EuroGP-2024. Mario Giacobini, Bing Xue, Luca Manzoni *Eds.*, Aberystwyth. 3-5 April 2024. Springer.

Genetic Improvement of Last Level Cache

William B. Langdon and David Clark

W.Langdon@cs.ucl.ac.uk david.clark@ucl.ac.uk
CREST, Department of Computer Science,
UCL, Gower Street, London, WC1E 6BT, UK

Abstract. With increasing reliance on multi-core parallel computing performance is evermore dominated by interprocessor data communication typically provided by last level cache (LLC) shared between CPUs. In an 8 core 3.6 GHz desktop using multiple local searches, the Magpie parameter tuning genetic improvement (GI) system was able to reduce L3 cache access (load + stores) four fold on an existing open source 7000 line C PARSEC parallel computing VIPS image benchmark.

Keywords hill climbing, SBSE, Software Engineering, automatic code optimisation, srcml, XML, parameter tuning, reduced search space, Linux perf

1 Introduction

The computing industry grew up in the presence of Moore's Law [1] ensuring in the early days software producers could always access more powerful computers by the time their programs were ready for release. The days of clock speeds doubling every two years are long gone, however silicon chip manufactures are using the still increasing number of transistors to pack more processing cores and larger cache memory onto their devices. This will continue into the foreseeable future with more cores becoming available and so increasing importance of communications between CPUs. Excluding specialised hardware, such as FPGAs and GPUs, many current parallel applications communicate between CPUs via shared memory. In multi-core silicon chips this can be highly effective. However modern CPUs run far faster than main memory and a complex hierarchy of cache memory is needed to try to keep data close to the individual computing engines. In many parallel multi-threaded applications the last level cache (LLC) is shared between cores and provides the main communication between threads running on different CPUs within the same chip. In most cases control of the cache hierarchy remains proprietary. Although cache memory will increase in size, it will remain both the main bottleneck limiting performance for many applications and outside programmer control. We show genetic improvement can in principle be used to automatically tune open source software to minimise use of the shared LLC cache, obtaining a 4.0× reduction, without deep access to the operating system or inner workings of the silicon chip.

PARSEC (Princeton Application Repository for Shared-Memory Computers) is a benchmark suite of parallel computing programs, which focuses on emerging

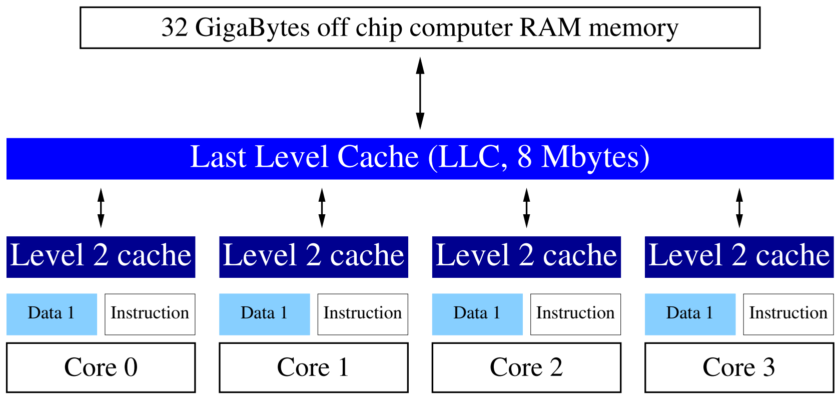

Fig. 1. Schematic of three level on-chip cache hierarchy. The last level (here level 3) cache is by far the largest on chip cache. (In our desktop each data and instruction L1 cache is 32 Kbytes, L2 each 256 Kbytes and the LLC (L3) is 8 Mbytes.) As well as interfacing with off chip memory, LLC also provides communication between the compute cores (just 4 cores are shown).

workloads [2, page 73]. It includes VIPS [3], which is an image processing library written in C. We selected the VIPS thumbnail image processing benchmark as it is multi-threaded and can be easily scaled to cover the critical size of modern chip caches. Indeed we use the Linux perf tool to measure its cache use during its multi-threaded generation of a small "thumbnail" image from an image exceeding the cache size (see Figure 2 `https://github.com/wblangdon/vips`). To avoid potentially complicated trade-offs between cache, image size and image quality (available in much more complicated image formats such as JPEG), for both images we use non-compressed P6 raster scan full colour images (see Section 7.3) and insist the mutated code produces identical output.

Fig. 2. 128×96 thumbnail image generated by VIPS.

In Section 3 we describe our use of the Magpie [4] genetic improvement system to simultaneously tune application specific parameters, compiler and linker options, and the VIPS C source code. The VIPS benchmark is detailed in Section 4. Whilst Section 5 describes how we use the Linux perf API to measure last level cache LLC usage and measures to combat noise. Section 6 shows in many cases Magpie is able to reduce cache usage by on average 75% (±4%). In the discussion (Section 7) we note that, amongst other changes, most successful mutations involved VIPS application parameters and (in Section 7.2) run and report additional experiments just tuning them. Section 7.3 summarises the VIPS thumbnail code and proposes an explanation for why Magpie's patches work. We conclude (Section 8) that despite noise, Magpie can find a single parameter change which reduces LLC cache use four fold. But first we give the background.

2 Background

Until recently genetic improvement (GI) [5,6] has applied genetic programming (GP) [7,8,9], to existing human written software, however in principle any optimisation technique, such as search-based software engineering [10], Grammatical Evolution [11]–[14], Novelty Search [15], Fuzzy Systems [16], or AI [17]–[20], can be used. Recently Magpie [4] has shown the power of local search in GI [21,22,23]. Already genetic improvement has been applied to automatic porting [5], transplanting code [24,25] code optimisation [26,27], including JavaScript [28] and Clang LLVM IR intermediate code [29,30], hardware design [31], automatic software testing [14] and cryptographic code [32]. Genetic improvement has been demonstrated on GPU applications [33]—[36] including BarraCUDA [37], the first GI code to be accepted into actual use [38]. At EuroGP'19 [39] we showed GI could also speed up parallel CPU code. The resulting GIed RNAfold [40] was accepted into production and like the GI version of BarraCUDA has been downloaded many thousands of times (for example [41]).

Previously we [42] showed GP optimising L1 cache but L1 is much more tightly bound to the CPU running the application (see Figure 1) and in half the cases we were unable to find an improvement. Jimenez et al. [43] use a genetic algorithm (GA) to improve the LLC cache but their approach is to improve future generic cache designs rather than improving specific multi-core applications. Klinkenberg et al. [44] describe H2M which is a heuristic tool for managing data placement in complex memory architectures in high performance computers (HPC, i.e. super computers). However, they use fixed hand made heuristics and are concerned only with runtime rather than seeking to show evolution can in general optimise last level cache (LLC) use by application software. They agree that managing diverse memory in parallel computing environments by hand is hard and yet will become increasingly important. Cloud White [45] is a tool for monitoring LLC contention between different customers' virtual machines (VMs) when they run on the same multi-core cloud computer server. Pons et al. [45] claim low overhead, but Cloud White is a black-box tool for monitoring Quality of Service (QoS) rather than an optimisation tool. Whereas Clite [46] uses Bayesian Optimization to try to get the best mix of existing VMs rather than optimising individual applications.

