*IEIE Transactions on Smart Processing and Computing*

# Performance Comparison between LLVM and GCC Compilers for the AE32000 Embedded Processor

**Chanhyun Park, Misun Han, Hokyoon Lee, Myeongjin Cho, and Seon Wook Kim**

Compiler and Microarchitecture Laboratory, School of Electrical and Computer Engineering, College of Engineering, Korea University, Seoul, Korea {yasutaxi, mesunyyam, hokyoon79, linux, seon}@korea.ac.kr

* Corresponding Author: Seon Wook Kim

*Received November 20, 2013; Revised December 27, 2013; Accepted February 12, 2014; Published April 30, 2014*

* *Extended from a conference: Preliminary results of this paper were presented at the ICEIC 2014.This present paper has been accepted by the editorial board through the regular reviewing process that confirms the original contribution.*

* *Short Paper*

***Abstract*:** The embedded processor market has grown rapidly and consistently with the appearance of mobile devices. In an embedded system, the power consumption and execution time are important factors affecting the performance. The system performance is determined by both hardware and software. Although the hardware architecture is high-end, the software runs slowly due to the low quality of codes. This study compared the performance of two major compilers, LLVM and GCC on a32-bit EISC embedded processor. The dynamic instructions and static code sizes were evaluated from these compilers with the EEMBC benchmarks.LLVM generally performed better in the ALU intensive benchmarks, whereas GCC produced a better register allocation and jump optimization. The dynamic instruction count and static code of GCCwere on average 8% and 7% lower than those of LLVM, respectively.

***Keywords*:** GCC, LLVM, Optimization, EISC, Performance, Code quality

## 1. Introduction

Nowadays, the embedded processor market is becoming increasingly larger with the appearance of mobile devices, such as smartphones and tablets. In an embedded system, the power consumption and execution time are important performance factors due to the resourcelimitations and mobility characteristics. Even if the hardware specifications becomes tremendously high, the hardware cannot be executed efficiently without proper assistance from a compiler. Therefore, higher performance on the same hardware can be achieved when a program is compiled with better compilers.

Traditionally, GCC [1] has been used and developed over a long period, and it provides many useful techniques for achieving higher performance, such as loop unrolling, constant propagation, aggressive register allocation. LLVM [2] has become available recently, and it is being popularly used because of its versatility, flexibility, and reusability. Despite this, a performance comparison of the two compilers has not been researched sufficiently.

The aim of this study was to evaluate the code quality of the two compilers and better understand their optimization techniques. For this purpose, the EEMBC benchmark [3], which represents a wide variety of workload for embedded systems, was used. The number of dynamic instructions and code size have been used as the basic performance metrics.

The remainder of this paper is organized as follows. Section 2 introduces the target embedded processor, AE32000 [4], along with its features and instruction set architecture. Section 3 describes the frameworks of the LLVM and GCC compilers. In Section 4, the experimental setup and results are described. Finally, the conclusion is reported in Section 5.

## 2. Target Processor and ISA

This section describes the features of the target processor (AE32000) and the characteristics of its instruction set, called the EISC (Extendable Instruction Set Computer).

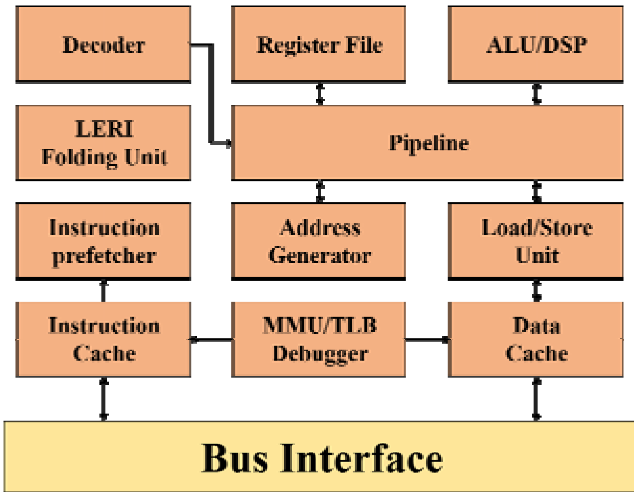



Fig. 1. AE32000 architecture [5].

