Proceedings of the International MultiConference of Engineers and Computer Scientists 2011 Vol I,
IMECS 2011, March 16 - 18, 2011, Hong Kong

# Fast Instruction Set Simulation Using LLVM-based Dynamic Translation

Claude Helmstetter, Vania Joloboff[1] Zhou Xinlei, Gao Xiaopeng[2]

***Abstract*—In the development of embedded systems, Instruction-Set Simulators (ISS) plays an important role. When using an ISS, simulation speed is a significant issue. In this paper, we present a dynamic translation technique that uses the LLVM open-source compiler infrastructure to increase the simulation speed. Our dynamic translation technique translates hot basic blocks of the target instruction set into LLVM bitcode, and compiles LLVM bitcode into host binary code using the LLVM Just-In-Time (JIT) compiler. We have simulated the same programs using LLVM-based dynamic translation and using traditional dynamic translation to compare their performance. The experiments show that the dynamic translation based on LLVM increases simulation speed.**

***Index Terms*—Instruction Set Simulation, binary dynamic translation, computer architecture, compilation, LLVM.**

## I. Introduction

Instruction-Set Simulators (ISS) are widely used tools for studying new architectures or developing software closely related to hardware such as operating systems or embedded systems applications.

An ISS is used to emulate the behavior of a target processor on a simulation host machine. The main task of an ISS is to carry out the computations that correspond to each instruction and maintain correct state of the simulated target processor.

There are several alternatives to achieve such simulation. In *interpretive simulation*, such as in popular Simplescalar [1] simulator, each instruction of the target program is fetched from memory, decoded, and finally executed.

Given that decoding is time-consuming and that instructions are generally executed many times, simulation can be sped up by translating and caching the result of the decoding phase. This is called *dynamic translation* [2]. The decoder output, i.e. the *translated code*, can be more or less optimized. Obviously, stronger optimization implies longer translation time.

In this paper, we present an ISS with two dynamic translation modes, which are complementary. The first mode translates the code quickly into an intermediate representation, allowing pretty fast simulations. This representation is *executable* in the sense that it uses objects with an *execute* method, but it does not consist of native code for the host machine.

In order to simulate even faster the time-critical parts, we added a second translation mode that uses the LLVM [3] library to optimize and compile the intermediate representation to native code.

This ISS is integrated in a SystemC module [4], and uses Transaction Level Modeling based on the OSCI TLM-2.0.1 standard [5] for communications with other simulation models, making it compatible with third-party components developed using the same standards.

This ISS is now part of the SimSoC open-source simulator [6] from INRIA, since version 0.7.

This paper is structured as follows. Section II details some close related work. Next, the new translation mode based on LLVM is described in Section III. We have carried out experiments, whose results are presented in Section IV. Finally, Section V concludes the paper.

## II. Related Work

There has been many papers on Instruction-Set Simulators, After the early days of interpretive simulation, several systems have successfully attempted to improve performance with dynamic translation starting with the Shade [2] and Embra simulators [7].

The JIT-CCS simulator [8] introduced the technique that we use in our so-called DT2 mode (for historical reasons) detailed in the next section. The code for simulating individual simulation operations is coded in C or C++, manually coded or generated (in our case generated C++ for ARM V6, and manually for PowerPC). The dynamic translator generates and caches a data structure with references to these operations to re-execute them. This is a fast method relatively easy to implement and provides a basis to compare performance enhancements.

The heart technology of SimIt [9] is a simulation engine capable of mixed interpretive and compiled simulation. To increase simulation speed, it enables to distribute the tasks involved in binary translation to other processors. While the simulator interprets target instructions it generates profiling statistics for selecting frequently executed pages to compile. When the execution count for a page exceeds a predefined threshold, it is compiled by GCC into a shared library which is loaded at runtime. This is indeed native host translation but as it is generating C and invoking GCC, it creates a latency that is only worthwhile for long simulations.

