SAND2016-2954R

Hyper-Heuristics to Automatically Target Code to Computer Architectures

Jesse Hughes

Faculty Advisor
Dr. Daniel Tauritz, Department of Computer Science

Natural Computation Laboratory

March 25, 2016

Sandia National Laboratories is a multi-program laboratory managed and operated by Sandia Corporation, a wholly owned subsidiary of Lockheed Martin Corporation, for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-AC04-94AL85000.

Abstract

With each new generation of High Performance Computing (HPC) architecture, the gap between peak theoretical performance and the observed performance is growing. The goal of this research is to develop a tool to utilize hyper-heuristics to target code to a computational environment. In order to test this, sorting algorithms will be evolved on several different architectures. The final solutions will then migrate to all other architectures and their fitnesses compared. If the natively-evolved algorithms out-perform all others, then it can be concluded that the tool successfully targeted its solutions to the architecture of origin. This is the first step towards creating a program-agnostic tool for optimizing code to the native environment. The results are pending testing on the high performance cluster. If it can be shown that the tool is able to optimize solutions for the environment, then the door opens to automatically optimizing entire programs.

1 Introduction

Computing architectures are constantly evolving but software is decreasingly being optimized for the nuances of these shifting environments. These evolving environments have led to an increasingly large gap between the theoretical peak performance and the actual observed performance. Software functionality is the primary focus of programmers across the board, as it should be: Non-functional properties - power consumption, execution time, and memory efficiency - are secondary goals and often difficult to integrate into the development cycle. This is a major failing of the modern software development process: it would be counter-productive to force programmers to spend time on non-functionality when the time demands are already so high for purely functional programming.

Furthermore, the layers of abstraction created by compilers, interpreters, and target platform result in an exceedingly complicated system. Small changes on the source code propagate downwardly on compilation and lead to starkly different behaviors at execution time [1]. Thusly, there is an apparent need for automated code optimization and targeting. The goal of this research is to develop and test a tool with hyper-heuristics for automatically optimizing code as a first step towards the automated optimization of entire programs. By evolving within the environment to be targeted and using the efficiency metric – run time and cache efficiency – as the fitness scale, the evolutionary process will be guided by the nuances of that machine.

While code optimization is not a new concept in the field of Evolutionary Computing, targeting code to an architecture is. This research is performed with the intent of developing a methodology for exploiting memory hierarchies and other nuances of an HPC architecture. Once this has been established as feasible, the door will open to optimizing whole programs and services.

In order to achieve generality, the class of problem to be solved is the Search algorithm. These algorithms are generally memory-intensive and there exists a large amount of space to explore

for optimization. The primitives will start as python “opcodes” which are abstractions of lower-level assembly functions. If these fail to have the resolution necessary for architecture exploitation, then the primitives will gradually shift layers of abstraction lower until the necessary resolution is captured. These macro functions will be an assortment of assembly-like functions in order to restrict the search space.

Modern compilers and interpreters are built upon layers of abstraction that have given modern languages the advanced functionality and simplicity programmers have come to expect. With each additional coupling of functions into macro operations some generalization was introduced. The sum of all these generalities has led to a wide arsenal of macro functions to choose from, but has made writing code while targeting specific hardware impractical.

To counter this trend, this hyper-heuristic was developed and tested to explore its potential for use in architecture-targeted code generation. This would be the first step towards automating the optimization of entire modules and programs. This would have a significant impact in almost any area of computation. It would allow for some de-coupling of functional and non-functional programming paradigms thus enabling developers to more efficiently address the functionality of their code without forfeiting efficient resource utilization.

2 Implemented Concepts

To create this optimization tool, several different concepts have been implemented from the field of Evolutionary Computing (EC).

2.1 Evolutionary Algorithm

An evolutionary algorithm (EA) is essentially a way of guiding a population of individuals within some environment that has limited resources. Competition for these resources creates an environment of natural selection in which the fittest individuals are more likely to contribute to the next generation [2].

The average life cycle of an EA can be seen in Figure 1. The EA initializes an initial population of individuals. These individuals are evaluated and the parents are chosen. Then, the EA performs recombination and mutation to generate the child population. These are combined with the general population and from this the survivors for the next generation are chosen. This cycle continues until the termination criteria is met (often when an individual of the population has reached peak fitness or the set max number of evaluations has been exceeded).