Table 1. Characteristics of ISAs [7].

| | RISC | CISC | EISC |
|---|---|---|---|
| ISA structure | Simple | Complex | Moderate |
| Program code Size | Large | Small | Moderate |

## 2.1 AE32000

Fig. 1 presents an overview of the architecture of AE32000. AE32000 is a 32-bit embedded microprocessor that is aiming at a high code density and high performance. The processor uses a 16-bit fixed length instruction set, and has a typical 5-stage pipeline, 16x32-bit GPRs (General Purpose Registers) and 9x32-bit SPRs (Special Purpose Registers). The target supports the AMBA AHB/AXI Bus and the SIMD architecture. In addition, it supports up to 3 coprocessors.

## 2.2 EISC (Extendable Instruction Set Computer)

EISC is an instruction set architecture that has been developed by ADChips [6]. The EISC ISA combines the hardware simplicity of RISC (Reduced Instruction Set Computer) and the small code size of CISC (Complex Instruction Set Computer) into one. Table 1 lists the characteristics of RISC, CISC, and EISC.

EISC uses the Extension Register (ER) and Extension Flag (E) to express the variable immediate operands. ER stores an extendable operand value and E-flag shows whether the ER stores the immediate value. A LERI instructionis used to set the ER.

Fig. 2 shows how the LERI instruction cooperates with the MOV instruction to move a large immediate value. When the LERI instruction is executed, a value in the immediate field is copied into the ER register. The following MOV instruction then checks E-flag. If E-flag is set, a value in ER is shifted to the left by 8 and concatenated with the immediate value in the MOV instruction. As a result, a 22bit-width value can be moved at once. Two LERI instructions followed by MOV can decode an immediate value up to [31:0] in the same manner.

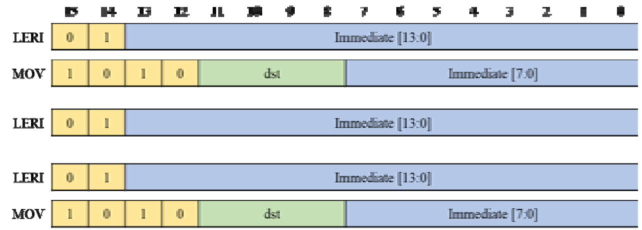



Fig. 2. LERI and LDI instructions to load the variable immediate values [7].

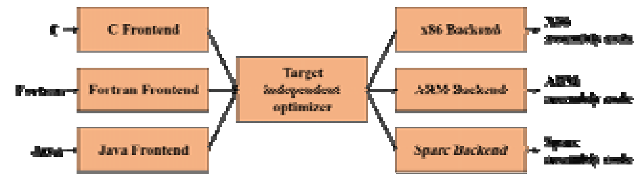



Fig. 3. Common compilation flow.

## 3. Code Generation

GCC and LLVM have similar features in that the front-ends reada source code written in various languages, such as C, C++, Fortran, Java, and generate IR (Intermediate Representation). Subsequently, the IR is optimized by the target independent optimizers. The compiler back-ends translate the optimized IR into the target processor assembly code, as described in Fig. 3. This strategy allows the reuse optimizers with various high-level languages and targets. An AE32000 back-end was added to each compiler to port LLVM and GCC to the AE32000 processor.

## 3.1 GCC Back-end

To generate the target assembly code, a compiler uses hardware-specific information, such as registers, instruction cost, and assembly mnemonics. For this purpose, the GCC back-end uses the machine description files. The machine description files consist of two parts: instruction patterns (.md file) and C code that contains the code generation rules.

The .md file contains IR patternsas RTL templates and the corresponding assembly mnemonics. The GCC back-end matches the generated IR against the RTL templates. If the IR matches successfully, GCC emits the assembly code for a target processor. Otherwise, GCC reconstructs the IR.