The Edinburgh High Speed (EHS) simulator [10] has two simulation modes: one is an interpretive mode and the other is a dynamic binary translation (DBT) mode. In EHS simulator, the translation units are Large Translation Units(LTU). LTU is a group of basic blocks connected by control flow arcs,

[1]LIAMA FORMES project, Tsinghua University, FIT building Beijing 100084, China, [claude.helmstetter,vania.joloboff]@inria.fr

[2]State key Laboratory of Software Development Environment Beijing University of Aeronautics and Aerospace, Beijing 100191, China, [zhouxinlei,gxp]@buaa.edu.cn

ISBN: 978-988-18210-3-4
ISSN: 2078-0958 (Print); ISSN: 2078-0966 (Online)

which may have several entry and exit points. Each translation-unit is translated into a C code function that simulates the target instructions. The functions are compiled by GCC into a shared library which is loaded by the dynamic linker. EHS simulator profiles the target program's execution in order to discover frequently taken paths (*hot paths*) rather than to identify frequently executed blocks.

Rapido [11] uses dynamic compilation with LLVM. Hot basic blocks are grouped into regions when specified threshold has been reached. A region is compiled into a LLVM function which contains only a single entry and without other restrictions. A region is the translation-unit of this simulator. It means that a region may contain loops, and then interrupts may not be checked for accurately. At compilation various optimization passes are invoked by simulator that decides which optimization pass to apply. Compared simulation speeds of the interpreter and the translator for MIPS and CHILI shows that the translator is up to 500 times faster for the longer running benchmarks.

QEMU [12] is a fast machine simulator which uses an original portable dynamic translator. Each target instruction is split into fewer simpler instructions called micro operations. The micro operations have been pre-compiled offline into an object file. The compile time tool called *dyngen* uses the object file as input to dynamically generate code with sort of a "copy and paste" of the micro-instructions. This dynamic code generator is invoked at run-time to generate and link complete host functions which concatenates several micro operations.

The project *llvm-qemu* [13] uses components of the LLVM compiler infrastructure to modify the QEMU dynamic translator to increase the performance of QEMU. Instead of directly emitting code for the host architecture QEMU is running on, the micro instructions are first translated to LLVM intermediate representation (IR) , then a selection of LLVM's optimization functions are applied to the IR and the LLVM JIT is used to generate code from the optimized IR for the host architecture. This is similar to our work, but no performance has been published as of this writing, making comparison difficult.

The library *libcpu* [14] is claiming to improve its architecture with LLVM, but it is not available yet.

## III. Dynamic Translation with LLVM

### A. Previous work

The SimSoc simulator is implementing four kinds of instruction simulation corresponding to four modes that the simulator can run in. It can simulate several architectures, but in this section, we will focus on the PowerPC architecture.

The first mode, named DT0, is purely interpretive simulation. Each instruction of the target program is fetched from memory, decoded, and executed, when simulator runs. This method is flexible and easy to implement, but the simulation speed is slow as it wastes a lot of time in decoding. It however provides a basis from which one can fairly compare performance with other simulation modes, for the same host machine and the same application program. The second mode DT1 is a simple minded dynamic cache translation, with no optimization, that makes it possible to evaluate the performance of the cache technique over DT0.

The third mode, named DT2, is dynamic cached translation with optimization. In this mode each type and variant of an instruction has a class structure corresponding to it. For example, the PowerPC *add* instruction corresponds to the `PPC_add` class, the *stw* instruction corresponding to `PPC_stw` class and so on. These instruction instances store all information obtained from the instruction decoding. Such information include for example operand registers, target registers, immediate values, and include a reference to the execution function. This mode also uses a partial evaluation technique at decoding time to possibly specialize each instruction into a more specialized execution function.

DT2 mode shows the performance improvement obtained with optimized dynamic translation compared to simple DT1. The benchmarks results in [15] show that simulation speed vary from 9.5 Mips in DT0 mode, to more than 30 Mips in DT1, to reach close to over 70 Mips in DT2. In average DT2 mode is between five to ten times faster than DT0 mode.

### B. Dynamic compilation

Our work is about introducing a new mode, named DT3, with dynamic translation based on LLVM.

LLVM is a Low Level Virtual Machine [3] that has been designed to serve as intermediate representation in compilers suitable for complex optimizations. LLVM consists in an abstract instruction set, each instruction having well defined semantics. An LLVM program can be interpreted directly using the LLVM interpreter, or compiled to machine code. The code generation can be done either with a JIT compiler or a batch compiling phase. It contains a complete set of high-level compiler optimizations, ranging from simple scalar simplifications to complex loop transformations.

