Comparison of LLVM and GCC on the ARM Platform

Jae-Jin Kim, Seok-Young Lee, Soo-Mook Moon
VM&O Laboratory
Seoul National University
Seoul, Korea
{kjj7999, sylee, smoon}@altair.snu.ac.kr

Suhyun Kim
Imaging Media Center
Korea Institute of Science and Technology
Seoul, Korea
dr.suhyun.kim@gmail.com

Abstract—**Virtual machines provide platform independence by using intermediate code. Program source code is compiled into intermediate code, which can be executed on many different platforms. Even though there are many virtual machines, there are not many virtual machines that can support C/C++, especially with high performance. Recently, Low Level Virtual Machine (LLVM) provides limited degree of architecture independence for C programs. Though it does not support multi-platform execution of complex programs, it is able to run simple benchmarks, enabling us to look into important aspects. In this paper, we analyze the performance impact of using intermediate code for C programs in the embedded system. Compared to the traditional native execution, LLVM has a little weakness in handling memory accesses, but generally shows competitive performance in other optimizations. Especially, in a benchmark with frequent function calls, LLVM exhibits even better performance than GCC.**

Keywords – Intermediate Code, LLVM, GCC, Embedded System, ARM, Performance Analysis, Code Optimization

I. INTRODUCTION

A Virtual Machine (VM) is a software implementation of a machine that executes programs [1]. One of the most well-known virtual machines is Java Virtual Machine (JVM) which is developed by SUN Microsystems [2]. Java source code is translated into Java bytecode, and then it can be executed in every environment with Java Virtual Machine. Such platform independence is the biggest advantage and the most important feature of virtual machine. In addition, Java Virtual Machine provides garbage collection and memory protection to strengthen robustness. However, the fact that bytecode is executed by a software virtual machine, not the hardware, involves the penalty of slower execution

Recently, in addition to the traditional desktop PC, new information devices to run software are getting important, including smartphones and interactive televisions or set-top boxes. With this growing variety of platforms, the needs for virtual machines to provide platform independence are growing. If developers write programs based on a virtual machine, the program can run on multiple platforms without recompiling the program and can acquire platform compatibility. Thus, if we can decrease performance degradation by the virtual machine, virtual machines can be widely accepted in embedded software platforms.

In addition, internet-based cloud computing technology is beginning to attract big attention [3]. One of its key ideas is the concept of software as a service (SaaS). If richer interaction than just a basic web browser is needed, some form of virtual machine, also known as rich internet application (RIA), is required. To take best advantage of local resources, a virtual machine with good performance is highly desired.

The Low Level Virtual Machine (LLVM) [4] originally developed at the University of Illinois at Urbana-Champaign is one of the virtual machines that have the potential to meet the requirement of high performance. LLVM is designed for compile-time, link-time, runtime, and "idle-time" optimization of programs written in arbitrary programming languages [5]. As shown in Figure 1, LLVM provides GCC-based C/C++ front-end tool called llvm-gcc and also provides back-end tool which compiles intermediate code into assembly code of various target machines. Source code is compiled into intermediate code using C/C++ front-end.

The platform independence is not the current goal of LLVM, but its intermediate code has a limited degree of architecture independence and intermediate code of simple programs satisfying certain restrictions can be executed on a variety of platforms through compilation by back-end. It can be a good reference for estimating the impact of using intermediate code for C programs.

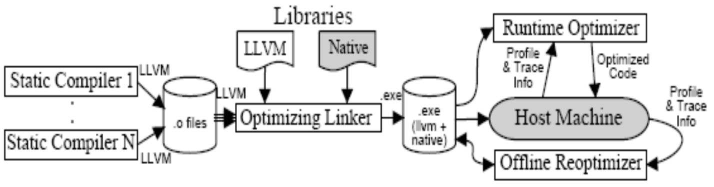

Figure 1. LLVM System Architecture Diagram

This paper evaluates the performance of LLVM and GCC, and compares the code generated by them from the same C programs. Benchmarks are tested on Linux-ARM smartphones, NeoFreeRunner [10]. By thorough examination of assembly

This work was supported by the IT R&D program of MKE/KEIT. [KI002119, Development of New Virtual Machine Specification and Technology]

code, we analyze which optimizations make the difference in performance of GCC and LLVM.