The other part of the machine description is written in C code. The C code describes the code generation rules in detail, such as caller-callee convention, register allocation priority, prologue/epilogue expansion, etc.

## 3.2 LLVM Back-end

The back-end of LLVM is similar to that of GCC.

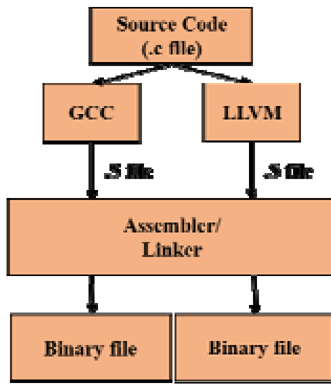



Fig. 4. Compilation flow of GCC and LLVM.

LLVM IRs are translated into the target instructions based on the target description (.td file) and cpp files. The target description files describe the register information, instruction information and caller-callee information for a target processor. Special routines, such as prologue, epilog expansion, and target specified instructions, are described in the cpp files.

## 3.3 LERI Instruction Support

The LERI is a special instruction for a large immediate value used by the EISC processor. The LERI instruction is inserted immediately before an instruction that requires a large immediate value. The AE32000 assembler is in charge of this process. Each source code is compiled into an assembly file (.s file) separately by LLVM and GCC. The assembler and linker generate binary files for AE32000. Fig. 4 shows a total compilation flow.

## 4. Performance Evaluation

The EEBMC benchmark on the AE32000 simulator was used to evaluate the performance of LLVM and GCC. The EEMBC benchmark was compiled by LLVM 3.1 and GCC 4.7.1 compilers with the option, '-O2'. In addition, the AE32000 assembler was used for both LLVM and GCC.

Fig. 7 shows the dynamic instruction count and the static code size of LLVM normalized to those of GCC. The results show that the dynamic instruction count and

```
signalOutLow1 = (varsize)(signal_in *
(*coefficient1++));
if( (signalOutLow1%COEF_SCALE)>(COEF_SCALE/2))
{
     signalOutLow1 += COEF_SCALE / 2;
}
signalOutLow1 /= COEF_SCALE;
//COEF_SCALE is constant.
```

Fig. 5. Code snippet of the iirflt benchmark.

```
C code
signalOutLow1 /= COEF_SCALE; // COEF_SCALE=50

(a)
 jal ___divsi3

(b)
 ldi  1374389535, %r9
 mul  %r9, %r8
 mfmh %r8
 mov  %r8, %r9
 lsr  31,  %9
 asr  4,   %r8
 add  %r9, %r8
```

Fig. 6. Assembly code generated by (a) GCC and (b) LLVM.

the static code size of GCCwere on average8% and 7% lower than those of LLVM, respectively. This section analyzes some representative benchmarks in detail.

### 4.1 iirflt

This benchmark shows a significant difference in performance between LLVM and GCC. As shown in Fig. 5, iirflt has a modulo operation and divides the codes with a single constant and single variable operand. Because AE32000 does not support modulo and division functional units in the hardware, the compiler back-ends generate a library call or an alternative instruction sequence instead of these unsupported operations.

As shown in Fig. 6, GCC generates a library call (__divsi3), whereas LLVM generates a code sequence for the division operator. The GCC manner has two disadvantages. First, the library call incurs call linkage overheads. Second, there are many branches in the library routine of GCC. The library checks whether a divisor is zero and calls another library to obtain the value required.