In the DT3 mode, our translation-unit is a *Basic block*, a straight sequence of code with only one entry point and only one exit, with a branch instruction at the end. That is to say, all instructions from a basic block will certainly be executed when it is entered. The idea is to compile each basic block into a linear simulation function that does not contain any control flow instruction, which allows fast translation.

Below is an example of a basic block of PowerPC instructions to be translated into an LLVM function:

```
addis r9, r0, 385
lwz   r0, 1076 (r9)
or    r1, r0, r0
bl    0xffffff70
```

To translate a basic block to LLVM, we first create an LLVM function, containing a single LLVM block `entry`. This LLVM function has a parameter `%proc` that holds the processor state. Then, for each instruction, we generate a call to the corresponding execution function, which must be defined by LLVM code. The implementations of the LLVM execution functions are stored in a LLVM bitcode library, whose generation is explained below. For example, the instructions

ISBN: 978-988-18210-3-4
ISSN: 2078-0958 (Print); ISSN: 2078-0966 (Online)

addis and lwz are translated to specialized llvm function calls to corresponding functions addis_ra0 and lwz_raS. Each instruction is followed by a function call to update the value of the PC register. The status returned by an execution function tells whether a branch has occurred; by definition, all status but the last tell that no branch occurred.

Thus, the basic block above is translated to the following LLVM function.

```
define void @bb_687 (%"struct.Proc"* %proc) {
entry:
 %status = call i32 @addis_ra0(%"struct.
                Proc"* %proc, i8 9, i32 385)
 call void @inc_pc(%"struct.Proc"* %proc)
 %status1 = call i32 @lwz_raS(%"struct.
          Proc"* %proc, i8 0, i8 9, i32 1076)
 call void @inc_pc(%"struct.Proc"* %proc)
 %status2 = call i32 @or(%"struct.
             Proc"* %proc, i8 0, i8 1, i8 0)
 call void @inc_pc(%"struct.Proc"* %proc)
 %status3 = call i32 @bl(%"struct.
                      Proc"* %proc, i32 -144)
 call void @inc_pc_if_no_branch(i32 %status3,
                       %"struct.Proc"* %proc)
 ret void
}
```

When a basic block has been constructed, one can use LLVM optimization functions at will. In particular, we systematically call the *AlwaysInline* optimization first so that all the code of the execution functions is actually inlined, and thus available for further optimizations. Next, other optimizations can be accomplished. For example, LLVM will reduce the $K$ successive calls to inc_pc() inlined functions into a single addition of $K \times 4$ to the PC when the PC variable is never read. In general, after the *AlwaysInline* pass, we apply the LLVM optimization passes named *GVNPass*, *InstructionCombiningPass*, *CFGSimplificationPass*, and *DeadStoreEliminationPass*.

After the LLVM optimization passes, we call the LLVM JIT compiler to compile LLVM bitcode into host binary code. Then we update the instruction cache so that this optimized binary code is called instead of the DT2 simulation function.

As it is much easier to write C++ code than LLVM bitcode, to obtain the LLVM library, we start from a library of C++ functions that we compile into a LLVM library prior to simulation, using the llvm-g++ compiler. All the instructions implemented in C++ code for the PowerPC instruction set are stored in the file ppc_llvm_lib.cpp. Using the llvm-g++ compiler, we generate the file ppc_llvm_lib.bc, which contains the corresponding LLVM bitcode.

As an example, here is the C++ code implementing the PowerPC *add* instruction:

```
extern "C" PseudoStatus ppc_add
          (Proc &proc, u8 rt, u8 ra, u8 rb) {
 const uint32_t a = proc.cpu.gpr[ra];
 const uint32_t b = proc.cpu.gpr[rb];
 proc.cpu.gpr[rt] = a+b;
 return OK;
}
```