The rest of this paper is organized as follows. In section 2, we describe the features and components of LLVM. In section 3, we compare the performance of LLVM and GCC through simple benchmarks. In section 4, we analyze assembly code generated by them to know why the difference in performance occurs and find out the difference between their optimization. A summary follows in section 5.

II. LLVM

In this section, we look into the relevant features and components of LLVM.

A. *Features of LLVM*

The Low Level Virtual Machine (LLVM) is a compiler infrastructure which focuses on compile-time, link-time, runtime optimization of programs written in multiple languages. Unlike most other compilers, LLVM compiles source code not directly into binary code, but into intermediate code with its own virtual instruction set. This low level intermediate code looks like a simple 3-address RISC instructions, but provides language independent type system. This intermediate code is in the Static Single Assignment (SSA) form to facilitate optimization [6]. Based on this intermediate code, the LLVM system includes language-independent and machine-independent optimization.

The intermediate code itself is not designed to be target-independent. The primary goal of its design is to provide as much information as possible to the link-time optimizer. However, significant portion of the intermediate code is the same across many architectures and allows the execution of simple programs on various target machines such as x86, ARM, and so on [4].

B. *Components of LLVM*

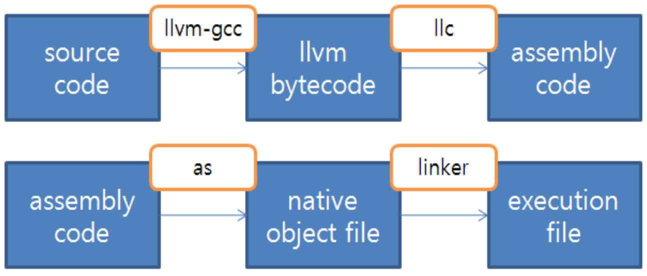

Figure 2. Compile Process of LLVM

Figure 2 shows the compile process of LLVM. We can get the language-independent and largely machine-independent bitcode from the source code using llvm-gcc, and make the assembly code of target machine using llc. Using assembler and linker of each machine, this assembly will become an execution file [7].

We can also get assembly code of target machine or execution file of that directly, not creating LLVM bitcode, by using llvm-gcc. If the source code consists of several files, created several bitcodes can be combined into one bitcode by llvm-link or llvm-ld. In this process, link time optimization is in progress.

1) *llvm-gcc*

llvm-gcc is C front end based on GCC that compiles C programs into LLVM bitcode depending on the options. By default, it compiles to object or execution file just like GCC. If the '-emit-llvm' option is given, It will generate LLVM bitcode. When '-S' option is given with '-emit-llvm' option, we will get human readable LLVM assembly language (.ll file). Instead If '-c' option is given with '-emit-llvm' option, we will get LLVM bitcode which can be compiled (.bc file). Like GCC, we are able to give optimization options such as -O1, -O2, -O3 [4].

2) *llc*

The llc command compiles LLVM bitcode files to assembly language of each machine. By default, it compiles to assembly language of host machine. If we want to compile to assembly language of the other machine, we will give 'march-arch ' option. arch can be x86, x86-64, sparc, arm, thumb, mips and so on. Optimization levels are not implemented [4].

3) *llvm-link, llvm-ld*

The source code can comprise one C file, but in many cases are composed of multiple files. One C file translated into one LLVM bitcode file (.bc file) by llvm-gcc. We can merge several LLVM bitcode files into one bitcode file by LLVM linker (llvm-link or llvm-ld).

llvm-link simply links several files into one, because that process does not affect performance. However, llvm-ld performs 'link time' optimization as well as merges LLVM bitcodes. Most of these optimizations are inter-procedural optimizations.

III. Experimental Results

In this section, we compare code optimization performances between LLVM and GCC using some benchmarks in CaffeineMark, MiBench.

A. *Experimental Environment*

The experiments are performed on openmoko's NeoFreeRunner as target machine to compare the optimization of LLVM with that of GCC [10]. Openmoko is a project to create open source mobile phones. It consist of Openmoko Linux, a Linux based operating system designed for mobile phones and the development of hardware devices on which Openmoko Linux runs. The NeoFreeRunner is the second phone designed by the openmoko project. It uses Linux kernel, and touch screen support. Its CPU is a 400 MHz ARM processor, and it has 128MB SDRAM memory and 256MB Flash memory. The Operating System is Openmoko Linux (kernel v2.6.24).

