15-745 Class Project Report

Brendan Meeder (bmeeder@cs),
Jamie Morgenstern (jaimiemmt@cs),
Richard Peng (richard.peng@gmail.com)

April 27, 2011

Abstract

Selecting an ordering for running optimizations on programs is a non-trivial task: the speedup imparted by different passes depends upon those passes already executed and the nature of the program. We employ technique from machine learning to construct an estimator which, when given given feature of the program and a set of optimizations to execute, will predict the speedup factor of the program under those optimizations. From these estimates, an ordered set of optimizations can be found that maximizes the estimated speedup factor. Our results show this method to be promising given a sufficient number of programs and optimization passes for the learning procedure.

1 Introduction

A central question in optimizing compilers is determining which optimization passes should be run. Even with exorbitant amounts of computation time, the order of optimizations can greatly affect the impact of the optimizations. When the resources available for performing optimization passes is bounded, the choice of optimization passes and their ordering is even more critical.

In this project[1], we consider the problem of choosing which four basic optimizations to execute on a program. We employ techniques from machine learning to build a regression model that predicts the speedup of a program under a set of optimizations. Such models can be used to select optimizations that it expects to perform well. This problem isn't straightforward since the order in which optimization passes are executed can greatly affect the performance of the code. For example, if we run aggressive dead code elimination before running liveness analysis, and compare the performance to performing the passes in the opposite order, we would expect the second order to produced a greater speedup. However, the dependence between particular analysis and transformation passes is not always clear, and often depends upon the structure of the code being optimized.

We train a neural network using data about the speedup that different orderings of optimization passes gave for linked SPEC benchmark code and use it to predict the ordering for other SPEC benchmarks which will achieve large speedup factors. Neural networks aim to model a function given its inputs X (in this case, the ordered optimizations and features extracted about the program) to the output Y (the expected speedup factor for using that ordering of optimizations on code with those features). Once we train the neural net, we do brute-force search over then orderings of optimizations to use for a different program. A neural net is a particularly good model in this setting: no assumptions are made about the independence of the feature space, and regression (estimation of the speedup factor) is quite fast once the net is trained. The efficiency of regression is important if we choose to do brute-force search over a large set of optimizations. In practice, one could also choose some fixed set of orderings to try between and use the net to predict which of those orders would provide the best performance for a program in question.

[1] addressed a similar problem of finding the set of passes that optimizes code performance. The related problem of reducing compilation time by selecting a useful set of passes was the main focus of [4]. The list of features listed on page 22 of [4] will also act as a starting point for our initial passes for identifying features of the program. However, techniques such as the ones used in [3] to automatically finding features might also be worth exploring.

[1] The website for this project is `http://www.cs.cmu.edu/~bmeeder/15745/proposal.html`

2 SPEC benchmarks and Training Data Collection

We selected 14 different optimization passes built-in from LLVM, and divided them (somewhat arbitrarily) into two groups: the first 9 are transformations(changes but not necessarily deletions of code) and the last 5 reductions (passes that primarily got rid of dead code). The list follows:

- Mem2Reg
- always-inline
- argpromotion
- Constprop
- loop invariant code motion
- loop strength reduction
- loop-unroll
- reassociate
- indvars
- agressive dead code elimination
- deadstoreelimination
- instcombine
- loop-deletion
- tail-call-elim

We created 100 random optimization orderings as follows. We randomly (with replacement) chose 2 transformations and 2 reductions. We then ran the optimizations in the ordering (transformation$_1$, reduction$_1$t, trasformation$_2$, reduction$_2$). For each of the SPEC benchmarks (in training mode), we compiled the code into (unoptimized) llvm bytecode, ran the optimization passes, and then measured the performance increase for each setting of the optimizations with respect to the runtime of the unoptimized code.

The speedup factors of each benchmark, compiled with each of the optimization settings, along with the features of the program itself and the optimization settings for each run comprised the training data for the learning algorithm described below.

3 Features from Programs

The feature data from the programs used are listed in table 3. Four analysis passes were used to gather these data, and they were aggregated using the STATISTICS macro in llvm. This macro allows certain variables defined by it to be gathered and outputted to by running opt with the -stats command.

