software (e.g., compression, image/video), scientific software (e.g., machine learning, genomics), or games.

Number of contributors. Industry practices, as well as large open-source software, involve many contributors that often don’t share a clear or deep understanding of the system in its entirety. As with other characteristics, software written by a single developer, a small team, multiple teams, or many infrequent contributors, may expose different types of inefficiencies.

5 Threats to Validity

Keywords used in the repository search may not cover all relevant literature. Due to its restrictive nature, there might be whole types of relevant work that a keyword search is not able to reveal. To mitigate that threat, we conducted first a preliminary search to discover all potential keywords. We tried to hand-pick a large number of papers (100) using very diverse traits, including research fields, types of non-functional properties improved, programming language, various synonyms, etc. We then extracted from their titles and abstracts generic terms and performed a frequency analysis, subsequently used to select the most potentially effective keywords.

Not all major publishers have been directly queried. Indeed, online libraries such as Springer Link or Science Direct haven’t been considered due to their lack of complex query ability. To mitigate this threat, we considered a healthy combination of primary and secondary sources of work. First, we choose ACM and IEEE, two of the main publishers in computer science[6], both for conferences and journals. Then, we considered Scopus, as in addition to being the online library of Elsevier it also indexes many other publishers and in particular Springer, who publishes the Lecture Notes in Computer Science (LNCS) series covering many conference proceedings in all areas of computer science. Finally, we considered both Google Scholar and ArXiV to further increase the potential coverage of the survey.

The repository search wasn’t exhaustive. Another threat is the very high number of papers returned by the digital libraries (as clearly shown in Table 2), despite the restrictive compound

---

[6] https://www.spinellis.gr/blog/20170915/index.html

queries. We choose to consider the 200 first papers returned by each query, rather than use even more restrictive queries, to make sure not to miss papers otherwise covered. We also assume that 200 hundred papers, i.e., for example ten pages down on Google Scholar, is a reasonable limit to what would be investigated manually. The repository search methodology, adapted from [173], then ensures that such a large number of papers (up to 5000) can be effectively considered.

Relevant work may not have been correctly identified. The identification of relevant work was a long repetitive manual task, and involuntary errors may impact conclusions drawn from our survey. The main concern here is the relevance of the paper selection during the survey. To mitigate this threat, we sampled 100 entries uniformly at random from the 3749 unique papers yielded by the repository search and cross-checked that the resulting relevant work was the same when processed independently by both authors of the survey.

Finally, **the classification used in Section 3's survey may not reflect the state of the art.** In order to validate both our results and our conclusions we turned toward prominent authors of the literature. In total 1080 authors have been identified throughout our main survey step (on average 2.8 authors per paper). More precisely, we found 31 authors of three relevant papers, nine authors of four papers, and seven authors of five or more papers. All 47 authors thus identified have been contacted and their feedback has been used to improve this survey.

6 Conclusions

Our comprehensive survey of benchmarks in empirical work on the improvement of non-functional properties of software provides several key insights:

1. We observed a substantial body of literature on this topic, dating back approximately 25 years and gaining significant traction in recent years.
2. We were only able to find very few prominent software benchmarks, which poses a challenge to reproducibility and hinders fast comparison with state of the art.
3. We identified clear discrepancies between the characteristics of software studied in academic research and those reported by industry and online surveys.

To address these issues, we compiled a detailed list of benchmarks used in the literature and formulated specific recommendations for future work. Our findings emphasize the need for standardized benchmarks and a better alignment between academic research and real-world software practices. We hope our survey and the accompanying artifact will drive further research and foster improvements in the enhancement of non-functional properties of software. All the details regarding the 425 identified papers, including the raw data pertaining to the systematic survey, are available on our dedicated webpage: https://bloa.github.io/nfunc_survey.

Acknowledgments

This work was supported by UK EPSRC Fellowship EP/P023991/1. We would like to thank the members of the community who kindly provided feedback on an earlier draft of this paper.

References

[1] Khaled Abdelaal and Martin Kong. 2021. Tile size selection of affine programs for GPGPUs using polyhedral cross-compilation. In *ICS 2021*. ACM, 13–26.
[2] Sarah Abdulsalam, Donna Lakomski, Qijun Gu, Tongdan Jin, and Ziliang Zong. 2014. Program energy efficiency: The impact of language, compiler and implementation choices. In *IGCC 2014*. IEEE, 1–6.
[3] Mathieu Acher, Luc Lesoil, Georges Aaron Randrianaina, Xhevahire Tërnava, and Olivier Zendra. 2023. A Call for Removing Variability. In *VaMoS 2023*. ACM, 82–84.
[4] Felix Adler, Gordon Fraser, Eva Gründinger, Nina Körber, Simon Labrenz, Jonas Lerchenberger, Stephan Lukasczyk, and Sebastian Schweikl. 2021. Improving Readability of Scratch Programs with Search-based Refactoring. In *SCAM 2021*. IEEE, 120–130.

[5] Philipp Adler and Wolfram Amme. 2014. Speculative optimizations for interpreting environments. *Softw. Pract. Exp.* 44, 10 (2014), 1223–1249.

[6] Ioannis Agadakos, Di Jin, David Williams-King, Vasileios P. Kemerlis, and Georgios Portokalidis. 2019. Nibbler: Debloating binary shared libraries. In *ACSAC 2019*. ACM, 70–83.

[7] Aatira Anum Ahmad, Abdul Rafae Noor, Hashim Sharif, Usama Hameed, Shoaib Asif, Mubashir Anwar, Ashish Gehani, Junaid Haroon Siddiqui, and Fareed Zaffar. 2022. TRIMMER: An Automated System for Configuration-based Software Debloating. *IEEE Trans. Softw. Eng.* (2022).

[8] Dana Van Aken, Dongsheng Yang, Sebastien Brillard, Ari Fiorino, Bohan Zhang, Christian Billian, and Andrew Pavlo. 2021. An Inquiry into Machine Learning-based Automatic Configuration Tuning Services on Real-World Database Management Systems. *Proc. VLDB Endow.* 14, 7 (2021), 1241–1253.

[9] Takuya Akiba, Shotaro Sano, Toshihiko Yanase, Takeru Ohta, and Masanori Koyama. 2019. Optuna: A Next-generation Hyperparameter Optimization Framework. In *KDD 2019*. ACM, 2623–2631.

[10] Juan Pablo Sandoval Alcocer, Alexandre Bergel, and Marco Tulio Valente. 2016. Learning from Source Code History to Identify Performance Failures. In *ICPE 2016*. ACM, 37–48.

[11] Farhana Aleen, Vyacheslav P. Zakharin, Rakesh Krishnaiyer, Garima Gupta, David Kreitzer, and Chang-Sun Lin Jr. 2016. Automated compiler optimization of multiple vector loads/stores. In *CF 2016*. 82–91.

[12] Mohammad A. Alkandari, Ali Kelkawi, and Mahmoud O. Elish. 2021. An Empirical Investigation on the Effect of Code Smells on Resource Usage of Android Mobile Applications. *IEEE Access* 9 (2021), 61853–61863.

[13] Abdullah Almogahed, Mazni Omar, Nur Haryani Zakaria, Ghulam Muhammad, and Salman A. AlQahtani. 2023. Revisiting Scenarios of Using Refactoring Techniques to Improve Software Systems Quality. *IEEE Access* 11 (2023), 28800–28819.

[14] Peter Amidon, Eli Davis, Stelios Sidiroglou-Douskos, and Martin C. Rinard. 2015. Program fracture and recombination for efficient automatic code reuse. In *HPEC 2015*. IEEE, 1–6.

[15] Gabin An, Aymeric Blot, Justyna Petke, and Shin Yoo. 2019. PyGGI 2.0: Language Independent Genetic Improvement Framework. In *ESEC/FSE 2019*. ACM, 1100–1104.

[16] Hina Anwar, Dietmar Pfahl, and Satish Narayana Srirama. 2019. Evaluating the Impact of Code Smell Refactoring on the Energy Consumption of Android Applications. In *SEAA 2019*. IEEE, 82–86.

[17] Hamid Arabnejad, João Bispo, Jorge G. Barbosa, and João M. P. Cardoso. 2018. An OpenMP Based Parallelization Compiler for C Applications. In *BDCloud 2018*. IEEE, 915–923.

[18] Davide Arcelli, Vittorio Cortellessa, and Catia Trubiani. 2012. Antipattern-based model refactoring for software performance improvement. In *QoSA 2012*. ACM, 33–42.

[19] Rafael Asenjo, Rosa Castillo, Francisco Corbera, Angeles G. Navarro, Adrian Tineo, and Emilio L. Zapata. 2008. Parallelizing irregular C codes assisted by interprocedural shape analysis. In *IPDPS 2008*. IEEE, 1–12.

[20] Amir H. Ashouri, Giovanni Mariani, Gianluca Palermo, Eunjung Park, John Cavazos, and Cristina Silvano. 2016. COBAYN: Compiler Autotuning Framework Using Bayesian Networks. *ACM Trans. Archit. Code Optim.* 13, 2 (2016), 21:1–21:25.

[21] Thanassis Avgerinos and Konstantinos Sagonas. 2009. Cleaning up Erlang code is a dirty job but somebody's gotta do it. In *ERLANG 2009*. ACM, 1–10.

[22] Naeem Z. Azeemi. 2006. Compiler Directed Battery-Aware Implementation of Mobile Applications. In *ICET 2006*. IEEE, 251–256.

[23] Naeem Z. Azeemi. 2006. Multicriteria Energy Efficient Source Code Compilation for Dependable Embedded Applications. In *IIT 2006*. IEEE, 1–5.

[24] Abdel-Hameed A. Badawy, Aneesh Aggarwal, Donald Yeung, and Chau-Wen Tseng. 2001. Evaluating the impact of memory system performance on software prefetching and locality optimizations. In *ICS 2001*. ACM, 486–500.

[25] Mourad Badri, Linda Badri, Oussama Hachemane, and Alexandre Ouellet. 2017. Exploring the Impact of Clone Refactoring on Test Code Size in Object-Oriented Software. In *ICMLA 2017*. IEEE, 586–592.

[26] Otto Skrove Bagge, Karl Trygve Kalleberg, Magne Haveraaen, and Eelco Visser. 2003. Design of the CodeBoost Transformation System for Domain-Specific Optimisation of C++ Programs. In *SCAM 2003*. IEEE, 65.

[27] J. Eugene Ball. 1979. Predicting the effects of optimization on a procedure body. In *CC 1979*. ACM, 214–220.

[28] Saeid Barati, Lee Ehudin, and Hank Hoffmann. 2021. NEAT: A Framework for Automated Exploration of Floating Point Approximations. *CoRR* abs/2102.08547 (2021).

[29] Rajkishore Barik, Jisheng Zhao, and Vivek Sarkar. 2013. Interprocedural strength reduction of critical sections in explicitly-parallel programs. In *PACT 2013*. IEEE, 29–40.

[30] Earl T. Barr, Yuriy Brun, Premkumar T. Devanbu, Mark Harman, and Federica Sarro. 2014. The plastic surgery hypothesis. In *SIGSOFT FSE 2014*. ACM, 306–317.

[31] Alberto Bartoli, Andrea De Lorenzo, Eric Medvet, and Fabiano Tarlao. 2014. Playing regex golf with genetic programming. In *GECCO 2014*. ACM, 1063–1070.

[32] Rajeev Barua, Walter Lee, Saman P. Amarasinghe, and Anant Agarwal. 2001. Compiler Support for Scalable and Efficient Memory Systems. *IEEE Trans. Computers* 50, 11 (2001), 1234–1247.

[33] Michail Basios, Lingbo Li, Fan Wu, Leslie Kanthan, and Earl T. Barr. 2018. Darwinian data structure selection. In *ESEC/FSE 2018*. ACM, 118–128.

[34] Muthu Manikandan Baskaran, J. Ramanujam, and P. Sadayappan. 2010. Automatic C-to-CUDA Code Generation for Affine Programs *(LNCS, Vol. 6011)*. Springer, 244–263.

[35] Thomas Henry Beach and Nicholas J. Avis. 2009. An Intelligent Semi-Automatic Application Porting System for Application Accelerators. In *UCHPC-MAW@CF 2009*. ACM.

[36] Alexandre Bergel, Stéphane Ducasse, and Oscar Nierstrasz. 2005. Classbox/J: Controlling the scope of change in Java. In *OOPSLA 2005*. ACM, 177–189.

[37] Benoît Bertholon, Sébastien Varrette, and Sebastien Martinez. 2013. ShadObf: A C-Source Obfuscator Based on Multi-objective Optimisation Algorithms. In *NIDISC@IPDPS 2013 in IPDPS 2013 Workshops*. IEEE, 435–444.

[38] Suparna Bhattacharya, Kanchi Gopinath, and Mangala Gowri Nanda. 2013. Combining concern input with program analysis for bloat detection. In *OOPSLA 2013*. ACM, 745–764.

[39] Suparna Bhattacharya, Mangala Gowri Nanda, Kanchi Gopinath, and Manish Gupta. 2011. Reuse, Recycle to De-bloat Software. In *ECOOP 2011 (LNCS, Vol. 6813)*. Springer, 408–432.