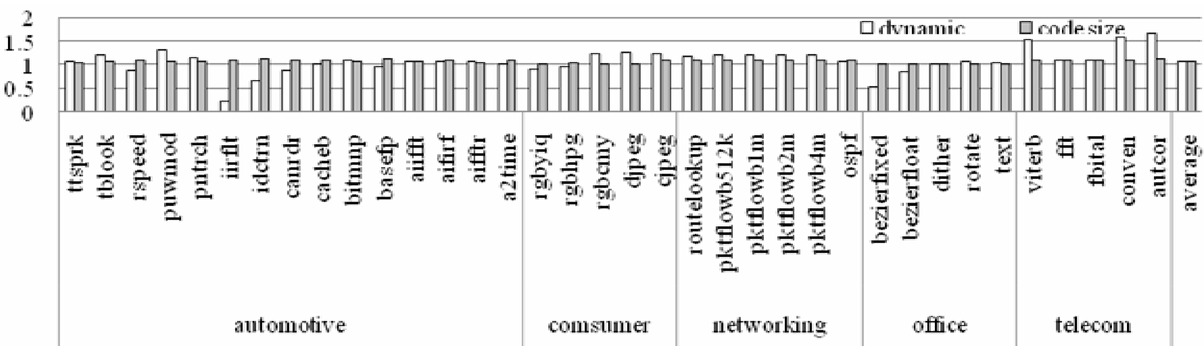



Fig. 7.Dynamic instruction count and static code size of LLVM normalized to GCC [8].

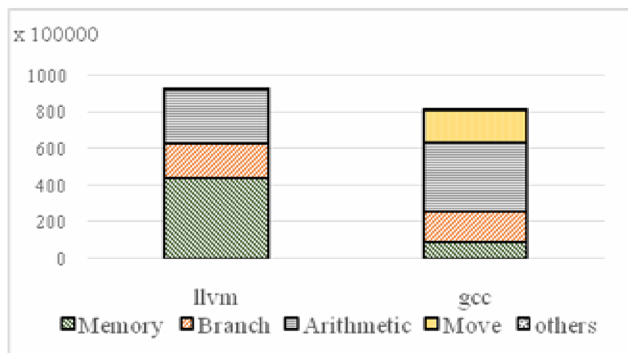



**Fig. 8. Number of dynamic instructions in the 5 groups in pntrch01.**

```
for(i=0;i<ROWS; i++)
 for(j=0; j<COLS; j++)
  for(k=0; k<COLS; k++)
   F_1[i][j]+=f_1[i][k]*cosMatrixA[k][j];
```

**Fig. 9. Hotspot code section of the idctrn benchmark.**

Therefore, LLVM could achieve better performance than GCC.

These strength reductions are slightly different in GCC and LLVM. GCC inserts 64bit multiplication during this optimization, but AE32000 does not support this; it results in a library call. On the other hand, LLVM inserts other instructions with the same effects. This makes LLVM apply the optimization.

## 4.2 pntrch

To compare the performance of LLVM and GCC in detail, the EISC instructions were categorized into five groups: Memory, Branch, Arithmetic, Move, and others. Fig. 8 shows the number of total dynamic instructions in the groups. The number of executed memory instructions of LLVM was 4.78 times higher than that of GCC. In contrast, the number of executed move instructions of LLVM was 0.8% of those by GCC.

The main reason for the difference is memory optimization in loops. First, in the LLVM code, a loop induction variable is spilled. Second, GCC performs basic block reordering. As a result, GCC uses more register-to-register instructions by reducing the LD/ST instruction, as shown in Fig. 8.

## 4.3 idctrn

The hotspot region of idctrn is a two dimension array multiplication (matrix multiplication) that is 3-nested loops, as shown in Fig. 9.

LLVM unrolls the loop by a factor of 8, precisely the same as a constant COLS. This loop-unrolling [9] removes the innermost loop – 8 comparison instructions, 8 jump instructions and arithmetic instructions of a loop induction variable. As a result, LLVM can reduce the dynamic instructions significantly, but increases the static code size. On the other hand, GCC does not perform loop unrolling with the –O2 option.

