Black Box Approach for Selecting Optimization Options Using Budget-Limited Genetic Algorithms

Guy Bashkansky and Yaakov Yaari

{guy, yaari}@il.ibm.com
IBM Haifa Research Lab

Abstract. Modern compilers present a large number of optimization options covering the many alternatives to achieving high performance for different kinds of applications and workloads. Selecting the optimal set of optimization options for a given application and workload becomes a real issue since optimization options do not necessarily improve performance when combined with other options. The ESTO framework described here searches the option set space using various types of genetic algorithms, ultimately determining the option set that maximizes the performance of the given application and workload. ESTO regards the compiler as a black box, specified by its external-visible optimization options. For the IBM XLC compiler, with some 60 optimization options, we achieved +13% gain over an aggresive base, using 60 iterations on average. We studied a number of search policies given a fixed iterations budget, and showed that exponentially decreasing the width of the search beam gives the best results.

1 Introduction

In modern compilers and optimizers, the set of possible optimizations is usually very large. In general, for a given application and workload, optimization options do not accrue toward ultimate performance. To avoid selection complexity, users tend to use standard combination options, such as -O, which provide stable, high performance for the "average" application. In general, however, these standard options are not the optimal set for a specific application executing its representative workload.

Genetic algorithms (GA) [17, 3] present an attractive solution to this problem of selecting an optimal set of options. The problem is easily mapped to the original problem of gene optimization. The extended time required to reach to a preferred solution is justified by the much longer life of the optimized program. Past related works (e.g. [12], [15]) dealt with tuning specific compilation heuristics. The Expert System for Tuning Optimization (ESTO) was developed to study this GA solution to the general compiler options optimization. The program first computes an initial result using the best-known optimization set, e.g., -O3, and forms an initial generation; randomly, or using some initial knowledge.

Then, at each iteration, the group of organisms (i.e., option sets) that comprise the generation is evaluated on the input workload. The results are sorted and pass through a breeding and mutation stage to form the next generation. This process continues until a termination condition is reached, where the generation results show some kind of stability (by themselves, and/or with respect to previous generations).

Since the option optimization problem exists regardless of the compiler, operating system, or application, we designed ESTO to be fully configurable along all these axes, so that a given compiler can be approached as a black box using solely its user-visible optimization options. Using this reconfigurability, we held studies on two compilers, GCC, and XLC, as well as on the post-link optimizer FDPR-Pro [10]. Each of these optimizers has between 40 to 60 optimization options, some of which have additional parameters. Measuring the SPEC2000/INT suite, we achieved an average gain of +22% over -O1 for GCC, +13% over -O3 for XLC, and +6% over -O3 for FDPR-Pro. The configurability also allows testing various search policies and parallel execution modes. These are described in the body of the paper.

The paper's main contribution is in the application of GA for the problem of selecting an optimal option set for a specific application and workload. Secondly, an adaptive GA is proposed to improve search potential in multi-optima spaces. Finally, we propose a number of decreasing search beam policies to meet a fixed iterations budget.

The paper is organized as follows. Section 2 discusses related work. Section 3 is the core of the paper, presenting the main problem of option selection, and the details of ESTO's genetic algorithm with its different policies. Section 4 explains the configurability aspects of the framework. Experimental results are examined in Sect. 5, and Sect. 6 concludes the paper.

2 Related work

The general approach for tuning compiler optimization for a given application and workload, is by iterative compilation, measuring the performance and using it to direct successive iterations. Some researchers propose tuning specific compiler functions or their order. Stephenson et al. [15] use genetic programming to optimize the specific heuristic associated with compiler optimization work. Cooper et al. [5] use a genetic algorithm to find the preferred order of optimization phases that generates smaller code in an embedded system environment. Kulkarni et al. [11] describe a solution for making an exhaustive search of the optimization phase order space. Bodin et al. propose a special iterative compilation system [4], which efficiently explores a large transformation space consisting of small number of optimization options (3), where each option is associated with a numerical parameter. Instead of exploring available optimization options, Frank et al., [6], suggests to have the iterative stochastic search explore the space of source-level transformation, thereby overcome the limitations imposed by fixed set of optimization options.

The large number of evaluations inherent in the iterative approaches is addressed by Agakov et al.[2]. Using trained predictive models their system is able to reduce number of iterations to as few as two. Fursin et al. [7] address this problem by exploiting stable program phases to test different versions of functions, instead of dedicating full runs for that.