[40] Suparna Bhattacharya, Karthick Rajamani, Kanchi Gopinath, and Manish Gupta. 2012. Does lean imply green? – A study of the power performance implications of Java runtime bloat. In *SIGMETRICS 2012*. ACM, 259–270.

[41] Priyam Biswas, Nathan Burow, and Mathias Payer. 2021. Code Specialization through Dynamic Feature Observation. In *CODASPY 2021*. ACM, 257–268.

[42] Brian Blaha and Donald C. Wunsch II. 2002. Evolutionary programming to optimize an assembly program. In *CEC 2002*. IEEE, 1901–1903.

[43] Aymeric Blot, Holger H. Hoos, Laetitia Jourdan, Marie-Éléonore Kessaci-Marmion, and Heike Trautmann. 2016. MO-ParamILS: A Multi-objective Automatic Algorithm Configuration Framework. In *LION 10 (LNCS, Vol. 10079)*. Springer, 32–47.

[44] Aymeric Blot and Justyna Petke. 2020. Comparing Genetic Programming Approaches for Non-Functional Genetic Improvement – Case Study: Improvement of MiniSAT's Running Time. In *EuroGP 2020 (LNCS, Vol. 12101)*. Springer, 68–83.

[45] Aymeric Blot and Justyna Petke. 2021. Empirical Comparison of Search Heuristics for Genetic Improvement of Software. *IEEE Trans. Evol. Comput.* 25, 5 (2021), 1001–1011.

[46] Mahmoud A. Bokhari, Bradley Alexander, and Markus Wagner. 2018. In-vivo and offline optimisation of energy use in the presence of small energy signals – A case study on a popular Android library. In *MobiQuitous 2018*. ACM, 207–215.

[47] Mahmoud A. Bokhari, Bobby R. Bruce, Brad Alexander, and Markus Wagner. 2017. Deep Parameter Optimisation on Android Smartphones for Energy Minimisation – A Tale of Woe and a Proof-of-Concept. In *GI@GECCO 2017 in GECCO 2017 companion*. ACM, 1501–1508.

[48] Murat Bolat and Xiaoming Li. 2009. Context-aware code optimization. In *IPCCC 2009*. IEEE, 256–263.

[49] Uday Bondhugula, Albert Hartono, J. Ramanujam, and P. Sadayappan. 2008. A practical automatic polyhedral parallelizer and locality optimizer. In *PLDI 2008*. ACM, 101–113.

[50] Talal Bonny and Jörg Henkel. 2008. FBT: Filled buffer technique to reduce code size for VLIW processors. In *ICCAD 2008*. IEEE, 549–554.

[51] Martí Bosch, Pierre Genevès, and Nabil Layaïda. 2014. Automated refactoring for size reduction of CSS style sheets. In *DocEng 2014*. ACM, 13–16.

[52] David Branco and Pedro Rangel Henriques. 2015. Impact of GCC optimization levels in energy consumption during C/C++ program execution. In *ISCI 2015*. IEEE, 52–56.

[53] Jon Brandvein and Yanhong A. Liu. 2016. Removing runtime overhead for optimized object queries. In *PEPM 2016*. ACM, 73–84.

[54] Déaglán Connolly Bree and Mel Ó Cinnéide. 2020. Inheritance versus Delegation: which is more energy efficient?. In *IWoR@ICSE 2020 in ICSE 2020 Workshops*. ACM, 323–329.

[55] Christopher Brown and Simon J. Thompson. 2010. Clone detection and elimination for Haskell. In *PEPM 2010*. ACM, 111–120.

[56] Michael D. Brown and Santosh Pande. 2019. CARVE: Practical Security-Focused Software Debloating Using Simple Feature Set Mappings. In *FEAST@CCS 2019*. ACM, 1–7.

[57] Michael D. Brown and Santosh Pande. 2019. Is Less Really More? Towards Better Metrics for Measuring Security Improvements Realized Through Software Debloating. In *CSET 2019*. USENIX.

[58] Alexander E. I. Brownlee, Nathan Burles, and Jerry Swan. 2017. Search-Based Energy Optimization of Some Ubiquitous Algorithms. *IEEE Trans. Emerg. Top. Comput. Intell.* 1, 3 (2017), 188–201.

[59] Alexander E. I. Brownlee, Michael G. Epitropakis, Jeroen Mulder, Marc Paelinck, and Edmund K. Burke. 2022. A systematic approach to parameter optimization and its application to flight schedule simulation software. *J. Heuristics* 28, 4 (2022), 509–538.

[60] Alexander E. I. Brownlee, Justyna Petke, and Anna F. Rasburn. 2020. Injecting Shortcuts for Faster Running Java Code. In *CEC 2020*. IEEE, 1–8.

[61] Bobby R. Bruce, Justyna Petke, and Mark Harman. 2015. Reducing Energy Consumption Using Genetic Improvement. In *GECCO 2015*. ACM, 1327–1334.

[62] Bobby R. Bruce, Justyna Petke, Mark Harman, and Earl T. Barr. 2019. Approximate Oracles and Synergy in Software Energy Search Spaces. *IEEE Trans. Softw. Eng.* 45, 11 (2019), 1150–1169.

[63] Bobby R. Bruce, Tianyi Zhang, Jaspreet Arora, Guoqing Harry Xu, and Miryung Kim. 2020. JShrink: In-depth investigation into debloating modern Java applications. In *ESEC/FSE 2020*. ACM, 135–146.

[64] Iulian Brumar, Marc Casas, Miquel Moretó, Mateo Valero, and Gurindar S. Sohi. 2017. ATM: Approximate Task Memoization in the Runtime System. In *IPDPS 2017*. IEEE, 1140–1150.

[65] Hugo Brunie, Costin Iancu, Khaled Z. Ibrahim, Philip Brisk, and Brandon Cook. 2020. Tuning floating-point precision using dynamic program information and temporal locality. In *SC 2020*. IEEE, 50.

[66] Nathan Burles, Edward Bowles, Alexander E. I. Brownlee, Zoltan A. Kocsis, Jerry Swan, and Nadarajen Veerapen. 2015. Object-Oriented Genetic Improvement for Improved Energy Consumption in Google Guava. In *SSBSE 2015 (LNCS, Vol. 9275)*. Springer, 255–261.

[67] Felipe Canales, Geoffrey Hecht, and Alexandre Bergel. 2021. Optimization of Java Virtual Machine Flags using Feature Model and Genetic Algorithm. In *ICPE 2021*. ACM, 183–186.

[68] Anthony Canino, Yu David Liu, and Hidehiko Masuhara. 2018. Stochastic energy optimization for mobile GPS applications. In *ESEC/FSE 2018*. ACM, 703–713.

[69] Heling Cao, YangXia Meng, Jianshu Shi, Lei Li, Tiaoli Liao, and Chenyang Zhao. 2020. A Survey on Automatic Bug Fixing. In *ISSR 2020*. IEEE, 122–131.

[70] Rong Cao, Liang Bao, Kaibi Zhao, and Panpan Zhangsun. 2024. ETune: Efficient configuration tuning for big-data software systems via configuration space reduction. *J. Syst. Softw.* 209 (2024), 111936.

[71] Antonin Carette, Mehdi Adel Ait Younes, Geoffrey Hecht, Naouel Moha, and Romain Rouvoy. 2017. Investigating the energy impact of Android smells. In *SANER 2017*. IEEE, 115–126.

[72] Steve Carr and Ken Kennedy. 1994. Improving the Ratio of Memory Operations to Floating-Point Operations in Loops. *ACM Trans. Program. Lang. Syst.* 16, 6 (1994), 1768–1810.

[73] Tiago Carvalho and João M. P. Cardoso. 2018. An approach based on a DSL + API for programming runtime adaptivity and autotuning concerns. In *SAC 2018*. ACM, 1211–1220.

[74] John Cavazos, Grigori Fursin, Felix V. Agakov, Edwin V. Bonilla, Michael F. P. O'Boyle, and Olivier Temam. 2007. Rapidly Selecting Good Compiler Optimizations using Performance Counters. In *CGO 2007*. IEEE, 185–197.

[75] Milind Chabbi, Jin Lin, and Raj Barik. 2021. An Experience with Code-Size Optimization for Production iOS Mobile Applications. In *CGO 2021*. 363–377.

[76] Yin Chan, Ashok Sudarsanam, and Andrew Wolfe. 1994. The effect of compiler-flag tuning on SPEC benchmark performance. *ACM SIGARCH Comput. Archit. News* 22, 4 (1994), 60–70.

[77] Daniel G. Chavarría-Miranda and John M. Mellor-Crummey. 2002. An Evaluation of Data-Parallel Compiler Support for Line-Sweep Applications. In *PACT 2002*. IEEE, 7–17.

[78] Shuai Che, Jeremy W. Sheaffer, and Kevin Skadron. 2011. Dymaxion: Optimizing memory access patterns for heterogeneous systems. In *SC 2011*. IEEE, 13:1–13:11.

[79] Yonggang Che, Zhenghua Wang, and Xiaomei Li. 2005. Reduction transformations for optimization parameter selection. In *HPC Asia 2005*. IEEE, 281–288.

[80] N. A. B. Sankar Chebolu and Rajeev Wankar. 2016. Comparative study of the impact of processor architecture on compiler tuning. In *ICACCI 2016*. IEEE, 585–592.

[81] Sunbal Cheema and Gul N. Khan. 2023. GPU Auto-tuning Framework for Optimal Performance and Power Consumption. In *GPGPU 2023*. ACM, 1–6.

[82] Boyuan Chen, Zhen Ming Jiang, Paul Matos, and Michael Lacaria. 2019. An Industrial Experience Report on Performance-Aware Refactoring on a Database-Centric Web Application. In *ASE 2019*. IEEE, 653–664.

[83] Chao Chen, Jinhan Xin, and Zhibin Yu. 2024. TIE: Fast Experiment-Driven ML-Based Configuration Tuning for In-Memory Data Analytics. *IEEE Trans. Computers* 73, 5 (2024), 1233–1247.

[84] Jie Chen and Guru Venkataramani. 2016. enDebug: A hardware-software framework for automated energy debugging. *J. Parallel Distributed Comput.* 96 (2016), 121–133.

[85] Junjie Chen, Ningxin Xu, Peiqi Chen, and Hongyu Zhang. 2021. Efficient Compiler Autotuning via Bayesian Optimization. In *ICSE 2021*. IEEE, 1198–1209.

[86] Tao Chen and Miqing Li. 2021. Multi-objectivizing software configuration tuning. In *ESEC/FSE 2021*. ACM, 453–465.

[87] Tao Chen and Miqing Li. 2024. Adapting Multi-objectivized Software Configuration Tuning. *Proc. ACM Softw. Eng.* 1, FSE (2024), 539–561.

[88] John J. Cherian, Andrew G. Taube, Robert T. McGibbon, Panagiotis Angelikopoulos, Guy Blanc, Michael Snarski, Daniel D. Richman, John L. Klepeis, and David E. Shaw. 2020. Efficient hyperparameter optimization by way of PAC-Bayes bound minimization. *CoRR* abs/2008.06431 (2020).

[89] Ioannis Chionis, Maria Chroni, and Stavros D. Nikolopoulos. 2013. Evaluating the WaterRpg software watermarking model on Java application programs. In *PCI 2013*. ACM, 144–151.

[90] Youngchul Cho and Kiyoung Choi. 2009. Code decomposition and recomposition for enhancing embedded software performance. In *ASP-DAC 2009*. IEEE, 624–629.

[91] Eui-Young Chung, Luca Benini, and Giovanni De Micheli. 2001. Source code transformation based on software cost analysis. In *ISSS 2001*. ACM/IEEE, 153–158.

[92] Eui-Young Chung, Luca Benini, Giovanni De Micheli, Gabriele Luculli, and Marco Carilli. 2002. Value-sensitive automatic code specialization for embedded software. *IEEE Trans. Comput. Aided Des. Integr. Circuits Syst.* 21, 9 (2002), 1051–1067.

[93] Brendan Cody-Kenny, Michael Fenton, Adrian Ronayne, Eoghan Considine, Thomas McGuire, and Michael O'Neill. 2017. A search for improved performance in regular expressions. In *GECCO 2017*. ACM, 1280–1287.

[94] Brendan Cody-Kenny, Edgar Galvan Lopez, and Stephen Barrett. 2015. locoGP: Improving Performance by Genetic Programming Java Source Code. In *GI@GECCO 2015 in GECCO 2015 companion*. ACM, 811–818.

[95] Brendan Cody-Kenny, Michael O'Neill, and Stephen Barrett. 2018. Performance Localisation. In *GI@ICSE 2018*. ACM, 27–34.

[96] Alessio Colucci, Dávid Juhász, Martin Mosbeck, Alberto Marchisio, Semeen Rehman, Manfred Kreutzer, Günther Nadbath, Axel Jantsch, and Muhammad Shafique. 2021. MLComp: A Methodology for Machine Learning-based Performance Estimation and Adaptive Selection of Pareto-Optimal Compiler Optimization Sequences. In *DATE 2021*. IEEE, 108–113.

[97] Jason Cong, Bin Liu, and Zhiru Zhang. 2009. Behavior-level observability don't-cares and application to low-power behavioral synthesis. In *ISLPED 2009*. ACM, 139–144.

[98] Keith D. Cooper and Timothy J. Harvey. 1998. Compiler-Controlled Memory. In *ASPLOS 1998*. ACM, 2–11.