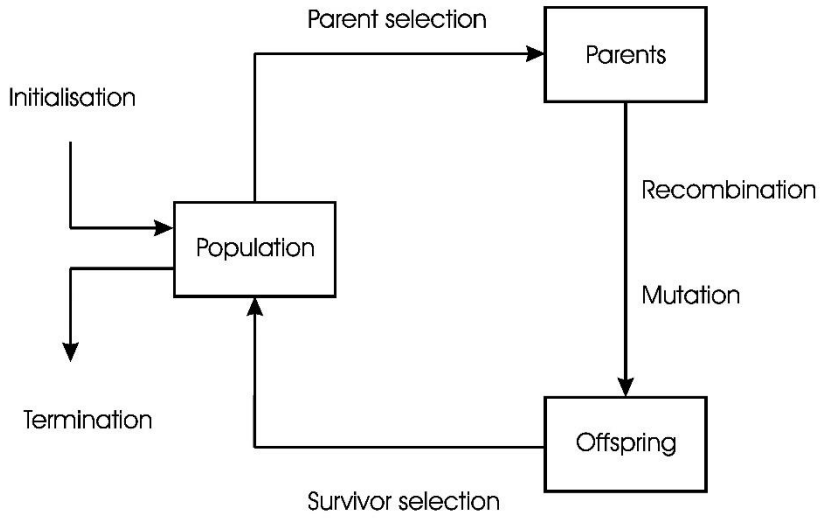

Figure 1: EA Lifecycle

2.2 Genetic Program

A genetic program (GP) is a specific type of EA where, instead of attempting to maximize payoff of the input to a known model, the intention is to optimize the model itself. This is the case when both the input and expected output are known, and the model itself is not. These models are the individuals of the population and their fitness evaluation depends on the quality of their output with respect to the expected output from a specific input [3]. Genetic programs are usually represented with primitives in the form of functions, inputs, and outputs in varying orientations.

2.3 Hyper-Heuristic

A hyper-heuristic is a heuristic (a technique for finding an approximate solution) which searches a space of heuristics, as opposed to a space of solutions directly [4]. Within our research, the hyper-heuristics implemented are considered genetic algorithm hyper-heuristics. They are evolving populations of computer programs represented as arrays. The contents of the indexes of these arrays correspond to the primitive functions selected for this problem domain.

2.4 Linear Genetic Program

The representation chosen for the genetic program is Linear GP. Linear GP is in contrast to tree GP where, instead of representing individuals as trees where the nodes are functions and the leaves inputs, the individual is an array or linear graph [5].

3 Related Work

In David R. White, et. al.'s paper, "Evolutionary Improvement of Programs", it is demonstrated that it is possible to optimize functions to perform better than their base counterparts [1]. It is shown that, using hyper-heuristics and seeding, it is possible to generate algorithms with non-obvious optimizations. These optimizations are placed into the domain of "non-functional" improvements: lower power consumption, better memory utilization, and execution time. What this research has failed to do is to show whether or not similar methods actually target the architecture they are evolved upon. The question also arises whether it is possible to better tune these systems to focus on memory hierarchy. This research directly addresses this question and attempts to show that architectures can be targeted with hyper-heuristics.

In work performed by Eric Schulte et. al., titled, "Post-compiler Software Optimization for Reducing Energy", it was shown that compiled code can be optimized for lowering power consumption [6].Their approach, coined as a Genetic Optimization Algorithm, targets measurable non-functional aspects of software execution that can be improved without altering functional requirements. While novel, this approach limits to what extent the code can be optimized. Once compiled, the functional flow of the algorithm is locked into place. Any novel deviations algorithmically can no longer occur and possible improvements are therefore lost. My research is operating on raw code in an attempt to find useful algorithmic and data-handling approaches for improving upon the non-functional aspects of the program.

With respect to HPC systems specifically, Thomas Weise et. al. performed research on the possibility of evolving distributed algorithms using genetic programming (GP) [7]. It was shown that GP is capable of improving the performance of software on distributed systems. This demonstrates that the networking hierarchy within distributed systems can be exploited, but their work fails to analyze what benefits targeting the processor hierarchy specifically. The major difference between my research and previous work is determining whether or not a hyper-heuristic is capable of performing optimization on a low enough level to exploit the individual nuances of each environment. With this knowledge in hand, it will be possible to explore new avenues of automated hardware targeting in HPC systems and beyond.