3 Magpie

MAGPIE (Machine automated general performance improvement via evolution of software) [4] is a freely available genetic improvement system written in Python[1] and designed to be applied to software written in any programming language. The current release was downloaded from GitHub[2]. Magpie is well documented. For example, its GitHub pages include examples and tutorials. Also there is a more formal description [4]. Although we have used GI to optimise both code and parameters before [47], Magpie is unique in being a general genetic improvement framework that can optimise simultaneously parameters and any programming language. Parameters to be optimised might be, for example: constants[3], program command line and execution parameters (Section 4.2) and/or compiler options (Section 4.3). While much existing GI work has been based on lines of source code (which Magpie also supports), we use its ability to work with source code at the compiler's AST level by using XML trees.

4 PARSEC VIPS Thumbnail Benchmark

The VIPS image processing library was downloaded as part of PARSEC 3.0 from GitHub[4]. PARSEC as a whole is enormous, even the VIPS source library (sub directory `pkgs/apps/vips/···/src/libvips`) contains more than 90 000 lines of code (mostly C source code).

4.1 Profiling VIPS thumbnail, targeting C code, generating XML

As mentioned in the introduction, we chose the VIPS thumbnail benchmark from the VIPS library. vipsthumbnail.exe was compiled and linked following the VIPS installation documentation and profiled using the Linux perf profiling utility (perf version 3.10.0) operating at its maximum sampling frequency (40 000 Hz). perf collected data from ten runs with a variety of number of concurrent threads (–vips-concurrency). In all cases run time was dominated by the shrink_gen function. Remember VIPS is essentially a library, most of which is not used by an individual application. To extricate the important code used by vipsthumbnail.exe, we took the union of functions in the hierarchical call of shrink_gen and any function sampled by perf. This gave us 37 .c source files containing 10 829 lines of C code. Notice this is not the whole of the VIPS thumbnail benchmark, it is still necessary to link to the libvips.so shared object library, but the 37 files do contain important code which we wish Magpie to optimise. A further filtering operation was done to select just the functions that are used during

[1] We use Python 3.10.1

[2] `https://github.com/bloa/magpie` (last update before submission 2 October 2023)

[3] Our work evolving 50 000 parameters for RNAfold's free energy minimisation algorithm [48] and evolving 512 floating point values to convert the GNU C square root function into other functions [49], was done before Magpie was available.

[4] `https://github.com/bamos/parsec-benchmark` Version 3.0 for 64-bit x86

fitness testing (Section 5), reducing the 37 files to a total of 7 328 lines of code. These were converted to XML using scrml version 1.0.0 and made available to Magpie to tune.

4.2 VIPS thumbnail parameters

The VIPS thumbnail benchmark allows up to 12 command line parameters, however some of these change the output. Excluding these, left five (`vips-concurrency`, `vips-tile-width`, `vips-tile-height`, `vips-thinstrip-height` and `vips-fatstrip-height`) all of which were made available via a parameter file to Magpie to tune. Magpie starts its search from the default values.

4.3 GCC compiler and linker options

The GNU compiler/linker version 10.2.1 has several hundred command line options. Rather than use them all, we selected those that appear in the installation scripts for VIPS plus some commonly used compilation options. For each, we set the default to the value used in the VIPS installation process but allowed Magpie the full range of allowed values. For example, VIPS uses -O2, so by default Magpie uses -O2. Although -O3 is available to Magpie, it was not used in the successful patches (Section 6). GCC options available to Magpie to tune are: `-fPIC -O -DNDEBUG -fvisibility -std=c99 -msse4.1 -fno-exceptions -ffat-lto-objects -flto -fno-strict-aliasing -fopenmp -fstack-protector -fstack-protector-strong -ftree-vectorize -g -m64 -mtune=generic -nostdlib --param=ssp-buffer-size -pipe -std=c++11 -std=c++98`

4.4 Magpie local search parameters

Due to the noisy nature of LLC cache usage, Magpie was run 100 times but each run was allowed only 100 local search steps. Otherwise the Magpie defaults, such as default time out for fitness evaluation (30 seconds) and limit on output generated during fitness testing (10 000 bytes), were used.

Almost the full suite of Magpie's XML mutation operators were enabled: literal numbers, StmtReplacement, StmtInsertion, StmtDeletion, ComparisonOperatorSetting, ArithmeticOperatorSetting, NumericSetting, RelativeNumericSetting.

Magpie keeps track of the C source code it has mutated (via XML) and so only the mutated C code needs to be recompiled. In contrast, if the compiler command line is changed, all 37 C source files must be recompiled.

The mutated vipsthumbnail.exe was run with a command line generated by Magpie from the five variable VIPS thumbnail command line parameters (see Section 4.2).

5 Fitness Function

The experiments were run on a standard networked Centos 7 desktop. Even when apparently idle, it has more than two hundred active Linux processes, all of which use the LLC cache. LLC cache measurements are noisy (see, for example, Figure 4. We used the Linux perf tool's API to measure cache usage during the critical multi-threaded image processing operations which create the thumbnail[5]. This allows us to isolate it from mundane operations, like processing the command line and reading and writing the image files. However initial experiments to reduce noise by placing the LLC cache in a defined state before starting perf measurements were unsuccessful. Instead the unmutated code was used as a reference and fitness is based on running it and then running the mutant as soon as possible, then setting fitness to the (signed) difference between their cache usage. Unfortunately even this paired approach is still quite noisy. (LLC measurements for the original unmutated code show that the coefficient of variation is 14%.) Indeed Section 6 suggests Figure 4) shows running original and mutant as a pair failed to eliminate fitness cache measurement noise.

To summarise there are multiple aspects of a mutation's fitness: 1) If Magpie mutated one or more XML files (Sections 4.1 and 4.4) or it changed the GCC options (Section 4.3), do the C source files still compile and link without error, 2) Does the mutant program run ok with the possibly mutated command line (Section 4.2), 3) Does it, within the two second timeout, produce an output file[6], 4) Is the output identical, 5) Finally, fitness is the number of LLC cache accesses by the reference program minus that of the mutant (remember Magpie minimises fitness). If any of the tests 1)–4) fail, Magpie discards the mutation.

6 Results

Magpie was run one hundred times on an otherwise idle 8 core 3.6 GHz Intel i7-4790 networked desktop with an 8 Mbyte LLC (L3) cache. Each time Magpie was allowed up to one hundred search steps (see Figure 3). The whole 100 runs took less than five hours, cf. horizontal axis in Figure 5. Mean Magpie run time 2:47 minutes:seconds each.