[99] Keith D. Cooper, Philip J. Schielke, and Devika Subramanian. 1999. Optimizing for Reduced Code Space using Genetic Algorithms. In *LCTES 1999*. ACM, 1–9.

[100] Jared Coplin and Martin Burtscher. 2015. Effects of source-code optimizations on GPU performance and energy consumption. In *GPGPU@PPoPP 2015*. ACM, 48–58.

[101] Alexandre Courbot, Mariela Pavlova, Gilles Grimaud, and Jean-Jacques Vandewalle. 2006. A Low-Footprint Java-to-Native Compilation Scheme Using Formal Methods. In *CARDIS 2006 (LNCS, Vol. 3928)*. Springer, 329–344.

[102] Luis Cruz and Rui Abreu. 2017. Performance-Based Guidelines for Energy Efficient Mobile Applications. In *MOBILESoft 2017*. IEEE, 46–57.

[103] Luis Cruz, Rui Abreu, John C. Grundy, Li Li, and Xin Xia. 2019. Do Energy-Oriented Changes Hinder Maintainability?. In *ICSME 2019*. IEEE, 29–40.

[104] Luis Cruz, Rui Abreu, and Jean-Noel Rouvignac. 2017. Leafactor: Improving Energy Efficiency of Android Apps via Automatic Refactoring. In *MOBILESoft 2017*. IEEE, 205–206.

[105] Zoltan Czako, Gheorghe Sebestyen, and Anca Hangan. 2021. AutomaticAI: A hybrid approach for automatic artificial intelligence algorithm selection and hyperparameter tuning. *Expert Syst. Appl.* 182 (2021), 115225.

[106] Anderson Faustino da Silva, Bernardo N. B. de Lima, and Fernando Magno Quintão Pereira. 2021. Exploring the space of optimization sequences for code-size reduction – Insights and tools. In *CC 2021*. ACM, 47–58.

[107] Anthony Danalis, Lori L. Pollock, and D. Martin Swany. 2007. Automatic MPI application transformation with ASPhALT. In *IPDPS 2007*. IEEE, 1–8.

[108] Eva Darulova, Einar Horn, and Saksham Sharma. 2018. Sound mixed-precision optimization with rewriting. In *ICCPS 2018*. IEEE/ACM, 208–219.

[109] Roshan Dathathri, Chandan Reddy, Thejas Ramashekar, and Uday Bondhugula. 2013. Generating efficient data movement code for heterogeneous architectures with distributed-memory. In *PACT 2013*. IEEE, 375–386.

[110] Shuhaizar Daud, R. Badlishah Ahmad, and Nukala S. Murthy. 2009. The effects of compiler optimisations on embedded system power consumption. *Int. J. Inf. Commun. Technol.* 2, 1/2 (2009), 73–82.

[111] Chirag Dave and Rudolf Eigenmann. 2009. Automatically Tuning Parallel and Parallelized Programs. In *LCPC 2009 (LNCS, Vol. 5898)*. Springer, 126–139.

[112] Fábio de Almeida Farzat, Márcio de Oliveira Barros, and Guilherme H. Travassos. 2018. Challenges on applying genetic improvement in JavaScript using a high-performance computer. *J. Softw. Eng. Res. Dev.* 6 (2018), 12.

[113] Fábio de Almeida Farzat, Márcio de Oliveira Barros, and Guilherme H. Travassos. 2021. Evolving JavaScript Code to Reduce Load Time. *IEEE Trans. Softw. Eng.* 47, 8 (2021), 1544–1558.

[114] Ewerton Daniel de Lima, Tiago Cariolano de Souza Xavier, Anderson Faustino da Silva, and Linnyer Beatrys Ruiz. 2013. Compiling for performance and power efficiency. In *PATMOS 2013*. IEEE, 142–149.

[115] Frédéric de Mesmay, Arpad Rimmel, Yevgen Voronenko, and Markus Püschel. 2009. Bandit-based optimization on graphs with application to library performance tuning. In *ICML 2009 (ACM ICPS, Vol. 382)*. ACM, 729–736.

[116] Saumya K. Debray and William S. Evans. 2002. Profile-Guided Code Compression. In *PLDI 2002*. ACM, 95–105.

[117] Zhaochu Deng, Jianjiang Li, and Jie Lin. 2021. A Synchronization Optimization Technique for OpenMP. In *ICCRD 2021*. IEEE, 95–103.

[118] Nikita P Desai. 2009. A Novel Technique for Orchestration of Compiler Optimization Functions Using Branch and Bound Strategy. In *IADCC 2009*. IEEE, 467–472.

[119] Diego Desani, Veronica Gil-Costa, Cesar Augusto Cavalheiro Marcondes, and Hermes Senger. 2016. Black-Box Optimization of Hadoop Parameters Using Derivative-Free Optimization. In *PDP 2016*. IEEE, 43–50.

[120] Breanna Devore-McDonald and Emery D. Berger. 2020. Mossad: Defeating software plagiarism detection. *Proc. ACM Program. Lang.* 4, OOPSLA (2020), 138:1–138:28.

[121] Matthew DeVuyst, Dean M. Tullsen, and Seon Wook Kim. 2011. Runtime parallelization of legacy code on a transactional memory system. In *HiPEAC 2011*. ACM, 127–136.

[122] Lamia Djoudi and William Jalby. 2008. Automatic Analysis for Managing and Optimizing Performance-Code Quality. In *SAW 2008*. ACM, 30–38.

[123] Sébastien Doeraene and Tobias Schlatter. 2016. Parallel incremental whole-program optimizations for Scala.js. In *OOPSLA 2016*. ACM, 59–73.

[124] Jonathan Dorn, Jeremy Lacomis, Westley Weimer, and Stephanie Forrest. 2019. Automatically Exploring Tradeoffs Between Software Output Fidelity and Energy Costs. *IEEE Trans. Softw. Eng.* 45, 3 (2019), 219–236.

[125] Hui Dou, Yilun Wang, Yiwen Zhang, and Pengfei Chen. 2022. DeepCAT: A Cost-Efficient Online Configuration Auto-Tuning Approach for Big Data Frameworks. In *ICPP 2022*. ACM, 67:1–67:11.

[126] Hui Dou, Xing Wei, Kang Wang, Yiwen Zhang, Pengfei Chen, and Yuee Huang. 2023. EFTuner: A Bi-Objective Configuration Parameter Auto-Tuning Method Towards Energy-Efficient Big Data Processing. In *2023*. ACM, 292–301.

[127] Milenko Drinic, Darko Kirovski, and Hoi Vo. 2003. Code Optimization for Code Compression. In *CGO 2003*. IEEE, 315–324.

[128] Junhan Duan, Yudi Yang, Jie Zhou, and John Criswell. 2020. Refactoring the FreeBSD Kernel with Checked C. In *SecDev 2020*. IEEE, 15–22.

[129] Hassan Eldib and Chao Wang. 2013. An SMT based method for optimizing arithmetic computations in embedded software code. In *FMCAD 2013*. IEEE, 129–136.

[130] Jianbin Fang, Henk J. Sips, Pekka Jääskeläinen, and Ana Lucia Varbanescu. 2014. Grover: Looking for Performance Improvement by Disabling Local Memory Usage in OpenCL Kernels. In *ICPP 2014*. IEEE, 162–171.

[131] Iffat Fatima, Hina Anwar, Dietmar Pfahl, and Usman Qamar. 2020. Detection and Correction of Android-specific Code Smells and Energy Bugs – An Android Lint Extension. In *QuASoQ@APSEC 2020 (CEUR, Vol. 2767)*. CEUR-WS.org, 71–78.

[132] Yunsi Fei, Srivaths Ravi, Anand Raghunathan, and Niraj K. Jha. 2004. Energy-Optimizing Source Code Transformations for OS-driven Embedded Software. In *VLSI Design 2004*. IEEE, 261–266.

[133] Yunsi Fei, Srivaths Ravi, Anand Raghunathan, and Niraj K. Jha. 2007. Energy-optimizing source code transformations for operating system-driven embedded software. *ACM Trans. Archit. Code Optim.* 7, 1 (2007), 2:1–2:26.

[134] Dustin Feld, Thomas Soddemann, Michael Jünger, and Sven Mallach. 2015. Hardware-Aware Automatic Code-Transformation to Support Compilers in Exploiting the Multi-Level Parallel Potential of Modern CPUs. In *COSMIC@CGO 2015*. ACM, 2:1–2:10.

[135] Benito Fernandes, Gustavo Pinto, and Fernando Castor. 2017. Assisting non-specialist developers to build energy-efficient software. In *ICSE 2017*. IEEE, 158–160.

[136] James Fischer, Vincent Natoli, and David Richie. 2006. Optimization of LAMMPS. In *HPCMP-UGC 2006*. 374–377.

[137] Subash Chandar G., Mahesh Mehendale, and R. Govindarajan. 2006. Area and Power Reduction of Embedded DSP Systems using Instruction Compression and Re-configurable Encoding. *J. VLSI Signal Process.* 44, 3 (2006), 245–267.

[138] Unai Garciarena and Roberto Santana. 2016. Evolutionary optimization of compiler flag selection by learning and exploiting flags interactions. In *GI@GECCO 2016 in GECCO 2016 companion*. ACM, 1159–1166.

[139] Luca Gazzola, Daniela Micucci, and Leonardo Mariani. 2019. Automatic Software Repair: A Survey. *IEEE Trans. Softw. Eng.* 45, 1 (2019), 34–67.

[140] K. Geetha and N. Ammasai Gounden. 2008. Compressed Instruction Set Coding (CISC) for Performance Optimization of Hand Held Devices. In *ADCOM 2008*. 241–247.

[141] Léonard Gérard, Adrien Guatto, Cédric Pasteur, and Marc Pouzet. 2012. A modular memory optimization for synchronous data-flow languages – Application to arrays in a lustre compiler. In *LCTES 2012*. ACM, 51–60.

[142] Masoud Ghaffarinia and Kevin W. Hamlen. 2019. Binary Control-Flow Trimming. In *CCS 2019*. ACM, 1009–1022.

[143] Fady Ghanim, Uzi Vishkin, and Rajeev Barua. 2018. Easy PRAM-Based High-Performance Parallel Programming with ICE. *IEEE Trans. Parallel Distributed Syst.* 29, 2 (2018), 377–390.

[144] Seyedhamed Ghavamnia, Tapti Palit, Shachee Mishra, and Michalis Polychronakis. 2020. Temporal System Call Specialization for Attack Surface Reduction. In *USENIX Security 2020*. USENIX, 1749–1766.

[145] Somnath Ghosh, Margaret Martonosi, and Sharad Malik. 1999. Cache miss equations: a compiler framework for analyzing and tuning memory behavior. *ACM Trans. Program. Lang. Syst.* 21, 4 (1999), 703–746.

[146] J. Paul Gibson, Thomas F. Dowling, and Brian A. Malloy. 2000. The Application of Correctness Preserving Transformations to Software Maintenance. In *ICSM 2000*. IEEE, 108.

[147] Sylvain Girbal, Nicolas Vasilache, Cédric Bastoul, Albert Cohen, David Parello, Marc Sigler, and Olivier Temam. 2006. Semi-Automatic Composition of Loop Transformations for Deep Parallelism and Memory Hierarchies. *Int. J. Parallel Program.* 34, 3 (2006), 261–317.

[148] Dirk Grunwald, Benjamin G. Zorn, and Robert Henderson. 1993. Improving the Cache Locality of Memory Allocation. In *PLDI 1993*. ACM, 177–186.

[149] Ruidong Gu and Michela Becchi. 2020. GPU-FPtuner: Mixed-precision Auto-tuning for Floating-point Applications on GPU. In *HiPC 2020*. IEEE, 294–304.

[150] Giovani Guizzo, Justyna Petke, Federica Sarro, and Mark Harman. 2021. Enhancing Genetic Improvement of Software with Regression Test Selection. In *ICSE 2021*. IEEE, 1323–1333.

[151] Jingzhe Guo and Mingsheng Ying. 2020. Software Pipelining for Quantum Loop Programs. *CoRR* abs/2012.12700 (2020).

[152] Sumit Gupta, Rajesh K. Gupta, Miguel Miranda, and Francky Catthoor. 2000. Analysis of High-Level Address Code Transformations for Programmable Processors. In *DATE 2000*. IEEE/ACM, 9–13.

[153] Irene Lizeth Manotas Gutiérrez, Lori L. Pollock, and James Clause. 2014. SEEDS: A software engineer's energy-optimization decision support framework. In *ICSE 2014*. ACM, 503–514.

[154] Vladimír Guzma, Teemu Pitkänen, and Jarmo Takala. 2011. Effects of loop unrolling and use of instruction buffer on processor energy consumption. In *SoC 2011*. IEEE, 82–85.

[155] Gadi Haber, Moshe Klausner, Vadim Eisenberg, Bilha Mendelson, and Maxim Gurevich. 2003. Optimization Opportunities Created by Global Data Reordering. In *CGO 2003*. IEEE, 228–240.

[156] Bastian Hagedorn, Larisa Stoltzfus, Michel Steuwer, Sergei Gorlatch, and Christophe Dubach. 2018. High performance stencil code generation with lift. In *CGO 2018*. ACM, 100–112.