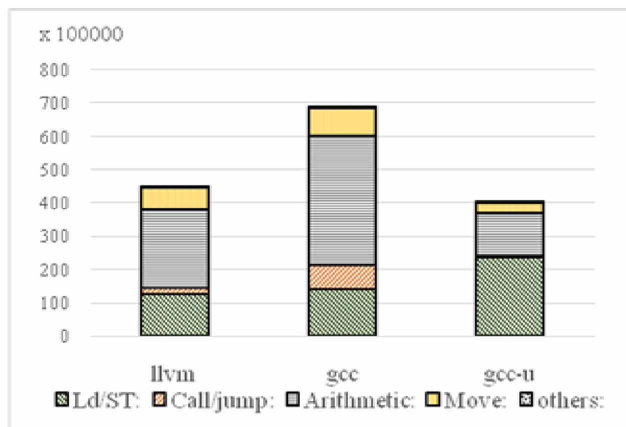



**Fig. 10. Number of dynamic instructions into 5 groupsin idctrn.**

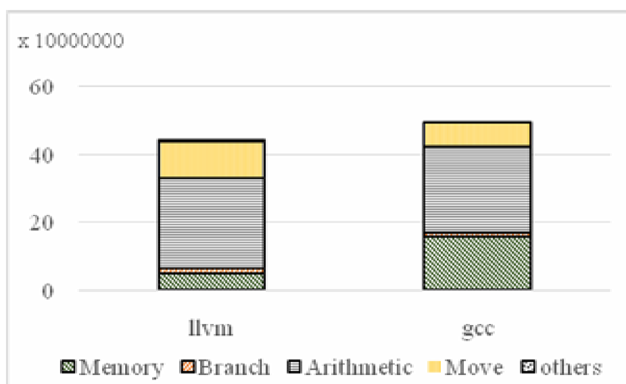



**Fig. 11. Number of dynamic instructions in the 5 groups in rgbyiq.**

For a fair comparison, the -funroll-loops option was applied when the benchmark was compiled with GCC. The simulation showed that the dynamic instruction count is 10% smaller than that of LLVM. On the other hand, the static code size of GCC is 28% larger than that of LLVM. LLVM unrolls only the inner most loop, but GCC unrolls the nested loops. Therefore, LLVM generates smaller code.

Fig. 10 shows the number of total dynamic instructions of the 5 groups in idctrn where gcc-u means it applied the loop unrolling optimization.

## 4.4 rgbyiq

The benchmark loads a 76816-byte RGB image and converts it to a YIQ format image. GCC has many more memory instructions than LLVM for the following reasons.

AE32000 provides a multiplication instruction with an immediate value. On the other hand, the GCC implementation does not use this instruction. Therefore, one more instruction is needed to load the immediate value from a register. In addition, in the main function loop, GCC uses 14 registers, whereas LLVM uses 12 registers. The difference is that 2 registers hold irrelevant values to the hot spot in the LLVM case. The use of more registers in GCC removes some move instructions, and GCC requires fewer move instructions.

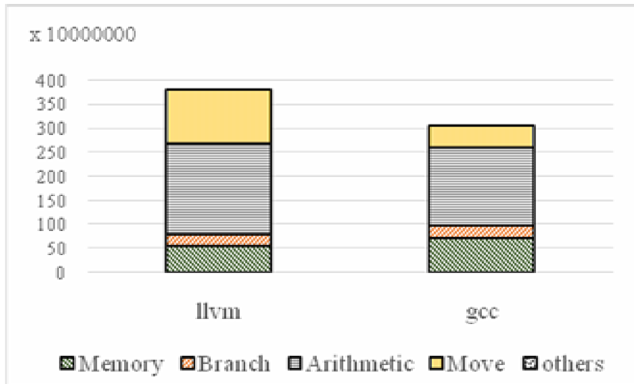



**Fig. 12. Number of dynamic instructions in the 5 groups in rgbcmy.**

```
C code
if(proute->dst_addr==trie_root->key)
{
   next_node=next_node->left;
}
else
{
   next_node=next_node->right;
}
(a)
  cmp %r1 %r2
  jeq label1:
  jmp label2:
label1:
  mov %4 %r3
  jmp label3:
label2:
  mov%r5 %r3
label3:
(b)
  move %r4 %r3
  cmp %r1 %r2
  jeq label1:
  move%r5 %r3
label1:
```

**Fig. 13. Code snippet in the routelookup and corresponding assembly code (a) LLVM, (b) GCC.**

## 4.5 rgbcmy

The benchmark rgbcmy loads RGB data, converts it to CMYK format, and stores into new memory. RGB and CMYK are stored in arrays with strides 3 and 4, respectively. In the source code, the source and destination memory locations are obtained with a base pointer, an offset and a stride. LLVM calculates the memory address for every loop, whereas GCC performs loop invariant factoring and strength reduction in the loop. Therefore, GCC generates the add instructions, but LLVM generates the add and move instructions to obtain the memory address. As a result, the number of move instructions in GCC is lower than those in LLVM, as shown in Fig. 12.