And here is the LLVM bitcode generated by llvm-g++:

```
define i32 @ppc_add(%"struct.Proc"* nocapture
       %proc, i8 zeroext %rt, i8 zeroext %ra,
                  i8 zeroext %rb) nounwind {
entry:
 %0 = zext i8 %ra to i64;
 %1 = geteleptr inbounds %"struct.Proc"*
          %proc, i64 0, i32 2, i32 4, i64 %0;
 %2 = load i32* %1, align 4;
 %3 = zext i8 %rb to i64;
 %4 = geteleptr inbounds %"struct.Proc"*
          %proc, i64 0, i32 2, i32 4, i64 %3;
 %5 = load i32* %4, align 4;
 %6 = add i32 %5, %2;
 %7 = zext i8 %rt to i64;
 %8 = geteleptr inbounds %"struct.Proc"*
          %proc, i64 0, i32 2, i32 4, i64 %7;
 store i32 %6, i32* %8, align 4
 ret i32 0
}
```

### C. Profiling and compilation threshold

On average, a program spends a lot of time to execute a small portion of its code. Since translation to LLVM is costly, an idea to speed up simulation, already exploited in JIT-CCS [8], is to only translate that small portion of code, whereas the remaining code might only execute once or only a few times and the extra time spent to generate the optimized code would not pay off. Therefore we need to find out the frequently executed basic blocks. To solve this problem we add a counter $C$ for each basic block. When the counter has reached a specified threshold $CT$ the basic block is identifies as a *hot* basic block. The value of the $CT$ threshold is a run time parameter. Only hot basic blocks are compiled into LLVM bitcode.

This is of course optimistic, hoping that blocks executed frequently in the past will be executed frequently in the future.

## IV. Validation and Performances

The dynamic translation based on LLVM is now available in SimSoC version 0.7. Some tests were already written to test dynamic translation. We reused them to test the new dynamic compilation, and all the tests worked well in the DT3 mode.

In this paper, we consider three benchmark programs that we have written to test the performance of our simulator, named *"loop"*, *"sorting"*, and *"crypto"*. The *loop* program is a simple loop, the *sorting* program executes many sorting algorithm, and the *crypto* program is a more complex cryptographic program using functions from the XYSSL library. We have compiled *sorting* and *crypto* using different optimization options: a first time with optimization (-O3) and a second time without (-O0) .

### A. Simulation speed of the compiled code

First, we measured the results with the compilation threshold $CT$ set to 1. That is to say, all basic blocks that are executed at least twice were compiled (For technical reasons not detailed here, it is not possible to compile a basic block

ISBN: 978-988-18210-3-4
ISSN: 2078-0958 (Print); ISSN: 2078-0966 (Online)

TABLE I
COMPILATION AND SIMULATION TIME

| | total time − compil. = simul. | DT2 |
|---|---|---|
| crypto-O0 | 31.67 s − 30.32 s = 1.35 s | 2.92 s |
| crypto-O3 | 14.50 s − 13.50 s = 1.00 s | 2.77 s |
| loop | 1.03 s − 0.02 s = 1.01 s | 1.58 s |
| sorting-O0 | 4.13 s − 2.64 s = 1.49 s | 3.36 s |
| sorting-O3 | 3.62 s − 2.03 s = 1.59 s | 3.29 s |
| total | 54.95 s − 48.51 s = 6.44 s | 13.92 s |

before its first execution, and so $CT = 0$ is not feasible). The results are detailed in Table I.

We can see that the compiled code (DT3) is more than twice as fast as the simply-translated code (DT2, not using LLVM). Given that the five compiled benchmarks total up to 1,018 millions of simulated instructions, the simulated speed of the compiled code $S_C$ is 158 Mips on average, whereas the speed of the DT2 mode $S_T$ was 73 Mips.

However, we notice that the runtime compiler itself is slow. Summing up the five benchmarks, 34,548 instructions in 1,500 basic blocks were compiled. Thus, on average, one can compile only $C = 712$ instructions per seconds. It results that the total simulation time is smaller only for the *loop* benchmark whose binary code is very short. That is the rationale to use a compilation threshold.

### B. Speed of the runtime compiler

Regarding speed of the runtime compiler, Fig. 1 shows the relation between the size of a basic block and its compilation time, using the same benchmarks than above. It appears that a constant time of about 10 ms adds to the compilation time of any basic blocks, and after that compile time is linear with block size. Thus, very small blocks are less interesting to translate than large blocks.