Figure 3 plots the training fitness of patches during each run. 38 Magpie runs terminated early and did not produce a best of run patch (black $\circ$). Figure 3 splits the remaining 62 Magpie runs into 21 $+$ whose best of run patch failed to generalise and 41 $\times$ where it did (see also Figures 4 and 5). Figure 3 shows by half way through, runs whose best patch will generalise are doing better (remember we are minimising) than the others, whose fitness tends to be scattered about the mean performance of the original C code (cf. 100% on vertical axis in Figure 3). (In an effort to deal with noise, Magpie by default, performs three warmup fitness

[5] There is a vipsthumbnail command line option to allow the user to control the number of threads used during thumbnail image creation: –vips-concurrency, Section 4.2.

[6] The Linux `limit filesize` command can be used to restrict the total size of files generated but this was not necessary in these experiments.

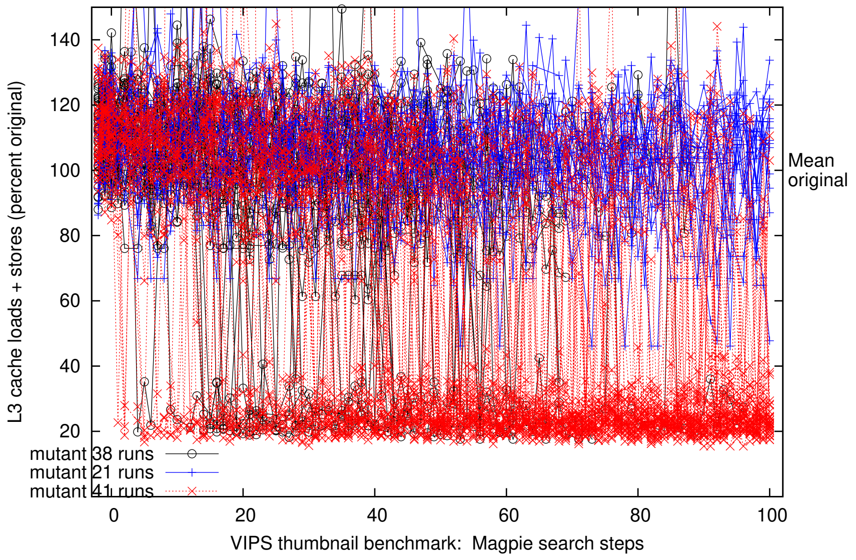

Fig. 3. 100 runs minimising last level cache accesses (load + stores) during multiple threaded processing reducing 3264×2448 image (23 970 833 bytes) to 128×96 (Figure 2). 38 runs did not complete ∘, 21 produced poor patches +, 41 reduced LLC cache usage 4.0 fold ×. See also Figures 4 and 5.

evaluations before commencing its search, hence in Figure 3 the horizontal axis starts at -2 rather than 1.)

To counter over fitting and measure out-of-sample performance, the 62 best of run patches each were individually tested again 100 times. As with fitness testing (Section 5), each test consists of executing the reference unmutated code and the mutant as a pair, measuring their cache usage and checking the mutant still produces identical output. Figure 4 shows the cache use of the original code (x-axis) and of the mutant (y-axis). Notice noisy scatter of the data. Figure 4 (blue dots top) shows 21 mutants failed to give an improvement when run again. Their individual performance is pretty close to that of the original code (cf. the diagonal line in Figure 4). Indeed the noise is also similar, giving rise to the approximately circular pattern at the top of Figure 4). Also note the circular pattern indicates little correlation between the two measurements, suggesting the pairing of the reference and mutated code is ineffective at noise suppression in this case. Which hints that any pattern (if any) in the noise takes place faster than the < 30 milliseconds between running the reference and mutated programs. In contrast 41 patches, despite the noise, always give better performance. (Plotted as red dots in the lower part of Figure 4.) Although each of the 41 patches is in detail different, their performance are remarkably similar and each gives a $\approx 4.0\times$ reduction in LLC cache usage, Figure 5.

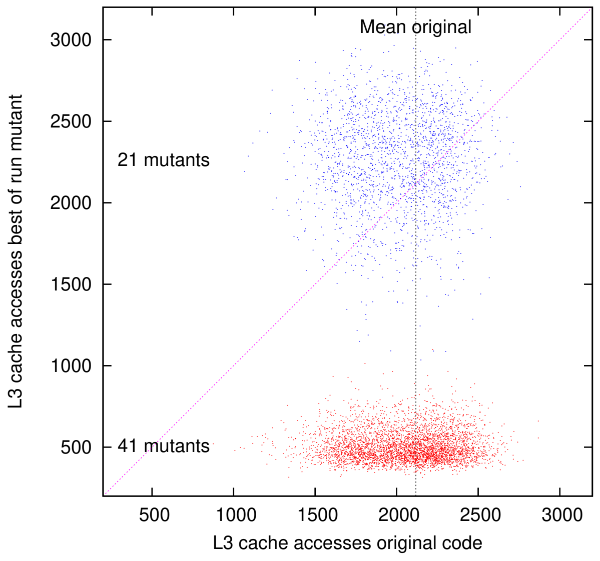

Fig. 4. LLC cache use of 62 best of run mutants, each tested 100 times (vertical axis). Horizontal axis LLC cache use of original code. Blue dots (near diagonal line) show 21 mutants which fail to improve on original code. Red dots (lower) show 41 mutants are always better than the unmutated code. Also Figure 5

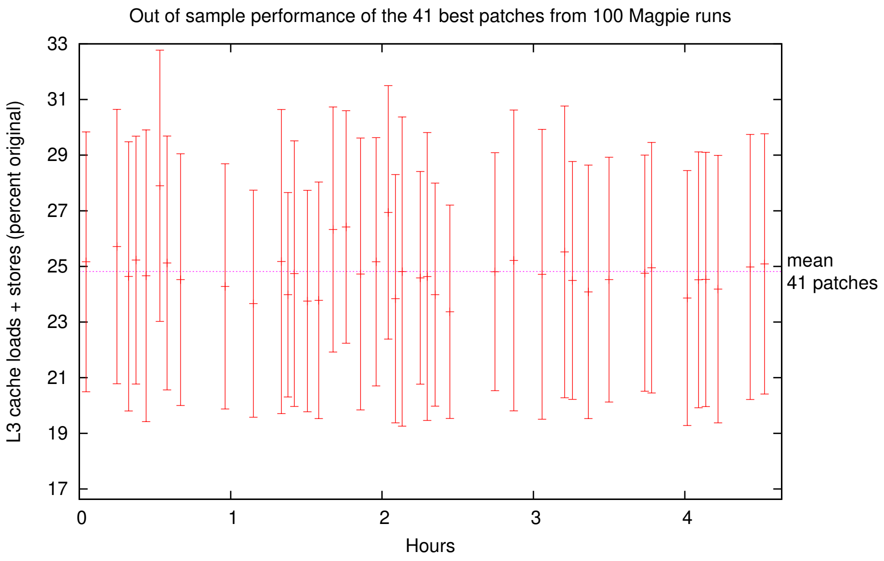

Fig. 5. 100 Magpie runs. Mean + and estimated standard deviation (error bars) of last level cache (LLC) use of the 41 good best of run patches found by Magpie (average of one hundred samples, cf. Figure 4). The horizontal axis shows when each patch was reported. The first successful mutant (left most) was found by Magpie after 2:42 minutes:seconds.