Two of the passes, instruction count and region based analysis are standard llvm analysis passes, they were the only ones among the built in passes that provided information that we considered useful. Some of the information about instruction counts are already very strong indicators of the type of program. For example, programs with a high count of floating point operations are typically associated with linear algebra programs, on which optimizations favoring longer loops are much more preferable.

The register pressure information was obtained using the live variable analysis described in class. Some difficulties were encountered in adapting the analysis to deal with some of the larger program that has functions with many variables.

The LoopInfo analysis was used to extract out the loop hierarchy structure in an intermediate form. Then statistics about the commands are computed by iterating over the basic blocks and examining their position in the structure obtained.

4 Experimentation Methodology

We performed a comprehensive array of experiments that underwent several iterations. The basic premise behind our method is to learn the speedup-factor of a program given its features and the set of optimizations used. If such a model can be learned, then we can proceed to choose optimization tuples in one of two ways. Combined with the extracted features for a new program P, we could try each of the

Benchmark	Lang.	Program type	Description
400.perlbench	C	Programming Language	Derived from Perl V5.8.7. The workload includes SpamAssassin, MHonArc (an email indexer), and specdiff (SPEC's tool that checks benchmark outputs).
401.bzip2	C	Compression	Julian Sward's bzip2 version 1.0.3, modified to do most work in memory, rather than doing I/O.
403.gcc	C	C Compiler	Based on gcc Version 3.2, generates code for Opteron.
429.mcf	C	Combinatorial Optimization	Vehicle scheduling. Uses a network simplex algorithm (which is also used in commercial products) to schedule public transport.
445.gobmk	C	Artificial Intelligence	Plays the game of Go, a simply described but deeply complex game.
456.hmmer	C	Search Gene	Sequence Protein sequence analysis using profile hidden Markov models (profile HMMs)
458.sjeng	C	Artificial Intelligence	Chess A highly-ranked chess program that also plays several chess variants.
462.libquantum	C	Physics / Quantum	Computing Simulates a quantum computer, running Shor's polynomial-time factorization algorithm.
464.h264ref	C	Video Compression	reference implementation of H.264/AVC, encodes a videostream using 2 parameter sets. The H.264/AVC standard is expected to replace MPEG2
471.omnetpp	C++	Discrete Event	Simulation Uses the OMNet++ discrete event simulator to model a large Ethernet campus network.
473.astar	C++	Path-finding	Algorithms Pathfinding library for 2D maps, including the well known A* algorithm.
483.xalancbmk	C++	XML Processing	modified version of Xalan-C++, which transforms XML documents to other document types.

Table 1: Listing of SPEC 2006 integer based benchmarks used in the project.

Benchmark	**Lang**	**Program type**	**Description**
433.milc	C	Physics / Quantum Chromodynamics	A gauge field generating program for lattice gauge theory programs with dynamical quarks.
444.namd	C++	Biology / Molecular Dynamics	Simulates large biomolecular systems. The test case has 92,224 atoms of apolipoprotein A-I.
447.dealII	C++	Finite Element Analysis	deal.II is a C++ program library targeted at adaptive finite elements and error estimation. The testcase solves a Helmholtz-type equation with non-constant coefficients.
450.soplex	C++	Linear Programming, Optimization	Solves a linear program using a simplex algorithm and sparse linear algebra. Test cases include railroad planning and military airlift models.

Table 2: Listing of SPEC 2006 floating-point based benchmarks used in the project.

#	Description	Method Obtained
1	Number of Br instructions	instruction count pass (-instcount)
2	Number of Ret instructions	
3	Number of Load instructions	
4	Number of Store instructions	
5	Number of Alloca instructions	
6	Number of memory instructions	
7	Number of basic blocks	
8	Number of FAdd instructions	
9	Number of FCmp instructions	
10	Number of FDiv instructions	
11	Number of FMul instructions	
12	Number of FPToSI instructions	
13	Number of FSub instructions	
14	Number of Unreachable instructions	
15	Number of regions	region analysis pass (-regions)
16	Number of simple regions	
17	Max number of variables in a function	live variable analysis
18	Maximum number of registers live simultaneously	
19	Total number of variables	
20	Average depth of a statement inside a loop	iteration + LoopInfo
21	Maximum loop nesting depth	
22	Percentage of statements contained in some loop	

Table 3: Listing of the 22 extracted program features.

