In *Machine Learning for Systems Problems* (NIPS-07 Workshop),
Whistler, British Columbia, Canada, December 2007.

Policy Search Optimization for Spatial Path Planning

Matthew E. Taylor, Katherine E. Coons, Behnam Robatmili,
Doug Burger, and Kathryn S. McKinley
Department of Computer Sciences
The University of Texas at Austin
Austin, TX 78712-0233
{mtaylor,coonske,beroy,mckinley,dburger}@cs.utexas.edu

1 EDGE ISAs

Chip designers have traditionally relied reduced feature sizes and increased the clock rates to improve performance in computer chips. More recently, to better exploit parallelism and achieve faster speeds, more processors and functional units have been placed on a single chip. RISC and CISC architectures require significant programming efforts in order to use these resources efficiently. *Explicit Data Graph Execution* (EDGE) Instruction Set Architectures (ISAs) have recently been developed to support out of order execution, with low power consumption, by passing dataflow information through the instruction set to the multiple functional units, allowing the compiler and processors to better exploit concurrency. TRIPS [2] is an implementation that supports the EDGE ISA, which can schedule eight *hyperblocks*, each one containing a total of 128 instructions, on a 4×4 grid of execution units. The placement of instructions on the units has a direct effect on the execution speed of the a hyperblock. One of the primary challenges for a compiler targeting the TRIPS architecture is mapping a *dataflow graph*, with up to 128 instructions, onto functional units such that the communication overhead is minimized.

2 Problem Formulation

One approach to this problem is to use *Spatial Path Scheduling* [1], which calculates a *placement cost* for each instruction in each possible location. A heuristic is used for this calculation to allow the scheduler to quickly evaluate every instruction, at every possible location, and then greedily place each instruction. If the heuristic can be improved, the speed of programs which are compiled with such a scheduler will likely be improved.

We choose to formulate this problem using the *reinforcement learning* [4] (RL) framework because there are no correct or incorrect labels available to train on, but we can measure the quality of any given schedule. We consider the placement cost as a type of *value function*, which estimates the expected *longterm return* received when an instruction is placed at a location (the return will be related to the execution count for the compiled program). Given some value function, we can evaluate each instruction and determine the costs for placing them at different locations. After the scheduler deterministically places the instructions we execute the program (either in hardware or in a simulator) and record the number of cycles used during program execution. The goal is to minimize the cycle count of the compiled program. We define the RL agent's *reward* to the inverse of the cycles used to execute the program.

To learn an appropriate placement cost function, we first must define features that adequately describe instructions and their relative placement. Beginning with features chosen previously [1], we select a set of 11 features which should allow approximation of an accurate value function. Our features include booleans (i.e., "Is this instruction a load?") and integers (i.e., "What is the critical path length through this instruction if it is placed at this location?").

3 Neural Evolution of Augmenting Topologies

NeuroEvolution of Augmenting Topologies (NEAT) [3] is a type of genetic algorithm that can be used for many different kinds of optimization problems, including RL. NEAT is able to train neural networks to approximate functions of different complexities. Most neuroevolutionary systems require the network topology, such as the number of hidden nodes and their connections, to be fixed. In contrast, NEAT automatically evolves the topology by combining the search for network weights with evolution of the network structure.

In NEAT, a population of genomes, each of which describes a single neural network, is evolved over time: each genome is evaluated and the fittest individuals reproduce through crossover and mutation. NEAT begins with a population of simple networks with no hidden nodes and inputs connected directly to outputs. Two special mutation operators,

add hidden node and *add link*, introduce new structure incrementally, but only structural mutations that improve performance tend to survive evolution. Thus NEAT can find an appropriate level of complexity for a given problem.

When we apply NEAT to scheduling for TRIPS, each neural network represents a different function for calculating the relative placement cost for an instruction at a location, given a set of input features. In each *generation*, the scheduler uses each neural network to compile a given program, each generated executable is run, and the executable's cycle count is used to evaluate every network in the population. We then evolve the population so that, over time, the performance of the scheduler improves.