For users of traditional compilation systems such as GCC, who have no control over phase order nor any knowledge of compiler internals, the above works do not provide a relevant solution. In this context, the only control the user has is over the set of optimization options. Not many works are available here, apart from the commercial applications [1], and PathOpt tuning tool of EKOPath [13]. Pinkers et al. [14] use orthogonal arrays (OA) to iteratively trim down the number of actual optimization options used. Though the number of iterations is small, the number of total compilations can be quite large. Nisbet [12] uses a genetic algorithm to select loop restructuring transformations in Fortran programs. This work come closest to our work with its approach to selection of optimization options. There are many differences, however, in the details of the algorithms, as discussed in Sect. 3.

3 Iterative Optimization and Tuning

3.1 The Problem

In modern compilers and optimizers, the set of possible optimizations is usually very large. Some of the options require an additional parameter, which makes the selection process even more complex. Selecting the optimal set for a given application is complex because optimization options do not necessarily add to performance when combined with other options, and/or when used for certain application and a certain workload. As a result of this complexity, users tend to use standard combination options, such as -O, -O2, and -O3. These combinations select a subset of the optimization options that are available to provide stable, high performance for a large number of applications. Thus, in general, these options are not the optimal set for a specific application executing its representative workload.

Table 1. Estimated Amounts of Optimization Options

Optimizer	Binary	Parameterized	Effective
FDPR-Pro	22	12	58
GCC	55	5	70
XLC	52	1 (see note)	55

Table 1 gives an estimate for the number of options in different optimizers. The exact assessment for GCC and especially for XLC is difficult, because some

options have complex multilevel syntaxes and dependencies (Note: for that reason only one one parametrized option was configured for XLC). Generally, we can see that the problem is highly multidimensional, with many binary and multi-value/continuous parameter dimensions. To estimate the effective search space, we assume parameterized options have eight discrete possible values (which is conservative, see typical cases in [4]). Now the total number of option combinations becomes $2^B \cdot 8^P = 2^{B+3P} = 2^E$, where B and P are the number of binary and parameterized options respectively, and $E = B + 3P$ the number of effective binary options. Study of such problems [4] shows that the performance landscape in this highly multidimensional space is nonlinear with many local optima. The ability of genetic algorithms to search efficiently in such an irregular multi-dimensional space makes them a preferred choice.

3.2 Genetic Algorithm

Algorithm 1 ESTO specific GA implementation

```
configure optimizer options, GA parameters and init organism
create population of randomized organisms and one init
evaluate fitness of init organism
for generation = 1 to generation.limit do
  evaluate fitness of all organisms
  sort all organisms by fitness
  print all organisms with gain relative to init
  if generation.best.result has improved then
    no.improvement.counter ⇐ 0
  else
    no.improvement.counter ⇐ no.improvement.counter + 1
    if no.improvement.counter = no.improvement.limit then
      terminate
    end if
  end if
  if variable.population.size then
    reduce population.size
    if population.size = 1 then
      terminate
    end if
  end if
  replace lower population half by upper population half {natural selection}
  randomly blend 1st quarter organisms into 4th, and 2nd into 3rd {crossover}
  randomly mutate lower 3/4 of population with option.mutation.rate
  if adaptive.policy then
    fully re-randomize all organisms worse than init
  end if
end for
```

Genetic Algorithm (GA) [17, 3] is a search technique inspired by evolutionary biology concepts such as inheritance, mutation, selection, and crossover (recombination). Optimization parameters are encoded as *genes* of separate *individuals*. Each individual's combination of parameters represents a possible candidate solution to the optimization problem. A *population* of individuals evolves during multiple *generations* toward better solutions. The evolution usually starts with a population of randomly generated individuals. In each generation, the fitness of every individual in the population is evaluated, multiple individuals are selected based on their fitness, then mutated and recombined to form a new population. This process repeats until a termination condition is reached.

In our specific GA implementation, each gene represents an optimizer *option*, either *binary* (on/off), or with a parameter in a certain *range* of type *int, float, power2*, and *enum*. These options are assembled into *option sets* that correspond to the optimizer invocation arguments, typically specified in a single command line. Each option set is encapsulated in an *organism*, i.e., a GA individual. A number of organisms constitute a population, which evolves for a limited number of generations (see Algorithm 1). The evolution stops when there is no improvement of the best result for a few generations, or when the *generation limit* is exhausted, or when the population is reduced to a single organism (see Sect. 3.4 on population size alterations).