The driving force behind much of the Genetic Programming community's interest in software optimization stems from the research done by William B. Langdon and Mark Harman. In their work, they found that they were able to optimize a software package of 50,000 lines of C++ code to gain a performance increase by a factor of 70 (the program ran 70X faster after optimization with their methodology than it did compiled from source) [8]. This further demonstrates that modern compilers are failing to optimize code to its full potential. Langdon's work is the proof that software can be optimized, but not that it can be targeted. The purpose of this research is to take the first step towards optimizing whole programs with the architecture being the target environment and fitness metric.

4 Methodology

To test the extent to which a hyper-heuristic can optimize code to an architecture a problem space with significant memory requirements and room for optimization will be used. Another criterion for this problem class is that, due to the amount of computation to be devoted to evolution, it needs to be calculable in linear time. With these factors in mind, the problem class chosen is sorting.

This tool is intentionally problem-agnostic. Linear GP will be utilized due to its ease of implementation and implicit parsimony pressure on non-effective code [5]. This parsimony pressure will remove operations that have no effect on the output and performance quickly throughout the sort.

In order to remain problem-agnostic, the fitness function and primitives will be kept separate from the structure of the hyper-heuristic. This way, testing a different problem class will be as simple as swapping out fitness functions and potentially the primitives.

This tool will be tested on The Forge at Missouri University of Science and Technology. The Forge is the queueing system available to all students for accessing the school's high-performance computing cluster. In Table 1, the three different architectures present are shown. The two AMD processors are not likely to show large differences in performance due to their similarities. The biggest differences are expected to occur between the two AMD processors and the Intel Processor due to their architectural differences.

Processor Type	**Version**
AMD Opteron	6174
AMD Opteron	6238
Intel Xeon	E5-2698 v3

Table 1: Processor List

For the first phase, jobs will run until at least 30 optimizations have been performed on each of the architectures. A table of host names with their respective processors will be used to specify where the job is run and will distribute their operation across as many different individual nodes as possible. This will be to ensure that any manufacturing or age-related defects in the processors have minimal effect on the results.

During evolution, the hyper-heuristic will be assumed to have converged once the difference between average fitness of the last 100 solutions and the current individual is near zero. At this point, the global best solution will be placed into a file and the program terminated. Convergence is guaranteed to occur due to the limits of the environment being the performance ceiling – solutions cannot improve their performance indefinitely.

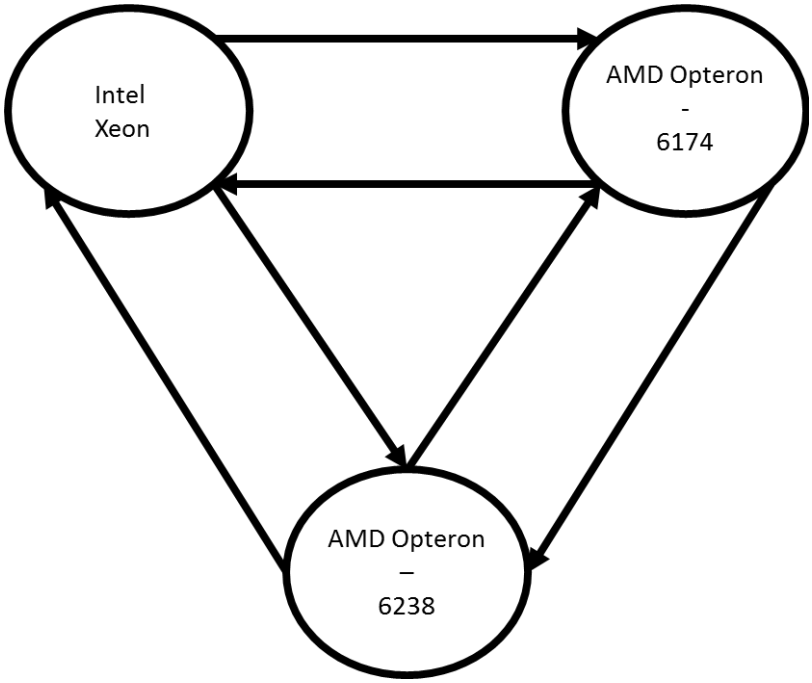

Figure 2: Abstraction of Solution Trading Between Processors

In the second phase, the generated solutions will be pitted against each other on each of the architectures. Once each solution has been run on each of the processors, the results will be collected for analysis. In order to ensure accurate data, each solution will be tested on a data set of 30 different arrays and their performance averaged. An abstraction of this solution trading is shown in Figure 2.

