A Genetic Algorithm to Improve Linux Kernel Performance on Resource-Constrained Devices

James Kukunas
jkukunas@acm.org

Robert D. Cupper
bcupper@allegheny.edu

Gregory M. Kapfhammer
gkapfham@allegheny.edu

Department of Computer Science
Allegheny College

ABSTRACT

As computers become increasingly mobile, users demand more functionality, longer battery-life, and better performance from mobile devices. In response, chipset fabricators are focusing on elegant architectures to provide solutions that are both low-power and high-performance. Since these architectures rely on unique $x86$ extensions rather than fast clock speeds and large caches, careful thought must be placed into effective optimization strategies for not only user applications, but also the kernel itself, as the typical default optimizations used by modern compilers do not often take advantage of these specialized features. Focusing on the Intel Diamondville platform, this paper presents a genetic algorithm that evolves the compiler flags needed to build a Linux kernel that exhibits reduced response times.

Categories and Subject Descriptors: D.3.4 [**Programming Languages**]: Processors-*compilers, optimization*

General Terms: Performance

Keywords: Genetic Algorithm, Linux Kernel, Compilers

1. INTRODUCTION

Resource constrained mobile devices, such as netbooks, have become increasingly popular due to their low cost and minimal size and weight. Many motivations exist for optimizing the Linux kernel for netbooks, such as increased battery life, reduced heat generation, and improved kernel performance. Intel offers an optimizing C compiler that is capable of performing architecture-specific optimizations for Intel chipsets, including the Intel Atom processors currently available in most netbooks. The LinuxDNA project patches the Linux kernel to build with the Intel C compiler, thus allowing the kernel to take advantage of these architecture-specific optimizations. Initial results from LinuxDNA found up to a 40% increase in performance within the kernel [4].

Careful consideration must be used when choosing compiler optimizations. While certain optimizations complement other ones and provide further opportunity for performance gains, some hinder the performance of others and thus can lead to kernels that are both larger and slower. The most straightforward approach to solving this problem is to perform exhaustive compilation by building each possible kernel and then selecting the best. However, this is not a feasible approach due to the size of the search space. The system described by this paper examines 107 different compiler flags, which would yield 2^{107} different kernels. We use a genetic algorithm (GA) to perform a focused search, thereby reducing the time to find good compiler flags.

Previous work using GAs to evolve compiler flags has seen significant success. Cooper et. al. presented a genetic algorithm to evolve compiler flags to reduce the space overhead of code and achieved up to 40% smaller code size [1]. Davidson et. al. presented a genetic algorithm to evolve optimization orderings within the VPO compiler on the ARM platform, and achieved near-optimal performance [3]. Unlike previous work, this paper focuses on a new and currently unstudied architecture, the Intel Diamondville, and on a different compiler, the Intel C Compiler, and uses response time as an optimization metric rather than code size.

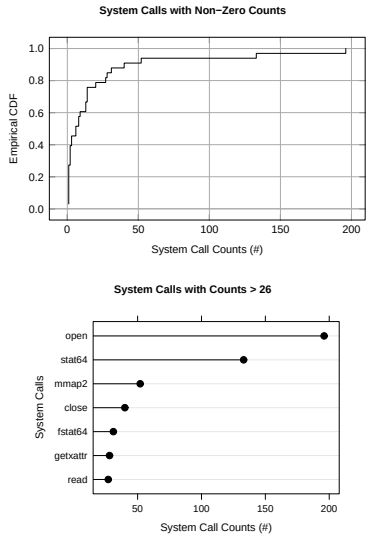
The Intel Diamondville is a micro-architecture designed for netbooks and mobile devices. The Diamondville includes the Intel Atom $n270$ processor and the $945GME$ chipset. Operating at a steppable frequency of 1.6 GHz and a front-side bus speed of 533 MHz, the Atom $n270$ operates significantly more slowly than recent Intel processors. The $n270$ utilizes an in-order instruction scheduler designed to reduce the footprint, heat generation, and power consumption of the processor. As a result, the $n270$ is more vulnerable to dependency stalls in the pipeline that are caused by poorly placed instructions. Thus, it becomes the compiler's job to emulate the pipeline and produce an efficient instruction ordering. Since most $x86$ processors contain out-of-order instruction schedulers and many compilers do not perform this optimization by default, instruction throughput may suffer when programs are compiled for the $n270$ [2].

2. GENETIC ALGORITHM

Depending on the compiler flags, processor, and kernel configuration, compiling a Linux kernel can take anywhere from 30 to 60 minutes. When designing the GA, which needs to build a large number of kernels in order to evaluate fitness, performance was a high priority in the design.

Fitness. To properly measure kernel performance the fitness operator only considers the time spent in the kernel. Since user applications interact with the kernel through system calls, timing these operations can adequately character-

GECCO'10, July 7–11, 2010, Portland, Oregon, USA.