7 Follow up Experiment: Optimising VIPS parameters

7.1 Types of improvement found in 41 successful Magpie runs

In the best of run mutants in the 41 runs which produced good patches there were between 1 and 10 individual changes (mean 4.9, total 201). There are only 15 GCC command line changes, none of which seem related to optimisation. For example, -O is not used. This may be because the available compiler and linker options aim to reduce the time taken for computation rather than data access. In contrast, all but two of the 41 patches tune one or more VIPS application parameters. Similarly, all but one of these 41 mutations changes one or more C source code files. In total there are: 8 C code insertions, 9 replacements and 23 statement deletions. Of the "smaller" code mutations, there are 20 arithmetic operator and 24 comparison changes, 27 direct changes to numbers and 30 relative changes (e.g. increasing by 50%). And 60 GCC or VIPS parameter changes. It is difficult to evaluate all the code changes but some appear not to matter as, despite removing whole unused functions before running Magpie (Section 4.1) they change code that is not executed or make a syntax change but the code semantics are unchanged, e.g. replace 0 by (0/2). Table 1 further summarises the 201 genes from the successful patches by mutation type and C source file.

7.2 Optimising vips-thinstrip-height and vips-fatstrip-height

From the top of Table 1 it is clear that VIPS parameter tuning stands out amongst the changes in the 41 successful Magpie runs. Therefore a second set of Magpie experiments were run to optimise LLC cache use by tuning only the two VIPS application parameters which occurred in almost all of the 41 successful mutants found in Section 6. Magpie was set up identically except: the GCC command line arguments and XML files and XML mutations were not used. Only vips-thinstrip-height and vips-fatstrip-height where included in the Magpie VIPS parameters. As before Magpie started from their default values (1 and 16 respectively) and again for both Magpie chose integer mutation values uniformly at random from the range 1 to 1200. Since it was no longer necessary to compile and link each mutant, Magpie runs were much faster (18.5 seconds).

In these final Magpie runs, the search space is greatly reduced, from effectively infinite to $[1200]^2$ (1 440 000). With 100 runs, each with up to 100 samples, in total Magpie was able to approximately sample the search space (see Figure 6). The combined sampling suggests that vips-thinstrip-height correlates well with LLC cache usage. Therefore Figure 7 concentrates upon it. Figure 6 shows values above vips-thinstrip-height = 6 simply scatter about the mean[7] and so they are omitted from Figure 7). Instead their mean is plotted with a horizontal line in Figure 7. Allowing for the noise, Figure 7 suggests a monotonic reduction in LLC usage as vips-thinstrip-height is increased from its default value 1 and that vips-thinstrip-height ≥ 6 gives a 4.0 fold reduction in LLC use.

[7] Although Table 2 suggests a slight downward trend in perf last level cache LLC measurements with increasing vips-thinstrip-height, this is not visible in Figure 6, which includes both vips-thinstrip-height and vips-fatstrip-height.

Table 1. Distribution of 201 genes in the 41 best of run Magpie patches. The table is sorted by frequency, most important first, and is read left to right in row order. Thus VIPS thumbnail command line parameters and GCC compilation switches are mixed with source code arithmetic constants and XML changes. The numbers give each gene's frequency.

Type	parameter name/file		Type	parameter name/file	
VIPS	vips-thinstrip-height	39	Arith XML	im_shrink.c.xml	10
Arith XML	im_guess_prefix.c.xml	7	Arith XML	threadpool.c.xml	6
Arith XML	rw_mask.c.xml	5	Arith XML	memory.c.xml	5
Arith XML	im_vips2ppm.c.xml	5	Arith XML	im_prepare.c.xml	5
Arith XML	im_demand_hint.c.xml	5	Arith XML	format.c.xml	5
XML	im_demand_hint.c.xml	4	XML	check.c.xml	4
Arith XML	time.c.xml	4	Arith XML	im_embed.c.xml	4
Arith XML	check.c.xml	4	Arith XML	buffer.c.xml	4
XML	window.c.xml	3	XML	meta.c.xml	3
XML	im_init_world.c.xml	3	VIPS	vips-tile-width	3
VIPS	vips-fatstrip-height	3	GCC	-fstack-protector	3
Arith XML	object.c.xml	3	Arith XML	init.c.xml	3
Arith XML	im_conv.c.xml	3	Arith XML	im_affine.c.xml	3
Arith XML	debug.c.xml	3	XML	time.c.xml	2
XML	region.c.xml	2	XML	interpolate.c.xml	2
XML	init.c.xml	2	XML	im_guess_prefix.c.xml	2
XML	debug.c.xml	2	GCC	-mtune=generic	2
GCC	-ftree-vectorize	2	GCC	-fopenmp	2
Arith XML	window.c.xml	2	Arith XML	util.c.xml	2
Arith XML	region.c.xml	2	Arith XML	rect.c.xml	2
Arith XML	im_open.c.xml	2	Arith XML	im_copy.c.xml	2
Arith XML	im_close.c.xml	2	XML	util.c.xml	1
XML	semaphore.c.xml	1	XML	object.c.xml	1
XML	memory.c.xml	1	XML	im_vips2ppm.c.xml	1
XML	im_prepare.c.xml	1	XML	im_copy.c.xml	1
XML	im_convsep_f.c.xml	1	XML	im_convsep.c.xml	1
XML	im_affine.c.xml	1	XML	buffer.c.xml	1
GCC	-m64	1	GCC	-g	1
GCC	-fvisibility	1	GCC	-fstack-protector-strong	1
GCC	-fno-strict-aliasing	1	GCC	-flto	1
Arith XML	sinkdisc.c.xml	1	Arith XML	semaphore.c.xml	1
Arith XML	interpolate.c.xml	1			

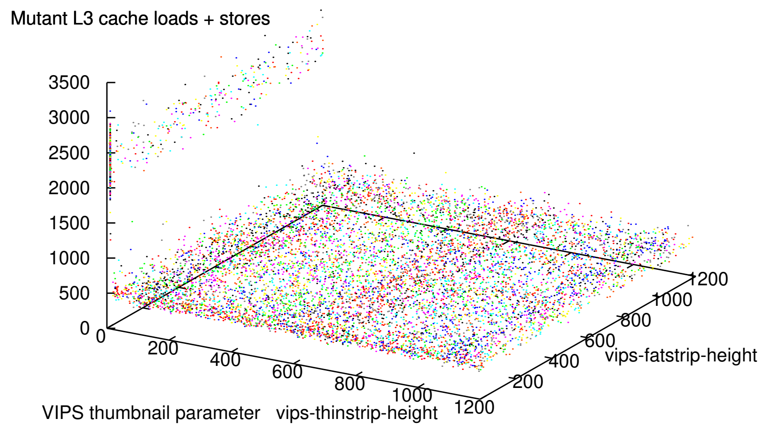

Fig. 6. 2nd Magpie experiment. LLC cache measurements during 100 runs to simultaneously tune vips-thinstrip-height and vips-fatstrip-height. Data from the same run have the same colour. The vertical line of dots above (1,16) are the initial default starting point for all 100 runs.

