An Automatic Compiler Optimizations Selection Framework for Embedded Applications

Shih-Hao Hung1,2, Chia-Heng Tu1, Huang-Sen Lin2, Chi-Meng Chen1
1Graduate Institute of Networking and Multimedia
2Department of Computer Science and Information Engineering
National Taiwan University
Taipei, 106 Taiwan

{hungsh, d94944008, r94922102 and r94944018}@csie.ntu.edu.tw

Abstract

Optimizing compilers provide users with compiler options to maximize program performance. The selection of compiler options is important as the resulted performance can vary significantly. The best combination of compiler options is not only dependent on the program itself, but it also is highly related to the configuration of the system and the architecture of the processor that the program runs on. The determination of the best combination of compiler options is very complicated, as its complexity grows exponentially with the number of the optimization options the compiler offers. Many previous work attempts to shorten the search time by reducing the complexity of the problem. However, most of them focus on computational intensive applications, which run with little or no invocation of kernel functions and device input/output activities, which often dominate system performance in specific embedded environment, such as network appliance.

This paper aims at system-wide compiler optimizations selection for embedded applications. We proposed an automated framework to judiciously select the compiler options not only for the control software in the user space but also for the associated kernel functions which perform the I/O operations for an embedded application. For this framework, we implemented compiler optimization selection algorithms and evaluated its efficiencies with and without performance monitoring hardware support. We argue that our framework is a platform-independent and system-level compiler options selection framework. Our experience in optimizing the performance of the embedded application on a production storage appliance show that an I/O-intensive application composed by various kernel modules device drivers under Linux can be optimized effectively and systematically.

1. INTRODUCTION

Modern compilers provide various optimization options for users to choose to obtain higher program performance. Proper selection of optimization options is not easy, especially for average users who do not have in-depth understanding of the compiler options, the effects of the optimizations and the interactions among the options. Therefore, most compliers offer a few optimization levels, and each optimization level performs a certain code analyses to determine the settings for a fixed set of optimization options. For example, three optimization levels, -O1, -O2 and -O3, are provided by the GNU Compiler Collection (GCC) [1]. To obtain the best performance, a user usually applies the highest optimization level, -O3, to have the compiler perform the most extensive code analysis and expects the compile-generated code to deliver the highest performance.

Studies have been made on compiler option searching algorithms to select a set of compiler options for optimal performance. However, most of these works were developed to optimize user-space programs and computer-intensive codes [2-10], rather than system software (control-plane codes). As Today’s embedded applications often handle input/output (I/O) activities with concurrent system operations, the overall system performance may be dominated by the device-dependent (driver) operations and the operating system (kernel) functions. For example, a control software usually comes with user codes and system software to form a complete embedded application. Thus, to optimize the embedded application, both layers of software need to be optimized as a whole.

In this paper, we propose to optimize embedded applications by setting compiler optimization options properly via a framework which extends the previous compiler options search methods to cover multiple layers of software. The framework contains tools which help a developer conduct a heuristic search to find a set of compiler options for the entire application to perform well: During the run time of the targeted application, a profiling tool may be deployed by the framework to record detailed performance metrics; the framework then adopts a compiler options selection method to decide on the optimization options for the compiler to produce a new binary code for the following execution. The procedure is repeated
automatically to refine the options and finds the best set of options eventually.

To demonstrate the ability of the framework, we first implement a quick prediction based on the Logistic Regression Model (LRM) [10], which finds a set of optimization options for the application immediately by matching the application’s runtime profile, captured by hardware monitor devices, against the knowledge database that was established via machine learning. Then, we implement the Combined Elimination (CE) algorithm [3] and improve the CE algorithm to do the same thing without underlying hardware support. We show that our framework is platform independent and can be applied on various hardware platforms either with or without hardware counters support.

The paper is organized as follows: Section 2 of this paper gives the background of this research by introducing the previous works, which focus only on user-space programs. We have not yet seen any work which tackles kernel function and I/O drivers, which are major components in an embedded application. Section 3 describes a framework which includes existing search algorithms. Section 4 highlights the algorithms implemented in the framework. Section 5 summarizes the results in this paper and the conclusion of this paper is given in Section 6.

