# Evaluating power-aware optimizations within GCC compiler

Dmitry Zhurikhin[1], Andrey Belevantsev[1], Arutyun Avetisyan[1], Kirill Batuzov[1], and Semun Lee[2]

[1] Institute for System Programming, Russian Academy of Sciences
{zhur,abel,arut,batuzovk}@ispras.ru
[2] Samsung Corp.
semun.lee@samsung.com

**Abstract.** This paper summarizes the results of evaluating several optimizations targeted at reducing power consumption for ARM architecture within the GCC framework. The optimizations tried were off-line dynamic voltage scaling (DVS) and bit-switching minimization. Also, we have experienced with tunings of memory-related GCC optimizations.

## 1 Introduction

Research on power consumption optimizations for embedded systems is GROWing every year. Important approaches that can be named include dynamic frequency and voltage scaling (DVS), various memory access optimizations, power-aware chip design. However, there is not much work done on this in the GCC compiler. It is also unclear which approaches are the most promising within the GCC framework.

To overcome this issue, we have completed a project sponsored by Samsung Corp. on evaluating the most popular approaches to compiler-directed power consumption optimizations for ARM architecture and GCC. We have tried three approaches: off-line profile-based dynamic voltage scaling, bit-switching minimization within instruction scheduler and combiner, and tuning of memory-related optimizations – prefetching and loop optimizations. We have evaluated our code on two ARM-based test boards, OMAP2430 [5] and MV320 [4], provided by Samsung. Our results show that DVS is the only optimization that provides at least some power savings within GCC for the test boards we were using. Power-aware loop optimizations are also promising, but for real work with them the Graphite framework [2] is required, which was not available during the course of this project.

We selected three sets of tests for use in experiments on power consumption. They are the Aburto test suite, representing many common scientific applications, the MediaBench test suite, representing applications working with audio-video and image processing, and MiBench, representing other application types common for the embedded systems. The latter two test suites are specifically aimed at the embedded systems, while the Aburto test suite tends to be system-independent. We have removed some tests from the MiBench suite that are also present in MediaBench. We have also prepared new data inputs for remaining MediaBench tests because the supplied data sets were not representative enough as it only took fractions of a second to execute most of the tests.

Rest of the paper is organized as follows. Section 2 reports on our implementation of a DVS approach and experimental results on the MV320 board. Section 3 describes our usage of instruction encodings for bit-switching minimization. Section 4 explains our experiments with prefetching, loop transformations, and vectorization. Section 5 concludes.

## 2 Dynamic Voltage Scaling

The basic idea of DVS is to change the voltage of the power supply unit in some places (called power management points, PMPs) during program execution, such that energy consumption decreases but the needed performance is still maintained. This is possible because energy consumption has quadratic behavior depending on voltage, and processor frequency (and hence performance) depends on voltage only linearly.

There are several types of DVS approaches, namely on-line, off-line and mixed. They differ in moments at which decisions are made for placement of PMPs and for the value of voltage change. On-line approaches make all decisions while executing the program, and off-line approaches make them while compiling the program. Mixed approaches calculate possible PMPs at compile-time, but the values of the voltage change are computed at run time.

During the project we implemented an off-line DVS approach prototype based on [3]. The main idea of this approach is to insert PMPs in memory-dominated parts of code so that reduced performance of the processor will be hidden by latencies of memory accesses. The prerequisite of the approach is that the processor and the memory unit have independent power supplies. The placement of PMPs and the values of voltage change are calculated statically by analyzing the profile of the program. The approach is notable because it was tested on real hardware and considers many implementation details, such as energy and execution time costs of voltage-changing instructions. Many other approaches were only tested on simulators and provide only theoretical models of DVS optimizations.

Other studied off-line DVS-based approaches consist of some loop optimizations, such as loop rotating or software pipelining, with DVS mechanism applied to the optimized loops so that performance gain due to the loop optimization is counterbalanced by performance penalty of reduced CPU performance [1, 6]. We think that such DVS optimizations have just an effect of combining regular loop optimizations with voltage change that is done separately.