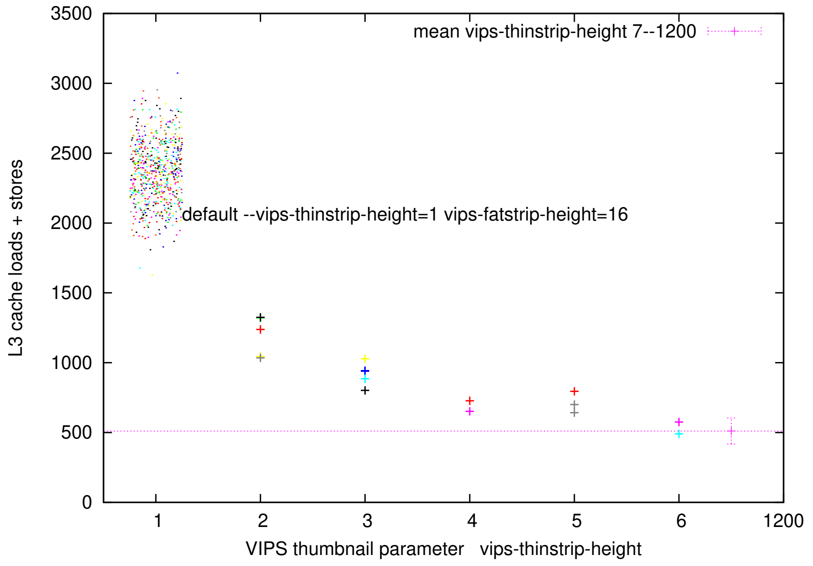

Fig. 7. 2nd Magpie experiment. LLC cache measurements during 100 runs to simultaneously tune vips-thinstrip-height and vips-fatstrip-height (the same data as Figure 6). Values above vips-thinstrip-height=6 not plotted as data are simply scattered about the mean (dotted line). Horizontal noise added to spread data for vips-thinstrip-height=1.

7.3 vips-thinstrip-height

This section tries to explain Magpie's results in terms of the VIPS application. Two dimensional images, such as photographs, are usually laid out on disk and in memory as rows of consecutive pixels. Starting at the left of the top edge and moving along it to the right edge and then moving down to the left hand edge of the second row. This pattern is repeated, working left to right across each row and progressively down the image until we reach the right hand end of its bottom row.

Although it is now common place for computers to have enough main RAM memory to store uncompressed the whole image, often images are too big to fit into the cache. For example, excluding metadata, a full colour P6 (3 bytes per pixel) 3264×2448 image occupies $3 \times 3264 \times 2448 = 23\,970\,816$ bytes, whereas these experiments were run on a computer with a LLC cache of 8 388 608 bytes. At more than 7000 lines of deeply nested [50] multi-threaded code, it is difficulty to be exactly sure which actions impact the LLC cache and in which ways. However, to exploit multi-threading, the VIPS library (ignoring small overlaps and edge effects) divides the input image into equal sized rectangular tiles. These are processed by separate threads and so random timing effects mean that they are processed in a different order in different runs, but they approximately follow the left to right top to bottom ordering normally used for image processing. The tiles are 128 pixels (384 bytes) wide. Depending upon alignment, they occupy 6 or 7, 64 byte cache lines. With the default value of vips-thinstrip-height (1), they are 1 pixel high, so for our 3264×2448 example each row takes $\lceil 3264/128 \rceil =$ 26 (25.5) tiles, and there are a total of 63 648 tiles.

On average, see Figure 4 and Table 2, processing the image takes 2161 LLC cache accesses. 1909 are LLC cache loads, the rest are LLC cache stores. Meaning on average each LLC cache load access fetches 12 600 bytes of the image. What appears to be happening is the LLC cache is asked for 26 tiles of data. These occupy 9 9792 bytes (depending upon alignment, this is 153 or 155 cache lines). Even though these data requests arrive at different times from different threads, the cache hierarchy appears to be able to consolidate these into a single LLC access. (9 9792 is within 28% of the average LLC cache load size. See column 9 in top row of Table 2.) When these data arrive, the 8 threads are able to process the 26 tiles and then request data for the next row (again 153 or 155 cache lines). It is not clear why the cache hierarchy is able to consolidate requests for 25.5 tiles but not for multiple rows.

When vips-thinstrip-height is increased to 2, the VIPS tiles are increased from 128×1 to 128×2 pixels (again ignoring overlaps). Again depending upon alignment, each tile now occupies 12 or 14 cache lines. and the number of LLC cache read accesses falls on average to 924, i.e. 25 900 bytes each. This roughly doubling of the mean size of data per LLC cache load is consistent with the idea that the cache hierarchy is able to consolidate 26 scattered request for cache reads but something about reading the end of two rows of consecutive bytes prevents further consolidation.

Table 2 shows as vips-thinstrip-height is increased to 3, 4, 5 and 6, the average number of bytes fetched per LLC load access increases and remains approximately the same as the size of one row of tiles. I.e., the last but one column in Table 2 remain near 1.0.

Although vips-thinstrip-height can be increased above 6, from Table 2 it is clear that this has only a small effect. It may be the VIPS code itself places a limit on the tile height. Alternatively the computer's RAM and/or the cache hierarchy may limit pre-fetch sizes to 64 Kbytes (65 536 bytes), cf. column 6 in Table 2. Above vips-thinstrip-height=96 (a magic number for VIPS' tile height) there appear to be no further reductions.

It is clear that the interaction between vips-thinstrip-height and the LLC is noisy and complicated but its interaction with internal VIPS tile height seems like a reasonable first step at explaining how it works and why vips-thinstrip-height stands out in Magpie's "hands clean" optimisation.

