Fast Dynamic Translation Using LLVM On Multi-Core Hosts

Zuyu Zhang[1], Vania Joloboff[2], Xinlei Zhou[3], Claude Helmstetter[2], Guoyin Zhang[1]

***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 experiments and comparison between several dynamic translation techniques. In addition to an existing technique which serves as a reference, we have developed a new on-the-fly translation technique using the LLVM open-source compiler infrastructure to enhance simulation speed. This dynamic translation technique translates hot basic blocks of the target instruction set into LLVM bitcode, then compiles LLVM bitcode into host binary code using the LLVM Just-In-Time (JIT) compiler. As the translation time using LLVM increases to the detriment of the overall simulation speed, we also present a mixed mode, where only the frequently executed chunks of code are compiled. This translation technique was then extended to support larger translation units compared to the previous. Finally the paper ends with presentation of an orthogonal solution to dispatch dynamic translation to a translation server to take advantage of multi-processor hosts.**

***Index Terms*—Instruction Set Simulation, Dynamic Binary Translation, LLVM, Hot Path, Parallel Simulation**

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 to carry out the computations that correspond to each instruction and maintain correct state of the simulated target processor.

Because binary decoding is time-consuming and instructions are generally executed many times, simulation can be accelerated by translating and caching on-the-fly the result of the target decoding phase. This is called *dynamic binary translation*. The decoded output, i.e. the *translated code*, can be more or less optimized. Stronger optimization usually implies longer translation time and there are trade-offs to be made between translation time and execution time to eventually obtain shorter overall simulation time.

We have used as code base the SimSoC open-source simulator [1] that includes an ISS integrated as a SystemC module [2] that uses Transaction Level Modeling for communications with other simulation models, making it compatible with third-party components developed using the same standards. This ISS has several runtime modes, each using a different style of dynamic translation so that one can compare performance of each mode on the same benchmarks.

The first mode, which is a base reference, translates the target code, using a fast translation technique, into an intermediate representation. This representation is *executable* in the sense that it uses objects with associated methods, but it does not consist of native code for the simulation host machine. As it uses partial interpretation compiler techniques it can however reach simulation speed of over 50 Millions instructions per second (Mips), including the MMU simulation. This ISS is now standard mode of the SimSoC simulator.

To increase simulation speed, a second dynamic translation mode was added, which uses the LLVM [3] library to translate on the fly target code into LLVM, and then use the existing LLVM Just-In-Time compiler to generate native code. The obtained results are presented, showing increase in execution speed.

The translation time from target code to LLVM, next from LLVM to native, can become lengthy and ultimately defeat the speed-up in execution time. Thus, we have implemented a third mixed translation mode with a method to evaluate and select only "hot path" code so that the LLVM translation is not systematic, but only operates on such hot paths, the remaining code being simulated with the cached translation. This effectively provides overall faster simulation.

In the mixed mode above, translation to native code is achieved on a basic block basis. We have also experimented another mode with larger translation units, by dynamically determining strongly coupled basic blocks in the control flow graph. This technique provides better execution performance but increases translation time, and also raises issues with regards to interrupts handling.

Finally, in order to benefit from multi-core simulation hosts, a distributed dynamic translation mechanism has been experimented. In that mode, the native code translation is achieved by a separate dynamic translation server, which runs concurrently with the ISS on other processors.

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. Then it details the techniques used for dynamic binary translation in the different modes. For each dynamic translation mode, tables are provided with a selected set of benchmarks to compare the execution timings and the simulation timings obtained in the different mode, with comments on the trade offs made to maximize overall simulation speed. Finally, Section V concludes the paper.

[1]Harbin Engineering University, China, hitzzy@gmail.com

[2]International FORMES project, INRIA and Shenzhen Institute of Advanced Technologies, China, [claude.helmstetter,vania.joloboff]@inria.fr

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

II. Related Work

There has been much work done around Instruction-Set Simulation. In order to reach high simulation speed, one must describe the hardware with a higher level of abstraction than the Cycle Accurate models achieved by the hardware designers. Two popular instruments to describe higher abstraction level of hardware models are SystemC [2] and Transaction Level Modeling [4]–[7], widely used in industry and academia. They provide flexible abstraction level to implement either Cycle Accurate, Loosely (or Approximately) Timed, or timeless models.

Rising the abstraction level is not the only way to achieve higher speed. Optimization techniques related to compiler technology and operating systems technology can be used to implement fast simulators, whether or not they are based on SystemC, and many efforts have been done to improve Instruction-Set-Simulator (ISS) speed. Whereas the early ISS's used interpretive simulation [8], most recent ISS's have used some kind of dynamic cached translation to accelerate simulation. Because it works on the real code at run time, *dynamic binary translation* [9]–[14] has been favored.