The crossover of genes and the natural selection are done by cross-breeding corresponding organisms from the upper two quarters of the population, and replacing the lower two quarters with the resulting "children". (*Cross-breeding* means that each option is separately inherited from either parent, with a given *crossover rate* probability for the lower parent. For an option with a parameter, its value is randomly dealt between the parents' values.) Then, the upper quarter is left untouched (*elitist selection*), and the lower three quarters are mutated with a given option-wide *mutation rate*, i.e., mutation probability for each option. (*Mutation* means dealing a new value to the option, irrespective of the old value.)

In our implementation, the initial random population is complemented by a pre-configured "initial" individual, which can reflect either the composition of a known-good combination like -O3, or a "zero" option set, or a previous search result, or some arbitrary user choice. This *init* organism serves as a kind of pivot for further comparisons and intermediate gain computations.

3.3 Efficient Use of Search Budget

As shown in Sect. 5, GA experimental results are quite good, but they are achieved by large number of optimization + measurement cycles, typically above one hundred per application. This could be a time-consuming investment of resources, especially with large real-life applications whose representative workloads might run for a long time. It is therefore important to maximize the efficiency of the GA search by using the given fixed search budget to reach the best possible performance gain (see [9]).

3.4 Search Beam Width

One way to use a given test budget more efficiently is by shaping GA population size as a function of the generation number. Basic GA keeps the number of individuals constant throughout the whole run. Our intuition is that extending the random coverage in the beginning, and focusing on convergence in the end, improves the cost/performance efficiency. To this extent, we can use the term *search beam width*, analogous to the typical searchlight operation, where in the beginning we use a wide searchlight beam to choose among many potential target areas, and then narrow it to focus on a specific promising area. The implemented searchlight width policies (i.e., population size alterations) appear in Fig. 1 and are described below. Note that the areas below each policy graph (i.e., total amounts of tests) are roughly same, corresponding to the *budget limit* mentioned in the title of this paper.

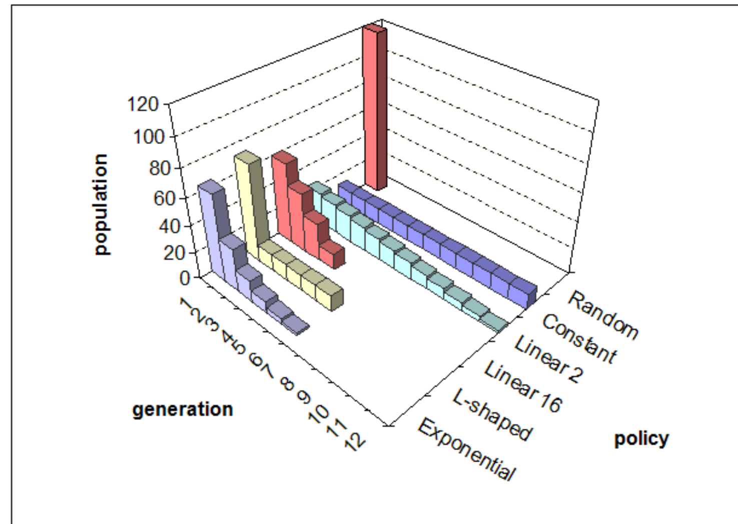

Fig. 1. Search beam width policies

Random – Pure random search, a single generation of 126 organisms + *init*. A trivial algorithm which serves as a sanity check for the proposed algorithms.

Constant – Like the above GA implementation with 12 generations of 12 organisms (total 144 tests), but without the '"reduce $population - size$" step.

Adaptive – Like *Constant*, but fully randomizes any organism with fitness below *init*. We regard this as search beam width policy, because "underperformers" from any generation are re-randomized, which likens them to individuals in the first generation and thus effectively reshapes the population size over generations. This technique was proved advantageous for the result gain and is used in the following methods.

Linear 2, 16 – Like *Adaptive*, but with linearly decreasing number of organisms in generations, either by 2: 24, 22, ... 2 (total 156), or by 16: 60, 44, 28, 12 (total 144). The dependence between the population size N and the generation number g is:

$$N(g+1) = N(g) - gradient \tag{1}$$

L-shaped – Like *Adaptive*, but with 72 organisms in the first generation and 12 organisms in the remaining 5 generations (total 132 tests).

