Deep Parameter Tuning of Concurrent Divide and Conquer Algorithms in Akka

David R. White[1], Leonid Joffe[1], Edward Bowles[2], and Jerry Swan[2]

[1] CREST, University College London, Malet Place, London, WC1E 6BT, UK
{leonid.joffe.14,david.r.white}@ucl.ac.uk,
[2] Computer Science, University of York, Deramore Lane, York, YO10 5GH, UK
{eab530,jerry.swan}@york.ac.uk

Abstract. Akka is a widely-used high-performance and distributed computing toolkit for fine-grained concurrency, written in Scala for the Java Virtual Machine. Although Akka elegantly simplifies the process of building complex parallel software, many crucial decisions that affect system performance are deferred to the user. Employing the method of Deep Parameter Tuning to extract embedded 'magic numbers' from source code, we use the CMA-ES evolutionary computation algorithm to optimise the concurrent implementation of three widely-used divide-and-conquer algorithms within the Akka toolkit: Quicksort, Strassen's matrix multiplication, and the Fast Fourier Transform.

Keywords: Genetic Improvement, Concurrency, Scala, JVM, Akka, Deep Parameter Tuning, Divide and Conquer, FFT, Matrix Multiplication, Quicksort.

1 Introduction

Support for concurrency is an important requirement when developing software for modern multicore systems, but the cognitive and development-time overheads of creating and manually-configuring concurrent and parallel systems are high [1]. In mediation of this difficulty, frameworks offering powerful concurrency support are now routinely used. Akka is the *de facto* standard for concurrency in Java and Scala; the strengths of Akka's concurrency model include:

- Immutability: the absence of mutable data eliminates many race conditions.
- Lightweight: many hundreds of processes can share a thread.
- Fault Tolerance: via the 'let it crash' philosophy popularised by Erlang [2].

In this paper, we define a generic template for divide and conquer algorithms in Akka and use it to express concurrent versions of three well-known and ubiquitous algorithms: the Fast Fourier Transform (FFT) [3], quicksort [4], and Strassen's matrix multiplication [5] . We then apply the method of Deep Parameter Tuning (DPT) [6] to optimise these algorithms for execution time.

2 Algorithms

Here we provide brief descriptions of the algorithms under optimisation, all of which are widely used and applicable to a large range of different problem areas.

2.1 Fast Fourier Transform

The FFT is an algorithm for computing the Discrete Fourier Transform (DFT) or its inverse of a sequence of complex numbers. The FFT is ubiquitous in signal and image processing and analysis, in order to refocus images, remove pattern noise, recover unclear images, pattern recurrence etc [7]. The naïve approach uses a series of multiplications and additions of sinusoidal waves, resulting in $\mathcal{O}(n^2)$ complexity. In contrast, the FFT achieves $O(n \log n)$ complexity by decomposing the DFT into even and odd components, calculating their transforms and then fusing them back together. This asymptotic complexity is also beneficial in polynomial arithmetic where it is often preferred over the Karatsuba [8] or naïve) algorithms of $O(n^{\log 3})$ and $O(n^2)$ complexities respectively. There are different variants of the FFT, but all utilise the same properties of the DFT — periodicity and complex conjugate symmetry. The version implemented here is due to Cooley and Tukey [9].

2.2 Quicksort

Quicksort is a divide and conquer sorting algorithm [4], and is popular due to its average case time complexity of $\mathcal{O}(n \log(n))$. A heuristically-selected element is chosen as a *pivot point*, and the sequence is partitioned such that all of the elements in one subsequence are 'less than' the value of the element in the pivot position, and all the elements in the other subsequence are 'greater than' it. The sorting process is then recursively applied to the subsequences. The sequential performance of the algorithm is in practice heavily-dependent on the choice of pivot point, but due to its recursive nature, it is naturally suited to parallelisation. Previous work optimising quicksort for energy consumption used Genetic Programming to obtain an improved pivot function [10].

2.3 Strassen's Matrix Multiplication

Strassen's algorithm [5] is a divide and conquer approach that reduces the complexity of matrix multiplication from $\mathcal{O}(n^3)$ to $\mathcal{O}(n^{2.8074})$. Asymptotically faster algorithms exist, but are rarely used since their high constant factor makes them impractical. The gain in Strassen's algorithm is achieved by reducing the number of recursive calls. Although this comes at a greater storage cost, the trade-off is often preferable.