### 2.1 DVS algorithm implemented

The algorithm operates on basic and combined regions. A basic region is just a basic block or an outermost loop as reported by GCC loop analysis. A combined region is a single entry/single exit piece of code consisting of basic regions, which is dominated by its entry and postdominated by its exit. This definition, unlike forming a region from certain control flow patterns as proposed in [3], allows handling more general constructs while retaining the possibility of easy calculation of region properties. Still, there are several restrictions on the regions imposed solely by our implementation. First, a region should not contain any function calls, as the initial implementation is intraprocedural. Second, there should not be edges that cross loop boundaries within a region. Third, we do not consider regions that are too small (i.e., the number of instructions inside a region is less than a certain threshold). This is because the cost of changing frequency is usually quite high, and the region should be large enough to pay off for switching frequency on it. A loop region is always eligible for optimization.

The algorithm consists of five main steps:

- Construct basic and combined regions for a given function.
- For every basic region, profile overall execution time at each available processor frequency, $T(R, f)$, and the number of times a region is executed, $N(R)$.
- Estimate $T(R, f)$ and $N(R)$ for combined regions. $N(R)$ is taken from the basic region that is at the entry of the combined region $R$. $T(R, f)$ is computed as sum of $T(BR, f)$ over all basic regions $BR$ that form the combined region $R$.
- Find the best region (basic or combined) and its frequency which minimizes power consumption while performance (running time) is not decreased beyond given percent $p$ of original[3]. The value of $p$ can be set by the user.
- Insert frequency setting instructions at the entry and the exit of the selected region.

The algorithm (as many other DVS approaches) strongly depends on profiling results. The GCC compiler allows gathering profiling information and using it inside the compiler. By default, frequencies of basic blocks, frequencies of edges, and branch probabilities are profiled. Additionally, we modify the profiling mechanism to record execution time and counter for every basic block that is either large enough to be a basic region or is a member of a large enough combined region. For loops, profile code is inserted only before the loop and after the loop to minimize profile overhead. Basic block counters are devised from the existing profile information, and execution times are profiled via inserting appropriate system calls before and after profiled basic blocks. If possible, hardware counters are used for measuring execution times.

Current implementation does not support finding an optimal set of regions within a function for changing the frequency. Instead, we are looking for a single region in each function for CPU frequency change. Also, we chose to have just two CPU frequency modes for this prototype – standard mode with maximal CPU frequency and energy-saving mode with reduced CPU frequency. This choice simplifies the implementation, as we just need to iterate over all regions

[3] The formal problem to which we want to find a solution can be found in [3].

and to estimate the total power consumption of the program given that the frequency was reduced for the current region. This also reduces the time needed for profiling, as it should be performed for each CPU frequency separately.

Finally, the interface for changing frequency is implemented as builtins from the GCC side and the syscalls from the kernel side. On the MV320 board, the syscalls are equivalent to setting the appropriate operation points via modifying the `/sys/devices/system/cpu/cpu0/op` file.

## 2.2 Experimental results

We have tested our prototype implementation of the DVS compiler optimization on the Aburto test suite on the MV320 test board. We have removed from the test suite several tests that are self-calibrating and thus perform different amount of work depending on the CPU frequency. The basic optimization level used was `-O2`.

Aburto test suite contains 196 functions. Our prototype can find regions suitable for DVS in 144 functions. To find profitable regions, we have tried several parameters of power dissipation of standard mode and energy-saving mode, $P_{hi}$ and $P_{lo}$. We have succeeded in finding profitable regions (i.e., such that switching CPU frequency on them seems to be worthwhile given the measured profiling data) only with $P_{hi} = 13$ and $P_{lo} = 9$, i.e. $P_{hi}/P_{lo} = 1.4$. The number of the regions found depends on the value of performance degradation threshold, $p$. When $p$ varies from 10% to 40%, the number of the profitable regions increases from 3 to 14 and steadies there no matter how we increase $p$.

We made experiments to measure the power consumption of the optimized programs with the power meter and the MV320 board. The DVS optimized tests ran about 8 minutes and the power meter showed the overall energy consumed of 750 mWh. Non-DVS programs ran 7 minutes and 30 seconds requiring about 720 mWh. We also measured static power consumption of the test board for 45 seconds and found out that the test board consumes about 59 mWh in 45 seconds while the CPU is idle. Given this data, it can be found that the processor only had consumed 120 mWh with the DVS optimization and 130 mWh without it, meaning CPU power consumption reduction by 7% with overall runtime increase by 6.6%.