[157] Ashok Halambi, Aviral Shrivastava, Partha Biswas, Nikil D. Dutt, and Alexandru Nicolau. 2002. An Efficient Compiler Technique for Code Size Reduction Using Reduced Bit-Width ISAs. In *DATE 2002*. IEEE, 402–408.

[158] Oumayma Hamdi, Ali Ouni, Eman Abdullah AlOmar, Mel Ó Cinnéide, and Mohamed Wiem Mkaouer. 2021. An Empirical Study on the Impact of Refactoring on Quality Metrics in Android Applications. In *MOBILESoft 2021*. IEEE, 28–39.

[159] Hamza Hamza and Steve Counsell. 2013. Exploiting slicing and patterns for RTSJ immortal memory optimization. In *PPPJ 2013*. ACM, 159–164.

[160] Masayo Haneda, Peter M. W. Knijnenburg, and Harry A. G. Wijshoff. 2006. On the impact of data input sets on statistical compiler tuning. In *IPDPS 2006*. IEEE.

[161] Saemundur O. Haraldsson, Ragnheidur D. Brynjolfsdottir, John R. Woodward, Kristin Siggeirsdottir, and Vilmundur Gudnason. 2017. The use of predictive models in dynamic treatment planning. In *ISCC 2017*. IEEE, 242–247.

[162] Saemundur O. Haraldsson, John R. Woodward, Alexander E. I. Brownlee, Albert V. Smith, and Vilmundur Gudnason. 2017. Genetic Improvement of Runtime and its fitness landscape in a Bioinformatics Application. In *GI@GECCO 2017 in GECCO 2017 companion*. ACM, 1521–1528.

[163] Haochen He, Zhouyang Jia, Shanshan Li, Yue Yu, Chenglong Zhou, Qing Liao, Ji Wang, and Xiangke Liao. 2022. Multi-Intention-Aware Configuration Selection for Performance Tuning. In *ICSE 2022*. ACM, 1431–1442.

[164] Geoffrey Hecht, Naouel Moha, and Romain Rouvoy. 2016. An empirical study of the performance impacts of Android code smells. In *MOBILESoft 2016*. ACM, 59–69.

[165] Kris Heid, Jakob Wenzel, and Christian Hochberger. 2018. Improved Parallelization of Legacy Embedded Software on Soft-Core MPSoCs through Automatic Loop Transformations. In *FSP@FPL 2018*. 1–8.

[166] Jari Heikkinen, Tommi Rantanen, Andrea G. M. Cilio, Jarmo Takala, and Henk Corporaal. 2003. Immediate optimization for compressed transport triggered architecture instructions. In *ISSOC 2003*. IEEE, 65–68.

[167] John L. Hennessy and Thomas R. Gross. 1983. Postpass Code Optimization of Pipeline Constraints. *ACM Trans. Program. Lang. Syst.* 5, 3 (1983), 422–448.

[168] Kihong Heo, Woosuk Lee, Pardis Pashakhanloo, and Mayur Naik. 2018. Effective Program Debloating via Reinforcement Learning. In *CSS 2018*. ACM, 380–394.

[169] Oscar R. Hernandez, Barbara M. Chapman, and Haoqiang Jin. 2008. A performance tuning methodology with compiler support. *Sci. Program.* 16, 2-3 (2008), 135–153.

[170] Karine Heydemann, François Bodin, and Henri-Pierre Charles. 2005. A Software-Only Compression System for Trading-Offs between Performance and Code Size. In *SCOPES 2005 (ACM ICPS, Vol. 136)*. ACM, 27–36.

[171] Justin Holewinski, Louis-Noël Pouchet, and P. Sadayappan. 2012. High-performance code generation for stencil computations on GPU architectures. In *ICS 2012*. ACM, 311–320.

[172] Timo Hönig, Heiko Janker, Christopher Eibel, Oliver Mihelic, and Rüdiger Kapitza. 2014. Proactive Energy-Aware Programming with PEEK. In *TRIOS 2014*. USENIX, 6.

[173] Max Hort, Maria Kechagia, Federica Sarro, and Mark Harman. 2022. A Survey of Performance Optimization for Mobile Applications. *IEEE Trans. Softw. Eng.* (2022).

[174] Kenneth Hoste, Andy Georges, and Lieven Eeckhout. 2010. Automated just-in-time compiler tuning. In *CGO 2010*. ACM, 62–72.

[175] Zhen Huang. 2024. Debloating Feature-Rich Closed-Source Windows Software. In *SANER 2024*. IEEE, 400–405.

[176] Shih-Hao Hung, Chia-Heng Tu, Huang-Sen Lin, and Chi-Meng Chen. 2009. An Automatic Compiler Optimizations Selection Framework for Embedded Applications. In *ICESS 2009*. IEEE, 381–387.

[177] Nguyen Hung-Cuong, Huynh Quyet Thang, and Tru Ba-Vuong. 2013. Rule-Based Techniques Using Abstract Syntax Tree for Code Optimization and Secure Programming in Java. In *ICCASA 2013 (LNICST, Vol. 128)*. Springer, 168–177.

[178] Frank Hutter, Holger H. Hoos, and Kevin Leyton-Brown. 2011. Sequential Model-Based Optimization for General Algorithm Configuration. In *LION 5 (LNCS, Vol. 6683)*. Springer, 507–523.

[179] Frank Hutter, Holger H. Hoos, Kevin Leyton-Brown, and Thomas Stützle. 2009. ParamILS: An Automatic Algorithm Configuration Framework. *J. Artif. Intell. Res.* 36 (2009), 267–306.

[180] Wen-mei W. Hwu and Pohua P. Chang. 1989. Achieving High Instruction Cache Performance with an Optimizing Compiler. In *ISCA 1989*. ACM, 242–251.

[181] Mostafa E. A. Ibrahim, Markus Rupp, and S. E.-D. Habib. 2009. Compiler-based optimizations impact on embedded software power consumption. In *NEWCAS 2009*. IEEE, 1–4.

[182] ibrahim Şanlıalp, Muhammed Maruf Öztürk, and Tuncay Yiğit. 2022. Energy Efficiency Analysis of Code Refactoring Techniques for Green and Sustainable Software in Portable Devices. *Electronics* 11, 3 (2022).

[183] Cos S. Ierotheou, Haoqiang Jin, Gregory Matthews, Steve P. Johnson, Robert Hood, and P. F. Leggett. 2007. Using an interactive software environment for the parallelization of real-world scientific applications. *Int. J. Comput. Math.* 84, 2 (2007), 167–175.

[184] Cos S. Ierotheou, Steve P. Johnson, P. F. Leggett, Mark Cross, E. W. Evans, Haoqiang Jin, Michael A. Frumkin, and Jerry C. Yan. 2001. The semi-automatic parallelisation of scientific application codes using a computer aided parallelisation toolkit. *Sci. Program.* 9, 2-3 (2001), 163–173.

[185] Amil A. Ilham and Kazuaki J. Murakami. 2011. Evaluation and optimization of Java object ordering schemes. In *ICEEI 2011*. IEEE, 1–6.

[186] Venkata N. Inukollu, Divya D. Keshamoni, Taeghyun Kang, and Manikanta Inukollu. 2014. Factors Influencing Quality of Mobile Apps – Role of Mobile App Development Life Cycle. *Int. J. Softw. Eng. & Appl.* 5, 5 (2014), 15–34.

[187] Kazuaki Ishizaki, Akihiro Hayashi, Gita Koblents, and Vivek Sarkar. 2015. Compiling and Optimizing Java 8 Programs for GPU Execution. In *PACT 2015*. IEEE, 419–431.

[188] Adrian Jackson, Fiona Reid, Joachim Hein, Alejandro Soba, and Xavier Saez. 2011. High Performance I/O. In *PDP 2011*. IEEE, 349–356.

[189] Thomas Jakobs and Gudula Rünger. 2018. Examining Energy Efficiency of Vectorization Techniques Using a Gaussian Elimination. In *HPCS 2018*. IEEE, 268–275.

[190] Choonki Jang, Jaejin Lee, Bernhard Egger, and Soojung Ryu. 2012. Automatic code overlay generation and partially redundant code fetch elimination. *ACM Trans. Archit. Code Optim.* 9, 2 (2012), 10:1–10:32.

[191] Abhinav Jangda and Uday Bondhugula. 2018. An effective fusion and tile size model for optimizing image processing pipelines. In *PPoPP 2018*. ACM, 261–275.

[192] Vladimir Janjic, Christopher Brown, Adam D. Barwell, and Kevin Hammond. 2021. Refactoring for introducing and tuning parallelism for heterogeneous multicore machines in Erlang. *Concurr. Comput. Pract. Exp.* 33, 14 (2021).

[193] Cheol Jeon and Yookun Cho. 2012. A robust steganography-based software watermarking. In *QSIC 2012*. IEEE, 333–337.

[194] Changjiang Jia and W. K. Chan. 2013. A Study on the Efficiency Aspect of Data Race Detection – A Compiler Optimization Level Perspective. In *QSIC 2013*. IEEE, 35–44.

[195] He Jiang, Guojun Gao, Zhilei Ren, Xin Chen, and Zhide Zhou. 2022. SMARTEST: A Surrogate-Assisted Memetic Algorithm for Code Size Reduction. *IEEE Trans. Reliab.* 71, 1 (2022), 190–203.

[196] Yufei Jiang, Dinghao Wu, and Peng Liu. 2016. JRed: Program Customization and Bloatware Mitigation Based on Static Analysis. In *COMPSAC 2016*. IEEE, 12–21.

[197] Haoqiang Jin, Michael A. Frumkin, and Jerry C. Yan. 2000. Automatic Generation of OpenMP Directives and Its Application to Computational Fluid Dynamics Codes. In *ISHPC 2000 (LNCS, Vol. 1940)*. Springer, 440–456.

[198] Kengo Jingu, Kohta Shigenobu, Kanemitsu Ootsu, Takeshi Ohkawa, and Takashi Yokota. 2018. An Implementation of LLVM Pass for Loop Parallelization Based on IR-Level Directives. In *CANDAR 2018*. IEEE, 501–505.

[199] Pramod G. Joisha, Robert S. Schreiber, Prithviraj Banerjee, Hans-Juergen Boehm, and Dhruva R. Chakrabarti. 2011. A technique for the effective and automatic reuse of classical compiler optimizations on multithreaded code. In *POPL 2011*. ACM, 623–636.

[200] Prashant V. Joshi and K. S. Gurumurthy. 2014. Analysing and improving the performance of software code for Real Time Embedded systems. In *ICDCS 2014*. IEEE, 1–5.

[201] Maurício Breternitz Jr. and Roger Smith. 1997. Enhanced Compression Techniques to Simplify Program Decompression and Execution. In *ICCD 1997*. IEEE, 170–176.

[202] René Just, Darioush Jalali, and Michael D. Ernst. 2014. Defects4J: A database of existing faults to enable controlled testing studies for Java programs. In *ISSTA 2014*. ACM, 437–440.

[203] Ismail Kadayif and Mahmut T. Kandemir. 2004. Quasidynamic Layout Optimizations for Improving Data Locality. *IEEE Trans. Parallel Distributed Syst.* 15, 11 (2004), 996–1011.

[204] Lester Kalms, Tim Hebbeler, and Diana Göhringer. 2018. Automatic OpenCL Code Generation from LLVM-IR using Polyhedral Optimization. In *PARMA-DITAM@HiPEAC 2018*. ACM, 45–50.

[205] Melanie Kambadur and Martha A. Kim. 2016. NRG-loops: Adjusting power from within applications. In *CGO 2016*. ACM, 206–215.

[206] Yasusi Kanada, Keiji Kojima, and Masahiro Sugaya. 1988. Vectorization techniques for Prolog. In *ICS 1988*. ACM, 539–549.

[207] Mahmut T. Kandemir, Alok N. Choudhary, J. Ramanujam, and Meenakshi A. Kandaswamy. 2000. A Unified Framework for Optimizing Locality, Parallelism, and Communication in Out-of-Core Computations. *IEEE Trans. Parallel Distributed Syst.* 11, 7 (2000), 648–668.

[208] Mahmut T. Kandemir and Ismail Kadayif. 2001. Compiler-directed selection of dynamic memory layouts. In *CODES 2001*. ACM, 219–224.

[209] Mahmut T. Kandemir, J. Ramanujam, Mary Jane Irwin, Narayanan Vijaykrishnan, Ismail Kadayif, and Amisha Parikh. 2004. A compiler-based approach for dynamically managing scratch-pad memories in embedded systems. *IEEE Trans. Comput. Aided Des. Integr. Circuits Syst.* 23, 2 (2004), 243–260.

[210] Ian Karlin, Jim McGraw, Esthela Gallardo, Jeff Keasler, Edgar A. León, and Bert Still. 2012. Abstract: Memory and Parallelism Exploration Using the LULESH Proxy Application. In *SC 2012*. IEEE, 1427–1428.

[211] Ian Karlin, Jim McGraw, Esthela Gallardo, Jeff Keasler, Edgar A. León, and Bert Still. 2012. Poster: Memory and Parallelism Exploration Using the LULESH Proxy Application. In *SC 2012*. IEEE, 1429.

[212] Manolis Katsaragakis, Lazaros Papadopoulos, Mario Konijnenburg, Francky Catthoor, and Dimitrios Soudris. 2020. Memory Footprint Optimization Techniques for Machine Learning Applications in Embedded Systems. In *ISCAS 2020*. IEEE, 1–4.