3 Implementation

We now describe the implementation details of our chosen algorithms within the Akka toolkit, give a little background on the Akka Dispatcher, and describe the optimisation framework that performs Deep Parameter Tuning (DPT).

3.1 Akka Message Dispatcher

Our algorithm implementations rely upon concurrency support from Akka. One of the essential building blocks of Akka concurrency is the `Future`, a widely-used notion in functional programming that acts as a kind of 'container'[3] for holding the eventual result of some concurrent operation. For example, an object of type `Future[Double]` will eventually yield a `Double` value. Our implementations use `Futures` to queue units of work as the problem is recursively subdivided.

Whilst the implementations themselves determine the division of the problem into subproblems represented as `Futures`, the details of how their concurrent execution is managed are deferred to Akka. The specific choice of concurrency policy to be used by Akka is encapsulated by an `ExecutionContext`. The `ExecutionContext` dispatcher manages the dispatch of threads used to execute `Futures`. We use the default dispatcher, known as the "fork-join-executor", which gives "excellent performance in most cases."[4] The fork-join executor has two integer parameters that are discovered by our Deep Parameter Tuning mechanism and exposed to the optimisation process:

1. *Parallelism Factor* — the number of threads to use relative to the number of physical cores on the machine.
2. *Throughput* — the fairness of resource sharing between threads.

3.2 Benchmark Implementation

Listing 1 shows our implementation of a `DivideAndConquer` template (c.f. [11]), which defines `concurrent`, an algorithm template that invokes the abstract methods `shouldDivide`, `sequential`, `divide`, and `merge`. These methods are subsequently defined in subclasses corresponding to each of our examples: `FFT`, `Quicksort`, and `Strassen`. The implementation of `concurrent` uses the Akka toolkit to represent an 'inversion of control' of the well-known recursion pattern of divide and conquer, with the divided arguments evaluated concurrently via a `Future`.

The results obtained via the completed `Futures` are then merged according to the subclass method implementation. Listing 2 gives the corresponding subclass implementation for `Quicksort`: a hard-coded `Threshold` parameter determines the point below which the sequential algorithm should be used — the implementations of `Strassen` and `FFT` also make equivalent use of a `Threshold`

[3] Strictly, a monad.
[4] http://doc.akka.io/docs/akka/current/scala/dispatchers.html

```scala
trait DivideAndConquer[Args,Result] {

  // Implemented by subclasses:

  def shouldDivide(args: Args): Boolean
  def sequential(args: Args): Result
  def divide(args: Args): Seq[Args]
  def merge(results: Seq[Future[Result]])
      (implicit ec: ExecutionContext): Future[Result]

  //////////////////////

  final def concurrent(args: Args): Future[Result] = {
    if( !shouldDivide(args) )
      Future.successful(sequential(args))
    else {
      val futures = divide(args).map { Future( concurrent(_) ) }
      Future.sequence(futures).flatMap { merge(_) }
    }
  }
}
```

Listing 1: Generic concurrent divide and conquer for Akka

parameter. The implementations of `divide` and `merge` can be seen to have a direct correspondence with the implementation of `sequential`. Listing 3 gives the unit test for `Quicksort`, which asserts that both the sequential and concurrent implementations correctly sort randomly-generated test data.

Our implementation of Strassen's algorithm utilises an additional tunable `Leaf` parameter, which determines whether the matrices should be recursively split further (down to a size of 1x1), or naïvely multiplied. As with the `Threshold` parameter, this decision is independent from Akka.

3.3 DPT Implementation

Deep Parameter Tuning [6] is a heuristic optimisation method that parses source code in order to identify performance-critical parameters that are not exposed via any external interface. The goal is to find 'magic numbers' or other variables that do not modify the semantics of the program, but are critical factors in determining non-functional properties. Here we implement such an approach in order to optimise the execution time of our algorithm implementations, by tuning parameters specific to the algorithms themselves along with those used by the Akka dispatcher to manage concurrent execution.

We implement a Deep Parameter Tuner (DPT) in Scala. Its top-level operation is as follows:

1. Parse the Scala source code of the application to be optimised: FFT.scala, Quicksort.scala, and Strassen.scala. Construct an abstract syntax tree.
2. Extract all embedded 'magic numbers', in this case, restricted to integer literals, by operating on the abstract syntax tree [12].
3. Perform a heuristic search over a parameter vector obtained from the extracted literals.