We were also suggested to set a deadline for the test suite and compare the total power consumption of the board as reported by the power meter while running optimized and non-optimized versions of the test suite within this deadline. 8 minutes can be used as a deadline for Aburto tests given that 7% slowdown compared with the non-DVS version is acceptable. We need to add the static power consumption during the remaining 30 seconds to the power consumption of non-DVS run. Given that the test board consumes about 59 mWh in 45 seconds when the CPU is idle, a value of $59 \times 30/45 = 39.3$ mWh was additionally consumed. The resulting power consumption of the non-DVS programs within the deadline is 759.3 mWh. Compared to the DVS optimized run result of 750 mWh, we observe 1.24% reduction of the total consumed power.

Unfortunately, due to the lack of time in the project and also due to the limited time we had access to the test boards, we could not produce more detailed data describing the above experiments. We are currently working on the enhanced interprocedural DVS optimization, and we hope to gather and analyze more data.

## 3 Bit-switching minimization

We have initially planned to experiment with power-aware instruction scheduling and/or combiner within GCC. However, this work requires building a power model of the processor in question, including estimating power cost of single instructions. There are several such models known, e.g. [9] and [10]. However, our experimental setup did not allow us to measure instruction power costs for two reasons. First, our power meter was able to measure only the consumption of the whole test board, not the consumption of the processor only. Second, the test board contained a lot of devices like LCD screen, camera, etc. provided for easier program development with the board, which meant that CPU power consumption was only a relatively small fraction of the total power consumption of the board.As a result, the difference between power consumption of tests containing different processor instructions could not be measured precisely enough.

Given this, we have decided to account for bit-switching created by an optimized program in the compiler. It is known [8] that switching bits on control and data buses of the processor accounts for significant amount of power consumption. As scheduler is the natural place in the compiler for moving instructions, we have tried to add a scheduling heuristic that takes into account the amount of created bit-switching.

First, we have examined the limits of energy saving that can be achieved via bit-switching. We have prepared tests that have instructions with as much different bit encodings as possible. For ARM, we have chosen `ands r6, r8, r0` and `bicne r9, r7, #0x3FC` instructions with encodings having only 3 out of 32 equal bits. The main part of the two test programs is a 1,000-instruction loop that has 500 `ands` instructions followed by 500 `bicne` instructions in the first case and 500 `ands-bicne` instruction pairs in the second case. In both tests the loop is executed a large number of times in order to capture effects on power consumption.

The experiments show increase in 1-2% of total power consumption for the second test compared to the first test on the OMAP test board and around 5% on the MV320 test board. As the processor power consumption is only a part of total power consumption, the percent of saved power of the processor should be around 10%. This gives the upper bounds of the savings that can be achieved with optimizations based on bit-switching heuristic.

Second, we have implemented a target hook for the ARM back-end that tries to predict bit encoding of the assembler instruction(s) into which the given RTL instruction is going to be translated. The hook predicts the whole instruction encoding except encodings of some types of instruction operands which contain address calculations or addresses that are not known at compile time. We have looked at some of the object code generated by the compiler and compared it to the predicted instruction encodings. The predicted encodings appeared to be exactly corresponding to the real one in almost all cases (except when the RTL instruction could be translated into several variants chosen later after scheduling and the less probable encoding had been chosen).

Third, we have written a scheduling heuristic that is based on bit-switching minimization using the implemented target hook. The heuristic uses a parameter to control its weight. The parameter can vary from 0 to 32, and it can be considered as the number of same bits on the processor control bus that increases instruction critical path priority by one. E.g., when the parameter equals to 5, and scheduler chooses between two instructions with priorities equal to 3 and 4, and these instructions are predicted to switch 7 and 22 bits on the instruction bus of the processor respectively, the scheduler increases the priority of the first instruction by $(32 - 7)/5 = 5$ and of the second by $(32 - 22)/5 = 2$ and chooses the first instruction to be scheduled next.

We ran our bit-switching heuristic implementation on the Aburto test suite. Even with the maximal weight of the bit-switching heuristic the overall minimization of switched bits reached 7% at most on Sim benchmark and was about 3% on average[4]. Our runs of optimized tests showed that there is no any power consumption minimization compared to non-optimized tests. One of the reasons for this might be that our tests have many floating point operations implemented via library calls, which makes impossible to predict their encodings until linking. Generally, it seems that to change the power consumption on 1%, one needs to change bit-switching on 10% or more, which cannot be achieved from within the compiler.