## 4.6 routelookup

The hotspot function in the routelookup benchmark consists of ‘if-else’ statements, as shown in Fig. 13.

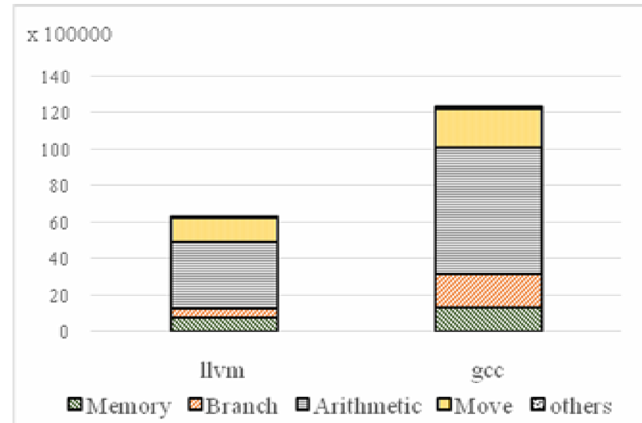



**Fig. 14. Number of dynamic instructions in the 5 groups in bezierfixed.**

As mentioned in Section 4.2, LLVM has a weakness in basic block reordering. LLVM tends to maintain the sequence of the basic block, as described in Fig. 13. For example, there are two unconditional jumps and one conditional jump instruction in assembly code obtained by LLVM, whereas there is only one unconditional jump instruction with GCC. The control flow in the code generated by LLVM is easy to understand because the basic blocks are quite similar to the original source code. The inefficiency of LLVM reduces the chance of optimization, which leads to decreasing dynamic instruction counts and static code size.

## 4.7 bezierfixed

The benchmark takes the curves as an input and calculates the Bezier curve points. This benchmark consists of nested loops and division instructions. LLVM compiles these division instructions into instruction sequences, as mentioned in Section 4.1. In addition, LLVM unrolls the innermost loop by a factor of 4. As a result, there are fewer branch instructions and arithmetic instructions of LLVM than GCC, as shown in Fig. 14.

## 5. Related work

Few studies have compared GCC and LLVM. ‘Kim et al, Comparison of LLVM and GCC on the ARM Platform”, EMC 2010’ [10] is an example of a study of LLVM and GCC. In their study, they focused on the inter-procedure optimization performed by ‘llvm-ld’. In addition, they showed that the other optimizations, GCC, performed better, in various true and false flags. They gave ‘-O2’ to both compilers, but they did not mention loop unrolling. LLVM might have been improved since then because their LLVM version was 2.6.

## 6. Conclusion

This study analyzed and compared the performance of

the generated codes by LLVM and GCC in terms of the dynamic instruction counts and static code sizes. For the performance evaluation, two backend compilers for the AE32000 processor were implemented and the EEMBC benchmarks were used.

GCC performed better in optimizing the memory accesses and basic block ordering, whereas LLVM did better in optimizing the arithmetic instructions. In addition, LLVM sometimes optimized quite aggressively in loop unrolling, whereas GCC was good at register allocation in the view of saving instructions using more registers. On the other hand, if there is an opportunity for loop optimization, the program performance can be enhanced by adding '-funroll-loops' with GCC.

In the ALU intensive EEMBC benchmarks, LLVM showed better performance. On the other hand, GCC showed better performance in memory intensive benchmarks. Overall, the dynamic instruction counts and the static code sizes of the benchmarks compiled by GCC were 8% and 7% on average lower than those of LLVM, respectively.

## Acknowledgement

This study was supported by Basic Science Research Program through the National Research Foundation of Korea (NRF) funded by the Ministry of Education, Science and Technology (No. 2011-0010262).