The benchmarks we used are CaffeineMark [8] which is simple and easy to compare and part of MiBench [9], a free, commercially representative embedded benchmark suite. Because CaffeineMark is made for evaluate performance of

Java Application, we translate it to C program. Performance of MiBench is to be the inverse of the execution time.

We use ARM cross compiler toolchain GCC version 4.4.1, llvm-gcc based on GCC version 4.2.1 (build 5646) and llc version 2.6. The optimization level of GCC and llvm-gcc is commonly used level, '-O2'.

We evaluate the performance using llvm-gcc and llc through LLVM bitcode, the performance using the assembly code obtained by only llvm-gcc front-end directly without LLVM bitcode and the performance using llvm-gcc, llvm-ld, llc to perform link-time optimization of LLVM bitcodes. And we compare these 3 LLVM performances with the performance using the assembly code obtained by GCC.

B. *Experimental Results*

The experimental result of using CaffeineMark (Float, Logic, Loop, Method, Sieve) is shown below.

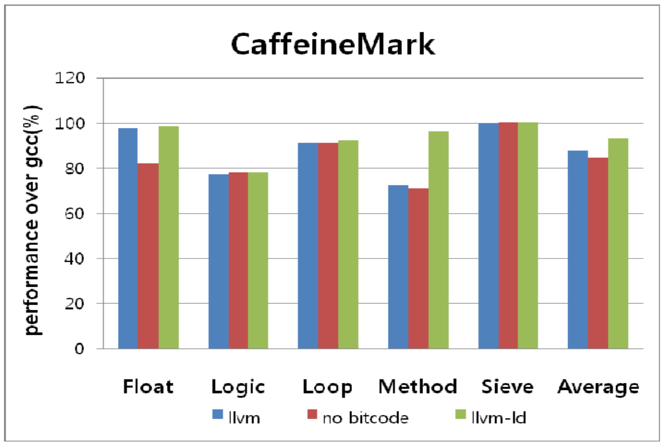

Figure 3. Performance Ratio over GCC Using CaffeineMark

In the Figure 3, 'llvm' means the performance using llvm-gcc and llc through LLVM bitcode. 'no bitcode' means the performance using llvm-gcc only without LLVM bitcode. llvm-ld means the performance using llvm-gcc, llvm-ld, llc using link time optimization on LLVM bitcode. The baseline (100%) is the performance using GCC.

On average, the LLVM performance through bitcode was 87.7% of GCC using CaffeineMark. 'Method' benchmark is the worst as 72.7% of GCC. 'Logic' is 77.3% of GCC. 'Loop' is 90.7% of GCC. There is little difference between GCC and LLVM at 'Float' and 'Sieve'.

When we use llvm-gcc only without LLVM bitcode, we can see the performance degradation at 'Float'. On average, the performance using llvm-gcc only without LLVM bitcode was 84.6% of GCC using CaffeineMark.

When we use llvm-ld to perform link time optimization merging LLVM bitcodes, we can found the performance of 'Method' significantly elevated to 96.3% of GCC. We can see good performance using link time optimization nearly 93.2% compared to GCC.

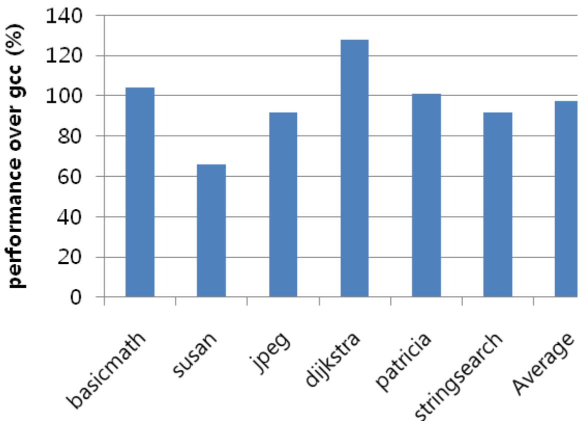

Figure 4. Performance Ratio over GCC Using Mibench

With MiBench, we measured only the performance after link-time optimization. The performance of LLVM is almost as good as that of GCC with up and down. In 'dijkstra' benchmark, the performance is better than that of GCC, 128% of GCC. 'susan' benchmark exhibits especially low performance, or 66% of GCC. 'jpeg' and 'stringsearch' benchmarks shows similar performance degradation, 8% worse than GCC. The other two benchmarks, 'basicmath' and 'patricia' shows a little better performance with LLVM compared to with GCC. In average, the performance with LLVM is 97.2% compared to that with GCC.