[213] Satnam Kaur, Lalit K. Awasthi, and A. L. Sangal. 2021. A brief review on multi-objective software refactoring and a new method for its recommendation. *Arch. Computat. Methods Eng.* 28 (2021), 3087–3111.

[214] Engin Kayraklioglu, Erwan Favry, and Tarek A. El-Ghazawi. 2019. A Machine Learning Approach for Productive Data Locality Exploitation in Parallel Computing Systems. In *CCGrid 2019*. IEEE, 361–370.

[215] Engin Kayraklioglu, Erwan Favry, and Tarek A. El-Ghazawi. 2021. A Machine-Learning-Based Framework for Productive Locality Exploitation. *IEEE Trans. Parallel Distributed Syst.* 32, 6 (2021), 1409–1424.

[216] Vasilios Kelefouras and Karim Djemame. 2018. A methodology for efficient code optimizations and memory management. In *CF 2018*. 105–112.

[217] Vasilios Kelefouras and Karim Djemame. 2019. A methodology correlating code optimizations with data memory accesses, execution time and energy consumption. *J. Supercomput.* 75, 10 (2019), 6710–6745.

[218] Md. Muhib Khan and Weikuan Yu. 2021. ROBOTune: High-Dimensional Configuration Tuning for Cluster-Based Data Analytics. In *ICPP 2021*. ACM, 60:1–60:10.

[219] Raffi Khatchadourian, Yiming Tang, Mehdi Bagherzadeh, and Syed Ahmed. 2019. Safe automated refactoring for intelligent parallelization of Java 8 streams. In *ICSE 2019*. IEEE/ACM, 619–630.

[220] Doohwan Kim, Jang-Eui Hong, Ilchul Yoon, and Sang-Ho Lee. 2018. Code refactoring techniques for reducing energy consumption in embedded computing environment. *Clust. Comput.* 21, 1 (2018), 1079–1095.

[221] William Klieber, Ruben Martins, Ryan Steele, Matt Churilla, Mike McCall, and David Svoboda. 2021. Automated Code Repair to Ensure Spatial Memory Safety. In *APR@ICSE 2021*. IEEE, 23–30.

[222] Hakduran Koc, Mounika Garlapati, and Pranitha P. Madupu. 2020. Data Compression and Re-computation Based Performance Improvement in Multi-Core Architectures. In *CCWC 2020*. IEEE, 390–395.

[223] Zoltan A. Kocsis, John H. Drak, Douglas Carson, and Jerry Swan. 2016. Automatic Improvement of Apache Spark Queries using Semantics-preserving Program Reduction. In *GI@GECCO 2016 in GECCO 2016 companion*. ACM, 1141–1146.

[224] Hyungjoon Koo, Seyedhamed Ghavamnia, and Michalis Polychronakis. 2019. Configuration-Driven Software Debloating. In *European Workshop on Systems Security*. 9:1–9:6.

[225] Herb Krasner. 2022. *The Cost of Poor Quality Software in the US – A 2022 report*. Technical Report. Consortium for Information & Software Quality.

[226] Chidamber Kulkarni, Francky Catthoor, and Hugo De Man. 1998. Code Transformations for Low Power Caching in Embedded Multimedia Processors. In *IPPS/SPDP 1998*. IEEE, 292–297.

[227] Prasad A. Kulkarni, Wankang Zhao, Stephen Hines, David B. Whalley, Xin Yuan, Robert van Engelen, Kyle A. Gallivan, Jason Hiser, Jack W. Davidson, Baosheng Cai, Mark W. Bailey, Hwashin Moon, Kyunghwan Cho, and Yunheung Paek. 2006. VISTA: VPO interactive system for tuning applications. *ACM Trans. Embed. Comput. Syst.* 5, 4 (2006), 819–863.

[228] Rakesh Kumar, Alejandro Martínez, and Antonio Gonzalez. 2021. A Variable Vector Length SIMD Architecture for HW/SW Co-designed Processors. *CoRR* abs/2102.13410 (2021).

[229] T. S. Rajesh Kumar, R. Govindarajan, and C. P. Ravikumar. 2003. Optimal Code and Data Layout in Embedded Systems. In *VLSI Design 2003*. IEEE, 573–578.

[230] Farley Lai, Daniel Schmidt, and Octav Chipara. 2015. Static memory management for efficient mobile sensing applications. In *EMSOFT 2015*. IEEE, 187–196.

[231] Riyane Sid Lakhdar, Henri-Pierre Charles, and Maha Kooli. 2020. Data-layout optimization based on memory-access-pattern analysis for source-code performance improvement. In *SCOPES 2020*. ACM, 1–6.

[232] William B. Langdon. 2019. Genetic Improvement of Data gives double precision invsqrt. In *GI@GECCO 2019 in GECCO 2019 companion*. ACM, 1709–1714.

[233] William B. Langdon and David Clark. 2024. Genetic Improvement of Last Level Cache. In *EuroGP 2024 (LNCS, Vol. 14631)*. Springer, 209–226.

[234] William B. Langdon and Mark Harman. 2010. Evolving a CUDA kernel from an nVidia template. In *CEC 2010*. IEEE, 1–8.

[235] William B. Langdon and Mark Harman. 2015. Grow and Graft a better CUDA pknotsRG for RNA pseudoknot free energy calculation. In *GI@GECCO 2015 in GECCO 2015 companion*. ACM, 805–810.

[236] William B. Langdon and Mark Harman. 2015. Optimizing Existing Software With Genetic Programming. *IEEE Trans. Evol. Comput.* 19, 1 (2015), 118–135.

[237] William B. Langdon and Oliver Krauss. 2020. Evolving sqrt into 1/x via software data maintenance. In *GI@GECCO 2020 in GECCO 2020 companion*. ACM, 1928–1936.

[238] William B. Langdon, Brian Yee Hong Lam, Marc Modat, Justyna Petke, and Mark Harman. 2017. Genetic improvement of GPU software. *Genet. Program. Evolvable. Mach.* 18, 1 (2017), 5–44.

[239] William B. Langdon, Brian Yee Hong Lam, Justyna Petke, and Mark Harman. 2015. Improving CUDA DNA Analysis Software with Genetic Programming. In *GECCO 2015*. ACM, 1063–1070.

[240] William B. Langdon and Ronny Lorenz. 2017. Improving SSE Parallel Code with Grow and Graft Genetic Programming. In *GI@GECCO 2017 in GECCO 2017 companion*. ACM, 1537–1538.

[241] William B. Langdon, Marc Modat, Justyna Petke, and Mark Harman. 2014. Improving 3D medical image registration CUDA software with genetic programming. In *GECCO 2014*. ACM, 951–958.

[242] William B. Langdon and Justyna Petke. 2019. Genetic improvement of data gives binary logarithm from sqrt. In *GECCO 2019 companion*. ACM, 413–414.

[243] William B. Langdon, David R. White, Mark Harman, Yue Jia, and Justyna Petke. 2016. API-Constrained Genetic Improvement. In *SSBSE 2016 (LNCS, Vol. 9962)*. Springer, 224–230.

[244] James R. Larus and Paul N. Hilfinger. 1988. Restructuring Lisp Programs for Concurrent Execution. In *PPEALS 1988*. ACM, 100–110.

[245] Claire Le Goues, Stephanie Forrest, and Westley Weimer. 2013. Current challenges in automatic software repair. *Softw. Qual. J.* 21, 3 (2013), 421–443.

[246] Sheayun Lee, Jaejin Lee, Chang Yun Park, and Sang Lyul Min. 2004. A Flexible Tradeoff Between Code Size and WCET Using a Dual Instruction Set Processor. In *SCOPES 2004 (LNCS, Vol. 3199)*. ACM, 244–258.

[247] Seong-Won Lee, Soo-Mook Moon, Won-Ki Jung, Jin-Seok Oh, and Hyeong-Seok Oh. 2010. Code Size and Performance Optimization for Mobile JavaScript Just-in-Time Compiler. In *Interaction between Compilers and Computer Architecture*. ACM, 1–7.

[248] Sanghoon Lee and James Tuck. 2011. Automatic parallelization of fine-grained meta-functions on a chip multiprocessor. In *CGO 2011*. IEEE, 130–140.

[249] Charles Lefurgy, Peter L. Bird, I-Cheng K. Chen, and Trevor N. Mudge. 1997. Improving Code Density Using Compression Techniques. In *MICRO 1997*. ACM/IEEE, 194–203.

[250] Edgar A. León, Ian Karlin, Ryan E. Grant, and Matthew G. F. Dosanjh. 2016. Program optimizations: The interplay between power, performance, and energy. *Parallel Comput.* 58 (2016), 56–75.

[251] David Leopoldseder, Lukas Stadler, Thomas Würthinger, Josef Eisl, Doug Simon, and Hanspeter Mössenböck. 2018. Dominance-based duplication simulation (DBDS) – Code duplication to enable compiler optimizations. In *CGO 2018*. ACM, 126–137.

[252] Rainer Leupers and Peter Marwedel. 1999. Function inlining under code size constraints for embedded processors. In *ICCAD 1999*. IEEE, 253–256.

[253] Ding Li and William G. J. Halfond. 2015. Optimizing energy of HTTP requests in Android applications. In *DeMobile@SIGSOFT FSE 2015*. 25–28.

[254] Ding Li, Angelica Huyen Tran, and William G. J. Halfond. 2014. Making web applications more energy efficient for OLED smartphones. In *ICSE 2014*. ACM, 527–538.

[255] Xueliang Li and John P. Gallagher. 2016. A Source-Level Energy Optimization Framework for Mobile Applications. In *SCAM 2016*. IEEE, 31–40.

[256] Yuhao Li and Benjamin C. Lee. 2022. Phronesis: Efficient Performance Modeling for High-dimensional Configuration Tuning. *ACM Trans. Archit. Code Optim.* 19, 4 (2022), 56:1–56:26.

[257] Yuancheng Li and Jiaqi Shi. 2019. CRbS: A Code Reordering Based Speeding-up Method of Irregular Loops on CMP. In *ASAP 2019*. IEEE, 34.

[258] Soo Ling Lim, Peter J. Bentley, Natalie Kanakam, Fuyuki Ishikawa, and Shinichi Honiden. 2015. Investigating Country Differences in Mobile App User Behavior and Challenges for Software Engineering. *IEEE Trans. Softw. Eng.* 41, 1 (2015), 40–64.

[259] Chen Lin, Junqing Zhuang, Jiadong Feng, Hui Li, Xuanhe Zhou, and Guoliang Li. 2022. Adaptive Code Learning for Spark Configuration Tuning. In *ICDE 2022*. IEEE, 1995–2007.

[260] Zhen Lin, Mohammad A. Alshboul, Yan Solihin, and Huiyang Zhou. 2019. Exploring Memory Persistency Models for GPUs. *CoRR* abs/1904.12661 (2019).

[261] Michael D. Linderman, Matthew Ho, David L. Dill, Teresa H. Meng, and Garry P. Nolan. 2010. Towards program optimization through automated analysis of numerical precision. In *CGO 2010*. ACM, 230–237.

[262] Cullen Linn and Saumya K. Debray. 2003. Obfuscation of executable code to improve resistance to static disassembly. In *CCS 2003*. ACM, 290–299.

[263] Jhe-Yu Liou, Muaaz Awan, Steven Hofmeyr, Stephanie Forrest, and Carole-Jean Wu. 2022. Understanding the Power of Evolutionary Computation for GPU Code Optimization. In *IISWC 2022*. IEEE, 185–198.

[264] Jhe-Yu Liou, Stephanie Forrest, and Carole-Jean Wu. 2019. Genetic Improvement of GPU Code. In *GI@ICSE 2019*. ACM, 20–27.

[265] Jhe-Yu Liou, Xiaodong Wang, Stephanie Forrest, and Carole-Jean Wu. 2020. GEVO: GPU Code Optimization Using Evolutionary Computation. *ACM Trans. Archit. Code Optim.* 17, 4 (2020), 33:1–33:28.

[266] Lixia Liu and Zhiyuan Li. 2010. A compiler-automated array compression scheme for optimizing memory intensive programs. In *ICS 2010*. ACM, 285–294.

[267] Víctor R. López-López, Leonardo Trujillo, and Pierrick Legrand. 2018. Novelty Search for Software Improvement of a SLAM system. In *GI@GECCO 2018 in GECCO 2018 companion*. ACM, 1598–1605.

[268] Víctor R. López-López, Leonardo Trujillo, and Pierrick Legrand. 2019. Applying genetic improvement to a genetic programming library in C++. *Soft Comput.* 23, 22 (2019), 11593–11609.

[269] Víctor R. López-López, Leonardo Trujillo, Pierrick Legrand, and Gustavo Olague. 2016. Genetic Programming: From design to improved implementation. In *GI@GECCO 2016 in GECCO 2016 companion*. ACM, 1147–1154.

[270] Pingjing Lu, Yonggang Che, and Zhenghua Wang. 2009. A Framework for Effective Memory Optimization of High Performance Computing Applications. In *HPCC 2009*. IEEE, 95–102.

[271] Walter Lucas, Rodrigo Bonif'acio, Edna Dias Canedo, Diego Marcilio, and Fernanda Lima. 2019. Does the Introduction of Lambda Expressions Improve the Comprehension of Java Programs?. In *SBES 2019*. ACM, 187–196.

