In *The 24th Conference on Artificial Intelligence, New Scientific and Technical Advances in Research (NECTAR) track* (AAAI-10), Atlanta, GA, July 2010.

Evolving Compiler Heuristics to Manage Communication and Contention

Matthew E. Taylor,* **Katherine E. Coons, Behnam Robatmili,**
Bertrand A. Maher, Doug Burger,† and **Kathryn S. McKinley**
$\star$ Lafayette College, *taylorm@cs.lafayette.edu*
The University of Texas at Austin, {*coonske, beroy, bmaher, mckinley*}*@cs.utexas.edu*
† Microsoft Research, *dburger@microsoft.com*

Abstract

As computer architectures become increasingly complex, hand-tuning compiler heuristics becomes increasingly tedious and time consuming for compiler developers. This paper presents a case study that uses a genetic algorithm to learn a compiler policy. The target policy implicitly balances communication and contention among processing elements of the TRIPS processor, a physically realized prototype chip. We learn specialized policies for individual programs as well as general policies that work well across all programs. We also employ a two-stage method that first classifies the code being compiled based on salient characteristics, and then chooses a specialized policy based on that classification.

This work is particularly interesting for the AI community because it 1) emphasizes the need for increased collaboration between AI researchers and researchers from other branches of computer science and 2) discusses a machine learning setup where training on the custom hardware requires weeks of training, rather than the more typical minutes or hours.

Introduction

Modern compilers employ many heuristic functions to generate high-performance code for today's complex computer architectures. These heuristics are typically hand-written by compiler developers, who use a combination of laborious benchmarking and human intuition to develop heuristics that achieve good performance. To alleviate this burden from compiler writers and to achieve more optimal solutions within the complex space of heuristics, our work uses a genetic algorithm to learn better compiler policies (Coons *et al.* 2008). We represent a compiler heuristic function, a key component of the compiler's policy, as a neural network. Training improves the heuristic functions to optimize the performance of compiled benchmark code.

Using this methodology, we learn specialized heuristics for particular benchmark programs, achieving a 12% speedup over hand-tuned heuristics and 4% over a simulated annealing solution. We also learn a general heuristic by optimizing the average performance over the entire set of benchmarks, but achieve only a 1% improvement over the hand-tuned heuristic. To create general heuristics that approach the performance of the specialized heuristics, we then use clustering to group similar segments of code. By learning reusable, per-cluster heuristics, we achieve speedups of 4–6% over existing hand-tuned heuristics.

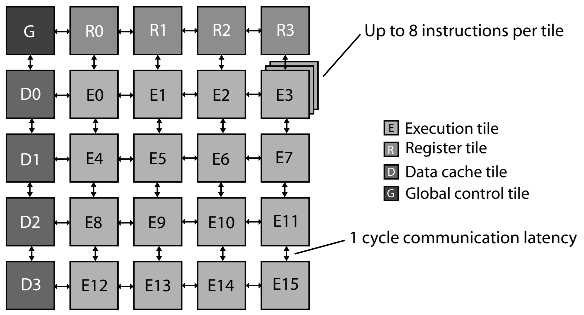

Figure 1: Instruction placement example

The remainder of this section provides an overview of the compiler problem we address: developing an instruction placement policy that balances communication and contention for a novel processor architecture. The Solution Techniques and Results section describes the methods we used to attack this problem at three levels of policy generality. Finally, the Discussion section argues that this type of collaboration between the AI and systems communities will become increasingly important and enumerates the lessons learned from this case study.

Instruction Placement for EDGE Architectures

Explicit Dataflow Graph Execution (EDGE) architectures are a recently developed class of instruction set architectures (ISAs) that express a program as a sequence of multi-instruction blocks. Each block contains a set of machine instructions, which are mapped to an array of functional units. The TRIPS processor, a physical instantiation of an EDGE ISA, has a 4×4 array of functional units (see Figure 1). Instructions within a block execute in *dataflow* order: each instruction waits until its operands arrive and then sends its result to a set of *target* instructions that use that result. Instruction placement is thus critical for fast execution.