IV. Analysis

In this section, we compare the ARM assemblies created by LLVM and GCC based on the results from the above benchmarks. So we look for the optimization difference of LLVM and GCC.

A. *CaffeineMark*

1) *Logic*

As shown below, Logic' benchmark is the benchmark which measure the speed continuing to vary true and false of bool type 'flag' variable.

```
   bool flag3 = true;
   bool flag4 = true;
   bool flag5 = true;

for (i = 0; i < wIterationCount; i++)
  if((flag || flag1) && (flag2 || flag3) …
  {
      flag7 = !flag7;
      flag8 = !flag8;
      flag9 = !flag9;
      flag10 = !flag10;
  }
```

Figure 5. Logic Benchmark Core Loop

We can see flag transition part in the for loop. When compared a part of ARM assembly code obtained by LLVM and GCC as follows

```
  rsbs  r1, r1, #1
  movcc   r1, #0
  rsbs  r4, r4, #1
  movcc   r4, #0
  eor   r3, r3, #1
  eor   ip, ip, #1
  eor   r5, r5, #1
.L14:
  add   r2, r2, #1
```

```
  mov r1, #1
  mov r2, #0
  mov r3, #1
  ......
.LBB2_4:
  cmp r12, #0
  mov r12, #0
  moveq r12, r3
  cmp lr, #0
  mov lr, #0
  moveq lr, r3
  cmp r4, #0
  flag10 = !flag10;
```

Figure 6. Logic Benchmark Assembly Obtained by GCC(left) and by LLVM(right)

On the left of ARM assembly code, obtained using the GCC, 'eor' instruction is repeated several times. After opcoded 'eor', there are 2 same register as operand, and #1. 'eor' instruction represents XOR (Exclusive OR). If the value of second and third operands is equal to each other then 0 is stored in the register which represents the first operand. If not, 1 is strored [11]. ‘eor r3, r3, #1’ instruction means that 0 is saved in r3, when r3 was 1 and 1 is saved in r3, when r3 was 0. So it has the role to switching flags.

When we see the assembly code obtain by LLVM, there are ‘cmp’, ‘mov’, and ‘moveq’ instructions repeatedly. It is to be found that the first operand of ‘cmp’, ‘mov’, and ‘moveq’ instructions are the same, and the second operands of ‘cmp’ and ‘mov’ instructions are 0, the second operand of ‘moveq’ instruction is the register ‘r3’. In the assembly code of LLVM, 1 is saved in the register ‘r3’. ‘cmp r12, #0 / mov r23, #0 / moveq r12, r3’ means that 0 is saved in r 12, when r12 was 0, 1 is saved in r12, when r12 was 0. Therefore they are in the role of switching flags.

Switching the flag, like 'eor r3, r3, #1', only one instruction is used in GCC, but in LLVM, like 'cmp r12, #0/ mov r12, #0/ moveq r12, r3' three instructions are used. We can infer performance differences occurred at this point. The number of instructions is less in GCC than in LLVM, and ‘eor’ instruction execute in one instruction cycle.

When LLVM's all of ‘cmp/mov/moveq’ instructions replaced by ‘eor’ instruction, the performance is increased to 103% compared with the GCC, it is similar to GCC performance. So it was confirmed that the difference occurs in this section, Strength Reduction optimization of LLVM is insufficient compared to that of GCC.

There is no significant difference whether through LLVM bitcode or not. Because 'Logic' benchmark has no function call, there is no performance gain due to inter-procedure optimization during link process.