[272] Gang Luo, Bing Guo, Yan Shen, HaiYan Liao, and Lei Ren. 2009. Analysis and Optimization of Embedded Software Energy Consumption on the Source Code and Algorithm Level. In *EMC 2009*. IEEE, 1–5.

[273] Victor De La Luz, Ismail Kadayif, Mahmut T. Kandemir, and Ugur Sezer. 2004. Access Pattern Restructuring for Memory Energy. *IEEE Trans. Parallel Distributed Syst.* 15, 4 (2004), 289–303.

[274] Victor De La Luz and Mahmut T. Kandemir. 2004. Array Regrouping and Its Use in Compiling Data-Intensive Embedded Applications. *IEEE Trans. Computers* 53, 1 (2004), 1–19.

[275] Yingjun Lyu, Ding Li, and William G. J. Halfond. 2018. Remove RATs from your code – Automated optimization of resource inefficient database writes for mobile applications. In *ISSTA 2018*. ACM, 310–321.

[276] Wenjing Ma and Gagan Agrawal. 2010. An integer programming framework for optimizing shared memory use on GPUs. In *HiPC 2010*. IEEE, 1–10.

[277] Wenjing Ma, Kan Gao, and Guoping Long. 2016. Highly Optimized Code Generation for Stencil Codes with Computation Reuse for GPUs. *J. Comput. Sci. Technol.* 31, 6 (2016), 1262–1274.

[278] Konner Macias, Mihir Mathur, Bobby R. Bruce, Tianyi Zhang, and Miryung Kim. 2020. WebJShrink: A web service for debloating Java bytecode. In *ESEC/FSE 2020*. ACM, 1665–1669.

[279] Alberto Magni, Christophe Dubach, and Michael F. P. O'Boyle. 2014. Exploiting GPU Hardware Saturation for Fast Compiler Optimization. In *GPGPU@ASPLOS 2014*. ACM, 99–106.

[280] Amin Majd, Mohammad Loni, Golnaz Sahebi, Masoud Daneshtalab, and Elena Troubitsyna. 2019. A Cloud Based Super-Optimization Method to Parallelize the Sequential Code's Nested Loops. In *MCSoC 2019*. IEEE, 281–287.

[281] Gregory Malecha, Ashish Gehani, and Natarajan Shankar. 2015. Automated software winnowing. In *SAC 2015*. ACM, 1504–1511.

[282] Josip Maras, Jan Carlson, and Ivica Crnkovic. 2012. Extracting client-side web application code. In *WWW 2012*. ACM, 819–828.

[283] Francesco Marino, Giovanni Squillero, and Alberto Paolo Tonda. 2016. A General-Purpose Framework for Genetic Improvement. In *PPSN XIV (LNCS, Vol. 9921)*. Springer, 345–352.

[284] Margaret Martonosi, Anoop Gupta, and Thomas E. Anderson. 1992. MemSpy: Analyzing Memory System Bottlenecks in Programs. In *SIGMETRICS 1992*. ACM, 1–12.

[285] Raghunandan Mathur, Hiroshi Matsuoka, Osamu Watanabe, Akihiro Musa, Ryusuke Egawa, and Hiroaki Kobayashi. 2015. A Case Study of Memory Optimization for Migration of a Plasmonics Simulation Application to SX-ACE. In *CANDAR 2015*. IEEE, 521–527.

[286] Kazuaki Matsumura, Hamid Reza Zohouri, Mohamed Wahib, Toshio Endo, and Satoshi Matsuoka. 2020. AN5D: Automated stencil framework for high-degree temporal blocking on GPUs. In *CGO 2020*. ACM, 199–211.

[287] Kathryn S. McKinley. 1998. A Compiler Optimization Algorithm for Shared-Memory Multiprocessors. *IEEE Trans. Parallel Distributed Syst.* 9, 8 (1998), 769–787.

[288] Nicholas Freitag McPhee, Thomas Helmuth, and Lee Spector. 2017. Using algorithm configuration tools to optimize genetic programming parameters – A case study. In *GECCO 2017 companion*. ACM, 243–244.

[289] Michael Med and Andreas Krall. 2007. Instruction Set Encoding Optimization for Code Size Reduction. In *IC-SAMOS 2007*. IEEE, 9–17.

[290] Igor Z. Milosavljevic and Marwan A. Jabri. 1999. Automatic Array Alignment in Parallel Matlab Scripts. In *IPPS/SPDP 1999*. IEEE, 285–289.

[291] Nikolai Moesus, Matthias Scholze, Sebastian Schlesinger, and Paula Herber. 2018. Automated Selection of Software Refactorings that Improve Performance. In *ICSOFT 2018*. SciTePress, 67–78.

[292] Martin Monperrus. 2018. Automatic Software Repair: A Bibliography. *ACM Comput. Surveys* 51, 1 (2018), 17:1–17:24.

[293] Martin Monperrus. 2018. *The Living Review on Automated Program Repair.* Technical Report hal-01956501. HAL.

[294] Ana Moreton-Fernandez, Arturo González-Escribano, and Diego R. Llanos. 2014. Exploiting distributed and shared memory hierarchies with Hitmap. In *HPCS 2014*. IEEE, 278–286.

[295] Mohsen Mosayebi and Manbir Sodhi. 2020. Tuning genetic algorithm parameters using design of experiments. In *GI@GECCO 2020 in GECCO 2020 companion*. ACM, 1937–1944.

[296] Girish Mururu, Chris Porter, Prithayan Barua, and Santosh Pande. 2019. Binary Debloating for Security via Demand Driven Loading. *CoRR* abs/1902.06570 (2019).

[297] Raana Saheb Nasagh, Mahnoosh Shahidi, and Mehrdad Ashtiani. 2021. A fuzzy genetic automatic refactoring approach to improve software maintainability and flexibility. *Soft Comput.* 25, 6 (2021), 4295–4325.

[298] Brandon Neth, Thomas R. W. Scogland, Bronis R. de Supinski, and Michelle Mills Strout. 2021. Inter-loop optimization in RAJA using loop chains. In *ICS 2021*. ACM, 1–12.

[299] Y. Nezzari and C. P. Bridges. 2017. Compiler extensions towards reliable multicore processors. In *AERO 2017*. IEEE, 1–6.

[300] Duc-Man Nguyen, Thang Q. Huynh, and Thanh-Hung Nguyen. 2016. Improve the Performance of Mobile Applications Based on Code Optimization Techniques Using PMD and Android Lint. In *IUKM 2016 (LNCS, Vol. 9978)*. Springer, 343–356.

[301] Nima Nikzad, Marjan Radi, Octav Chipara, and William G. Griswold. 2015. Managing the Energy-Delay Tradeoff in Mobile Applications with Tempus. In *Middleware 2015*. ACM, 259–270.

[302] François Noël, Luke Hornof, Charles Consel, and Julia L. Lawall. 1998. Automatic, Template-Based Run-Time Specialization – Implementation and Experimental Study. In *ICCL 1998*. IEEE, 132–142.

[303] Wellington Oliveira, Renato Oliveira, Fernando Castor, Benito Fernandes, and Gustavo Pinto. 2019. Recommending energy-efficient Java collections. In *MSR 2019*. IEEE/ACM, 160–170.

[304] Wellington Oliveira, Renato Oliveira, Fernando Castor, Gustavo Pinto, and João Paulo Fernandes. 2021. Improving energy-efficiency by recommending Java collections. *Empir. Softw. Eng.* 26, 3 (2021), 55.

[305] Vijay S. Pai and Sarita V. Adve. 1999. Code Transformations to Improve Memory Parallelism. In *MICRO 1999*. ACM/IEEE, 147–155.

[306] Zhelong Pan and Rudolf Eigenmann. 2006. Fast and Effective Orchestration of Compiler Optimizations for Automatic Performance Tuning. In *CGO 2006*. IEEE, 319–332.

[307] Maksim Panchenko, Rafael Auler, Bill Nell, and Guilherme Ottoni. 2019. BOLT: A Practical Binary Optimizer for Data Centers and Beyond. In *CGO 2019*. ACM, 2–14.

[308] Lazaros Papadopoulos, Charalampos Marantos, Georgios Digkas, Apostolos Ampatzoglou, Alexander Chatzigeorgiou, and Dimitrios Soudris. 2018. Interrelations between Software Quality Metrics, Performance and Energy Consumption in Embedded Applications. In *SCOPES 2018*. ACM, 62–65.

[309] Jung Gyu Park and Myong-Soon Park. 2002. Using indexed data structures for program specialization. In *ASIA-PEPM 2002*. ACM, 61–69.

[310] Jae Jin Park, Jang-Eui Hong, and Sang-Ho Lee. 2014. Investigation for Software Power Consumption of Code Refactoring Techniques. In *SEKE 2014*. Knowledge Systems Institute Graduate School, 717–722.

[311] Rui Pereira, Marco Couto, João Saraiva, Jácome Cunha, and João Paulo Fernandes. 2016. The influence of the Java collection framework on overall energy consumption. In *GREENS@ICSE 2016*. ACM, 15–21.

[312] Rui Pereira, Pedro Simão, Jácome Cunha, and João Saraiva. 2018. jStanley: Placing a green thumb on Java collections. In *ASE 2018*. ACM, 856–859.

[313] Justyna Petke, Saemundur O. Haraldsson, Mark Harman, William B. Langdon, David R. White, and John R. Woodward. 2018. Genetic Improvement of Software: A Comprehensive Survey. *IEEE Trans. Evol. Comput.* 22, 3 (2018), 415–432.

[314] Justyna Petke, Mark Harman, William B. Langdon, and Westley Weimer. 2014. Using Genetic Improvement and Code Transplants to Specialise a C++ Program to a Problem Class. In *EuroGP 2014 (LNCS, Vol. 8599)*. Springer, 137–149.

[315] Justyna Petke, Mark Harman, William B. Langdon, and Westley Weimer. 2018. Specialising Software for Different Downstream Applications Using Genetic Improvement and Code Transplantation. *IEEE Trans. Softw. Eng.* 44, 6 (2018), 574–594.

[316] Justyna Petke, William B. Langdon, and Mark Harman. 2013. Applying Genetic Improvement to MiniSAT. In *SSBSE 2013 (LNCS, Vol. 8084)*. Springer, 257–262.

[317] Pedro Pinto, João Bispo, João M. P. Cardoso, Jorge G. Barbosa, Davide Gadioli, Gianluca Palermo, Jan Martinovic, Martin Golasowski, Katerina Slaninová, Radim Cmar, and Cristina Silvano. 2022. Pegasus: Performance Engineering for Software Applications Targeting HPC Systems. *IEEE Trans. Softw. Eng.* 48, 3 (2022), 732–754.

[318] Teerit Ploensin, Krerk Piromsopa, and Norraphat Srimanobhas. 2021. Code Transformation Impact on Compiler-based Optimization – A Case Study in the CMSSW. In *ICAPM 2021 (JoP:CS, Vol. 1936)*. IOP Publishing, 012023.

[319] Gary Plumbridge and Neil C. Audsley. 2012. Translating Java for resource constrained embedded systems. In *ReCoSoC 2012*. IEEE, 1–8.

[320] Boldizsár Poór, Melinda Tóth, and István Bozó. 2020. Transformations towards clean functional code. In *ERLANG 2020*. ACM, 24–30.

[321] Chris Porter, Girish Mururu, Prithayan Barua, and Santosh Pande. 2020. BlankIt library debloating: Getting what you want instead of cutting what you don't. In *PLDI 2020*. ACM, 164–180.

[322] Kleanthis Psarris and Konstantinos Kyriakopoulos. 2004. An Experimental Evaluation of Data Dependence Analysis Techniques. *IEEE Trans. Parallel Distributed Syst.* 15, 3 (2004), 196–213.

[323] Chenxiong Qian, Hong Hu, Mansour Alharthi, Simon Pak Ho Chung, Taesoo Kim, and Wenke Lee. 2019. RAZOR: A Framework for Post-deployment Software Debloating. In *USENIX Security 2019*. USENIX, 1733–1750.

[324] Chenxiong Qian, Hyungjoon Koo, ChangSeok Oh, Taesoo Kim, and Wenke Lee. 2020. Slimium: Debloating the Chromium Browser with Feature Subsetting. In *CCS 2020*. ACM, 461–476.

[325] Anh Quach and Aravind Prakash. 2019. Bloat Factors and Binary Specialization. In *FEAST@CCS 2019*. ACM, 31–38.

[326] Faizur Rahman, Jichi Guo, and Qing Yi. 2011. Automated empirical tuning of scientific codes for performance and power consumption. In *HiPEAC 2011*. ACM, 107–116.

[327] Shah Mohammad Faizur Rahman, Jichi Guo, Akshatha Bhat, Carlos D. Garcia, Majedul Haque Sujon, Qing Yi, Chunhua Liao, and Daniel J. Quinlan. 2012. Studying the impact of application-level optimizations on the power consumption of multi-core architectures. In *CF 2012*. 123–132.

[328] Sreeranga P. Rajan, Masahiro Fujita, Ashok Sudarsanam, and Sharad Malik. 1999. Development of an optimizing compiler for a Fujitsu fixed-point digital signal processor. In *CODES 1999*. ACM, 2–6.