Table 2. Mean impact on last level cache LLC of running the original code using 8 threads with various values of –vips-thinstrip-height (column 1) 100 times. Numbers in brackets are estimates of standard deviation, except last column where () indicates standard error in the calculation of $\frac{\text{mean LLC load size}}{\text{rows size}}$ (column 9 = column 6 divided by column 8.)

thinstrip	LCC loads	stores	load bytes	rows bytes	Ratio
1	1909 (290)	252 (22)	12600 (1910)	9792	1.28 (0.020)
2	924 (191)	188 (19)	25900 (5340)	19584	1.32 (0.027)
3	764 (113)	173 (13)	31400 (4650)	29376	1.07 (0.016)
4	609 (108)	166 (13)	39300 (7010)	39168	1.00 (0.018)
5	578 (89)	165 (13)	41500 (6360)	48960	0.85 (0.013)
6	477 (92)	155 (12)	50200 (9690)	58752	0.85 (0.017)
7	450 (84)	152 (12)	53200 (9870)	68544	0.78 (0.014)
8	447 (82)	152 (12)	53700 (9830)	78336	0.69 (0.013)
9	455 (74)	151 (12)	52700 (8540)	88128	0.60 (0.010)
10	432 (80)	147 (12)	55500 (10230)	97920	0.57 (0.010)
20	424 (93)	149 (14)	56600 (12450)	195840	0.29 (0.006)
40	410 (81)	145 (11)	58500 (11580)	391680	0.15 (0.003)
60	371 (69)	145 (12)	64500 (11950)	587520	0.11 (0.002)
80	383 (75)	148 (12)	62600 (12250)	783360	0.08 (0.002)
96	341 (62)	145 (11)	70200 (12840)	940032	0.07 (0.001)
100	353 (72)	146 (11)	67800 (13830)	979200	0.07 (0.001)
500	350 (84)	146 (12)	68500 (16370)	4896000	0.01 (0.000)
1000	345 (61)	146 (11)	69400 (12340)	9792000	0.01 (0.000)
1500	345 (57)	145 (12)	69400 (11450)	14688000	0.00 (0.000)
2000	364 (75)	144 (10)	65900 (13580)	19584000	0.00 (0.000)
2448	342 (57)	146 (10)	70000 (11560)	23970816	0.00 (0.000)
2500	347 (64)	146 (11)	69000 (12640)	23970816	0.00 (0.000)

8 Conclusions

We have shown an “off the shelf” genetic improvement (GI) system Magpie [4] using multiple hill climbing runs can automatically tune a standard parallel processing benchmark to reduce four fold its use of last level cache (LLC) on modern multi-core computers.

The VIPS image processing thumbnail benchmark is representative of a large class of parallel processing programs. Its multi-processing is based on POSIX pthreads, which is heavily used in multi-core applications. Although we expect continued growth of hybrid computers which off load significant computation to accelerators (e.g. GPU [51], TPUs [52], FPGAs [53]), we expect they will remain hard to program effectively and so they may remain the preserve of artificial intelligence (AI) deep artificial neural networks [54], e.g. for training large language models (LLMs) and specific domains such as astronomy [55] and bioinformatics [56]. Instead we anticipate mundane applications will continue to be run on multi-core computers. To get the best of their increasing numbers of cores will require, not just optimising the code, but also increasingly optimising data communication. However it appears (cf. Section 7.1) existing compilers optimise computation not data access (“Compilers are not good at managing caches” [57, p44]). Therefore new tools will be needed [58] to make effective use of complex proprietary hardware cache hierarchies, whose operation is invisible to the user level programmer. By taking a “hands off” approach Magpie may be able to help programmers by optimising for them the last level cache which provides high bandwidth inter-core communication, which will be increasingly needed in what promises to be the dominant domain for future software development.

Acknowledgements

I am grateful for the help of Aymeric Blot and Dan Blackwell.

References

1. Moore, G.E.: Cramming more components onto integrated circuits. Electronics 38(8), 114–117 (April 19 1965)
2. Bienia, C., Sanjeev Kumar, Jaswinder Pal Singh, Kai Li: The PARSEC benchmark suite: characterization and architectural implications. In: Moshovos, A., Tarditi, D., Olukotun, K. (eds.) 17th International Conference on Parallel Architectures and Compilation Techniques, PACT 2008. pp. 72–81. ACM, Toronto, Ontario, Canada (October 25-29 2008), `http://dx.doi.org/10.1145/1454115.1454128`
3. Martinez, K., Cupitt, J.: VIPS - a highly tuned image processing software architecture. In: Proceedings of the 2005 International Conference on Image Processing, ICIP. pp. 574–577. IEEE, Genoa, Italy (September 11-14 2005), `http://dx.doi.org/10.1109/ICIP.2005.1530120`
4. Blot, A., Petke, J.: MAGPIE: Machine automated general performance improvement via evolution of software. arXiv (4 Aug 2022), `http://dx.doi.org/10.48550/arxiv.2208.02811`

5. Langdon, W.B., Harman, M.: Evolving a CUDA kernel from an nVidia template. In: Sobrevilla, P. (ed.) 2010 IEEE World Congress on Computational Intelligence. pp. 2376–2383. IEEE, Barcelona (18-23 Jul 2010), `http://dx.doi.org/10.1109/CEC.2010.5585922`
6. Petke, J., et al.: Genetic improvement of software: a comprehensive survey. IEEE Transactions on Evolutionary Computation 22(3), 415–432 (Jun 2018), `http://dx.doi.org/doi:10.1109/TEVC.2017.2693219`
7. Koza, J.R.: Genetic Programming: On the Programming of Computers by Means of Natural Selection. MIT Press, Cambridge, MA, USA (1992), `http://mitpress.mit.edu/books/genetic-programming`
8. Banzhaf, W., Nordin, P., Keller, R.E., Francone, F.D.: Genetic Programming – An Introduction. Morgan Kaufmann (1998), `https://www.amazon.co.uk/Genetic-Programming-Introduction-Artificial-Intelligence/dp/155860510X`
9. Poli, R., Langdon, W.B., McPhee, N.F.: A field guide to genetic programming. Published via `http://lulu.com` and freely available at `http://www.gp-field-guide.org.uk` (2008), `http://www.gp-field-guide.org.uk`, (With contributions by J. R. Koza)
10. Harman, M., Jones, B.F.: Search based software engineering. Information and Software Technology 43(14), 833–839 (Dec 2001), `http://dx.doi.org/10.1016/S0950-5849(01)00189-6`
11. Jhe-Yu Liou, Forrest, S., Carole-Jean Wu: Genetic improvement of GPU code. In: Petke, J., Shin Hwei Tan, Langdon, W.B., Weimer, W. (eds.) GI-2019, ICSE workshops proceedings. pp. 20–27. IEEE, Montreal (28 May 2019), `http://dx.doi.org/10.1109/GI.2019.00014`, Best Paper
12. Jhe-Yu Liou, Xiaodong Wang, Forrest, S., Carole-Jean Wu: GEVO: GPU code optimization using evolutionary computation. ACM Transactions on Architecture and Code Optimization 17(4), Article 33 (Dec 2020), `http://dx.doi.org/10.1145/3418055`
13. Schweim, D., et al.: Using knowledge of human-generated code to bias the search in program synthesis with grammatical evolution. In: Chicano, F., et al. (eds.) Proceedings of the 2021 Genetic and Evolutionary Computation Conference Companion. pp. 331–332. GECCO '21, Association for Computing Machinery, internet (Jul 10-14 2021), `http://dx.doi.org/10.1145/3449726.3459548`
14. Murphy, A., Laurent, T., Ventresque, A.: The case for grammatical evolution in test generation. In: Bruce, B.R., et al. (eds.) GI @ GECCO 2022. pp. 1946–1947. Association for Computing Machinery, Boston, USA (9 Jul 2022), `http://dx.doi.org/10.1145/3520304.3534042`
15. Griffin, D., Stepney, S., Vidamour, I.: DebugNS: Novelty search for finding bugs in simulators. In: Nowack, V., et al. (eds.) 12th International Workshop on Genetic Improvement @ICSE 2023. pp. 17–18. IEEE, Melbourne, Australia (20 May 2023), `http://dx.doi.org/10.1109/GI59320.2023.00012`
16. Yueke Zhang, Yu Huang: Leveraging fuzzy system to reduce uncertainty of decision making in software engineering automation. In: Bruce, B.R., et al. (eds.) GI @ GECCO 2022. pp. 1948–1949. Association for Computing Machinery, Boston, USA (9 Jul 2022), `http://dx.doi.org/10.1145/3520304.3533991`
17. Weimer, W.: From deep learning to human judgments: Lessons for genetic improvement. GI @ GECCO 2022 (9 Jul 2022), `http://geneticimprovementofsoftware.com/slides/gi2022gecco/weimer-keynote-gi-gecco-22.pdf`, Invited keynote