2. RELATED WORK

The PEAK System [2] implements an automatic performance tuning methodology that divides a program into several tuning sections and applies the CE algorithm to each section to find out the best set of compiler optimization options. The CE algorithm, proposed by Pan [3], first combines all the compiler optimization options that have negative effects to the target program together as a set, and then eliminates those options which have the most negative effects mostly within this set. The CE algorithm reduces the original search space from $O(2^n)$ to $O(n^5)$, where $n$ is the number of compiler optimization options which can be turned on or off. Consider that the number of optimization options is 43 for the -O3 optimization level of GCC compiler (version 3.4.2), the CE algorithm reduces an enormous search space to a manageable size ($43^5 = 1849$).

The FFD—fractional factorial design [9] approach divides options into several groups to slice the search space. For instance, in the above case of GCC, the FFD approach would divide the 43 optimization options into five groups, and requires that the options in one group have to be turned on or off all at the same time. Thus the initial search space is $2^{\text{number-of-groups}}$ ($2^5 = 32$) in this case. After finding the best combination from this initial search space, a group can be divided further into multiple groups in case the group has a strong impact on the performance. This method eventually finds a set of options by starting with few groups and continuing to divide the groups with strong impact.

Cavazos et al. [10] proposed a method to construct a mathematic model and predict the best set of compiler optimization option by making use of the features of a program extracted from the performance counters residing in the processor. The features of programs and their compiler optimization options are kept in a database. For a new program, its features are extracted from the performance counters, and the database is searched to find the record whose features match closely with the new program. This approach takes a relatively short time to provide the compiler optimization options to the user.

The method proposed by Agakov et al. [11] uses machine learning to reduce the huge search space. Programs which have been optimized are categorized into groups based on their characteristics, e.g., nested for-loop, non-linear array access, etc. A database is built to keep track of the best compiler options for the programs in a group. For a new program, the database finds its initial set of options based on the program’s characteristics. This method shortens the search time by suggesting a good starting point for the search.

3. AN AUTOMATIC PERFORMANCE TUNING FRAMEWORK

In this section, we describe the general design of the framework that we used to apply automatic compiler options selection algorithms for embedded applications. During the development process, the framework tries to optimize the application performance by selecting compiler options in building kernel modules, drivers, and user-space programs, i.e., the full blown embedded applications. During the runtime, system performance is monitored and analyzed to improve the compiler options for the compiler to compile the entire application for the next at host machine. The compiler options selection algorithm chosen by the user will decide how to improve the options and whether to stop the process based on the performance results.

Figure 1 illustrates the process of automatically selecting compiler options for kernel-space modules. The steps in the figure, from steps 1 to 10, are performed in timing sequence. For steps 1 to 5, the kernel image and modules are compiled at the host machine and loaded into the client machine (target system) for performance analysis. Then, at step 6 and 7, performance testing is performed and performance data is stored. Precisely, the performance data is IO operations per second (IOPS), and throughput.
(MB/sec) for a network-attached storage server. After the experiments, in step 8 and 9, the implemented algorithm determines proper compiler options for the next run if it is necessary. Otherwise, a suitable options combination has been found. This automated process is mostly done by the client machine by our tool except that our tool sends compiler options to the host machine to get the code recompiled by the compiler on the host machine.

In our experiment, the host-client communications and data exchanges were done by NFS (Network File System) [12] via a private Local Area Network (LAN). NFS allows users from one computer to access files resident in computers over the network as if the network devices were attached to its local disks.

To tune the application in user space, we have written a simple script to run the program, obtain the performance data and decide whether to terminate the selection process based on current algorithm. Different from user-space applications, for kernel-space applications (e.g., operating systems and kernel modules), selecting a good software version toward better performance requires rebooting system and acquires different software version (kernel image) across the LAN from the host machine. To facilitate this process, we setup an environment to conform PXE (Pre-Execution Environment) network booting protocol [13]. Following the protocol, the client machine will first initialize BIOS, perform DHCP request and PXE broadcast over the network, and download the PXE environment (and configurations) once connected with host machine (PXE server). Based on the pre-programmed configurations, the client machine can then boot from the updated kernel version and gather performance data during this run. After that the client machine is told to reboot and perform the above processes until the termination condition of current algorithm is met. In addition, to automate the whole options selecting process controlled by the client machine after booting up, auto login console in linux and SSH login without password [14] are used in our framework.