[329] Alex Ramírez, Luiz André Barroso, Kourosh Gharachorloo, Robert S. Cohn, Josep Lluís Larriba-Pey, P. Geoffrey Lowney, and Mateo Valero. 2001. Code layout optimizations for transaction processing workloads. In *ISCA 2001*. ACM, 155–164.

[330] Ari Rasch, Richard Schulze, Michel Steuwer, and Sergei Gorlatch. 2021. Efficient Auto-Tuning of Parallel Programs with Interdependent Tuning Parameters via Auto-Tuning Framework (ATF). *ACM Trans. Archit. Code Optim.* 18, 1 (2021), 1:1–1:26.

[331] Vaibhav Rastogi, Drew Davidson, Lorenzo De Carli, Somesh Jha, and Patrick D. McDaniel. 2017. Cimplifier: Automatically Debloating Containers. In *ESEC/FSE 2017*. ACM, 476–486.

[332] Mahesh Ravishankar, Roshan Dathathri, Venmugil Elango, Louis-Noël Pouchet, J. Ramanujam, Atanas Rountev, and P. Sadayappan. 2015. Distributed memory code generation for mixed Irregular/Regular computations. In *PPoPP 2015*. ACM, 65–75.

[333] Tushar Rawat and Aviral Shrivastava. 2015. Enabling multi-threaded applications on hybrid shared memory manycore architectures. In *DATE 2015*. ACM, 742–747.

[334] Siddharth Rele, Vipin Jain, Santosh Pande, and J. Ramanujam. 2001. Compact and efficient code generation through program restructuringon limited memory embedded DSPs. *IEEE Trans. Comput. Aided Des. Integr. Circuits Syst.* 20, 4 (2001), 477–494.

[335] Manman Ren, Ji Young Park, Mike Houston, Alex Aiken, and William J. Dally. 2008. A tuning framework for software-managed memory hierarchies. In *PACT 2008*. ACM, 280–291.

[336] Xiaoguang Ren, Yuhua Tang, Guibin Wang, Tao Tang, and Xudong Fang. 2010. Optimization and Implementation of LBM Benchmark on Multithreaded GPU. In *DSDE 2010*. IEEE, 116–122.

[337] Gabriel Rivera and Chau-Wen Tseng. 2000. Tiling Optimizations for 3D Scientific Computations. In *SC 2000*. IEEE, 32.

[338] Soumyaroop Roy, Srinivas Katkoori, and Nagarajan Ranganathan. 2007. A Compiler Based Leakage Reduction Technique by Power-Gating Functional Units in Embedded Microprocessors. In *VLSI Design 2007*. IEEE, 215–220.

[339] Soumyaroop Roy, Nagarajan Ranganathan, and Srinivas Katkoori. 2009. Exploring Compiler Optimizations for Enhancing Power Gating. In *ISCAS 2009*. IEEE, 1004–1007.

[340] Shane Ryoo, Christopher I. Rodrigues, Sara S. Baghsorkhi, Sam S. Stone, David Blair Kirk, and Wen-mei W. Hwu. 2008. Optimization principles and application performance evaluation of a multithreaded GPU using CUDA. In *PPoPP 2008*. ACM, 73–82.

[341] Akash Sachan and Bibhas Ghoshal. 2021. Learning based compilation of embedded applications targeting minimal energy consumption. *J. Syst. Archit.* 116 (2021), 102116.

[342] Cagri Sahin, Lori L. Pollock, and James Clause. 2014. How do code refactorings affect energy usage?. In *ESEM 2014*. ACM, 36:1–36:10.

[343] Nissy Saju, Jaya Garg, Rajni Sehgal, and Renuka Nagpal. 2021. Green Mining for Android Based Applications Using Refactoring Approach. In *ICRITO (TSF) 2021*. 1–6.

[344] Ryuichi Sakamoto, Masaaki Kondo, Kohei Fujita, Tsuyoshi Ichimura, and Kengo Nakajima. 2020. The Effectiveness of Low-Precision Floating Arithmetic on Numerical Codes – A Case Study on Power Consumption. In *HPC Asia 2020*. ACM, 199–206.

[345] Thayalan Sandran, M. Nordin Zakaria, and Anindya Jyoti Pal. 2011. An Optimized Tuning of Genetic Algorithm Parameters in Compiler Flag Selection Based on Compilation and Execution Duration. In *SocPros 2011 (LNCS, Vol. 131)*. Springer, 599–610.

[346] Thayalan Sandran, M. Nordin Zakaria, and Anindya Jyoti Pal. 2012. Performance profile of some hybrid heuristic search techniques using compiler flag selection as a seed example. In *CEC 2012*. IEEE, 1–5.

[347] John Sanguinetti. 1984. Program Optimization for a Pipelined Machine – A Case Study. In *SIGMETRICS 1984*. ACM, 88–95.

[348] Santonu Sarkar and Sayantan Mitra. 2014. Execution profile driven speedup estimation for porting sequential code to GPU. In *COMPUTE 2014*. ACM, 21:1–21:6.

[349] Nadathur Satish, Changkyu Kim, Jatin Chhugani, Hideki Saito, Rakesh Krishnaiyer, Mikhail Smelyanskiy, Milind Girkar, and Pradeep Dubey. 2012. Can traditional programming bridge the Ninja performance gap for parallel computing applications?. In *ISCA 2012*. ACM, 440–451.

[350] Yukinori Sato, Shimpei Sato, and Toshio Endo. 2015. Exana: An execution-driven application analysis tool for assisting productive performance tuning. In *SEPS@SPLASH 2015*. ACM, 1–10.

[351] Daniel J. Scales, Kourosh Gharachorloo, and Chandramohan A. Thekkath. 1996. Shasta: A Low Overhead, Software-Only Approach for Supporting Fine-Grain Shared Memory. In *ASPLOS 1996*. ACM, 174–185.

[352] Eric M. Schulte, Jonathan Dorn, Stephen Harding, Stephanie Forrest, and Westley Weimer. 2014. Post-compiler software optimization for reducing energy. In *ASPLOS 2014*. ACM, 639–652.

[353] Robert Sedgewick. 1978. Implementing Quicksort Programs. *Commun. ACM* 21, 10 (1978), 847–857.

[354] Rajni Sehgal, Deepti Mehrotra, Renuka Nagpal, and Ramanuj Sharma. 2020. Green software: Refactoring approach. *J. K. S. Univ. Comput. Inf. Sci.* (2020).

[355] Marija Selakovic, Thomas Glaser, and Michael Pradel. 2017. An actionable performance profiler for optimizing the order of evaluations. In *ISSTA 2017*. ACM, 170–180.

[356] Sangmin Seo, Jun Lee, Gangwon Jo, and Jaejin Lee. 2013. Automatic OpenCL work-group size selection for multicore CPUs. In *PACT 2013*. IEEE, 387–397.

[357] Seok-Won Seong and Prabhat Mishra. 2006. A bitmask-based code compression technique for embedded systems. In *ICCAD 2006*. IEEE, 251–254.

[358] Hashim Sharif, Muhammad Abubakar, Ashish Gehani, and Fareed Zaffar. 2018. TRIMMER: Application specialization for code debloating. In *ASE 2018*. ACM, 329–339.

[359] Anil Sharma and C. P. Ravikumar. 2000. Efficient Implementation of ADPCM Codec. In *VLSI Design 2000*. IEEE, 456–461.

[360] Qingchuan Shi, Henry Hoffmann, and Omer Khan. 2015. A Cross-Layer Multicore Architecture to Tradeoff Program Accuracy and Resilience Overheads. *IEEE Comput. Archit. Lett.* 14, 2 (2015), 85–89.

[361] Yao Shi, Bernard Blackham, and Gernot Heiser. 2013. Code optimizations using formally verified properties. In *OOPSLA 2013*. ACM, 427–442.

[362] Wen-Li Shih, Cheng-Yen Lin, Ming-Yu Hung, and Jenq Kuen Lee. 2016. A Probabilistic Framework for Compiler Optimization with Multithread Power-Gating Controls. In *ICPP Workshops 2016*. IEEE, 281–288.

[363] Alexander Shypula, Aman Madaan, Yimeng Zeng, Uri Alon, Jacob R. Gardner, Yiming Yang, Milad Hashemi, Graham Neubig, Parthasarathy Ranganathan, Osbert Bastani, and Amir Yazdanbakhsh. 2024. Learning Performance-Improving Code Edits. In *ICLR 2024*. OpenReview.net.

[364] Tajana Simunic, Giovanni De Micheli, Luca Benini, and Mat Hans. 2000. Source Code Optimization and Profiling of Energy Consumption in Embedded Systems. In *ISSS 2000*. ACM/IEEE, 193–199.

[365] Gagan Somashekar, Amoghavarsha Suresh, Saurabh Tyagi, Vikas Dhyani, Krishna Chaitanya Donkada, Anurag Pradhan, and Anshul Gandhi. 2022. Reducing the Tail Latency of Microservices Applications via Optimal Configuration Tuning. In *ACSOS 2022*. IEEE, 111–120.

[366] Seung Woo Son, Guangyu Chen, Ozcan Ozturk, Mahmut T. Kandemir, and Alok N. Choudhary. 2007. Compiler-Directed Energy Optimization for Parallel Disk Based Systems. *IEEE Trans. Parallel Distributed Syst.* 18, 9 (2007), 1241–1257.

[367] Xiaohu Song, Ying Wang, Xiao Cheng, Guangtai Liang, Qianxiang Wang, and Zhiliang Zhu. 2024. Efficiently Trimming the Fat – Streamlining Software Dependencies with Java Reflection and Dependency Analysis. In *ICSE 2024*. ACM, 103:1–103:12.

[368] César Soto-Valero, Nicolas Harrand, Martin Monperrus, and Benoit Baudry. 2021. A comprehensive study of bloated dependencies in the Maven ecosystem. *Empir. Softw. Eng.* 26, 3 (2021), 45.

[369] Johannes Spazier, Steffen Christgau, and Bettina Schnor. 2016. Automatic generation of parallel C code for stencil applications written in MATLAB. In *ARRAY@PLDI 2016*. 47–54.

[370] Ekaterina Stefanov and Anthony M. Sloane. 2004. Simple, Effective Code-Size Reduction for Functional Programs. In *IFL 2004 (LNCS)*. Springer, 211–225.

[371] Panagiotis Stratis and Ajitha Rajan. 2016. Test case permutation to improve execution time. In *ASE 2016*. ACM, 45–50.

[372] Manuel Strobel and Martin Radetzki. 2019. A Backend Tool for the Integration of Memory Optimizations into Embedded Software. In *FDL 2019*. IEEE, 1–7.

[373] Pengfei Su, Shasha Wen, Hailong Yang, Milind Chabbi, and Xu Liu. 2019. Redundant loads: A software inefficiency indicator. In *ICSE 2019*. IEEE/ACM, 982–993.

[374] Bjorn De Sutter, Bruno De Bus, Koen De Bosschere, and Saumya K. Debray. 2001. Combining Global Code and Data Compaction. In *OM@PLDI 2001*. ACM, 29–38.

[375] Indira Syawanodya, Dian Anggraini, Fajar Muhammad Al-Hijri, and Mochamad Iqbal Ardimansyah. 2024. An Empirical Evaluation on the Effect of Refactoring Code Smells Mobile Applications Android with ASATs on Resource Usage. *Int. J. Adv. Sci. Eng. Inf. Technol.* 14, 1 (2024), 214–223.

[376] Giuseppe Tagliavini, Andrea Marongiu, and Luca Benini. 2020. FlexFloat: A Software Library for Transprecision Computing. *IEEE Trans. Comput. Aided Des. Integr. Circuits Syst.* 39, 1 (2020), 145–156.

[377] Tat Kee Tan, Anand Raghunathan, and Niraj K. Jha. 2003. Software Architectural Transformations: A New Approach to Low Energy Embedded Software. In *DATE 2003*. IEEE, 11046–11051.

[378] Binxian Tao and Ju Qian. 2014. Refactoring Java Concurrent Programs Based on Synchronization Requirement Analysis. In *ICSME 2014*. IEEE, 361–370.

[379] Huda Tariq, Maliha Arshad, and Wafa Basit. 2020. Effects of Refactoring upon Efficiency of an NP-Hard Task Assignment Problem – A case study. In *ICACS 2020*. IEEE, 1–9.

[380] Sendhilraj Thangaraj, Sudhakar Gummadi, and Shanmugasundaram Radhakrishnan. 2006. Enhancement in ARM Code Optimization for Memory Constrained Embedded Systems. In *ADCOM 2006*. 483–486.

[381] Eli Tilevich and Yannis Smaragdakis. 2005. Binary refactoring: Improving code behind the scenes. In *ICSE 2005*. ACM, 264–273.

[382] Spyridon Triantafyllis, Manish Vachharajani, Neil Vachharajani, and David I. August. 2003. Compiler Optimization-Space Exploration. In *CGO 2003*. IEEE, 204–215.

[383] Dan Tsafrir, Robert W. Wisniewski, David F. Bacon, and Bjarne Stroustrup. 2009. Minimizing dependencies within generic classes for faster and smaller programs. In *OOPSLA 2009*. ACM, 425–444.