4 Preliminary Results and Future Work

To test the RL formulation described above we first selected a set of 47 microbenchmarks from a collection of standard programs that have behavior representative of much larger executables. We then trained NEAT on each of the 47 different benchmarks and compared their performance with the hand-coded scheduler. Figure 1 shows the successful results of training on four of these benchmarks. Although these results are very encouraging, when these individual policies are tested on the remaining 46 benchmarks, the speedup gains are more moderate (in the range of 1–2%).

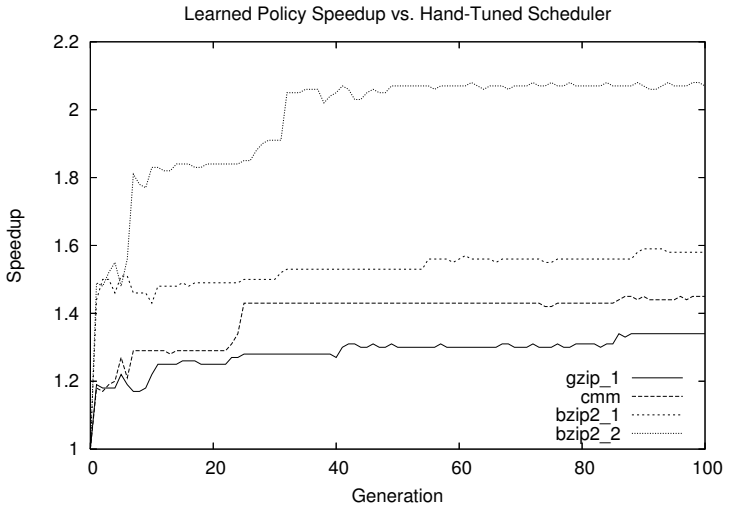

Figure 1: We use NEAT to learn scheduling policies for individual benchmarks. Over time, the policies improve to significantly outperform the hand-tuned scheduler.

We have also tried using NEAT to learn a policy for the entire set of 47 benchmarks, rather than for each benchmark individually. Such a method may allow us to avoid overtraining and discover a general scheduling policy that can outperform the handcoded policy. To date, this method has only produced modest speedups (up to 3%).

One potential enhancement is to attempt to cluster hyperblocks by properties of the dataflow graph. If we can determine clusters of similar hyperblocks, we may be able to train different policies for each cluster of hyperblocks. Then, after training, we can classify novel individual dataflow graphs into one of the existing clusters, and use a policy appropriate to the type of program currently being be compiled. Such a method may allow us to significantly speed up the hyperblocks within each cluster, as well as be able to successfully schedule novel programs.

A second possible enhancement would be to try using *temporal difference* methods, another popular RL solution technique, as they have been shown to outperform genetic algorithms in some circumstances [5]. Lastly, we could change the problem formulation so that rather than learning a value function, we could learn a full action selector. In such a system the entire scheduler could be replaced by a RL module that learned which instructions to place on which tiles, instead of greedily exploiting a learned (or hand-coded) placement cost function.

References

[1] K. Coons, X. Chen, S. Kushwaha, D. Burger, , and K. S. McKinley. A spatial path scheduling algorithm for edge architectures. In *The Twelth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2006.

[2] K. Sankaralingam, R. Nagarajan, R. McDonald, R. Desikan, S. Drolia, M. S. Govindan, P. Gratz, D. Gulati, H. Hanson, C. Kim, H. Liu, N. Ranganathan, S. Sethumadhavan, S. Sharif, P. Shivakumar, S. W. Keckler, and D. Burger. Distributed microarchitectural protocols in the trips prototype processor. In *MICRO 39: Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 480–491, Washington, DC, USA, 2006. IEEE Computer Society.

[3] K. O. Stanley and R. Miikkulainen. Evolving neural networks through augmenting topologies. *Evolutionary Computation*, 10(2):99–127, 2002.

[4] R. S. Sutton and A. G. Barto. *Introduction to Reinforcement Learning*. MIT Press, 1998.

[5] M. E. Taylor, S. Whiteson, and P. Stone. Comparing evolutionary and temporal difference methods in a reinforcement learning domain. In *Proceedings of the Genetic and Evolutionary Computation Conference*, pages 1321–28, July 2006.