![Proposed Framework for Automatic Compiler Options Selection of Control Software on Embedded System](image)

**Figure 1.** The flow chart for optimizing an embedded Linux-based network appliance with our framework.

### 4. COMPILER OPTIONS SELECTION ALGORITHM

Within the automatic framework, the algorithm used to find suitable compiler options is important, as it affects the total search time and the code quality. An ideal algorithm results in the best performance in short time. However, it is difficult for existing algorithms to achieve both goals. In addition, some algorithms require hardware assist. Thus, we integrated several algorithms into our framework to accommodate the requirements from the users, the
applications, and the hardware platforms. In this section, due to page length limitation, we briefly describe the usage of the two compiler options selection algorithms used in our framework, the Logistic Regression Model (LRM) algorithm [3] and the Combined Elimination (CE) algorithm [10]. As the LRM depends on capturing program features with hardware monitoring facilities built in processors, the CE algorithm might be a favor choice on embedded processors without hardware monitoring functions. In addition, regarding the redundant procedures performed in CE algorithm, we propose and implement a modified version of CE algorithm to shorten the time in finding suitable compiler options for programs.

4.1 Logistic Regression Model (LRM)

The Logistic Regression Model [10] is a machine learning technique used to reduce the complexity of selecting suitable compiler options for programs running on a specific hardware platform. For a large program, finding the optimal set of compiler options can take an enormous amount of time. To shorten the search time, LRM poses a different approach to address this problem.

The main idea of this approach is to collect correlation between program features and compiler options upon specific hardware platform at the training stage. The correlation information is then used to build a model and used later to guide compiler options selection whenever new programs arrive. Thus, selecting suitable compiler options is then automated.

Furthermore, to collect correlation between program features and compiler options is to learn the relationship between the pair of vectors, Performance Results (PR) and Compiler Options (CO). Performance Results refer to normalized values of performance counter events and implication of the speedup of the program while Compiler Options denotes specific combination of compiler optimizations with respect to the obtained PR. The relationship is learnt offline and served as a database, which is looked up for optimal configuration when a new program (executable) arrived. For using the model, when a new program is arrived, the program needed to be run once with the baseline compiler options set, -O0. The program results are obtained from the single run. The baseline compiler flags configuration along with program performance results on this single run is feed into pre-trained model to select suitable compiler options configuration for the new program. The concept is illustrated in Figure 2.

![Figure 2. Logistic Regression Model for compiler options selection.](image-url)

4.2 Combined Elimination Algorithm (CE) and Modified Combined Elimination Algorithm (Modified CE)

The Combined Elimination algorithm identifies and eliminates the optimizations with negative effects iteratively [3]. By the feedback-directed approach, CE algorithm can find a near-optimal set of compiler options. In the CE algorithm, Relative Improvement Percentage (RIP) is used to measure the impact of one compiler option on program performance. The definition of program performance compiled with specific compiler options refers to the execution time of the program, the program code
size, the power consumption or etc. In this work, we use RIP to measure the performance of IOPS with and without one optimization, $F_i$, in a set of options, $B$, which are denoted as $M_B(F_i=1)$ and $M_B(F_i=0)$, respectively. Further, $B(F_i=0)$ expresses in a set of compiler option $\{F_1=1, F_2=1, ... , F_i=0, ..., F_n=1\}$, where $n$ is the number of compiler options. The RIP is defined as formula:

$$ RIP(F_i) = \frac{M_B(F_i=0) - M_B(F_i=1)}{M_B(F_i=1)} $$ (1)

This formula indicates that the RIP of compiler option, $F_i$, is the performance effects given by turning off the option, $F_i$, in a set of compiler option $B$. For example, $RIP_B(F_i)<0$ means $M_B(F_i=1) > M_B(F_i=0)$. Namely, for the program, the performance improves relative to baseline when user turns off the $i$th compiler flag in option set $B$. Note that the baseline of this algorithm applies all optimizations.

To avoid redundant computation in CE algorithm, we propose a modified CE algorithm. Experiments show no performance degradation while we using the modified CE algorithm. Moreover, the proposed modification improves search time by 4% relative to the original CE algorithm, which results from removing redundant computation of $RIP_B(F_i)$ in the original CE algorithm, if the $RIP_B(F_i)$ in the previous run has been calculated. The corresponding modifications in bold font are given below.

**Algorithm 1 Modified CE Algorithm**

**Input variables:**
- $B$: the baseline (default) configuration.
- $F_i$: the configuration of $i$th compiler options (=on or off).
- $M_B$: performance of the baseline option combination.
- $n$: the number of compiler options available in the experiment.
- $SS$: search space of compiler options selection (=2$^n$, while available operations for those $n$ flags are either on or off).