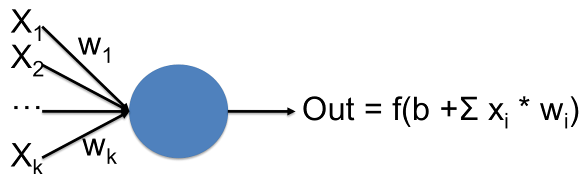

Figure 1: Illustration of a perceptron.

100 optimization tuples used in the benchmarking phase and pick the one we estimate will perform best. Alternatively, we could search over all 2,025 optimization tuples and pick the one that performs best.

Several methods are available for doing multivariate, nonlinear regression. Logistic regression and neural networks are two models that come to mind. Due to one author's familiarity with neural networks, we chose to them for learning the regression models.

4.1 Background on Neural-networks

Neural networks are a computation model that can solve a variety of classification and regression problems. The interested reader can browse through [2], which is a comprehensive exposition on the subject of neural networks. For the sake of this project, we consider feed-forward networks with one or two hidden-layers.

Network nodes. A neural network *node* is the basic unit of computation; it is also sometimes called a perceptron. The node takes a set of input signals, weights them, and *transfers* the signal according to a transfer function f. Formally, if the node has an input signal $\mathbf{x}$ and weight vector $\mathbf{w}$, then the output of the node is $f(\mathbf{x} \cdot \mathbf{w} + b)$, where b is the *bias* at the node. This process is illustrated in Figure 1. A feed-forward network can take on an arbitrary acyclic topology in which neural network nodes are vertices in a directed graph and there is an edge (u, v) if the output of node u is used in calculating the transfer function at node v. For the hidden nodes in the network, a hyperbolic tangent sigmoid transfer function, given by $f(x) = 2/(1 + \exp(-2 * x)) - 1$, is used at the hidden nodes, and a linear transfer function ($f(x) = \alpha x + \beta$) is used at the output node.

Network topologies. We investigate using complete one hidden-layer and two hidden-layer networks. The topologies of these types of networks are illustrated in Figures 2 and 3. In general, one must experiment with the number of hidden-layers and number of nodes; more nodes and hidden-layers allow for more complex functions to be learned, but suffer from over-fitting, longer training times, and potentially worse training performance because training methods converge to a local, and not a global, optima.

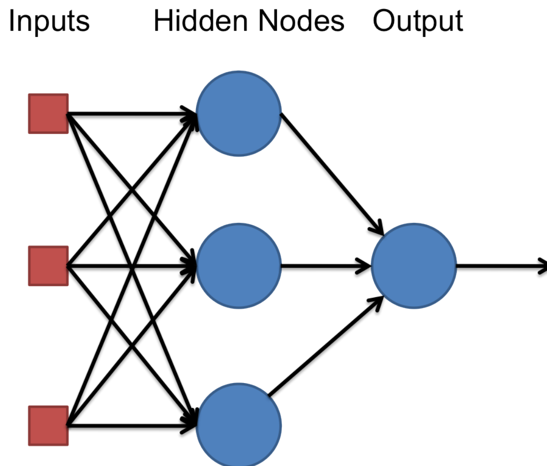

Figure 2: Topology of a single hidden-layer feed-forward neural network.

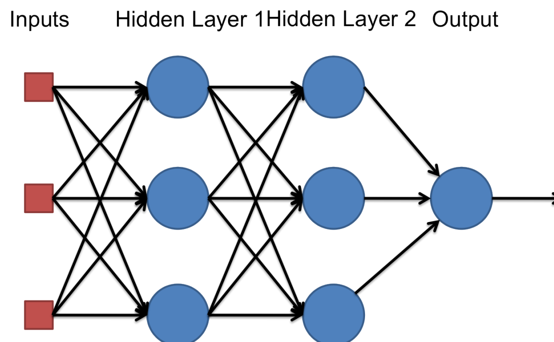

Figure 3: Topology of a two hidden-layer feed-forward neural network.

Network Training. We use the MATLAB neural network toolbox for training and evaluating the networks. First we describe the data preprocessing we performed. We then give an overview of the progression of training and evaluation that we used to come to the final set of classifiers used in the experiments.