An EDGE compiler transforms source code into blocks of machine instructions (Maher *et al.* 2006; Smith *et al.* 2006). The compiler represents instructions within a block as a directed acyclic graph, where the nodes are instructions and the edges are dependencies. Using this graph, the *instruction scheduler* assigns each instruction to the particular functional unit where it will execute at runtime. To achieve high performance, the scheduler must balance the desire for concurrency against the cost of communication. To maximize concurrency, instructions should be placed on separate functional units; to minimize communication, however, dependent instructions should be placed nearby or on the same functional unit.

Our prior work introduced *Spatial Path Scheduling* (SPS), an algorithm for EDGE architectures to select a location for each instruction based on a *placement cost* (Coons *et al.* 2006). SPS uses a heuristic function to compute the placement cost, which is a real-valued number, from several inputs: features of the instruction, the location under consideration, and the graph in which the instruction resides. The scheduler then uses this cost to place instructions, greedily placing the highest cost instruction on the functional unit with the lowest placement cost (i.e., a minimax strategy).

Prior instruction placement techniques rely on a combination of laborious benchmarking and human intuition to create hand-tuned heuristics. In this work, a genetic algorithm learns a placement cost heuristic. Prior work shows that the hand-tuned SPS heuristic provides results within 5% of those found via simulated annealing (Coons *et al.* 2006). Although this may suggest that this heuristic is close to an upper bound, results in this paper show that significant improvement over simulated annealing is possible.

To enable learning, we replaced the SPS heuristic function with a neural network that determines the placement cost for an instruction at a given location (see Figure 2). The placement cost fully specifies the scheduler's placement policy.

We focused on short-running benchmarks to minimize training time and selected programs with varied characteristics, including available parallelism and register/memory usage, to provide a range of placement requirements during training. The final benchmark set contained 47 benchmarks, large enough to ensure that the benchmark set sizes would be reasonable after clustering (discussed later). After compilation, the evolved networks were evaluated by executing the generated code on the TRIPS processor.

Solution Techniques and Results

This section describes the genetic algorithm used for optimization, then provides an overview of the three types of experiments conducted.

NEAT

This paper uses *Neuro-Evolution of Augmenting Topologies* (NEAT) (Stanley & Miikkulainen 2002), a genetic algorithm to learn neural networks that compute the placement cost. NEAT uses a neural network to specify each heuristic function and uses biologically-inspired operators to stochastically improve a *fitness* metric over time. Specifically, we use the variant *FS-NEAT* (Whiteson *et al.* 2005), which assists with feature selection by incrementally adding links from possible inputs into the neural network over time, using an *add link* mutation. This procedure allows evolution to determine the features that most improve the fitness of each network. Each network's fitness is the speedup, in processor cycles, of its compiled code over code produced by the baseline hand-tuned heuristic. There is a fixed-size *population* of networks, and FS-NEAT uses the networks' fitnesses to evolve and create the next *generation* of networks.

NEAT is an appropriate choice for multiple reasons. First, NEAT is a publicly available package with nine different im-

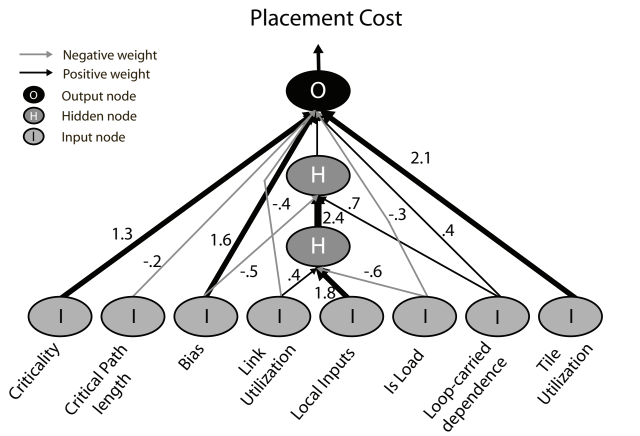

Figure 2: An example neural network including: inputs (features), output (placement cost), two hidden nodes, and edge weights

plementations. Second, NEAT is a well understood, domain-independent method that has been successfully applied to a variety of tasks. Third, the process of complexification has been empirically shown to reduce training times (Stanley & Miikkulainen 2002) and thus make large optimization problems tractable. Fourth, NEAT requires relatively little parameter tuning in practice for successful learning, which is critical for discovering a good heuristic quickly.