It should be emphasised that the DPT tool is agnostic about the nature of the program it is optimising.

The search mechanism used for parameter optimisation is CMA-ES [13], a well-known evolutionary search mechanism that guides the search process via an adaptive approximation to the second derivative of the fitness function. The fitness function supplied to CMA-ES performs wall-clock timing of a modified version of the original source code, in which the magic numbers in the source are replaced by the corresponding values of a candidate solution, with vector elements rounded to the nearest integer. The modified source code is then compiled by the Scala compiler and executed via the appropriate test harness (e.g. as per Listing 3 for Quicksort), which helps ensure correctness of the modified code. The evaluation of the fitness function is repeated 10 times, and the median value taken, in order to reduce the impact of nondeterminism on the optimisation process.

To summarise, for a candidate solution vector $\bar{v}$ consisting of the proposed literals, the associated fitness function $f(\bar{v})$ to be minimised is given by:

$$f(\bar{v}) = \begin{cases} \infty, \text{if the test case fails} \\ \text{otherwise, the median time in seconds to run the test case} \end{cases}$$

```
class Quicksort
extends DivideAndConquer[List[Int],List[Int]] {

  val Threshold = 100
  val Throughput = 10
  val ParallelismFactor = 3

  val threadDispatcher =
      configureAkka(Throughput,ParallelismFactor)

  ////////////////////

  override def shouldDivide(data: List[Int]): Boolean =
      data.length > Threshold

  // well-known recursive implementation:
  override def sequential(data: List[Int]): List[Int] = {
    if( data.isEmpty ) {
      data
    } else {
      val pivot = data.head
      val (left, right) = data.tail partition (_ < pivot)
      sequential(left) ++ (pivot :: sequential(right))
    }
  }

  override def divide(data: List[Int]): Seq[List[Int]] = {
      val pivot = data.head
      val (left,right) = data.tail partition(_ < pivot)
      Seq( left, List(pivot), right )
  }

  override def merge(data: Seq[Future[List[Int]]]):
      Future[List[Int]] = {
    Future.sequence( data ).map { l =>
      l.head ++ l.tail.head ++ l.tail.tail.head
    }
  }
}
```

Listing 2: Quicksort via concurrent Divide and Conquer framework

```
class TestQuicksort {
  @Test
  def test: Unit = {

    implicit val executionContext: ExecutionContext =
        ActorSystem().dispatcher

    val ArraySize = 1600000
    val testData = List.fill(ArraySize)(randomInt)

    val result1 = Quicksort.sequential(testData)
    val result2 = Quicksort.concurrent(testData)
    assertTrue(isSorted(result1))
    assertTrue(isPermutation(testData,result1))
    assertEquals(result1, result2)
  }
}
```

Listing 3: Unit Test for Quicksort

4 Empirical Evaluation

We evaluated DPT on our three algorithm implementations, to assess the efficacy of DPT in reducing execution time. We compared the optimised results to a set of default parameters, and also compared DPT with a random search strategy to confirm that the evolutionary search is exploiting information in the search space. For each algorithm, we ran DPT and Random Search 10 times each. All experiments were run on the same machine, which was a Windows 10 machine using a i7-2670QM processor with four physical cores at 2.20GHz and 8GB RAM. Due to ten repetitions per fitness evaluation, and the repeated runs for statistical testing, the main experiments took several days to complete.

4.1 Tuned Parameters

We evaluated DPT on our three algorithms, with the goal of reducing execution time by modifying performance-critical parameters in the source code, namely:

1. The size *Threshold* at which the algorithms terminates recursion and uses a sequential method instead.
2. The *Leaf* setting, a parameter specific to the Strassen example, which similarly controls recursive behaviour.
3. Akka's *Parallelism Factor*, the number of threads to use relative to the number of physical cores on the machine.

4. Akka's *Throughput* setting, which controls the fairness of resource sharing between threads.

These parameters were automatically extracted from the source code by our DPT tool. In general, there is no guarantee that such parameters will not affect the semantics of a program, a problem that can be mitigated by empirical evaluation using unit tests, and by manual inspection of the results. We implemented suitable unit tests and, once satisfied that the parameters did not impact the semantics of the code, we omitted the execution of those tests during the optimisation process to improve efficiency. Post-optimisation, we validated the results against unseen data, and our confidence is further increased through out knowledge of Akka and the system itself.