2) *Method*

'Method' benchmark is a program that repeatedly calls the function. The core ‘for’ loop of ‘Method’ benchmark is as belows.

```
for (i = 0; i < wIterationCount; i++) {
                        int k = arithmeticSeries(i);
                        int l = notInlineableSeries(i);
                        if (k > l) j += l;
                        else j += k;
}
```

Figure 7. Method Benchmark Core Loop

```
  movgt r4, r5
  ble .L39
.L37:
  mov r0, r5
  bl notInlineableSeries
l dr  r3,[r7, 0]
  cmp r0, r4
  add  r5, r5, #1
  addlt  r6, r6, r0
  addge r6, r6, r4
  cmp  r3, r5
  ble  .L31
  cmp r5, #0
  movne r3, r5
  movne r4, #0
  beq  .L40
.L28:
  subs r2, r3, #1
  add  r4, r4, r3
  beq  .L37
  mov r3, r2
  b  .L28
```

```
  mov r5, #0
  mov r4, r5
.LBB4_2:
  mov r0, r5
  bl arithmeticSeries
  mov r6, r0
  mov r0, r5
  bl notInlineableSeries
  cmp r6, r0
  movgt r6, r0
  add r4, r6, r4
  ldr r0, .LCPI4_0
  ldr r0, [r0]
  add r5, r5, #1
  cmp r5, r0
  blt .LBB4_2
.LBB4_3:
  mov r0, r4
```

Figure 8. Method Benchmark Assembly Obtained by GCC(left) and by LLVM(right)

'Method' benchmark has two function calls in the for loop (arithmeticSries, notInlineableSeries). As shown in the right of Figure 8. Although there are two subroutine calls to the in the assembly code obtained by LLVM, there is one subroutine call to the notInlineableSeries but not to the arithmeticSeries in the assembly code obtained by GCC.

```
int arithmeticSeries(int i)
{
          if(i == 0)  return 0;
          else return i + arithmeticSeries(i - 1);
}
```

Figure 9. The ‘arithmeticSeries’ Function of Method Benchmark

As shown Figure 9, arithmeticSeries(i) is a function to add numbers from 0 to i. In the .L28 block of left of Figure 8, r3 is i in the initial condition, and r4 is 0 in the initial condition. r3 is added to r4 repeatedly while r3 is reduced by 1. We can see that .L28 block and arithmeticSeries are in the same role.

So we can determine the 'arithmeticSeries' function is inlined in the assembly code obtained by GCC. Eventually, the cost of function calls are reduced by method inlining in the assembly of GCC, so that the performance of GCC is better than that of LLVM in 'Method' benchmark [12].

There is an implementation of the definition part of ‘arithmeticSeries’ function in the assembly code obtained by

GCC, but there is no subroutine call to the definition part of 'arithmeticSeries' function. The implementation of 'arithmeticSeries' function has no role. '.L28' block is replaced the role of that. But, the implementation of the 'arithmeticSeries' function of GCC is more efficient than that of LLVM. If we replaced the implementation of 'arithmeticSeries' of LLVM with that of GCC, the performance is increased by 15%.

Using llvm-ld to perform link-time optimization, performance was improved to 96.3% compared to the GCC. It does not inline the method, but devided block. So, it is not always executing entire 'arithmeticSeries' and 'notInlineableSeries', it is performed differently depending on the value of i. When there are function calls, we can know that link-time optimization of LLVM bitcode takes good effect.

B. *MiBench*

1) *basicmath*

'basicmath' benchmark performs simple mathematical calculations such as solving three-dimensional functions, obtaining the square root of an integer and translation between degree and radian. 'basicmath' spends most time executing 'SolveCubic' function repeatedly which solve three-dimensional functions.

```
for(a1=1;a1<10;a1++) {
 for(b1=10;b1>0;b1--) {
  for(c1=5;c1<15;c1+=0.5) {
   for(d1=-1;d1>-11;d1--) {
    SolveCubic(a1, b1, c1, d1, &solutions, x);
    }
  }
}
}
```

Figure 10. Basicmath Benchmark Core Loop

'SolveCubic' function doesn't have a loop, statements are performed sequentially. Compared assembly obtained by LLVM with assembly obtained by GCC, sequential registers are saved or loaded at once using 'stmib' and 'ldmib' instruction in GCC, but in LLVM 'ldr' and 'str' instructions are used multiple times. Increasing memory accesses of 'SolveCubic' function affect the performance difference of LLVM and GCC.