Specialized Solution

The most straightforward setup uses NEAT to learn one heuristic per benchmark. This experiment produced 47 populations of heuristics, each specialized for a different program. Although this required 47 separate training runs, these training runs could execute in parallel. As expected, we found that learning heuristics for individual benchmarks leads to overfitting. These overfit heuristics, however, resulted in better per-benchmark performance than hand-tuned heuristics or placements generated via simulated annealing. Although these specialized heuristics may not provide robust general solutions, this technique is appropriate for optimizing performance-critical programs or libraries.

Initial experiments indicated that training plateaued within 100 generations. After training for 100 generations, we compared results for specialized heuristics to results for both the instruction scheduler's hand-tuned heuristic, and a simulated annealing scheduler. Simulated annealing, as used in prior work, finds a *placement*, not a heuristic to produce placements (Coons *et al.* 2006). Thus, if the source program changes the entire annealing process must run again to accommodate the changes. With NEAT, however, the end result is a *heuristic* that can be used to place any program.

We implemented a simulated annealing scheduler for TRIPS and used the instruction scheduler's placement as the starting point. Table 1 shows the geometric mean of the speedup across 46 of the 47 benchmarks in the training suite normalized to the hand-tuned scheduler's performance on the same benchmark (one benchmark was omitted due to incompatibility with the annealer). These results show that 1) NEAT outperforms hand tuning, 2) NEAT outperforms simulated annealing as an optimization method in this problem, but 3) the policies produced by NEAT are not optimal and

Technique	Speedup
Hand-tuned	1.00
Hand-tuned + Annealing	1.08
NEAT	1.12
NEAT + Annealing	1.14

Table 1: Relative speedups for benchmark-specific heuristics

further tuning may yield still faster programs, as shown in the last line of Table 1 where simulated annealing improved upon NEAT-generated compiler policies.

General Solution

The previous section discussed training a separate heuristic for each benchmark; this section presents results of training a single heuristic across all benchmarks. Training NEAT to maximize the speedup across the entire set of benchmarks resulted in a more general heuristic that was not specialized for any particular benchmark. The best network trained using this approach provided an average speedup of 1.010 over the hand-tuned SPS heuristic. To test this general solution on new data, we ran the best network trained across the 47 benchmark set on a different (test) benchmark set, but the average speedup was only 1.005, nearly identical to the hand-tuned solution.

Although this result is significant given that it is compared to a highly hand-tuned result, it is modest compared to results observed in Stephenson et al.'s Meta Optimization work, which observed speedups of 1.44, 1.03, and 1.31 for hyperblock formation, register allocation, and data prefetching, respectively, using a general heuristic trained across sets of ten or fewer benchmarks (Stephenson *et al.* 2003). On novel benchmarks, they observed speedups of 1.09, 1.02, and 1.01 for the same three optimization problems.

One explanation for these limited speedups using the general solution may be that our baseline is better optimized for the general case; prior work showed that the baseline came within 5% of simulated annealing solutions (Coons *et al.* 2006). Another explanation might be the potential for improvement — Stephenson et al. focused on optimizations such as hyperblock formation, which may have greater impact on performance than instruction placement. Although we were unable to find general solutions that significantly outperformed the hand-tuned solution, the difference between the best specialized heuristics and the general heuristic suggests that further improvements are possible.

Clustered Benchmark Optimization

Different blocks of instructions may require different placement policies. For example, some blocks may contain many instructions that can execute in parallel, whereas other blocks may contain many dependences between instructions, forcing them to execute sequentially. When these block-level factors vary significantly, a heuristic specialized for that particular type of block may be preferable.