Exponential – Like *Adaptive*, but with 6 generations containing exponentially decreasing numbers of organisms: 64, 32 ... 2 (total 126 tests). The dependence between the population size N and the generation number g is:

$$N(g+1) = N(g)/2 \tag{2}$$

Note that this can be expressed in absolute (rather than iterative) terms, using the generation limit L:

$$N(g) = 2^{L-g+1} \tag{3}$$

Our hypothesis is that the *Exponential* policy is the most efficient one. It should have the best cost/performance ratio and thus utilize the budget limit in the best possible way. The rational behind this hypothesis is the following: Each passing generation decreases the *uncertainty* that we indeed found the optimum. It seems plausible that this decrease is roughly the same as that of the previous generation, so that the uncertainty decreases exponentially as function of the generation number. Thus, we can reduce the population size exponentially as well, without harming the above uncertainty decrease rate, akin to the per-generation convergence rate.

4 Configurability

ESTO can be used for a wide variety of optimization tools. It has already been applied successfully to GCC and XLC, as well as to FDPR-Pro, IBM feedback-directed post-link optimization tool [10]. It was adapted to Linux, AIX, and various embedded board setups, as well as to a number of benchmark configurations: direct application invocation, SPEC2000, and UMT2K [16].

Much of ESTO flexibility is due to its configuration file. The file consists of two sections: algorithm control and option specification (see Sects. 4.1 and 4.2). The latter section enables ESTO to regard the optimizing compiler as a black box, controlled solely through its user-visible optimization options.

4.1 Search Algorithm

The concrete *search algorithm modification* to use is specified in a configuration file line, and can be any of those described in Sect. 3.4, plus a few special investigative modifications. The algorithm can be configured with its own *arguments*, like *L-shaped* sizes, *Linear* gradients, and *Exponential* bases, and also these common GA parameters (defaults in parentheses): *mutation rate* (0.01), *crossing rate* (0.5), *initial population size* (12), *generation limit* (12), and *no improvement limit* (4).

4.2 Application

By the term *application*, we mean the whole range from the *option set* to the *reported time*, typically including the invocation of an *optimizer* on some *target program*, and measuring the resulting performance in an environment-specific fashion.

Optimizer Options – This configuration section specifies the set of options from which the optimal set should be selected. Each option line specifies its *name*, whether switched on in the *init* organism, and the *probability* of switching on upon mutation (e.g., can be set to the option occurrence rate in the historic ESTO results, to speed up convergence).

Two kinds of options can be specified: *binary* and *range* (parameterized). The latter, in turn, has a few types: *int, float, power2*, and *enum*. Each type should specify *low* and *high* boundaries of parameter value variations, an *initial* parameter value, and a parameter *step*, i.e., the natural or artificial parameter value alignment.

Application and Workload – The behavior of the default application and its workload is controlled by a few default scripts that can be provided by the user according to defined command interfaces. Alternatively, the user can extend and re-implement the application class interface.

Most importantly, the *application script* runs the user application with its required parameters, including an *option set* whose fitness has to be measured, and *transit arguments*, passed directly and transparently from ESTO command line invocation. There are a couple of other default command interfaces for error handling and synchronization.

As it runs, ESTO prints to standard output the performance results of each organism, and the sorted summary of each generation. Finally, it outputs the result line, which lists the chosen option set, its reported time, and the gain percentage.

4.3 Multiplexor

The *multiplexor* interface implements scattering of the *application* trials, gathering of their results, and synchronization as requested by the *algorithm*. ESTO supports single-threaded mode and a parent/workers style multiprocessing (default). Other multitasking implementations are possible: multithreading, grid or blade architecture, clusters, etc. The interface is also responsible for fault tolerance: identifying, getting rid of, and recovering from failing runs. The task is particularly important here because arbitrary selecting of options may bring compilers to untested regions and possible failures.

5 Experimental Results

5.1 Metric

Typically, maximizing an application's performance means either reducing its running time on a predefined workload, or performing more operations (processing larger workload) during a given time. We use running times, so our fitness metric is descending – the shorter is the running time, the better is the performance.

All of our experiments were done with SPEC2000 benchmark suite (*train* workloads) on Linux machines with Power architecture. ESTO invokes the user application script as mentioned in Sect. 4.2. That script typically invokes `runspec` as part of its operation, and then extracts results from the `reported_time` fields of the raw result files. These results are then piped back to ESTO for comparison inside GA. Detected failures are considered as `FLT_MAX`.

5.2 Comparison of Results for Different Optimizers

Although relatively inefficient, the *Constant* policy allows us to compare ESTO achievements for different optimizers relative to various starting points. This is due to the abundance of available historic results on machines of same type: 4-processor 1.5GHz Power5 running SUSE Linux Enterprise Server 9 operating system. The GA parameters were also identical: mutation rate 0.02, crossover rate 0.45, population size 12, generation limit 12, no-improvement limit 4.