**Output variables:**
- $B$: represents selected compiler options set.

**Steps:**

1. Setup $SS=\{F_{i_1}, F_{i_2}, ..., F_{i_n}\}$ and $B = \{F_1=1, F_2=1, ..., F_i=1, ..., F_n=1\}$.
2. Compile the program with baseline configuration $B$ and measure the performance under baseline configuration, $M_B$.
3. Calculate $RIP_B(F_i)$ for each optimization $F_i$, where $i$ is in $SS$ set. Check if $RIP_B(F_i)$ has already computed in the step 3 from previous iteration. If yes, use the pre-computed value.
4. Repeat 2 and 3. When $RIP_B(F_i) \geq 0$, for each option, $F_i$, in $B$, go to step 5.
5. $B$ represents the selected compiler option set.

**5. EXPERIMENTAL RESULTS**

This section demonstrates the application of the proposed framework on a Linux-based embedded storage server application. The target storage server is a production model from the Quanta Computer Inc., which supports RAID-0 disk arrays and fiber channel interfaces. The performance bottleneck is caused by the processor on the controller, which spends most of its time in executing the RAID-0 kernel module, handling file system operations, and driving the fiber channel interfaces. The entire software collection poses a great challenge for the programmers to optimize the compiler options manually, as it contains many kernel modules and the kernel modules and drivers can execute asynchronously and concurrently. Thus, it makes a good target to optimize with our compiler optimization framework.

We first list the experiment environment and describe the implementation of three heuristic algorithms. Then, we introduce the performance metric used to evaluate the storage server and our experience in optimizing the performance of the embedded application on a production storage appliance. We show that an I/O-intensive application composed by various kernel modules and device drivers under Linux can be optimized effectively and systematically.

**5.1 Experimental Environment**

Table 1 lists the settings in our experimental experiment. Two machines were used: host and client, referring to development PC and target storage server, respectively. Performance measurement was done on the client side with various performance tools such as Intel’s VTune Performance Analyzer [15] and our workload generator,
Internal IO Request Generator (IRG). 43 compiler options provided by GCC 3.4.2 are considered in this experiment. Detailed descriptions can be found in the GCC documentation [1].

Table 1. Experimental environment and benchmarks.

<table>
<thead>
<tr>
<th>Machine Description</th>
<th>Host Machine (Development PC)</th>
<th>Client Machine (Storage Server)</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU Model</td>
<td>AMD Athlon 64 3000+</td>
<td>Intel Pentium D 2.80 GHz</td>
</tr>
<tr>
<td>Memory Size</td>
<td>1GB DDR-333 (256 MB for OS, 768MB for Disk cache)</td>
<td>1 GB DDR2-533</td>
</tr>
<tr>
<td>Linux Kernel Version</td>
<td>2.6.15</td>
<td>2.6.15</td>
</tr>
<tr>
<td>Compiler</td>
<td>GCC 3.4.2</td>
<td>GCC 3.4.2</td>
</tr>
<tr>
<td>Performance Analyzer</td>
<td>Internal IO Request Generator (IRG) and Intel VTune Performance Analyzer 8.0</td>
<td></td>
</tr>
<tr>
<td>Testing Configuration</td>
<td>Read IO requests hit in the Memory (disk cache)</td>
<td></td>
</tr>
<tr>
<td>Workload</td>
<td>I/O requests generated by IRG</td>
<td></td>
</tr>
<tr>
<td>Benchmarks to build our LRM model</td>
<td>SPEC CPU 2000</td>
<td></td>
</tr>
</tbody>
</table>

We have implemented the Logistic Regression Model (LRM), the Combined Elimination (CE) algorithm, and modified Combined Elimination (Modified CE) algorithm in the proposed framework. For LRM algorithm, we ran the SPEC CPU 2000 benchmarks on the client machine and used the performance data collected by Intel VTune as the training data set to build the model. After the training period, the model is used to suggest the best optimization options for the application under different workloads, or for new applications. Note that the programs used to build the LRM model are critical to the applications of the LRM model. For simplicity, we chose to build the LRM model with the SPEC CPU 2000 benchmarks, and then we validated the functionality of our LRM model with the Media Bench II benchmarks.