Finally, we have also tried to utilize the bit-switching estimation in a combining optimization. In GCC, combine unites up to three instructions connected by use-def chains in a single one. We have attempted to limit the combining opportunities to those that do not increase bit-switching. However, this didn't work for two reasons. First, combine is done before the first scheduling pass, which is relatively early, and we couldn't predict the final bit encoding precisely enough. Second, combine doesn't change code much, so the influence of this optimization on bit-switching is even lower than in the scheduling case.

## 4 Tuning of memory optimizations

Memory subsystem is considered one of the most energy-consuming parts of embedded devices. Hence, minimization of memory accesses and optimization of cache behavior is an important approach for lowering power consumption. We decided to make a research on current status of memory optimizations implemented in different versions of GCC and on possibilities of some new approaches for memory optimizations available for the OMAP2430 test board.

[4] This data was calculated statically as bit-switching of successive instructions given by the `objdump -d` output, so it does not reflect the dynamic bit-switching.

### 4.1 Prefetching

Prefetching is available for ARM implementations that support the `pld` instruction [7]. The OMAP test board we had for experiments supported this instruction. GCC also supports prefetching for ARM5TE or later processors. We have tested the performance of this optimization with different sets of parameters. The most important parameters that are used by prefetching are the size of L1 cache line, number of available concurrent prefetches, and the latency of the prefetch operation. The size of L1 cache line is known from our experiments to be 32 bytes. The number of simultaneous prefetches is hard to estimate from experiments but it should not be large for embedded devices. Hence we used values of 1, 2 and 3 for our prefetching experiments. The latency of the prefetch operation is not known too, but we assumed that it should be close to the latency of memory access, which is about 20 cycles on the OMAP test board. We then decided to run prefetching experiments with different values ranging from 10 up to 100 cycles for the prefetching latency parameter. We used Aburto and MediaBench test suites for prefetching experiments.

The resulting data shows that prefetching provides a significant increase in performance of Nsieve test (14%) and significant decrease in performance of five tests: Sim (6-10%[5]), Heapsort (7-12%), Queens (2-5%), EPIC (7%), and JPEG (8%). The Matrix Multiplication tests showed unstable behavior so it is hard to say unambiguously how prefetching influences these tests.

### 4.2 Other memory optimizations in GCC

We have studied the source codes of GCC 4.2 for optimizations that can improve memory performance. We have found that such optimizations include automatic vectorization and loop linear transformations. We have prepared and ran tests for evaluating these optimizations on our test boards.

Automatic vectorization is an optimization that converts several scalar operations into one vector operation. As ARM supports loading of several registers from memory simultaneously in one operation, this optimization potentially offers decrease in memory access operations. Unfortunately, the only case when GCC succeeded in applying vectorization was in converting of accesses to char arrays into one access of an integer loading four array elements in one operation. We observed several cases of the same optimization performed on the Aburto test suite, but in general the increase in performance was negligible. The results from MediaBench test suite are similar.

Linear loop transformations are very limited in GCC. For GCC 4.2, a simple loop interchange in a loop nest traversing a two-dimensional array by row instead of by column was not performed, though this test is successfully transformed by GCC 4.3. The reason for this behavior is in weakness of the loop analysis of 4.2 and prior versions of GCC. This optimization doesn't change anything on the Aburto test suite.

[5] We have tried different values of other prefetch parameters, i.e. prefetch latency and simultaneous prefetches, thus we got different performance numbers.

For GCC 4.3, there are several others potential memory optimizations, such as array flattening, structure peeling, and auto increment/decrement generation. We have only performed testing of the structure peeling. This optimization splits an array of structures into several arrays that contain each field of the structure when one or several fields of structure is accessed considerably more often than others. This optimization successfully transformed our test program, but it was never applied on the Aburto test suite.

Experimental results show that there are no optimizations in GCC that can dramatically increase memory system performance of the target board compared to the performance obtained by the `-O2` optimization level. Most studied optimizations don't work well or at all on the ARM platform.

### 4.3 Other possible memory optimizations

We have selected two other promising optimization types for experiments. These are using of scratch-pad memories, otherwise called tightly-coupled memories (TCM), and optimization of allocation on memory banks. However, the first optimization does not apply on the test boards that we had for experiments. The OMAP test board offers on-chip SRAM memory which should have lower power consumption and latency characteristics than conventional RAM. However, our experiments showed that this is not the case.