`ldmia   r1, {r0-r1}`	`ldr r0, [sp, #+12]`
`bl      __divdf3`	`ldr r1, [sp, #+16]`
`mov     r2, r4`	`mov r2, r0`
`mov     sl, r0`	`mov r3, r1`
`mov     fp, r1`	`bl __muldf3`
`mov     r3, r5`	`str r0, [sp, #+8]`
`mov     r0, r4`	`str r1, [sp, #+4]`
`mov     r1, r5`	`ldr r0, [sp, #+56]`
`bl      __muldf3`	`ldr r1, [sp, #+60]`

Figure 11. Basicmath Benchmark Assembly Obtained by GCC(left) and by LLVM(right)

2) *stringsearch*

'stringsearch' benchmarks, literally, the benchmark looking for given word in the sentence.

```
for (i = 0; find_strings[i]; i++) {
    init_search(find_strings[i]);
    here = strsearch(search_strings[i]);
}
```

Figure 12. A Part Of Stringserch Benchmark

Because 'stringsearch' benchmark is made up simple function calls, there are little difference between LLVM and GCC. When link-time optimization of LLVM is applied during merging bitcodes, there is a big performance improvement. The performance with link time optimization of LLVM reaches 92% over GCC.

`mov    r5, sp`	`ldr r0, [r4]`
`mov    r6, #0`	`bl strlen`
`.L27:`	`mov r1, #1, 24`
`bl     init_search`	`.LBB1_15:`
`ldr    r0, [r4, r6]`	`cmp r1, #0`
`bl     strsearch`	`bne .LBB1_15`
`ldr    r0, [r5, #4]!`	`.LBB1_16:`
`cmp    r0, #0`	`cmp r0, #0`
`add    r6, r6, #4`	`beq .LBB1_19`
`bne    .L27`	`.LBB1_17:`
	`mov r1, #0`
	`.LBB1_18:`
	`add r1, r1, #1`
	`cmp r1, r0`
	`blo .LBB1_18`
	`.LBB1_19:`
	`ldr r0, [r4, #+4]!`
	`cmp r0, #0`
	`bne .LBB1_14`

Figure 13. Stringsearch Benchmark Assembly Obtained by GCC(left) and by LLVM with Link Time Optimization(right)

In the Figure 13, there are 'bl' instructions which call 'init_search' function and 'strsearch' function on the assembly obtained by GCC. But there is no 'bl' instruction on the assembly obtained by LLVM with link-time optimization.

'llvm-ld' merges several LLVM bitcode files into one LLVM bitcode file. In this process, LLVM perform link-time optimization. This optimization is almost inter-procedure optimization. The right of Figure 13 is the inlined 'init_search' in the assembly obtained by LLVM with link-time optimization. As shown that, the link-time optimization of LLVM can perform powerful inter-procedure optimization including 'method inlining' [13].

3) *dijkstra*

'dijkstra' benchmark makes a big graph represented adjacency matrix, and get the shortest path between two nodes using Dijkstra algorithm. Dijkstra algorithm is well-known solution for shortest path problem. It takes O(n) time.

```
for (i = 0; i < NUM_NODES; i++) {
  if ((iCost = AdjMatrix[iNode][i]) != NONE)  {
     if ((NONE == rgnNodes[i].iDist) ||
            (rgnNodes[i].iDist > (iCost + iDist))) {
             rgnNodes[i].iDist = iDist + iCost;
             rgnNodes[i].iPrev = iNode;
             enqueue (i, iDist + iCost, iNode); }}}
```

Figure 14. Dijkstra Benchmark Core Loop

As shown in Figure 14, 'dijkstra' which uses matrix needs many memory accesses. The difference of memory accesses bring about the significant performance difference between LLVM and GCC. Indeed, there was a difference of 'ldr' instructions over the 10 in 'dijkstra' function which actually perform Dijkstra algorithm, there was also the difference of the number of 'str' and 'ldr' in the loop.