GCC – ESTO-on-GCC setup includes GCC-specific SPEC2000 configuration and GCC optimization options configuration for ESTO (see Sect. 4.2). The *init* seed organism corresponds to the composition of -O1 combination. On most benchmarks ESTO obtains high gains over -O1 starting point. Even relative to -O3 these are reasonably good gains in half of the cases. If we would start from -O3 seed, ESTO results would not go below the -O3 result. See Fig. 2.

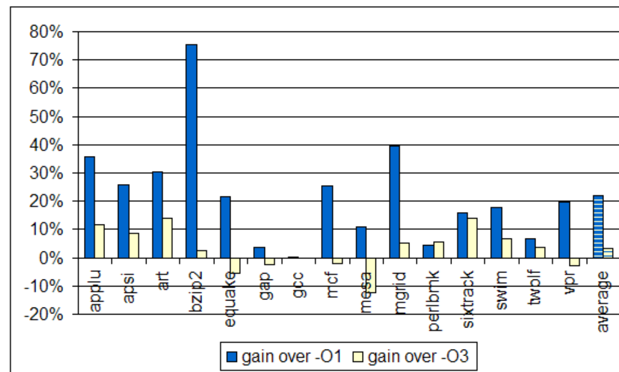

Fig. 2. ESTO gain over GCC -O1 and -O3

XLC – ESTO-on-XLC setup includes XLC-specific SPEC2000 configuration and XLC optimization options configuration for ESTO. The *init* seed organism has all options switched off (*zero* organism). On most benchmarks ESTO obtains high gains over -O3, with average of 13%. We compared also to the *peak* option set results reported[1] to SPEC for XLC on Power5 with SLES9. Most of these reports include XLC profile-driven feedback facility, so we extended ESTO application setup to run profiling phases, which prolongs optimization times significantly. As can be seen in Fig. 3, in some cases ESTO gains even over these, certifiably the best reported, results.

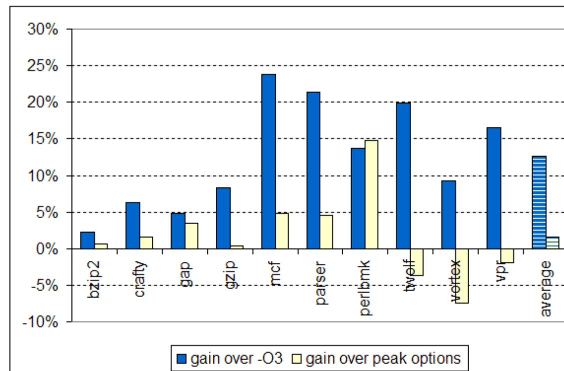

Fig. 3. ESTO gain over XLC -O3 and over peak option sets per benchmark

[1] www.spec.org/osg/cpu2000/results/res2004q4/cpu2000-20041018-03448.html

FDPR-Pro – ESTO-on-FDPR-Pro setup includes FDPR-Pro-specific SPEC2000 configuration with feedback-directed optimization stages and FDPR-Pro optimization options configuration for ESTO, which includes relatively large number of parameterized options. The *init* seed organism corresponds to the composition of -O3 combination. On most benchmarks ESTO obtains high gains over -O3 starting point, see Fig. 4. One interesting question in SPEC context is whether the option sets found by ESTO on the *train* workload can benefit the *ref* workload (the formal reference workload of SPEC) . In Fig. 4 the left set of bars contains ESTO gains over FDPR-Pro -O3 on *train* workload, and the right set depicts *ref* gains over -O3 with the option set found on *train*. In most cases there seems to be a significant correlation between the two, so *train* could be used as a predictor for *ref*. The practical significance of this finding stems from the fact that *train* runs some order of magnitude faster than *ref*. For the *ref* workload the above hundred tests, required by ESTO's, might be prohibitively slow, but can complete in reasonable time on *train*.