Once the data has been collected, it will be analyzed. The results from the second phase of testing will be compared to each other within the same computing architecture. The performance of each of the 30 solutions will be averaged across their respective average performance across the test data sets. These averages (of the 30 solutions) will be compared to the others tested in the same environment.

If the native solutions (those evolved on the architectures in question) are found to have performed significantly better than those evolved within the other two environments, then it can effectively be concluded that the code was optimized to the architecture. This is only if this significance exists across all three environments (that is, there is no set of 30 solutions better on all architectures, etc.)

5 Preliminary Results

Preliminary testing of sorting algorithm generation has shown that the hyper-heuristic is capable of successfully generating working solutions. Using the Hamming distance as a fitness function, it has produced solutions capable of successfully sorting arrays using the chosen set of primitive functions. An example solution is shown in Figure 3. The array "instrs" is the representation of the individual's primitives and the loop below it is the translator for this array to the function calls each number represents. The numbers and their mapped primitives can be seen in Table 2.

```
import os
import sys
sys.path.insert(0, "/mnt/dfs/jbhf39/Users/jbhf39/Research/HyHeCoAr/python_lgp")
import instructions

data = [47, 16, 30, 81, 37, 4, 36, 64, 17, 15, 69, 60, 18, 2, 96, 38, 25, 75, 80, 31, 72,
59, 92, 10, 41, 66, 26, 52, 98, 42, 44, 20, 68, 7, 54, 12, 71, 9, 33, 77, 73, 50, 99, 13,
93, 89, 1, 94, 19, 35, 14, 91, 95, 49, 70, 6, 65, 48, 34]

instrs = [2, 0, 1, 7, 5, 7, 5, 5, 3, 0, 0, 6, 7, 5]

_continue = True
program_counter = 0
main_program = instructions.Instructions(data)
while _continue:
    main_program.call(instrs[program_counter])
    program_counter = main_program.get_instr_index()
    if(program_counter >= len(instrs)):
        _continue = False
print "sorting fitness:" + str(main_program.get_sortedness())
```

Figure 3: Generated solution

Genotype Number	Corresponding Function	Action
0	min	Sets the 'j' counter to the index of the smallest valued index in the array at an index greater or equal to 'i'
1	swap	Swaps the values of the array at index 'i' and 'j'
2	inc_i	Increments 'i' by one
3	comp_i	If the value at index 'i' is less than the value at last index then set program counter to zero
4	reset_reg	Set 'i' and 'j' equal to zero
5	inc_j	Increments 'j' by one
6	comp_j	If the value at index 'j' is less than the value at the last index then set the program counter to zero
7	max	Sets the 'j' counter to the index of the largest valued index in the array at an index greater or equal to 'i'

Table 2: Primitives and Numerical Mappings

During testing, Valgrind (a programming tool for memory debugging, memory leak detection, and profiling) was used to easily simulate the cache sizes of a processor. By running the solutions within this program, it was possible to simulate cache misses (calling a variable that is not in the faster memory). This data created a search gradient to utilize for ranking the architectural efficiency of each solution.