For a program P, we use all of the instances of SPEC test suite programs $P' \neq P$ and optimization tuples $\mathcal{O}$ to train an ensemble (collection) of neural networks as described previously. Due to the difference in scales for each of the variables, the data is first normalized so that each feature component has zero mean and unit variance. This is standard technique speeds up training as the features have all been put into a similar range and the wide range in parameter scale does not need to be learned. Only the training data is used to determine these scaling transformations; it is possible that the held out data for program P will have program features that lie largely outside of these ranges.

We tried single-layer networks with 5, 10, 20, and 50 hidden nodes, as well as two-layer networks with (5,5) and (10,5) hidden nodes in each layer. As the MATLAB training process only uses a fraction of the training data, (it saves some for validation-stopping and for testing), we train each network multiple times to get a sense for the best-case and worst-case performance. Based on the performance of the validation and training errors, we decided that a two hidden-layer network with 5 nodes in each hidden layer provided the best compromise between training accuracy and testing accuracy. However, how do these training errors (mean-squared error between the estimated speedup-factor and actual speedup-factor) translate into picking good optimization tuples? For each network, we show how well the optimization tuple it predicts will do best performs compared to the actual best optimization tuple of the 100 we used. We show the results for these different sized networks on each of the benchmark programs in Figure 4.1.

As mentioned previously, we use an ensemble of five networks as any single network potentially suffers from the learning process getting stuck in a 'bad' local optimum, or from the training data being skewed. Regardless, we can view each network as a 'weak classifier' that can be used in a boosting process. Since we didn't have additional training data on which we could perform a boosting procedure, we simply average the output of all of the networks to get an estimated program speedup under a particular optimization tuple. In all of the experiments below, we train an ensemble *for each program* P using all of the data from programs $P' \neq P$ and optimization tuples. Thus, we really trained 75 many neural networks.

5 Experimental Results

5.1 Constrained Optimization Tuple Experiment

The first set of experiments we run measure how well the learned models select optimization tuples when the tuple is one of the 100 we used to generate the training data. Using this ensemble, we calculate the expected speedup factor for each optimization tuple $\mathcal{O}$ and pick the best one, $\hat{\mathcal{O}}$. We look at the speedup factor of P under optimizations $\hat{\mathcal{O}}$ and compare it to the maximum speedup factor of P under any of the 100 tested optimization tuples.

In Figure 5.1, we report the speedup factor of $\hat{\mathcal{O}}$ as a percentage of the maximum speedup factor. If the regression model were perfect, then the estimated optimal optimization parameters would be the best ones, and this ratio would be 1.0 for every test. We note that as our training regimen is actually

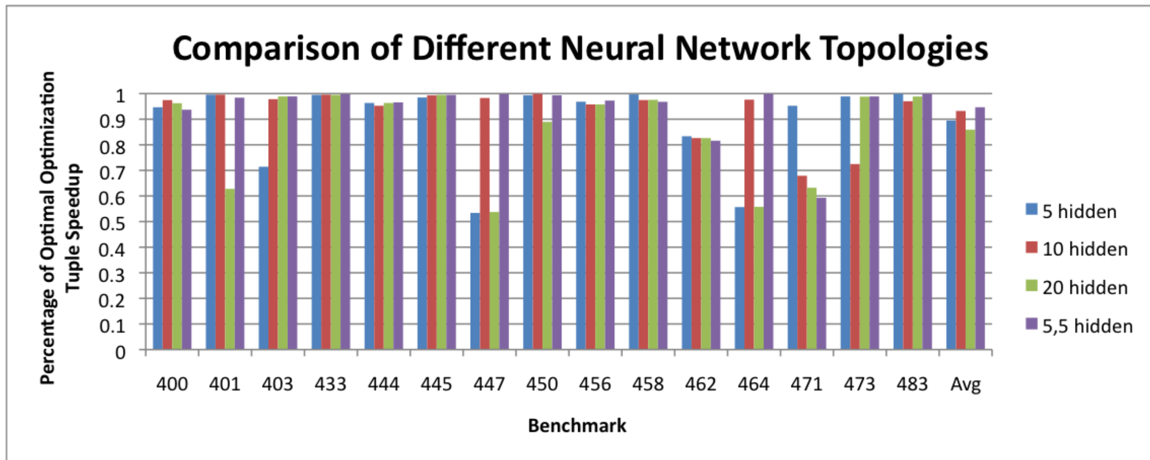

Figure 4: A comparison of the performance of different network topologies on the constrained optimization problem. Three network topologies with a single hidden-layer of 5, 10, and 20 nodes are shown, as well as a topology with two hidden-layers of 5 nodes each. The final column is the average performance of the network topology across all of the held-out data experiments.

a Leave-One-Out-Cross-Validation analysis, we find that these results are quite good and suggest this approach is reasonable.