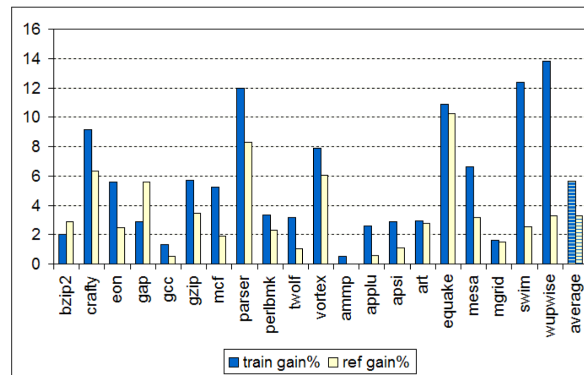

Fig. 4. ESTO gain over FDPR-Pro -O3 found on *train* workload, and *ref* workload gain with same option set

Figure 5 depicts the typical course of ESTO run on an example: *mesa train*. The upper line shows how the best gain in generation approximates the final result. The vertical range bars denote the distribution range of the upper half of that generation's individuals, in line with our GA natural selection implementation. We clearly see the reverse-exponential decrease of both the result approximation and the distribution range. This provides visual experimental support to our hypothesis that the uncertainty of the result decreases exponentially, with a constant ratio (convergence rate) between two successive generations.

Convergence Comparison – To assess the rate of convergence, we use the following convergence criterion: Two thirds of the population reached gain within 5% of their mean. Using this criterion on the accumulated ESTO logs of the

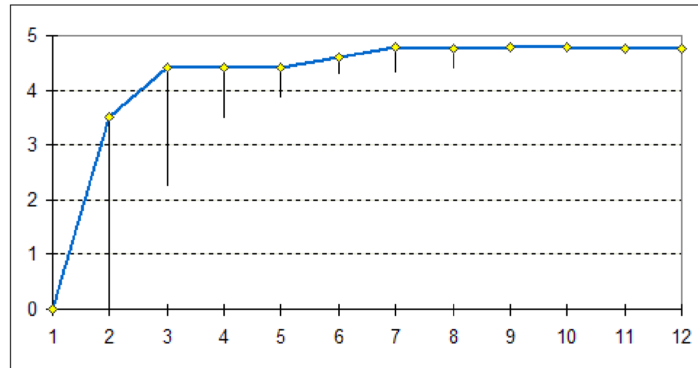

Fig. 5. ESTO best gain % and distribution range in each generation

above experiments, we measured the following average number of test iterations required for optimizers to converge:

- GCC – 119
- XLC – 59
- FDPR-Pro – 94

GCC's comparatively slow rate can be attributed to its large amount of options (see Table 1) with performance benefits more evenly spread between option combinations (wide distribution). XLC's fast convergence seems to be caused by a few very dominant option combinations which greatly affect performance, while others barely matter (narrow distribution). Thus the *effective* amount of options is quite smaller than Table 1 suggests. FDPR-Pro is somewhere in the middle in both senses – convergence rate and options benefits distribution.

5.3 Comparison of Population Size Policies

ESTO GA policy comparative study was conducted on a single-core Power970 2.2GHz blade machine with SUSE Linux operating system. The study compares the policies discussed in Sec. 3.4: *Random*, *Constant*, *Adaptive*, *Linear 2/16*, *L-shaped*, and *Exponential*. Fig. 6 shows the results of this study: ESTO *total* and *per-iteration* gains over FDPR-Pro -O3 when using different policies.

Total Gain – Represented by wide bars in Fig. 6.

There is a big jump from *Constant* to *Adaptive*, probably due to a better search space coverage created by re-randomizing of many "underperformers". Both *Linear* policies seem to interfere with this consideration by artificially reducing that coverage. *L-shaped* and *Exponential* are obviously less affected, perhaps because both rapidly get rid of those "underperformers" anyway. We see that *L-shaped* and *Exponential* policies reach maximum total performance gain.

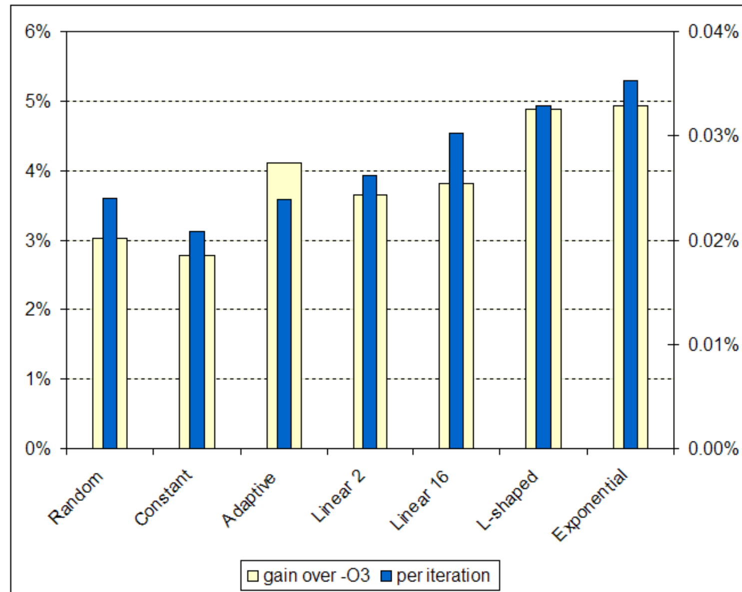

Fig. 6. ESTO total and per-iteration gain over FDPR-Pro -O3 with different policies