### C. Overall speed and best threshold value

Theoretically, we know that if a binary instruction is executed $N$ times, then its cost using the DT2 mode is $N \times S_T^{-1}$,

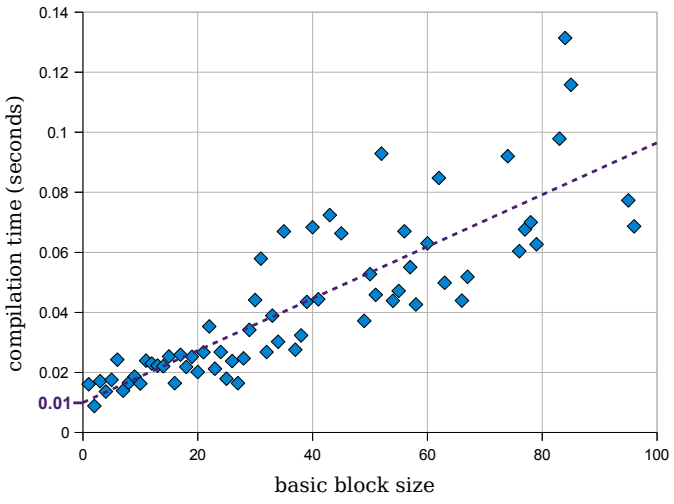


Fig. 1. Relationship between size of blocks and their compilation time

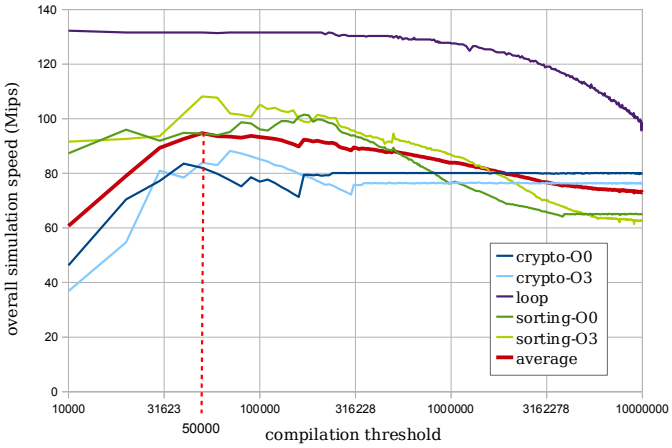


Fig. 2. Effect of the compilation threshold on the simulation speed

where $S_T$ is the speed of the translated code, and its cost using the DT3 mode is $C^{-1} + N \times S_C^{-1}$, where $S_C$ is the speed of the compiled code and $C$ the speed of the compiler. Consequently, compiling an instruction is paying off only if:

$$N > \frac{C^{-1}}{S_T^{-1} - S_C^{-1}} \approx \frac{1/712}{1/(73{\cdot}10^6) - 1/(158{\cdot}10^6)} \approx 190{\cdot}10^3.$$

So, we expect that the best compilation threshold value $CT$ should be in the same order of magnitude than 190,000. For this experiment, we tested the same benchmarks with different values of $CT$ in DT3 mode, and we compare the overall simulation speeds.

Fig. 2 shows that if the value of $CT$ is small, most basic blocks counters exceed the threshold, thus much time is spent compiling basic blocks which are not really *"hot blocks"*; the compile time is quite long and consequently the simulation speed is lower than DT2 mode. When increasing the compilation threshold, less blocks get compiled and the simulation speed is going up, above the DT2 speed. But when the value of $CT$ is too large, the number of compiled blocks decreases towards none. An infinite value of $CT$ means that any basic block counter can never exceed the threshold, and the whole simulation is done using the intermediate representation of the DT2 mode.

The users should run the simulations with an optimized value of $CT$, so that the simulation speed will reach its peak. According to Fig. 2, the best value is around 50,000. Using this value, our new DT3 mode is 29% faster than the previous DT2 mode. However, there are significant differences: the DT3 mode is more interesting for program with long execution but short binary code, or at least short hot sections. For example, the DT3 mode is not interesting for the crypto benchmark because the control flow never stay in the same function for a long time. However, if one does cryptographic computation during a longer time, (e.g., one minute instead of 3 seconds), then the DT3 mode will become advantageous.