5.2 Unconstrained Optimization Choices

Now that we have examined the task of picking the best optimization tuple from the 100 we ran, how well do the models perform when they can pick an arbitrary set of optimization parameters? In this set of experiments, for each program P we evaluate the network ensemble on the program features of P and *every possible optimization tuple.* Over these 2,025 points, we pick the optimization tuple that the ensemble estimates will result in the largest speedup-factor. Figure 5.2

Notice that the optimization tuples picked by the ensembles performs well compared to the best of the 100 we benchmarked. In one case (program 433), we pick a tuple which performs better than all of those we benchmarked. Furthermore, it's not the case that the ensembles simply pick a tuple that is one of the 100; in all of the 15 programs, a tuple *which is not* one of the 100 we benchmarked is chosen. This shows that the ensembles can make reasonable judgements outside of the set of optimizations over which they were trained.

6 Conclusions and Future Work

In this work we explore the possibility of using neural networks to estimate the speedup factor of programs under a limited set of optimizations. We find that a neural network with two hidden layers and a small number of hidden nodes can accurately estimate the speedup factor of programs under. Furthermore, by performing a leave-one-out cross validation experiment, we show that the method is robust and extensible.

The results we see show that the method is good at learning We also believe the method would benefit from having more training instances. In particular, the leave-one-out cross validation hurt the efficiency of code most in the case of the floating-point programs, where we had less training data (since we ran fewer of them than we did integer benchmarks). In future work, it would be interesting to train separate neural nets for different domains: for example, in this case, training one with the integer program data and another with the floating point training data.

In order for these results to be widely applicable, we believe a more assorted set of benchmarks is needed. Most importantly, we would need to consider a wider class of optimization regimes in which there is more freedom in picking optimization passes.

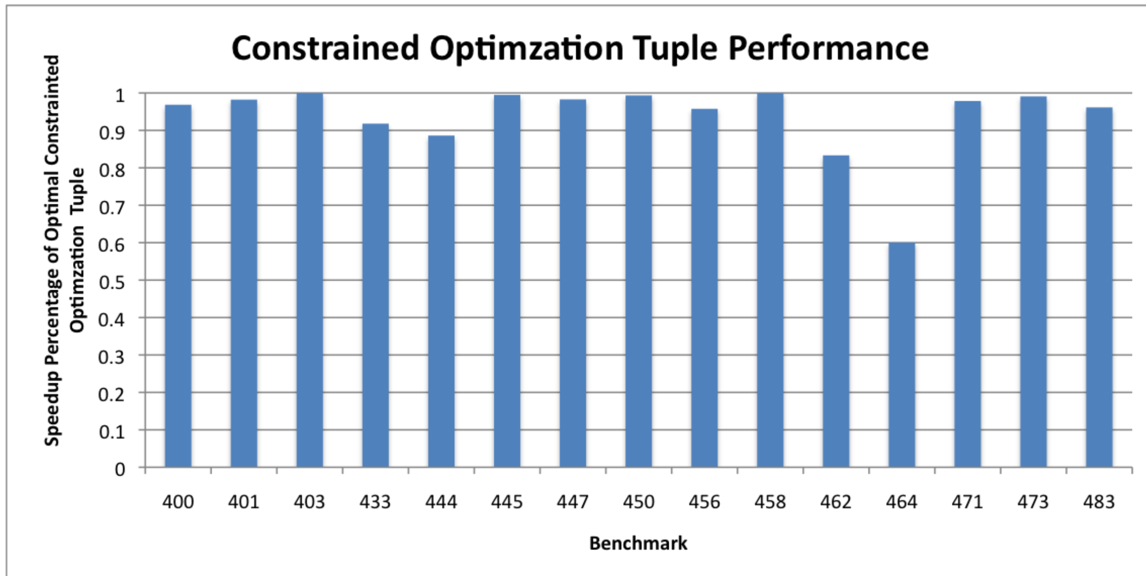

Figure 5: Performance of the network ensembles in estimating the best optimization tuple among the 100 we benchmarked. Each benchmark is labeled, and the vertical axis is the percent of the best speedup factor (across all 100 tuples) attained by the tuple chosen by the ensemble. A value of 1 means that the ensemble pick the best optimization tuple for that program.