The superiority of *Random* over *Constant* can be explained by *Constant*'s apparent "wasting" of the budget. In each generation, *Constant*'s iterations above *Exponential* (with same size of generation 1) – are redundant according to our hypothesis. *Random*, on the other hand, does not "waste" budget, it just does not "complete" the exponential tail befittingly with its only generation 1. See also [8].

Gain per Iteration – Represented by narrow bars in Fig. 6.

Notably, we see the gain rate steadily rising as the policy shape approaches the decreasing exponential curve. This algorithm efficiency indicator culminates when using the *Exponential* policy, as suggested by our hypothesis in Sect. 3.4.

5.4 Parallelization Study

Reduction of ESTO running time is an important technical challenge, because it improves response time and makes more efficient use of resources. ESTO total processing task for each SPEC2000 benchmark consists of multiple iterations of FDPR-Pro optimization and *train* workload measurement. The total duration of sequential ESTO runs on all benchmarks in the suite reached some 5-6 days on a 1.5GHz Power5 machine. Durations of the iterations' stages vary widely between benchmarks, from 16 seconds optimization for *parser* and 3 seconds measurement for *gcc*, to as much as 12 minutes optimization for *perlbmk* and 1.5 minute measurement for *ammp*. Overall, when *train* workload is used for

measurements (25 seconds on average), the optimization stages consume 3/4 of the iteration time (1.5 minutes on average). Of course, *ref* workload would have reversed this situation.

Superficially, we may want to evaluate the organisms of a generation in parallel. An important constraint here is that the measurement stages generally should not run in parallel on the same machine, because of the shared L2/L3 cache. The approach taken is then to perform the optimization stages of a generation in parallel, while the measurement stages are done serially. This allowed to reduce the above running time by half, using the machine's 4 cores (2 HW threads/core, i.e. 8 "logical" CPU).

We then studied the effect on measurement accuracy when this phase is done in parallel. The machine used was 1.5GHz Power5 system with 8 cores (2 HW threads/core, i.e. 16 "logical" CPU). The assumption was that for single-threaded medium-size applications, like the SPEC2000 suite, with one process per core (to avoid interference between HW threads of the same core), measurements will not interfere with each other. This was verified as shown in Fig. 7. Up to N parallel measurements on an N-core machine ($N = 8$) practically do not interfere, and above N there is a linear degradation with gradient about $1/2N$ per process addition.

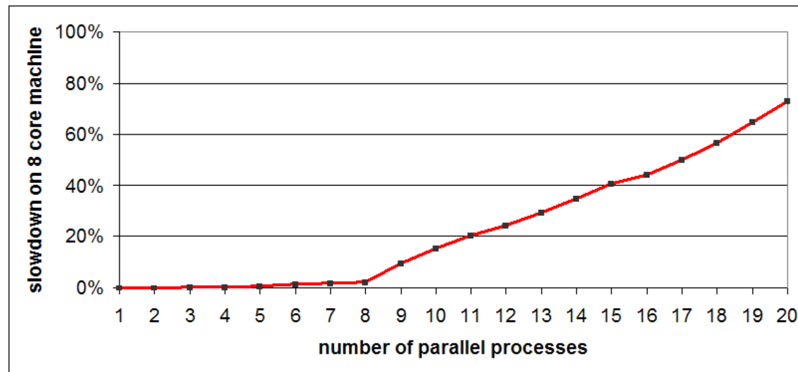

Fig. 7. Slowdown of SPEC2000 measurements (median) when parallel processes run the same benchmark

In our case, running ESTO on all benchmarks using 6 cores (12 logical CPU) out of the available 8 (16), while parallelizing *both* optimizations and measurements – completed in 16 hours only. In practical terms, this means that instead of waiting for the ESTO result for a working week, one can get the result overnight, thus greatly increasing the tool's usability.

Such technique can be used for single-threaded applications only. Parallelizing the tuning of multitasking applications calls for distribution of GA organisms between different machines.