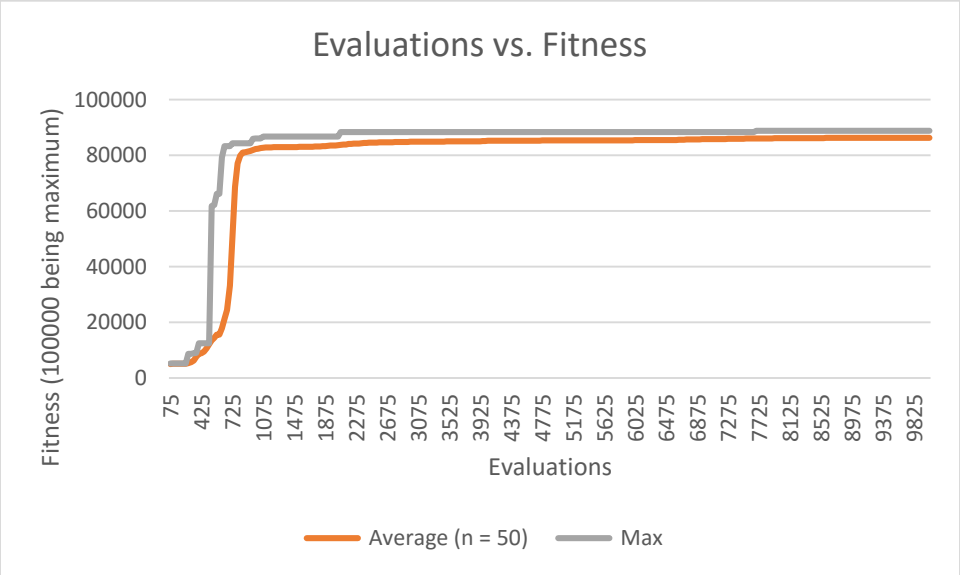

Figure 4: Evaluations vs. Fitness

In Figure 4, the number of evaluations performed versus the average and max fitness of the population are plotted from a standard experiment using Valgrind. The fitness value is a unit-less combination of both the sorting proficiency of the individual as well as its memory efficiency. The graph shows a sharp increase in fitness until the population reaches a local maximum and then makes incremental improvements for the remainder of the experiment.

The major drawback of Valgrind is the high overhead it adds to the simulation. Though it provided a useful metric to use in place of actual wall time, it is by no means a perfect simulation. It forces some amount of estimation and abstraction when interpreting the data for the fitness calculation. This highlights the necessity for actual testing on the cluster and on physical hardware.

6 Discussion

Pending the results from testing on the High Performance Cluster, it can be shown that targeting an architecture is possible. The methodology used can then be utilized to further automate the optimization of entire programs to an architecture. This automation would allow programmers to de-couple the functional and non-functional aspects of their development cycles. During the coding stages of development, the focus of coding would be on satisfying the functional constraints of the project. After these were met, the code would presumably be

deployed. In these environments, it would be possible to then utilize this tool to optimize the code to the architecture it would be running upon.

7 Future Work

The hope of this research is that it will act as a fundamental proof-of-concept to set the stage for the next phase of investigation. This phase will be testing the scalability of this methodology to entire libraries and programs. The biggest challenge in transitioning from this phase of research would be determining what level of code granularity would be most effective for optimization.

In addition, more work can be performed on the sorting fitness algorithm. While using the Hamming distance provides insight into the quality of the sort, it gives misleading results for many edge cases that more sophisticated algorithms can capture.

8 Acknowledgements

Acknowledgements go out to my faculty advisor, Dr. Daniel Tauritz and Dr. William Siever from Western Illinois University for their guidance and support throughout my experience on this project. I'd also like to thank Rebecca Curtis for laying much of the groundwork for this project and Ryan Wood for his incredible technical assistance and contributions to this project.

References

[1] D. R. White, A. Arcuri and J. A. Clark, "Evolutionary Improvement of Programs," *IEEE Transactions on Evolutionary Computation,* vol. 15, no. 4, pp. 515-538, August 2011.

[2] A. E. Eiben and J. E. Smith, "What is an Evolutionary Algorithm?," in *Introduction to Evolutionary Computing*, New York, Springer-Verlag Berlin Heidelberg, 2007, pp. 15-35.

[3] A. E. Eiben and J. E. Smith, "Genetic Programming," in *Introduction to Evolutionary Computing*, New York, Springer-Verlag Berlin Heidelberg, 2007, pp. 101-114.

[4] P. Ross, "Hyper-heuristics," in *Search Methodologies: Introductory Tutorials in Optimization and Decision Support Techniques*, Boston, Kluwer, 2005, pp. 529-556.

[5] M. Brameier, "On Linear Genetic Programming," Dissertation, University of Dortmund, Dortmund, Germany, 2004.

[6] E. Schulte, J. Dorn, S. Harding, S. Forrest and W. Weimer, "Post-compiler Software Optimization for Reducing Energy," *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems,* pp. 639-652, March 2014.

[7] T. Weise and K. Tang, "Evolving Distributed Algorithms with Genetic," *IEEE Transactions on Evolutionary Computation,* vol. 16, no. 2, pp. 242-265, April 2012.

[8] W. B. Langdon and M. Harman, "Optimizing Existing Software with Genetic Programming," *IEEE Transactions on Evolutionary Computation,* vol. 19, no. 1, pp. 118-135, February 2015.