7 Post-mortem and Final Notes

We propose distributing the credit as 28% to Richard, and 36% to Brendan and Jamie. Jamie should be praised for wrestling with the SPEC framework and LLVM. Our intention was to avoid the time-consuming tasks of dealing with LLVM internals and large codebases; however, it eventually became apparent that the setup outline in the proposal involved such work. Richard contributed the feature extraction passes. Brendan was responsible for doing all regression analysis, optimization, and other MATLAB / Excel / Powerpoint graphics wizardry. Each group member contributed the text for their corresponding section, and all group members edited this document.

We certainly underestimated the difficulty of getting SPEC and LLVM to play nicely together and do our bidding. Generally speaking, we are theory people and are afraid of MAKE. In hindsight we should have asked for assistance so that we could run a more extensive suite of tests.

The accuracy with which the ensemble of networks estimates the speedup factor is also surprising, especially given the limited variety of programs used for training. This gives us hope that the method could work well with an unconstrained sequence of optimization steps.

References

[1] 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 *In Proceedings of the International Symposium on Code Generation and Optimization (CGO*, pages 295–305, 2006.

[2] Christopher Bishop. *Neural Networks for Pattern Recognition.* Oxford University Press, USA, 1 edition, January 1996.

[3] Hugh Leather, Edwin Bonilla, and Michael O'Boyle. Automatic feature generation for machine learning based optimizing compilation. In *Proceedings of the 7th annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '09, pages 81–91, Washington, DC, USA, 2009. IEEE Computer Society.

[4] Gennady Pekhimenko. Machine learning algorithms for choosing compiler heuristics, 2008.

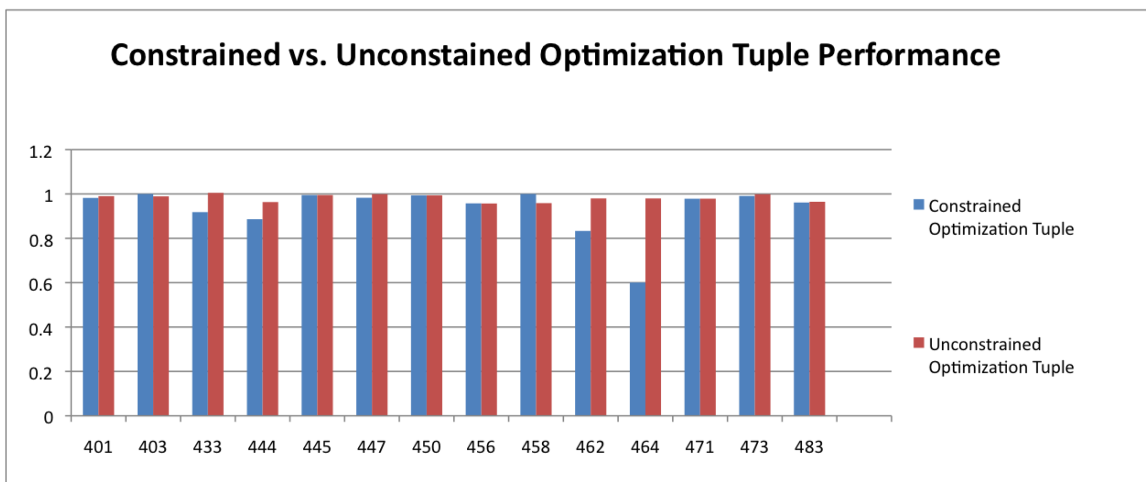

Figure 6: Performance of the best optimization tuple, as estimated by the network ensemble, compared to the speedup-factor of the best optimization tuple over the 100 we benchmarked. For comparison, we also show the competitive ratio attained by the ensembles when the choice is constrained to one of the 100 tuples we benchmarked.