We selected a set of default parameter settings, to act as a starting point for the CMA-ES search, and also as a baseline for comparison with the optimised settings. These default parameter settings were chosen based partly on Akka documentation, but also through human judgement: the effectiveness of any settings are dependent on both the program implementation and host machine. The defaults are given in Table 1.

Algorithm	Input Size	Threshold	Leaf	Parallelism Factor	Throughput
FFT	524288	100	N/A	3	100
Quicksort	1000000	100	N/A	3	100
Strassen	800	200	10	3	99

Table 1: Default Parameter Settings for each Algorithm

4.2 Test Data

Each algorithm accepts a numerical vector as input. The size of this vector was selected for each algorithm to ensure its execution time ran for less than 10 seconds using our test machine when using the default parameter settings. The FFT algorithm implementation requires an input size that is a power of two. The input sizes for each algorithm are given in Table 1.

4.3 Sample Size

The inherent nondeterminism of concurrent execution creates a noisy fitness function, which can be exacerbated by the exploration of parameter settings that, for example, reduce the fairness of scheduling threads. After some exploratory data analysis, we chose 10 repetitions to form the basis of our fitness evaluation;

we use the median of 10 measurements when evaluating the execution time of a given candidate solution. This is an imperfect measure, as it still means that a solution may be regarded by the search as superior only due to variance in execution time measurement. We evaluate the outputs of the optimisation separately when comparing against the baseline parameter settings.

4.4 CMA-ES Configuration

The CMA-ES implementation used was from Apache Commons Math [14], using its default parameter settings[5], with an initial sigma one order of magnitude greater than the default parameters. We execute the search for 100 steps, each consisting of the 10 executions of the program with a candidate solution of parameter settings. We take the final output of the search, in the form of the best parameters found for each benchmark.

4.5 Results

CMA-ES vs Default Parameters We executed ten runs of the CMA-ES search for each algorithm. Given the noise inherent in measuring concurrency performance, we wish avoid reporting the behaviour of a possible outlier. As a conservative measure of success, we therefore select the optimised parameters produced by the sixth best result, i.e. an approximation of the median, and compared the resulting performance to the default parameter settings for that algorithm. The parameter settings from that result are provided in Table 2. We took 30 measurements of execution time and report the median in Table 3. We then compared the two sets of measurements for a significant difference using a Mann-Whitney U-Test, and calculated the Vargha-Delaney $\hat{A}_{12}$ measure to quantify effect size. The timing information gathered using the default parameters was tested for normality using the Shapiro-Wilk test, and the test statistic was found to be less than the critical value for the Strassen and Quicksort benchmarks, meaning that we cannot assume a normal distribution of timing values. The Shapiro-Wilk test was chosen as the number of samples is sufficiently small to avoid biases. The Mann-Whitney U test was chosen as we cannot be sure as to the distribution of timing values and as such a parametric test would be inappropriate. Similarly, the Vargha-Delaney measure of effect size was chosen as it too is distribution-agnostic while also being able to handle inputs in the form of real numbers, as opposed to integers.

CMA-ES clearly made a very significant improvement to execution time, even when we only consider a representative, rather than best, result. The improvements in execution time are all significant at the $p < 0.0167$ level (a 0.05 p value Bonferroni-corrected to reflect our three separate benchmarks), and also have the strongest possible effect size. Examining the median execution times, we see an *order of magnitude* improvement for Strassen, and the execution time for FFT is more than halved.

[5] https://commons.apache.org/proper/commons-math/javadocs/api-3.6.1/index.html

	Algorithm	Threshold	Leaf	Parallelism Factor	Throughput
Default	FFT	100	N/A	3	100
	Quicksort	100	N/A	3	100
	Strassen	200	10	3	99
CMA-ES	FFT	1325	N/A	86	735
	Quicksort	2078	N/A	43	277
	Strassen	217	187	48	730

Table 2: Default Parameters compared to Optimised Parameter Settings from sixth best result found by CMA-ES

Algorithm	Default (s)	Optimised (s)	P Value	$\hat{A}_{12}$
FFT	9.16	3.67	2.87e−11	0.0
Quicksort	3.43	1.93	2.87e−11	0.0
Strassen	3.75	0.47	2.87e−11	0.0

Table 3: Median execution time (ET) over 30 runs for default parameter settings and a representative solution found by CMA-ES. Figures to 2 d.p.