The second optimization is based on observation that when some memory banks are not used by the system, they can be turned to idle modes for energy economy. The OMAP test board's PRCM supports idle modes of memory banks and should set this mode automatically on needed memory banks when there are no accesses to them during some time. A compiler can try to move data or rearrange data accesses in such way that only one memory bank is used most of the time. Unfortunately, this kind of optimization is not applicable for the OMAP test board because of the size of the memory banks. OS support is needed instead of a compiler support to put different processes into different banks.

## 5 Conclusions

We have completed the research on DVS optimizations, memory related optimizations, and power-aware scheduling for ARM architecture and GCC compiler. The most promising direction turned out to be DVS optimization. On a set of small benchmarks, our prototype implementation found several regions suitable for DVS. As a result, we were able to reduce the power consumption of processor alone by 7% and total power consumption by 1.2%, while increasing run time by 6.6%. The total power consumption reduction number can be bigger for production devices, where the percentage of processor consumption is greater.

We are working on improving the implementation to make find more regions for DVS by turning it into interprocedural optimization. Initial experiments show that the number of regions made available for DVS by a simple interprocedural prototype is doubled compared to the implementation described in the paper. Also, the implementation needs to be adopted for multiprocess environment, e.g. by writing an OS manager to resolve conflicts between different processes asking for different frequency levels.

We have studied a number of memory optimization techniques, both implemented in GCC and specifically targeted for power-efficient computing. We have found that current GCC memory optimizations (prefetching, vectorization, loop transformations) are either not applicable to ARM or trigger only on specially constructed test cases, but not on the benchmarks. As for specific power optimizations, the test board hardware did not allow us to use them. We think that the new framework for loop transformations, Graphite, which will be available in GCC 4.4, makes a better basis for further research on power-aware loop optimizations.

While researching on power-aware instruction scheduling, we didn't have information about power costs of single instructions on our test boards. We have tried using bit-switching heuristic, but the experimental results show that decreasing bit-switching for several per cents is not enough to produce substantial reductions in power consumption. We conclude that the information about power costs of instructions is crucial in making the existing machine-dependent optimizations power aware.

## References

1. Y. Chen, Z. Shao, Q. Zhuge, C. Xue, B. Xiao, E.H.-M. Sha. Minimizing Energy via Loop Scheduling and DVS for Multi-Core Embedded Systems. In 11th International Conference on Parallel and Distributed Systems – Workshops, pp. 2-6, July 2005.
2. GRAPHITE GCC framework. `http://gcc.gnu.org/wiki/Graphite`
3. C. Hsu. Compiler-Directed Dynamic Voltage and Frequency Scaling for CPU Power and Energy Reduction. Doctoral Thesis, Rutgers University, 2003.
4. MV320 ARM Board. `http://mvtool.co.kr/products/product.php?query=list&code=100101&lv=3&lang=`
5. OMAP2430 Development Board. `http://focus.ti.com/general/docs/wtbu/wtbugencontent.tsp?contentId=14645&navigationId=12013&templateId=6123`
6. H. Saputra, M. Kandemir, N. Vijaykrishnan, M.J. Irwin, J. Hu, C.-H. Hsu, U. Kremer. Energy conscious compilation based on voltage scaling. In ACM SIGPLAN Joint Conference on Languages, Compilers, and Tools for Embedded Systems and Software and Compilers for Embedded Systems, pp. 2-11, June 2002.
7. D. Seal. ARM Architecture Reference Manual. $2^{nd}$ Edition, Addison-Wesley, 2000.
8. Ching-Long Su, Chi-Ying Tsui, and A.M. Despain. Low power architecture design and compilation techniques for high-performance processors. Compcon Spring '94, Digest of Papers, pp.489-498, 1994.
9. V. Tiwari, S. Malik, A. Wolfe, M.T. Lee. Instruction level power analysis and optimization of software. Journal of VLSI Signal Processing Systems, vol.13, iss.2-3, August 1996, pp.223-238.
10. A. Varma, B. Jacob, E. Debes, I. Kozintsev, P. Klein. Accurate and fast system-level power modeling: An XScale-based case study. ACM Transactions on Embedded Computing Systems, vol.6, iss.4, September 2007.