To help NEAT take advantage of these higher-level differences between blocks, we introduced a clustering and classification stage to better optimize blocks based on their salient characteristics. One difficulty of this clustering approach, however, is that we can evaluate performance on a per-benchmark basis, yet we cannot extract meaningful features for an entire program. Conversely, while we can extract features from a block of instructions, we cannot evaluate the performance of a block in isolation. Furthermore, while the benchmarks we used were quite small, real programs are large and complex, and may need different heuristics for different parts of the same program.

These problems motivate a *critical block* approach, where we associate the performance of a benchmark with the performance of its most critical blocks — the blocks that together represent over 80% of that benchmark's execution time. Because the benchmarks were quite small, they typically contained only a few critical blocks. We cluster benchmarks based on their performance, and train a classifier for those clusters based on the features of the critical blocks in the cluster. This section describes this compromise approach, in which we 1) determine which benchmarks perform well with the same SPS heuristics, 2) use this similarity metric to cluster the benchmarks and their corresponding critical blocks, 3) train a classifier to label novel blocks into one of the specified clusters, 4) train a heuristic for each cluster, and 5) evaluate the classifier and trained heuristics.

Defining a Similarity Metric To cluster blocks, we defined a metric based on how well benchmarks perform when scheduled with the same heuristic. After training NEAT on each benchmark individually, we used the top ten networks in the population to compile every other benchmark. Using this data, we created a *similarity graph* in which the nodes represent benchmarks and the edges between nodes represent the tendency of those benchmarks to perform similarly with the same heuristics. We used the publicly available graph clustering tool *Graclus* to cluster benchmarks via the similarity graph (Dhillon, Guan, & Kulis 2007).

Critical Block Features and Classification We assigned the critical blocks within each benchmark to that benchmark's cluster and built a classifier to place new blocks into the appropriate cluster. After finding the class associated with each critical block (i.e., its benchmark's cluster), we extracted 16 block-level features[1] for the classifier to use to predict their class.

We tested four different classifiers in Weka (Witten & Frank 2005): a decision tree, a propositional rule learner, a support vector machine, and a multi-layer perceptron. We tested each classifier with 3, 4, 5, and 6 clusters. When many clusters were allowed, Graclus tended to produce clusters with one or two blocks, grouping the rest of the training set into a single cluster. By using cross validation, we found that the rule learning method (JRip, based on RIPPER (Cohen 1995)) with three classes (i.e., clusters) produced high accuracy results, as well as a fairly equal distribution of critical blocks across the different clusters.

After training the classifier, we again used NEAT to train heuristics for each of the three clusters. We found that one of the classes contained a catch-all for blocks that did not

[1]We did not use the same feature set as used to place instructions — features used to classify blocks must be representative of the entire block rather than individual instructions.

fit into either of the other classes, and we saw very little speedup by training on these blocks. We opted to use hand-tuned SPS to schedule those blocks to save training time. Thus, NEAT produced two different heuristics for the two clusters, one containing 17 benchmarks and the other 12.

Clustering and Classification Results We trained NEAT heuristics to optimize critical blocks per-cluster (excluding the third catch-all cluster). One experimental advantage to the clustering technique is that we could perform the training runs in parallel (per cluster). While it took multiple weeks to perform a complete training run with the general solution, it required less than a week to obtain significantly better results with classification. Across the 17 benchmarks that contained critical blocks belonging to the first cluster, we saw a speedup of 1.06. Among the 12 benchmarks containing critical blocks belonging to the second cluster, we saw a speedup of 1.04 after only a day of training.

Discussion

The work presented in this paper is an example of "use-inspired" learning: this project started when an AI student attended a practice talk for a compilers conference. In AI we sometimes focus on "toy problems," which are valuable for initial exploration, but provide little guidance when one must decide what algorithms to use when presented with a novel real-world problem. In addition to defining and addressing such a problem, this study is noteworthy because it describes a complex domain that necessitated long training times. Rather than learning a heuristic in a matter of seconds or minutes, learning time was measured in days or weeks. In light of this problem, feature selection became quite important. Enabling the genetic algorithm to search over a constrained policy space was *critical*, not just convenient. Even though we selected a learning algorithm (FS-NEAT) that was designed to handle many irrelevant features, we found that off-line feature selection using lasso regression was an important component to achieving high performance (Coons *et al.* 2008). Using FS-NEAT we determined that different benchmarks likely require different features to produce good placements: only 20% of the 47 benchmarks made use of all 12 of the hand-selected features.