We implemented CE and Modified CE algorithms based on [3] and Algorithm 1, respectively. During the option selection stage, the algorithm finds a good set of compiler options for the target system based on a single performance index, i.e., the average IOPS, as reported by IRG. We do not compare our performance results with that reported by [3], as its system configuration was different from what was used in our experiment. We validated the correctness of our algorithms by comparing the results generated by the modified CE algorithm to the results generated by the original CE algorithm. We found that the results generated by both algorithms are identical in the benchmarks that we evaluated.

5.2 Performance Results and Analysis

Figure 3 depicts the performance of the storage server in terms of IOPS when using different compiler options selection algorithms. All three search algorithms provided by our framework delivered better results than the compiler’s -O3 option setting. In this case, the performance improvement delivered by the CE algorithm was better than that obtained by the LRM. Some previous work, e.g., [10], reported that the LRM outperformed the CE in their case studies. We should not be too surprised to see this, as both search algorithms are heuristic, which is also why we decided to integrate several algorithms into our framework. However, it is interesting to note that some kernel modules behave quite differently from user-space programs. In our experiments, we trained the LRM model with the SPEC CPU 2000 benchmarks which run in the user space. This may explain why the LRM delivered less performance than the CE algorithm for our storage server application. With more training, the LRM may have the potential for improvement. Further, this implies that adopting LRM algorithm may require additional domain knowledge on both machine learning algorithms and the understanding/supporting of underlying system, including the software and underlying hardware.

Table 2 compares the average performance, the time required for each search algorithm to complete, and the relative speed of the search, from the storage server application delivered by our framework with different search algorithms. We show that, with less than 1% time, the LRM algorithm was much faster than the CE algorithm but achieved 98% of the performance obtained by the CE algorithm. In this experiment, the CE algorithm requires 34.5 hours to find the compiler options set; in contrast, our Modified CE algorithm delivers the same performance with 4% reduction of the search time.

We implemented CE and Modified CE algorithms based on [3] and Algorithm 1, respectively. During the option selection stage, the algorithm finds a good set of compiler options for the target system based on a single performance index, i.e., the average IOPS, as reported by IRG. We do not compare our performance results with that reported by [3], as its system configuration was different from what was used in our experiment. We validated the correctness of our algorithms by comparing the results generated by the modified CE algorithm to the results generated by the original CE algorithm. We found that the results generated by both algorithms are identical in the benchmarks that we evaluated.

In summary, the LRM requires a large amount of historical performance data to produce a model that may be used to suggest optimization options for by matching program features; it may be preferred in the cases where
quick search is needed and may be further used to improve application performance dynamically on-site based on the input. On the other hand, the CE algorithm is easy to apply on target system regardless of the capability of underlying hardware. This could be important when the target embedded processors are unable to provide sufficient/accurate performance profiles for LRM model due to lack of hardware performance monitoring support. Under the circumstance, the Modified CE algorithm becomes the primary choice.

### Table 2. Speedup of compiler options search time.

<table>
<thead>
<tr>
<th></th>
<th>CE</th>
<th>Modified CE</th>
<th>LRM</th>
</tr>
</thead>
<tbody>
<tr>
<td>Application Performance</td>
<td>1</td>
<td>1</td>
<td>0.98</td>
</tr>
<tr>
<td>Search Time (hours)</td>
<td>34.5</td>
<td>33.2</td>
<td>0.32</td>
</tr>
<tr>
<td>Relative Speed</td>
<td>1</td>
<td>1.04</td>
<td>106.6</td>
</tr>
</tbody>
</table>

6. CONCLUSION AND FUTURE WORK

In this paper, we proposed a framework to facilitate performance tuning for embedded applications which may invoke kernel functions and device I/O operations. We discussed three compiler options selection methods which were implemented in our framework to effectively support various applications. We presented our results with a full blown storage server application. The results showed that the compiler options selection algorithms delivered good performance for our kernel-space application with different search time. On a platform which has hardware performance profiling support, the user may choose the faster LRM algorithm to obtain the result quickly. Without performance profiling support, the user can still use the Modified CE algorithm provided in the framework. By removing redundant computations, our Modified CE algorithm reduced the options set search time by 4% with respect to the original CE algorithm.

In the future, we hope to incorporate more option selection algorithms and figure out ways to combine and apply the algorithms effectively and systematically with our framework.

7. ACKNOWLEDGEMENTS

This work was supported in part by a grant from the Quanta Computer Inc., a grant from the National Science Council (95C2443-2), and a grant from Excellent Research Projects of National Taiwan University (96R0062-AE00-07).

8. REFERENCES