18. Sungmin Kang, Shin Yoo: Towards objective-tailored genetic improvement through large language models. In: Nowack, V., et al. (eds.) 12th International Workshop on Genetic Improvement @ICSE 2023. pp. 19–20. IEEE, Melbourne, Australia (20 May 2023), http://dx.doi.org/10.1109/GI59320.2023.00013, Best position paper
19. Krauss, O.: Exploring the use of natural language processing techniques for enhancing genetic improvement. In: Nowack, V., et al. (eds.) 12th International Workshop on Genetic Improvement @ICSE 2023. pp. 21–22. IEEE, Melbourne, Australia (20 May 2023), http://dx.doi.org/10.1109/GI59320.2023.00014
20. Brownlee, A.E.I., et al.: Enhancing genetic improvement mutations using large language models. In: Arcaini, P., Tao Yue, Fredericks, E. (eds.) SSBSE 2023: Challenge Track. LNCS, vol. 14415, pp. 153–159. Springer, San Francisco, USA (8 Dec 2023), http://dx.doi.org/10.1007/978-3-031-48796-5_13
21. Blot, A., Petke, J.: Empirical comparison of search heuristics for genetic improvement of software. IEEE Transactions on Evolutionary Computation 25(5), 1001–1011 (Oct 2021), http://dx.doi.org/10.1109/TEVC.2021.3070271
22. Blot, A., Petke, J.: Using genetic improvement to optimise optimisation algorithm implementations. In: Hadj-Hamou, K. (ed.) 23ème congrès annuel de la Société Française de Recherche Opérationnelle et d'Aide à la Décision, ROADEF'2022. INSA Lyon, France (23–25 Feb 2022), https://hal.archives-ouvertes.fr/hal-03595447
23. Langdon, W.B., Alexander, B.J.: Genetic improvement of OLC and H3 with Magpie. In: Nowack, V., et al. (eds.) 12th International Workshop on Genetic Improvement @ICSE 2023. pp. 9–16. IEEE, Melbourne, Australia (20 May 2023), http://dx.doi.org/10.1109/GI59320.2023.00011
24. Marginean, A., Barr, E.T., Harman, M., Yue Jia: Automated transplantation of call graph and layout features into Kate. In: Labiche, Y., Barros, M. (eds.) SSBSE. LNCS, vol. 9275, pp. 262–268. Springer, Bergamo, Italy (Sep 5-7 2015), http://dx.doi.org/10.1007/978-3-319-22183-0_21
25. Marginean, A.: Automated Software Transplantation. Ph.D. thesis, University College London, UK (8 Nov 2021), https://discovery.ucl.ac.uk/id/eprint/10137954/1/Marginean_10137954_thesis_redacted.pdf, ACM SIGEVO Award for the best dissertation of the year
26. Langdon, W.B., Harman, M.: Optimising existing software with genetic programming. IEEE Transactions on Evolutionary Computation 19(1), 118–135 (Feb 2015), http://dx.doi.org/10.1109/TEVC.2013.2281544
27. Blot, A., Petke, J.: Comparing genetic programming approaches for non-functional genetic improvement case study: Improvement of MiniSAT's running time. In: Ting Hu, Lourenco, N., Medvet, E. (eds.) EuroGP 2020: Proceedings of the 23rd European Conference on Genetic Programming. LNCS, vol. 12101, pp. 68–83. Springer Verlag, Seville, Spain (15-17 Apr 2020), http://dx.doi.org/10.1007/978-3-030-44094-7_5
28. de Almeida Farzat, F., de Oliveira Barros, M., Horta Travassos, G.: Challenges on applying genetic improvement in JavaScript using a high-performance computer. Journal of Software Engineering Research and Development 6(12) (Dec 2018), http://dx.doi.org/10.1186/s40411-018-0056-2, 20th Iberoamerican Conference on Software Engineering
29. Shuyue Stella Li, et al.: Genetic improvement in the Shackleton framework for optimizing LLVM pass sequences. In: Bruce, B.R., et al. (eds.) GI @ GECCO 2022. pp. 1938–1939. Association for Computing Machinery, Boston, USA (9 Jul 2022), http://dx.doi.org/10.1145/3520304.3534000, winner Best Presentation