CMA-ES vs Random Search In order to demonstrate that CMA-ES produced these results through the exploitation of information within the search space, we implemented a simple random search algorithm as a baseline, and compared the distribution of optimised execution times against that found by CMA-ES over the ten runs. A summary of the results for each benchmark are given in Table 4, and boxplots are given in Figure 1.

Whilst random search was able to make some improvements to execution time, they appear small in comparison to the performance of CMA-ES. We performed a Mann-Whitney U test and calculated the Vargha-Delaney $\hat{A}_{12}$ statistic for the comparison on each benchmark. While the timings resulting from the random search optimisation technique are normally distributed, given that we only have ten samples, we felt that a nonparametric test was safer, and continuing to use the Mann-Whitney U test maintained consistency with our previous experiments. The results are given in Table 5. All tests are significant at the $p < 0.05$ level, and the effect is as strong as possible for FFT and Strassen, whilst still strong for Quicksort. This supports our alternative hypothesis, that is CMA-ES outperforms random search; there exists information in the parameter space that CMA-ES can exploit to tune concurrent performance.

	Algorithm	Min (s)	Max (s)	Median (s)
CMA-ES	FFT	3.54	3.97	3.84
	Quicksort	1.79	2.19	1.91
	Strassen	0.44	0.53	0.48
Random Search	FFT	8.85	8.96	8.92
	Quicksort	3.08	3.57	3.29
	Strassen	3.70	3.75	3.74

Table 4: Optimised execution time statistics from CMA-ES and Random Search. Figures to 2 d.p.

4.6 Threats to Validity

As observed above, wall-clock measurements of concurrent systems are inherently noisy. We have attempted to mitigate against this by using the largest input sizes that still allow learning to take place within a reasonable amount of time, and by taking the median of 10 execution time measurements as our fitness function. The underlying idea is that with larger input sizes, asymptotic behaviour will dominate over 'constant-of-proportionality' effects such as startup-transients caused by Just-In-Time compilation. In addition, we only time the method call to the benchmark itself, excluding JVM startup overhead.

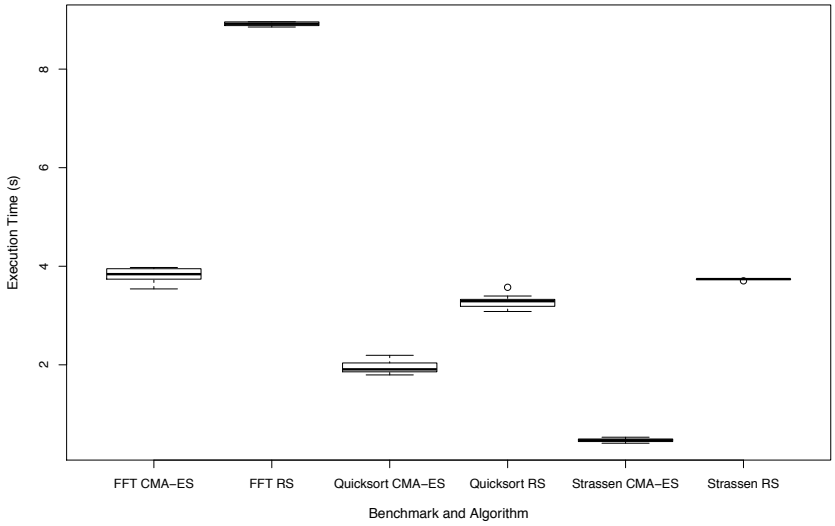

Fig. 1: Execution Times found by CMA-ES and Random Search (RS)

Benchmark	p-value	$\hat{A}_{12}$
FFT	2.80e−6	0.0
Quicksort	0.0095	0.223
Strassen	1.57e−4	0.0

Table 5: Results for Mann-Whitney U Test and Vargha-Delaney Effect Size comparison between CMA-ES and Random Search. Figures to 2 d.p.