6 Conclusions and Future Work

We show the feasibility of using the genetic algorithm for selecting preferred optimization options for a variety of compilers and workload, dealing with more then more then 60 effective options, and converging to the preferred result in around 100 iterations, depending on the option space. While we do not have formal proof, the stability of the results, when starting from an arbitrary organism, show that the preferred results are close to the optimum. This is comparable or better to the other alternatives discussed in this paper. The convergence speed, in gain/iteration, is a critical factor in iterative compilation. We studied a number of search beam policies and showed that an exponentially decreasing beam gives the best result.

The database of past results associates a specific workload to recommended option sets. We plan to exploit this database by using automatic feature selection and clustering to acquire quick prediction of the recommended option set for a given workload, or a recommended starting point for the GA search.

References

1. ACOVEA. Available at http://www.coyotegulch.com/products/acovea/index.html, 2006.
2. F. Agakov, E. Bonilla, J. Cavazos, B. Franke, G. Fursin, M.F.P. O'Boyle, J. Thomson, M. Toussaint, and C.K.I. Williams. Using machine learning to focus iterative optimization. In *Proceedings of the 4thAnnual International Symposium on Code Generation and Optimization (CGO)*, March 2006.
3. E.B. Baum, D. Boneh, and C. Garrett. Where genetic algorithms excel. *Evolutionary Computation*, 9(1):93–124, 2001.
4. F. Bodin, T. Kisuk, P. M. W. Knijnenburg, M. F. P. O'Boyle, and E. Rohou. Iterative compilation in a non-linear optimisation space. In *Proceedings of the Workshop on Profile and Feedback-Directed Compilation, in Conjunction with the International Conference on Parallel Architectures and Compilation Techniques (PACT '98)*, October 1998.
5. Keith D. Cooper, Philip J. Schielke, and Devika Subramanian. Optimizing for reduced code space using genetic algorithms. In *Proceedings of the ACM SIGPLAN 1999 Workshop on Languages, Compilers, and Tools for Embedded Systems (LCTES '99)*, pages 1–9, New York, NY, USA, 1999. ACM Press.
6. B. Franke, M. O'Boyle, J. Thomson, and G. Fursin. Probabilistic source-level optimisation of embedded systems software. In *Proceedings of the Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES'05)*, pages 78–86, June 2005.
7. Grigori Fursin, Albert Cohen, Michael O'Boyle, and Oliver Temam. A practical method for quickly evaluating program optimizations. In *Proceedings of the 1st International Conference on High Performance Embedded Architectures and Compilers (HiPEAC 2005)*, pages 29–46, November 2005.
8. G.G.Fursin, M.F.P.O'Boyle, and P.M.W. Knijnenburg. Evaluating iterative compilation. In *Proceedings of the 15th Workshop on Languages and Compilers for Parallel Computing (LCPC'02)*, 2002.

9. A. Grajdeanu and K. A. De Jong. Fixed budget allocation strategy for noisy fitness landscapes. In *Late Breaking Papers of the Genetic and Evolutionary Computation Conference (GECCO-2003)*, 2003.
10. G. Haber, E. A. Henis, and V. Eisenberg. Reliable post-link optimizations based on partial information. In *Proceedings of Feedback Directed and Dynamic Optimizations 3 Workshop*, pages 91 – 100, December 2000.
11. A. P. Kulkarni, D. B. Whalley, G. S. Tyson, and J. W. Davidson. Exhaustive optimization phase order space exploration. In *Proceedings of the International Symposium on Code Generation and Optimization*, 2006.
12. A. Nisbet. Gaps: Genetic algorithm optimised parallelisation. In *Proceedings of the 7th Workshop on Compilers for Parallel Computing*, 1998.
13. PathOpt. http://www.pathscale.com/pdf/PathOpt2-paper.pdf.
14. R. P. J. Pinkers, P. M. W. Knijnenburg, M. Haneda, and H. A. G. Wijshoff. Statistical selection of compiler options. In *Proceedings of the The IEEE Computer Society's 12th Annual International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunications Systems (MASCOTS '04)*, pages 494–501, Washington, DC, USA, 2004. IEEE Computer Society.
15. M. Stephenson, U. O'Reilly, M. Martin, and S. Amarasinghe. Genetic programming applied to compiler heuristic optimisation. In *Proceedings of The 6th European Conference on Genetic Programming*, 2003.
16. The UMT benchmark code. Available at http://www.llnl.gov/asci/platforms/purple/rfp/benchmarks/limited/umt/, 2002.
17. D. Whitley. A genetic algorithm tutorial. *Statistics and Computing*, 4(2):65–85, 6 1994.