In the assembly obtained by GCC, the loaded value is saved in register, and reused. But, in the assembly of LLVM, it is loaded whenever memory assesses occurs. There is little difference whether through LLVM bitcode or not. If link-time optimization of LLVM applied, the performance increased significantly.

```
ldr  r2, .L35+28                 str r12, [r2]
add  r3, r3, r3, asl #2          str r12, [r2, #+4]
add  r3, ip, r3, asl #2         ......
ldr  r3, [r2, r3, asl #2]        str r2, [r3, #+4]
ldr  r2, .L35+32                 bl enqueue
cmp  r3, r8                      ldr r0, .LCPI4_2
str  r3, [r2, #0]                ldr r4, [r0]
beq .L25                         cmp r4, #
ldr  r2, [r7, ip, asl #3]        blt .LBB4_14
cmp  r2, r8                     ......
beq  .L34                       .LBB4_5:
ldr  r1, [r9, #0]                ldr r0, .LCPI4_3
add  r1, r3, r1                  ldr r0, [r0]
                                 cmp r0, #
```

Figure 15. Dijkstra Benchmark Assembly Obtained by GCC(left) and by LLVM(right)

4) *patricia*

The patricia tree is a data structure which represents routing table in network applications. This benchmark gets IP traffics as an input.

```
phead = (struct ptree *)malloc(sizeof(struct ptree));
memset(phead, 0, sizeof(*phead));
phead->p_m = (struct ptree_mask *)malloc(
            sizeof(struct ptree_mask));
memset(phead->p_m,0, sizeof(*phead->p_m));
pm = phead->p_m;
pm->pm_data = (struct MyNode*)malloc(sizeof
      (struct MyNode));
memset(pm->pm_data, 0, sizeof(*pm->pm_data));
```

Figure 16. A Part Of Patricia Benchmark

As shown above, many pointer operations are executed. And memory accesses of LLVM are more than that of GCC. Like 'dijkstra' benchmark', LLVM treats memory access operations ineffectively compared to GCC. The performance difference of 'patricia' benchmark is less than that of 'dijkstra', because 'patricia' doesn't have a big loop like 'dijkstra'.

There is no significant difference whether the assembly code is generated through LLVM bitcode or not. However, link-time optimization of LLVM plays an important role in improving the overall performance.

V. Conclusion

With the growing number of diverse embedded platforms, the needs for virtual machines will be increased. C programming language is a very popular one, and a virtual machine that can support existing C programs is highly desired. Though it is not perfect currently, LLVM could be a candidate for that, and we tried to estimate the possibility with simple benchmarks

A Linux-based ARM smartphone was used to run CaffeineMark and MiBench, designed for embedded platform. In general, LLVM shows very competitive performance, near that of GCC in some benchmarks. Still, in some other benchmarks, there is a significant difference when compared to GCC.

By analyzing generated code, we found out that LLVM is insufficient at strength reduction optimization, elimination of loop invariant value, and method inlining compared to GCC. Especially when there are many memory accesses, LLVM handled it inefficiently. The interprocedural optimization during linking several bitcode files into one bitcode file has a large effect in some benchmarks with many function calls. Because most real programs are composed of many files and have frequent function calls, LLVM would show a competitive performance with them.

References

[1] J. Smith and R. Nair, Virtual Machines, Elsevier, 2005

[2] J. Gosling, B. Joy, and G. Steele, The Java Language Specification Reading, Addison-Wesley,1996.

[3] Luis M. Vaquero et al., A Break in the Clouds: Toward a Cloud Definition, ACM SIGCOMM, Volume 39, Issue 1 January 2009, Pages 50–55

[4] LLVM Home, http://www.llvm.org

[5] Chris Lattner, M.S. Thesis, LLVM: An Infrastructure for Multi-Stage Optimization Computer Science Dept., University of Illinois at Urbana-Champaign, Dec. 2002.

[6] Vikram Adve et al., LLVA: A Low-level Virtual Instruction Set Architecture, MICRO-3, SanDiego,CA,Dec.2003

[7] Chris Lattner, Introduction to the LLVM Compiler Infrastructure, 2006 Itanium Conference and Expo, San Jose, California, Apr. 2006.

[8] CaffeinMark 3.0, Pendragon Software Corporation, http://www.benchmarkhq.ru/cm30/index.html

[9] MiBench, a free, commercially representative embedded benchmark suite, http://www.eecs.umich.edu/mibench/

[10] http://wiki.openmoko.org/wiki/Neo_FreeRunner

[11] RM Architecture Reference Manual, ARM DDI0100I

[12] J. C. Huang et al., Generalized Loop-Unrolling: a Method for Program Speed-Up, Department of Computer Science, The UniversityofHouston , 1999

[13] http://en.wikipedia.org/wiki/Inline_expansion