ISBN: 978-988-18210-3-4
ISSN: 2078-0958 (Print); ISSN: 2078-0966 (Online)

## V. Conclusion

In this paper we presented a new simulation mode in SimSoc which is called dynamic translation based on LLVM. This approach compiles target instructions into LLVM bitcode, followed by several optimization passes invoked during compilation. We have tested five benchmark programs on this simulation mode, and the results demonstrate that it is faster than the DT2 mode.

With the technique exposed, we found that a reasonable value of the $CT$ threshold is relatively high. To decrease this threshold to a value that would be suitable to more applications, we must shorten the compilation time and increase the speed of the compiled code. Our future work to further increase simulation speed will thus concentrate on parallelizing the compilation phase, and exploring larger translation units with higher optimization of these units.

## Acknowledgments

This work is supported by the fund of the State Key Laboratory of Software Development Environment (No.SKLSDE-2009ZX-02) and the Aviation Research Foundation (ARF) grant 20081951032.

## References

[1] T. Austin, E. Larson, and D. Ernst, "SimpleScalar: an infrastructure for computer system modeling," *Computer*, vol. 35, no. 2, pp. 59–67, 2002.

[2] B. Cmelik and D. Keppel, "Shade: A fast instruction-set simulator for execution profiling," in *Proceedings of the 1994 ACM SIGMETRICS Conference on the Measurement and Modeling of Computer Systems*, May 1994, pp. 128,137.

[3] C. Lattner and V. Adve, "LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation," in *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*, Palo Alto, California, Mar 2004.

[4] *SystemC v2.2.0 Language Reference Manual (IEEE Std 1666-2005)*, Open SystemC Initiative, 2006, http://www.systemc.org/.

[5] "OSCI TLM-2.0 language reference manual," Open SystemC Initiative, 2009, http://www.systemc.org/.

[6] C. Helmstetter, V. Joloboff, and H. Xiao, "SimSoC: A full system simulation software for embedded systems," in *OSSC'09*, IEEE, Ed., 2009.

[7] E. Witchel and M. Rosenblum, "Embra: fast and flexible machine simulation," *SIGMETRICS Perform. Eval. Rev.*, vol. 24, no. 1, pp. 68–79, 1996.

[8] A. Nohl, G. Braun, O. Schliebusch, R. Leupers, H. Meyr, and A. Hoffmann, "A universal technique for fast and flexible instruction-set architecture simulation," in *Proceedings of the 39th annual Design Automation Conference*, ser. DAC '02. New York, NY, USA: ACM, 2002, pp. 22–27. [Online]. Available: http://doi.acm.org/10.1145/513918.513927

[9] W. Qin, J. D'Errico, and X. Zhu, "A multiprocessing approach to accelerate retargetable and portable dynamic-compiled instruction-set simulation," in *CODES+ISSS '06: Proceedings of the 4th international conference on Hardware/software codesign and system synthesis*. New York, NY, USA: ACM, 2006, pp. 193–198.

[10] D. Jones and N. Topham, "High speed cpu simulation using ltu dynamic binary translation," in *HiPEAC '09: Proceedings of the 4th International Conference on High Performance Embedded Architectures and Compilers*. Berlin, Heidelberg: Springer-Verlag, 2009, pp. 50–64.

[11] F. Brandner, A. Fellnhofer, A. Krall, and D. Riegler, "Fast and accurate simulation using the llvm compiler framework," in *Rapid Simulation and Performance Evaluation: Methods and Tools (RAPIDO'09)*, O. T. Smail Niar, Rainer Leupers, Ed. Paphos, Cyprus: HiPEAC, January 2009, pp. 1–6.

[12] F. Bellard, "Qemu, a fast and portable dynamic translator," in *ATEC '05: Proceedings of the annual conference on USENIX Annual Technical Conference*. Berkeley, CA, USA: USENIX Association, 2005, pp. 41–41.

[13] T. Scheller, "llvm-qemu, backend for QEMU using LLVM components," 2007, http://code.google.com/p/llvm-qemu/.

[14] M. Steil, O. Bassotto, and G. Guida, "libcpu, anything to anything binary translation," 2010, http://www.libcpu.org/wiki/Main_Page.

[15] H. Hongwei, S. Jiajia, C. Helmstetter, and V. Joloboff, "Generation of executable representation for processor simulation with dynamic translation," in *Proceedings of the International Conference on Computer Science and Software Engineering*. Wuhan, China: IEEE, 2008.