30. Langdon, W.B., Al-Subaihin, A., Blot, A., Clark, D.: Genetic improvement of LLVM intermediate representation. In: Pappa, G., Giacobini, M., Vasicek, Z. (eds.) EuroGP 2023: Proceedings of the 26th European Conference on Genetic Programming. LNCS, vol. 13986, pp. 244–259. Springer Verlag, Brno, Czech Republic (12-14 Apr 2023), `http://dx.doi.org/10.1007/978-3-031-29573-7_16`
31. Bruce, B.R.: Automatically exploring computer system design spaces. In: Bruce, B.R., et al. (eds.) GI @ GECCO 2022. pp. 1926–1927. Association for Computing Machinery, Boston, USA (9 Jul 2022), `http://dx.doi.org/10.1145/3520304.3534021`
32. Kuepper, J., et al.: CryptOpt: Verified compilation with randomized program search for cryptographic primitives. In: Foster, N. (ed.) 44th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2023. p. article no. 158. Association for Computing Machinery, Orlando, Florida (17-21 Jun 2023), `http://dx.doi.org/10.1145/3591272`, Gold winner 2023 HUMIES, PLDI Distinguished Paper
33. Langdon, W.B., Harman, M.: Genetically improved CUDA C++ software. In: Nicolau, M., et al. (eds.) 17th European Conference on Genetic Programming. LNCS, vol. 8599, pp. 87–99. Springer, Granada, Spain (23-25 Apr 2014), `http://dx.doi.org/10.1007/978-3-662-44303-3_8`
34. Langdon, W.B., Modat, M., Petke, J., Harman, M.: Improving 3D medical image registration CUDA software with genetic programming. In: Igel, C., et al. (eds.) GECCO '14: Proceeding of the sixteenth annual conference on genetic and evolutionary computation conference. pp. 951–958. ACM, Vancouver, BC, Canada (12-15 Jul 2014), `http://dx.doi.org/10.1145/2576768.2598244`
35. Langdon, W.B., Harman, M.: Grow and graft a better CUDA pknotsRG for RNA pseudoknot free energy calculation. In: Langdon, W.B., Petke, J., White, D.R. (eds.) Genetic Improvement 2015 Workshop. pp. 805–810. ACM, Madrid (11-15 Jul 2015), `http://dx.doi.org/10.1145/2739482.2768418`
36. Langdon, W.B., et al.: Genetic improvement of GPU software. Genetic Programming and Evolvable Machines 18(1), 5–44 (Mar 2017), `http://dx.doi.org/10.1007/s10710-016-9273-9`
37. Klus, P., et al.: BarraCUDA - a fast short read sequence aligner using graphics processing units. BMC Research Notes 5(27) (2012), `http://dx.doi.org/10.1186/1756-0500-5-27`
38. Langdon, W.B., Brian Yee Hong Lam: Genetically improved BarraCUDA. BioData Mining 20(28) (2 Aug 2017), `http://dx.doi.org/10.1186/s13040-017-0149-1`
39. Langdon, W.B., Lorenz, R.: Evolving AVX512 parallel C code using GP. In: Sekanina, L., Ting Hu, Lourenco, N. (eds.) EuroGP 2019: Proceedings of the 22nd European Conference on Genetic Programming. LNCS, vol. 11451, pp. 245–261. Springer Verlag, Leipzig, Germany (24-26 Apr 2019), `http://dx.doi.org/10.1007/978-3-030-16670-0_16`
40. Lorenz, R., Bernhart, S.H., Höner zu Siederdissen, C., Tafer, H., Flamm, C., Stadler, P.F., Hofacker, I.L.: ViennaRNA package 2.0. Algorithms for Molecular Biology 6(1) (2011), `http://dx.doi.org/10.1186/1748-7188-6-26`
41. Andrews, R.J., et al.: A map of the SARS-CoV-2 RNA structurome. NAR Genomics and Bioinformatics 3(2), lqab043 (June 2021), `http://dx.doi.org/10.1093/nargab/lqab043`
42. Langdon, W.B., Petke, J., Blot, A., Clark, D.: Genetically improved software with fewer data caches misses. In: Silva, S., et al. (eds.) Proceedings of the Companion Conference on Genetic and Evolutionary Computation. pp. 799–802. GECCO '23,

Association for Computing Machinery, Lisbon, Portugal (15-19 Jul 2023), http://dx.doi.org/10.1145/3583133.3590542
43. Jimenez, D.A., Teran, E., Gratz, P.V.: Last-level cache insertion and promotion policy in the presence of aggressive prefetching. IEEE Computer Architecture Letters 22(1), 17–20 (Jan-June 2023), http://dx.doi.org/10.1109/LCA.2023.3242178
44. Klinkenberg, J., et al.: H2M: exploiting heterogeneous shared memory architectures. Future Generation Computer Systems 148, 39–55 (2023), http://dx.doi.org/10.1016/J.FUTURE.2023.05.019
45. Lucia Pons, et al.: Cloud White: Detecting and estimating QoS degradation of latency-critical workloads in the public cloud. Future Generation Computer Systems 138, 13–25 (Januray 2023), http://dx.doi.org/10.1016/J.FUTURE.2022.08.012
46. Tirthak Patel, Tiwari, D.: CLITE: Efficient and QoS-aware co-location of multiple latency-critical jobs for warehouse scale computers. In: 2020 IEEE International Symposium on High Performance Computer Architecture (HPCA). pp. 193–206 (2020), http://dx.doi.org/10.1109/HPCA47549.2020.00025
47. Langdon, W.B., Brian Yee Hong Lam, Petke, J., Harman, M.: Improving CUDA DNA analysis software with genetic programming. In: Silva, S., et al. (eds.) GECCO '15: Proceedings of the 2015 Annual Conference on Genetic and Evolutionary Computation. pp. 1063–1070. ACM, Madrid (11-15 Jul 2015), http://dx.doi.org/10.1145/2739480.2754652
48. Langdon, W.B., Petke, J., Lorenz, R.: Evolving better RNAfold structure prediction. In: Castelli, M., Sekanina, L., Mengjie Zhang (eds.) EuroGP 2018: Proceedings of the 21st European Conference on Genetic Programming. LNCS, vol. 10781, pp. 220–236. Springer Verlag, Parma, Italy (4-6 Apr 2018), http://dx.doi.org/10.1007/978-3-319-77553-1_14
49. Langdon, W.B., Krauss, O.: Genetic improvement of data for maths functions. ACM Transactions on Evolutionary Learning and Optimization 1(2), Article No.: 7 (Jul 2021), http://dx.doi.org/10.1145/3461016
50. Langdon, W.B., Clark, D.: Deep mutations have little impact. In: Gabin An, et al. (eds.) 13th International Workshop on Genetic Improvement @ICSE 2024. ACM, Lisbon (16 Apr 2024), forthcoming
51. Owens, J.D., et al.: GPU computing. Proceedings of the IEEE 96(5), 879–899 (May 2008), http://dx.doi.org/doi:10.1109/JPROC.2008.917757, Invited paper
52. Jouppi, N.P., et al.: TPU v4: an optically reconfigurable supercomputer for machine learning with hardware support for embeddings. In: Proceedings of the 50th Annual International Symposium on Computer Architecture, ISCA. p. article no 82. ACM, Orlando, FL, USA (2023), http://dx.doi.org/10.1145/3579371.3589350
53. Santiago, A., et al.: Analysis and deployment of applications acceleration environment for Xilinx hardware-accelerated platforms. In: 37th Conference on Design of Circuits and Integrated Circuits (DCIS). IEEE, Pamplona, Spain (16-18 November 2022), http://dx.doi.org/10.1109/DCIS55711.2022.9970101
54. Silver, D., et al.: Mastering the game of Go without human knowledge. Nature 550(7676), 354–359 (2017), http://dx.doi.org/10.1038/nature24270
55. Adamek, K., Dimoudi, S., Giles, M., Armour, W.: GPU fast convolution via the overlap-and-save method in shared memory. ACM Transactions on Architecture and Code Optimization 17(3), article no 18 (aug 2020), http://dx.doi.org/10.1145/3394116

56. Robinson, T., Harkin, J., Shukla, P.: Hardware acceleration of genomics data analysis: challenges and opportunities. Bioinformatics 37(13), 1785–1795 (2021), `http://dx.doi.org/10.1093/bioinformatics/btab017`
57. Berger, M.: Compilers and computer architecture: Caches and caching. G5035, BSc/MComp Computer Science, University of Sussex (December 2019), `https://users.sussex.ac.uk/~mfb21/compilers/slides/15-handout.pdf`, Accessed November 2023
58. Yung-Chia Lin, Jenq-Kuen Lee, Bodin, F.: Guest editorial: Special issue on embedded multicore applications and optimization. Journal of Signal Processing Systems 91(3-4), 217–218 (2019), `http://dx.doi.org/10.1007/S11265-018-1431-2`