## References

[1] GCC, the GNU Compiler Collection. Article (CrossRef Link)

[2] The LLVM Compiler Infrastructure. Article (CrossRef Link)

[3] J. Poovey, T. Conte, M. Levy, and S. Gal-On, "A benchmark characterization of the EEMBC benchmark suite," *Micro, IEEE*, vol. 29, no. 5, pp.18–29, 2009. Article (CrossRef Link)

[4] Hyun-Gyu Kim, Dae-Young Jung, Hyun-Sup Jung, Young-Min Chio, Jung-Su Han, Byung-Gueon Min, and Hyeong-Cheol Oh, "AE32000B: a Fully Synthesizable 32-Bit Embedded Microprocessor Core," *ETRI Journal*, vol. 25, no. 5, pp. 337-344, Oct. 2003. Article (CrossRef Link)

[5] AE32000-Lucida. Article (CrossRef Link)

[6] Advanced digital chips inc. Article (CrossRef Link)

[7] EISC. Article (CrossRef Link)

[8] Chanhyun Park, Miseon Han, Hokyoon Lee, and Seon Wook Kim, "Performance Comparison of GCC and LLVM on the EISC Processor, "*International Conference on Electronics, Information and Communication (ICEIC)* in Kota Kinabalu, Malaysia, 2014.

[9] J. C.Huang, T. Leng, "Generalized loop-unrolling: a method for program speedup," *IEEE Symposium on Application-Specific Systems and Software Engineering and Technology (ASSET) in Richardson, Texas*, pp.244-248, 1999. Article (CrossRef Link)

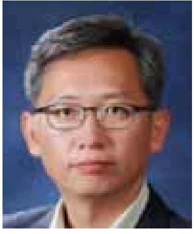
[10] Jae-Jin Kim, Seok-Young Lee, Soo-Mook Mook, Suhyun Kim, "Comparison of LLVM and GCC on the ARM Platform", *Embedded and Multimedia Computing (EMC), 2010 5th International Conference* on, vol., no., pp.1,6, 11-13 Aug.2010. Article (CrossRefLink)

**ChanhyunPark** received his B.S. degree in Electrical Engineering from Korea University, Seoul, Republic of Korea, in 2013. Currently he is a Ph.D. student in Korea University. His research interests include performance analysis in the Android mobile system. He is a student member of IEEE

**Mesun Han** received her B.S. degree in Electrical Engineering from Korea University, Seoul, Republic of Korea, in 2012. Currently, she is a Ph.D. student in Korea University. Her research interests include compiler support, microarchitecture, and memory designs (particularly DRAM and NAND Flash memory). She is a student member of IEEE.

**Hokyoon Lee**receivedhis B.S. degree in Electrical Engineering from Korea University, Seoul, Republic of Korea, in 2013. Currently, he is a Ph.D. student in Korea University. His research interests include compiler, embedded system, and microarchitecture.

**MyeongjinCho** received his BS, MS, and Ph.D.from the School of Electrical Engineering of Korea University, Seoul, Republic of Korea in 2006, 2008, and 2013, respectively. Currently, he is a research professor at Korea University. His research interests include Android performance analysis and optimization, high performance computing, and microarchitecture. He is a member of IEEE and ACM.



**Seon Wook Kim** received his BS in Electronics and Computer Engineering from Korea University, Seoul, Republic of Korea in 1988. He received his MS in Electrical Engineering from Ohio State University, Columbus, Ohio, USA, in 1990, and Ph.D. in Electrical and Computer engineering from Purdue University, West Lafayette, Indiana, USA, in 2001. He was a senior researcher at the Agency for Defense Development from 1990 to 1995, and a staff software engineer at Inter/KSL from 2001 to 2002. Currently, he is a professor with the School of Electrical and Computer Engineering of Korea University and Associate Dean for Research at the Collegeof Engineering. His research interests include compiler construction, microarchitecture, and SoC design. He is a senior member of ACM and IEEE.

Copyrights © 2014 The Institute of Electronics and Information Engineers