5 Related Work

Beginning with early work on compiler optimisation [15], there is an extensive body of work applying semantics-preserving transformations to improve the non-functional properties (NFPs) of software. Recent work in this area includes Kocsis et al. [16], which yield a 10,000-fold speedup of database queries on terabyte datasets within the Apache Spark analytics framework by eliminating redundant database joins and other transformations. Kocsis et al. also automatically repaired 451 systematic errors in the implementation of the Apache Hadoop HPC framework [17], whilst simultaneously significantly improving performance.

In addition to the work improving Quicksort for energy efficiency mentioned in Section 2, Burles et al. [18], also obtained a 24% improvement in energy consumption by optimising a single widely-used class, ImmutableMultimap, in Google's Guava collection library. They used a Genetic Algorithm and constrained the search space via the behavioural contracts of Object-Orientation. Recent work that explicitly addresses parallelism includes refactoring Haskell programs via rewrite rules [19].

Within the last decade there has been increasing interest in the use of stochastic search techniques to optimise NFPs [20], often described as "Genetic Improvement", relying on Genetic Programming as an optimisation method. Early work on execution time optimisation using search focused on obtaining patches to source code [21, 22]. More recently, Baudry and Yeboah-Antwi produced EC-SELR, a framework for *in-situ* runtime optimisation and slimming of software systems, which can optimize and trade-off functional and non-functional properties [23]. Goa et al. used a Genetic Algorithm to optimise webservice composition, with a focus on quality of service [24]. Calderón Trilla et al. also combined search with static analysis [1] to find *worthwhile parallelism*, i.e. semantics-preserving transformations that produce parallelism of meaningful granularity.

In contrast to the optimization of pre-existing software, Agapitos and Lucas used Genetic Algorithms to evolve sorting functions whose time complexity was measured experimentally [25]. Vasicek and Mrazek used Cartesian Genetic Programming to trade the solution quality of median-finding algorithms against NFPs such as power efficiency and execution time within embedded systems [26].

6 Conclusion

We applied the method of deep parameter tuning to extract and optimise literal values from the source code of concurrent versions of three well-known algorithms: FFT, quicksort, and Strassen's matrix multiplication, which make use of the Akka concurrency toolkit. We find that a DPT system based on the CMA-ES evolutionary algorithm achieves significant acceleration of all benchmarks, halving the execution time of FFT and producing an order of magnitude speed-up of Strassen's algorithm.

One of the major challenges in this work was the noisy execution time due to the inherent nondeterminism of the concurrent algorithms. Whilst CMA-ES did

produce good results, exploratory measurements suggest that finding a gradient in the search space is quite difficult. Algorithms more suited to noisy fitness functions may find further improvements.

The execution time of the benchmarks varied according to the architecture of the host machine: thus for best results it is likely that a certain amount of retuning would be required for a given machine. Recently, Sohn et al. [27] demonstrated the feasibility of *amortised optimisation*, that is searching the parameter space at runtime. Applying amortised optimisation to recursive concurrent software may serve as the next challenge for developing this technique.