[384] Yash Ukidave and David R. Kaeli. 2013. Analyzing Optimization Techniques for Power Efficiency on Heterogeneous Platforms. In *AsHES@IPDPS 2013 in IPDPS 2013 Workshops*. IEEE, 1040–1049.

[385] Dan Umeda, Takahiro Suzuki, Hiroki Mikami, Keiji Kimura, and Hironori Kasahara. 2015. Multigrain Parallelization for Model-Based Design Applications Using the OSCAR Compiler. In *LCPC 2015 (LNCS, Vol. 9519)*. Springer, 125–139.

[386] Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie Hendren, Patrick Lam, and Vijay Sundaresan. 2010. Soot: A Java Bytecode Optimization Framework. In *CASCON 2010*. IBM, 214–224.

[387] Sébastien Varrette, Frédéric Pinel, Emmanuel Kieffer, Grégoire Danoy, and Pascal Bouvry. 2019. Automatic Software Tuning of Parallel Programs for Energy-Aware Executions. In *PPAM 2019 (LNCS, Vol. 12044)*. Springer, 144–155.

[388] Hernán Ceferino Vázquez, Alexandre Bergel, Santiago A. Vidal, Jorge Andrés Díaz Pace, and Claudia A. Marcos. 2019. Slimming JavaScript applications: An approach for removing unused functions from JavaScript libraries. *Inf. Softw. Technol.* 107 (2019), 18–29.

[389] Hans-Nikolai Vießmann, Artjoms Sinkarovs, and Sven-Bodo Scholz. 2018. Extended Memory Reuse: An Optimisation for Reducing Memory Allocations. In *IFL 2018*. ACM, 107–118.

[390] Tobias J. K. Edler von Koch, Björn Franke, Pranav Bhandarkar, and Anshuman Dasgupta. 2014. Exploiting function similarity for code size reduction. In *LCTES 2014*. ACM, 85–94.

[391] Michael Voss and Rudolf Eigenmann. 2000. ADAPT: Automated De-Coupled Adaptive Program Transformation. In *ICPP 2000*. IEEE, 163–172.

[392] Cheng Wang, Wei-Yu Chen, Youfeng Wu, Bratin Saha, and Ali-Reza Adl-Tabatabai. 2007. Code Generation and Optimization for Transactional Memory Constructs in an Unmanaged Language. In *CGO 2007*. IEEE, 34–48.

[393] Qinglong Wang, Runzhe Wang, Yuxi Hu, Xiaohai Shi, Zheng Liu, Tao Ma, Houbing Song, and Heyuan Shi. 2023. KeenTune: Automated Tuning Tool for Cloud Application Performance Testing and Optimization. In *ISSTA 2023*. ACM, 1487–1490.

[394] Runzhe Wang, Qinglong Wang, Yuxi Hu, Heyuan Shi, Yuheng Shen, Yu Zhan, Ying Fu, Zheng Liu, Xiaohai Shi, and Yu Jiang. 2022. Industry practice of configuration auto-tuning for cloud applications and services. In *ESEC/FSE 2022*. ACM, 1555–1565.

[395] Shuai Wang, Dingyi Fang, Zheng Wang, Guixin Ye, Meng Li, Lu Yuan, Zhanyong Tang, Huanting Wang, Wei Wang, Fuwei Wang, and Jie Ren. 2019. Leveraging WebAssembly for Numerical JavaScript Code Virtualization. *IEEE Access* 7 (2019), 182711–182724.

[396] Yutong Wang and Cindy Rubio-González. 2024. Predicting Performance and Accuracy of Mixed-Precision Programs for Precision Tuning. In *ICSE 2024*. ACM, 15:1–15:13.

[397] Zhenjiang Wang, Chenggang Wu, and Pen-Chung Yew. 2010. On improving heap memory layout by dynamic pool allocation. In *CGO 2010*. ACM, 92–100.

[398] Matthias Weidmann. 1997. Design and Performance Improvement of a Real-World, Object-Oriented C++ Solver with STL. In *ISCOPE 1997 (LNCS, Vol. 1343)*. Springer, 25–32.

[399] Benjamin Welton and Barton P. Miller. 2018. Exposing Hidden Performance Opportunities in High Performance GPU Applications. In *CCGrid 2018*. IEEE, 301–310.

[400] Benjamin Welton and Barton P. Miller. 2020. Identifying and (automatically) remedying performance problems in CPU/GPU applications. In *ICS 2020*. ACM, 27:1–27:13.

[401] Jian Weng, Animesh Jain, Jie Wang, Leyuan Wang, Yida Wang, and Tony Nowatzki. 2021. UNIT: Unifying Tensorized Instruction Compilation. In *CGO 2021*. 77–89.

[402] Michael Werner, Lorenzo Servadei, Robert Wille, and Wolfgang Ecker. 2020. Automatic compiler optimization on embedded software through k-means clustering. In *MLCAD 2020*. ACM, 157–162.

[403] R. Clinton Whaley, Antoine Petitet, and Jack J. Dongarra. 2001. Automated empirical optimizations of software and the ATLAS project. *Parallel Comput.* 27, 1-2 (2001), 3–35.

[404] David R. White, Andrea Arcuri, and John A. Clark. 2011. Evolutionary Improvement of Programs. *IEEE Trans. Evol. Comput.* 15, 4 (2011), 515–538.

[405] Chris Wilcox, Michelle Mills Strout, and James M. Bieman. 2014. An optimization-based approach to lookup table program transformations. *J. Softw. Evol. Process.* 26, 6 (2014), 533–551.

[406] David Williams-King and Junfeng Yang. 2019. CodeMason: Binary-Level Profile-Guided Optimization. In *FEAST@CCS 2019*. ACM, 47–53.

[407] Felix Winterstein, Samuel Bayliss, and George A. Constantinides. 2014. Separation Logic-Assisted Code Transformations for Efficient High-Level Synthesis. In *FCCM 2014*. IEEE, 1–8.

[408] Yudong Wu, Mingyao Shen, Yi-Hui Chen, and Yuanyuan Zhou. 2020. Tuning applications for efficient GPU offloading to in-memory processing. In *ICS 2020*. ACM, 37:1–37:12.

[409] Bin Xiao, Zili Shao, Chantana Chantrapornchai, Edwin Hsing-Mean Sha, and Qingfeng Zhuge. 2002. Optimal Code Size Reduction for Software-Pipelined and Unfolded Loops. In *ISSS 2002*. ACM/IEEE, 144–149.

[410] Qinge Xie, Qingyuan Gong, Xinlei He, Yang Chen, Xin Wang, Haitao Zheng, and Ben Y. Zhao. 2022. Trimming Mobile Applications for Bandwidth-Challenged Networks in Developing Regions. *IEEE Trans. Mob. Comput.* (2022).

[411] Jinhan Xin, Kai Hwang, and Zhibin Yu. 2022. LOCAT: Low-Overhead Online Configuration Auto-Tuning of Spark SQL Applications. In *SIGMOD 2022*. ACM, 674–684.

[412] Qi Xin, Myeongsoo Kim, Qirun Zhang, and Alessandro Orso. 2020. Program debloating via stochastic optimization. In *ICSE (NIER) 2020*. ACM, 65–68.

[413] Qi Xin, Myeongsoo Kim, Qirun Zhang, and Alessandro Orso. 2020. Subdomain-Based Generality-Aware Debloating. In *ASE 2020*. IEEE, 224–236.

[414] Guoqing Xu, Nick Mitchell, Matthew Arnold, Atanas Rountev, Edith Schonberg, and Gary Sevitsky. 2014. Scalable Runtime Bloat Detection Using Abstract Dynamic Slicing. *ACM Trans. Softw. Eng. Methodol.* 23, 3 (2014), 23:1–23:50.

[415] Jinchen Xu, Guanghui Song, Bei Zhou, Fei Li, Jiangwei Hao, and Jie Zhao. 2024. A Holistic Approach to Automatic Mixed-Precision Code Generation and Tuning for Affine Programs. In *PPoPP 2024*. ACM, 55–67.

[416] Qiang Xu, James C. Davis, Y. Charlie Hu, and Abhilash Jindal. 2022. An Empirical Study on the Impact of Deep Parameters on Mobile App Energy Usage. In *SANER 2022*. IEEE, 844–855.

[417] Hua Yan, Yulei Sui, Shiping Chen, and Jingling Xue. 2016. Automated memory leak fixing on value-flow slices for C programs. In *SAC 2016*. ACM, 1386–1393.

[418] Fan Yang, Huanguo Zhang, Fei Yan, and Jian Yang. 2012. Testing Method of Code Redundancy Simplification Based on Program Dependency Graph. In *TrustCom 2012*. IEEE, 1895–1900.

[419] Yangzhao Yang, Naijie Gu, Kaixin Ren, and Bingqing Hu. 2014. An Approach to Enhance Loop Performance for Multicluster VLIW DSP Processor. In *ARCS 2014*. Springer, 1–8.

[420] Yi Yang, Ping Xiang, Jingfei Kong, and Huiyang Zhou. 2010. A GPGPU compiler for memory optimization and parallelism management. In *PLDI 2010*. ACM, 86–97.

[421] Kwaku Yeboah-Antwi and Benoit Baudry. 2015. Embedding Adaptivity in Software Systems using the ECSELR framework. In *GI@GECCO 2015 in GECCO 2015 companion*. ACM, 839–844.

[422] Kwaku Yeboah-Antwi and Benoit Baudry. 2017. Online Genetic Improvement on the Java virtual machine with ECSELR. *Genet. Program. Evolvable. Mach.* 18, 1 (2017), 83–109.

[423] Taylan Yemliha, Guangyu Chen, Ozcan Ozturk, Mahmut T. Kandemir, and Vijay Degalahal. 2007. Compiler-Directed Code Restructuring for Operating with Compressed Arrays. In *VLSI Design 2007*. IEEE, 221–226.

[424] Kamen Yotov, Xiaoming Li, Gang Ren, Michael Cibulskis, Gerald DeJong, María Jesús Garzarán, David A. Padua, Keshav Pingali, Paul Stodghill, and Peng Wu. 2003. A comparison of empirical and model-driven optimization. In *PLDI 2003*. ACM, 63–76.

[425] Tomofumi Yuki, Lakshminarayanan Renganarayanan, Sanjay V. Rajopadhye, Charles Anderson, Alexandre E. Eichenberger, and Kevin O'Brien. 2010. Automatic creation of tile size selection models. In *CGO 2010*. ACM, 190–199.

[426] Antonia Zhai, Christopher B. Colohan, J. Gregory Steffan, and Todd C. Mowry. 2004. Compiler Optimization of Memory-Resident Value Communication Between Speculative Threads. In *CGO 2004*. IEEE, 39–52.

[427] Yueyang Zhan, Rui Xi, Jianming Liao, Shuhuan Fan, and Mengshu Hou. 2024. KnobTune: A Dynamic Database Configuration Tuning Strategy Leveraging Historical Workload Similarities. In *CMLDS 2024*. ACM, 9:1–9:8.

[428] Xinyi Zhang, Zhuo Chang, Yang Li, Hong Wu, Jian Tan, Feifei Li, and Bin Cui. 2022. Facilitating Database Tuning with Hyper-Parameter Optimization – A Comprehensive Experimental Evaluation. *Proc. VLDB Endow.* 15, 9 (2022), 1808–1821.

[429] Hucheng Zhou, Wenguang Chen, and Fred C. Chow. 2011. An SSA-based algorithm for optimal speculative code motion under an execution profile. In *PLDI 2011*. ACM, 98–108.

[430] Keren Zhou, Xiaozhu Meng, Ryuichi Sai, Dejan Grubisic, and John M. Mellor-Crummey. 2022. An Automated Tool for Analysis and Tuning of GPU-Accelerated Code in HPC Applications. *IEEE Trans. Parallel Distributed Syst.* 33, 4 (2022), 854–865.

[431] Mingzhou Zhou, Xipeng Shen, Yaoqing Gao, and Graham Yiu. 2014. Space-efficient multi-versioning for input-adaptive feedback-driven program optimizations. In *OOPSLA 2014*. ACM, 763–776.

[432] Jiajing Zhu, Jay P. Hoeflinger, and David A. Padua. 2001. A synthesis of memory mechanisms for distributed architectures. In *ICS 2001*. ACM, 13–22.

[433] Qingfeng Zhuge, Zili Shao, and Edwin Hsing-Mean Sha. 2002. Optimal Code Size Reduction for Software-Pipelined Loops on DSP Applications. In *ICPP 2002*. IEEE, 613–620.

[434] Qingfeng Zhuge, Bin Xiao, and Edwin Hsing-Mean Sha. 2003. Code size reduction technique and implementation for software-pipelined DSP applications. *ACM Trans. Embed. Comput. Syst.* 2, 4 (2003), 590–613.

[435] Zan Zong, Lijie Wen, Xuming Hu, Rui Han, Chen Qian, and Li Lin. 2022. MespaConfig: Memory-Sparing Configuration Auto-Tuning for Co-Located In-Memory Cluster Computing Jobs. *IEEE Trans. Serv. Comput.* 15, 5 (2022), 2883–2896.

[436] Yun Zou and Sanjay V. Rajopadhye. 2018. A Code Generator for Energy-Efficient Wavefront Parallelization of Uniform Dependence Computations. *IEEE Trans. Parallel Distributed Syst.* 29, 9 (2018), 1923–1936.