It was informative to see what features FS-NEAT selected for the neural network inputs. By examining sets of high performance policies, we can develop an understanding of the most important features. AI researchers may often be satisfied with a neural network (or other "black boxes") as outputs, but the systems community is much more likely to want to *understand* which features are important to assist in the design of future systems.

Systems problems often involve many intertwined and complex components. In the final set of experiments using clustering and classification, NEAT effectively learned a placement heuristic given a particular compiler configuration, but the solutions were not robust when other parts of the compiler changed. For example, heuristics learned at a particular compiler optimization level were not effective with a different optimization level, even for the same benchmarks. Furthermore, the best heuristic for one class depended on the heuristics used for the other classes, indicating additional gains may come from co-evolving the different heuristics. Ideally any optimization technique used would be robust to such changes, better handling the complex interactions between the many parts of a system.

Although NEAT and FS-NEAT have many parameters, the default settings have been reported to work well in practice (Stanley & Miikkulainen 2002). While the results obtained were provably non-optimal, the reality of high training times precluded carefully optimizing NEAT's 24 parameters. Weka provides a general off-the-shelf solution to many classification and regression problems by providing a simple interface to tens of learning methods, but there is no such package for evolutionary algorithms. AI techniques will become more commonly used in systems research, in both hardware (Ipek *et al.* 2008) and software (Stephenson *et al.* 2003; Coons *et al.* 2008) — providing such packages would make our field's techniques more accessible.

Acknowledgements

We thank Steijn Kistemaker, W. Bradley Knox, Jason Tsai, and the anonymous reviewers for helpful comments and suggestions on this paper. Research described in this paper was conducted while all authors were at The University of Texas at Austin.

References

Cohen, W. W. 1995. Fast effective rule induction. In *ICML*.

Coons, K. E.; Chen, X.; Burger, D.; McKinley, K. S.; and Kushwaha, S. K. 2006. A spatial path scheduling algorithm for EDGE architectures. In *ACM Conf. on Architectural Support for Programming Languages and Operating Systems*.

Coons, K. K.; Robatmili, B.; Taylor, M. E.; Maher, B. A.; McKinley, K.; and Burger, D. 2008. Feature selection and policy optimization for distributed instruction placement using reinforcement learning. In *The Seventh Intl. Joint Conf. on Parallel Architectures and Compilation Techniques*.

Dhillon, I. S.; Guan, Y.; and Kulis, B. 2007. Weighted graph cuts without eigenvectors: A multilevel approach. *IEEE Trans. Pattern Anal. Mach. Intell.* 29(11):1944–1957.

Ipek, E.; Mutlu, O.; Martínez, J. F.; and Caruana, R. 2008. Self-optimizing memory controllers: A reinforcement learning approach. In *Intl. Sym. on Computer Architecture*.

Maher, B. A.; Smith, A.; Burger, D.; and McKinley, K. S. 2006. Merging head and tail duplication for convergent hyperblock formation. In *IEEE/ACM Intl. Sym. on Microarchitecture*.

Smith, A.; Burrill, J.; Gibson, J.; Maher, B.; Nethercote, N.; Yoder, B.; Burger, D.; and McKinley, K. S. 2006. Compiling for EDGE architectures. In *Intl. Sym. on Code Generation and Optimization*.

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

Stephenson, M.; Amarasinghe, S.; Martin, M.; and O'Reilly, U.-M. 2003. Meta optimization: Improving compiler heuristics with machine learning. In *ACM Conf. on Programming Language Design and Implementation*.

Whiteson, S.; Stone, P.; Stanley, K. O.; Miikkulainen, R.; and Kohl, N. 2005. Automatic feature selection in neuroevolution. In *GECCO*.

Witten, I. H., and Frank, E. 2005. *Data Mining: Practical machine learning tools and techniques, 2nd Ed.* Morgan Kaufmann.