Only 33/330=10% of System Calls Have Non-Zero Counts

Figure 1: Profile Data Used in Fitness Evaluation.

ize kernel performance. With over 300 system calls available, the GA focuses on those that that are frequently invoked by user programs. The GA framework uses the `ptrace` profiler to determine which system calls to consider during fitness evaluation. As input, the profiler takes a set of executable(s) for which the user wants to improve performance. Next, the profiler runs every executable and determines the total counts for each system call.

After profiling a small number of commonly used applications (e.g., `ls`, `mkdir`, and `du`), Figure 1 shows that a small number of system calls are frequently referenced, with only 33 system calls having a non-zero invocation count. The top graph in Figure 1 gives an empirical cumulative distribution function (ECDF) that shows the range of counts for the non-zero calls. The ECDF curve represents the probability that the count is less than or equal to a specific value on the horizontal axis. The ECDF shows that over 80% of system calls have an invocation count less than or equal to 50. The dot plot in Figure 1 also reveals that, for the 7 system calls with a count greater than 26, user programs most often call `open`. For this paper, the GA's fitness function focuses on `open`, `close`, `fstat64`, `read`, and `write`, although we intend to investigate others in subsequent studies.

The fitness operator times these system calls for all individuals in the population. Each individual is first searched for in a global lookup table, which persists between GA runs. This lookup table stores already evaluated compiler flags and their corresponding fitness, in order to obviate duplicated kernel compilations. If the fitness is not available in the lookup table, the compressed kernel image is built, and then sent to a netbook with the `System.map` file. After the netbook installs the new kernel and reboots into it, the fitness function uses the chosen system calls as microbenchmarks. By building and sending both the compressed kernel image and the `System.map` file, as opposed to the whole kernel, the GA eliminates approximately 30 minutes of compression, archive, and network transfer time. These optimizations, along with direct memory access network transfers and parallel compiler construction, support kernel compilation and installation in approximately 6 minutes, thereby improving the response time and throughput of fitness evaluation.

Kernel	Fork (ns)	Mmap (ns)	Socket (ns)
Fedora Default	254212	33552	31187
Evolved Kernel	191953	23498	25959
% Reduction	24.49%	29.97%	16.76 %

Table 1: System Call Performance Results.

Initialization. The initialization phase of the GA focuses on two main goals: parsing the chosen compiler flags and creating the initial population. The compiler options file follows a simple grammar in which each line is either a compiler option followed by a newline or an "EITHER m," where the next m lines contain compiler options which cannot be used in tandem. After parsing the options, the GA creates a random population where every member is represented by a binary string. Each bit in the binary string determines whether a specific compiler option is either on or off. For example, if we had three compiler options, O_0, O_1, and O_2, an individual, say 101, would correspond to using both optimization O_0 and O_2 together, since bit 0 corresponds to the first optimization and bit 2 corresponds to the third. Binary strings were chosen due to their efficient structure and low overhead, thus enabling each individual to be $\lceil n/8 \rceil$ bytes long when the GA considers n compiler flags.

Selection, Reproduction, and Mutation. The selection operator uses truncation to pick the top 75% of the population as candidates for crossover. Returning the population to the specified size, crossover takes the higher bits of the first parent and the lower bits from the second parent and combines them by using a bitwise `or` operator. With a 10% chance, the mutation operator reads a random byte from `/dev/urandom` and uses a bitwise `or` operator to combine it with a random byte from the chosen individual. The GA runs until it completes the specified number of generations.

3. PRELIMINARY RESULTS

The genetic algorithm was run with a population size of 10 for 5 generations. As shown in Table 1, we compared the performance of the evolved kernel with the default Fedora kernel by measuring the time overhead of three frequently invoked system calls not used in the GA's fitness function. The kernel resulting from the genetic algorithm performed approximately 17% to 30% better than the default Fedora kernel. This result stems from the fact that the GA selected compiler flags such as `-axSSE3`, which utilizes SSE instructions, `-vec`, which vectorizes loops, and `-vec-guard-write`, which avoids unnecessary stores during vectorized loops. It is anticipated that the profiling and inclusion of additional system calls and increasing both the population size and the number of generations will confirm these early results. We intend to conduct further experiments in order to study how well a wide variety of GA configurations produce Linux kernels that improve user-perceived performance for commonly executed netbook-based software applications.

4. REFERENCES

[1] K. D. Cooper, P. J. Schielke, and D. Subramanian. Optimizing for reduced code space using genetic algorithms. In *Proc. of LCTES*, 1999.

[2] Intel. *Mobile Intel Atom Processor N270 Single Core Datasheet*, May 2008.

[3] P. A. Kulkarni, D. B. Whalley, G. S. Tyson, and J. W. Davidson. Practical exhaustive optimization phase order exploration and evaluation. *Trans. Archit. Code Optim.*, 6(1):1–36, 2009.

[4] J. Ryan. LinuxDNA supercharges Linux with the Intel C/C^{++} compiler. *Linux Journal*, Feb. 2009.