References

1. J. M. Calderón Trilla, S. Poulding, and C. Runciman. Weaving Parallel Threads. In *SSBSE 2015*, 2015.
2. J. Armstrong. A History of Erlang. In *Proceedings of the Third ACM SIGPLAN Conference on History of Programming Languages*, HOPL III, 2007.
3. J. W. Cooley and J. W. Tukey. An algorithm for the machine calculation of complex Fourier series. *Math. Comput.*, 19:297–301, 1965.
4. C. A. R. Hoare. Algorithm 64: Quicksort. *Commun. ACM*, 4(7), July 1961.
5. V. Strassen. Gaussian elimination is not optimal. *Numerische Mathematik*, 13(4):354–356, 1969.
6. F. Wu, W. Weimer, M. Harman, Y. Jia, and J. Krinke. Deep Parameter Optimisation. In *GECCO Proceedings*, 2015.
7. D. N Rockmore. The FFT: An algorithm the whole family can use. *Computing in Science & Engineering*, 2(1):60–64, 2000.
8. A. A. Karatsuba. The complexity of computations. *Proceedings of the Steklov Institute of Mathematics-Interperiodica Translation*, 211:169–183, 1995.
9. J. W Cooley and J. W Tukey. An algorithm for the machine calculation of complex Fourier series. *Mathematics of computation*, 19(90):297–301, 1965.
10. Jerry Swan and Nathan Burles. *Templar – A Framework for Template-Method Hyper-Heuristics*, pages 205–216. Springer International Publishing, Cham, 2015.
11. J. R. Woodward and J. Swan. Template Method Hyper-heuristics. In *Proceedings of the Companion Publication of the 2014 Annual Conference on Genetic and Evolutionary Computation*, GECCO Comp '14. ACM, 2014.
12. J. Swan, M. G. Epitropakis, and J. R. Woodward. Gen-O-Fix: An embeddable framework for Dynamic Adaptive Genetic Improvement Programming. Technical Report CSM-195, Computing Science and Mathematics, University of Stirling, 2014.
13. N. Hansen and A. Ostermeier. Adapting arbitrary normal mutation distributions in evolution strategies: the covariance matrix adaptation. In *Proceedings of IEEE International Conference on Evolutionary Computation*, 1996.
14. N. Hansen and A. Ostermeier. Completely Derandomized Self-Adaptation in Evolution Strategies. *Evol. Comput.*, 9(2):159–195, 2001.
15. A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley Longman Publishing Co., Inc., 1986.
16. Zoltan A. Kocsis, John H. Drake, Douglas Carson, and Jerry Swan. Automatic improvement of apache spark queries using semantics-preserving program reduction. In *Proceedings of the 2016 on Genetic and Evolutionary Computation Conference Companion*, GECCO '16 Companion, pages 1141–1146, New York, NY, USA, 2016. ACM.

17. Zoltan A. Kocsis, Geoff Neumann, Jerry Swan, Michael G. Epitropakis, Alexander E. I. Brownlee, Saemundur O. Haraldsson, and Edward Bowles. *Repairing and Optimizing Hadoop hashCode Implementations*, pages 259–264. Springer International Publishing, Cham, 2014.
18. Nathan Burles, Edward Bowles, Alexander E. I. Brownlee, Zoltan A. Kocsis, Jerry Swan, and Nadarajen Veerapen. *Object-Oriented Genetic Improvement for Improved Energy Consumption in Google Guava*, pages 255–261. Springer International Publishing, Cham, 2015.
19. C. M. Brown, K. Hammond, and H. Loidl. Paraforming: Forming Parallel Haskell Programs Using Novel Refactoring Techniques. In *Trends in Functional Programming*, 2011.
20. D. R. White, J. Clark, J. Jacob, and S. M. Poulding. Searching for Resource-efficient Programs: Low-power Pseudorandom Number Generators. In *GECCO 2008 Proceedings*, 2008.
21. Mark Harman, William B. Langdon, Yue Jia, David R. White, Andrea Arcuri, and John A. Clark. The GISMOE Challenge: Constructing the Pareto Program Surface Using Genetic Programming to Find Better Programs (Keynote Paper). In *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*, ASE 2012, pages 1–14, New York, NY, USA, 2012. ACM.
22. T. Ackling, B. Alexander, and Ian Grunert. Evolving Patches for Software Repair. In *GECCO 2011 Proceedings*, 2011.
23. K. Yeboah-Antwi and B. Baudry. Embedding Adaptivity in Software Systems Using the ECSELR Framework. In *Proceedings of the Companion Publication of the 2015 Annual Conference on Genetic and Evolutionary Computation*, GECCO Companion '15, 2015.
24. Z. Gao, C. Jian, X. Qiu, and L Meng. QoE/QoS driven simulated annealing-based genetic algorithm for Web services selection. *The Journal of China Universities of Posts and Telecommunications*, 16:102–107, 2009.
25. A. Agapitos and S. M Lucas. Evolving efficient recursive sorting algorithms. In *2006 IEEE International Conference on Evolutionary Computation Proceedings*, 2006.
26. V. Mrazek, Z. Vasicek, and L. Sekanina. Evolutionary approximation of software for embedded systems: Median function. In *Proceedings of the Companion Publication of the 2015 Annual Conference on Genetic and Evolutionary Computation*, 2015.
27. J. Sohn, S. Lee, and S. Yoo. Amortised Deep Parameter Optimisation of GPGPU Work Group Size for OpenCV. In F. Sarro and K. Deb, editors, *SSBSE 2016*, 2016.