In order to translate the target binary code into simulation host code (typically Intel architecture), several techniques are possible. The heart technology of SimIt [15] is a simulation engine capable of mixed interpretive and compiled simulation. 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 generates C code equivalent to the target operations performed, then this code is compiled with GCC into a shared library which is loaded to replace the interpreted code. 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. To increase simulation speed, it enables to distribute the compiling tasks involved in binary translation to other processors. The Edinburgh High Speed (EHS) simulator [16] 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, 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 [17] 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.

Identification of hot paths and strongly coupled blocks of code is a technique that has been used in compilers or other translators such as JCOD [18] or UQDBT [19].

QEMU [20] 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 dynamic translation code generator invoked at run-time generates and links complete host functions which concatenates several micro operations.

The project *llvm-qemu* [21] 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.

In conclusion, to dynamically translate target code into host code, one has to generate some representation and then compile this representation into native code. This representation can be C code as in SimIt or EHS, or it can be a lower level representation, that can be more easily translated into native code. Building a robust, efficient Just In Time compiler represents a large and long term effort, which is why it was decided to take advantage of LLVM.

III. Dynamic Translation with LLVM

A. Previous work

The SimSoC simulator as released in open source is implementing three kinds of instruction simulation corresponding to modes that the simulator can run in. It can simulate several architectures, but in this paper the PowerPC architecture is used as a reference.

The first mode, named DT0, is purely interpretive simulation. Each instruction of the target program is fetched from memory, decoded, and executed. 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 JIT-CCS simulator [12] introduced the basic technique that is used in the DT1 mode: 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 [22], and manually for PowerPC). The dynamic translator then 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 good basis to compare performance enhancements.

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. For example, the PowerPC add instruction is specialized into two variants, one that does update the status register, and one that does not.

DT2 mode shows the performance improvement obtained with optimized dynamic translation compare to the simple DT1. The benchmarks results in [23] shows 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 new dynamic translation modes based on LLVM and trying to take advantage of multi-processor hosts.

LLVM is a common low-level toolchain infrastructure [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, an LLVM function is created, containing a single LLVM block `entry`. This LLVM function has a parameter `%proc` that holds the processor state. Then, for each instruction, a call to the corresponding execution function is generated. These functions are stored into a LLVM bitcode library, whose generation is explained below. For example, the instructions `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 or not a branch has 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, the *AlwaysInline* optimization is systematically called 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, the LLVM optimization passes named *GVNPass*, *InstructionCombiningPass*, *CFGSimplificationPass*, and *DeadStoreEliminationPass* are applied.

After the LLVM optimization, the LLVM JIT compiler is activated to compile LLVM bitcode into host binary code. Eventually the translation cache is updated that the resulting optimized native 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, a library of C++ functions is compiled into a LLVM library prior to simulation. All of the instructions implemented for the PowerPC are gathered into the C++ file `ppc_llvm_lib.cpp`. Using the `llvm-g++` compiler, the LLVM bitcode file `ppc_llvm_lib.bc` is obtained, and this file is dynamically loaded at simulation start-up.

As an example, below 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++` with optimization (-O3):

```
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 [12], 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 one needs to find out the frequently executed basic blocks. To that effect, a counter C is added for each basic block, counting the number of times this block has been executed. When the counter reaches a specified threshold CT, the basic block is identified as a *hot* basic block. Then, only hot basic blocks are compiled into LLVM bitcode. This is optimistic, hoping that blocks executed frequently in the past will also be executed frequently in the future, which may not be true.

The value of the CT threshold is a run time parameter, so one can study the effect of CT variations on performance.

IV. Validation and Performances

The dynamic translation based on LLVM is implemented as part of SimSoC. Some tests were already written to compare dynamic translation performance in DT1 and DT2 mode. They have been re-used to test the new dynamic compilation.

In this paper, we consider three benchmark programs that we have written to test the performance of our simulator, named *"loop"*, *"matrix"*, and *"sorting"*. The *loop* program consists of simple loops and only a few basic blocks. The *matrix* program performs a big block of simple matrix arithmetic. And the *sorting* program executes several sorting algorithms operating on tables stored in memory and consists of several hundreds blocks, but all are small blocks of less than 10 instructions. We have cross-compiled these benchmarks for PowerPC using different optimization options: a first time with optimization (-O3) and a second time without (-O0). When using no optimization, the compiler generates larger blocks, which is useful to study influence of block size.

A. Simulation speed of the compiled code

The first results were obtained with the compilation threshold CT set to 1. That is to say, all basic blocks that have been executed at least twice were compiled (For technical reasons not detailed here, it is not possible to compile a basic block before its first execution, and so $CT = 0$ is not feasible). The results are presented in Table I.

	DT3 with CT = 1			DT2
	total time	compil.	simul.	total.
loop-O0	30.958 s	0.121 s	30.837 s	85.457 s
loop-O3	8.633 s	0.041 s	8.592 s	18.661 s
matrix-O0	37.410 s	0.544 s	36.866 s	111.383 s
matrix-O3	6.032 s	0.364 s	5.668 s	11.629 s
sorting-O0	23.161 s	5.588 s	17.573 s	43.055 s
sorting-O3	7.020 s	3.627 s	3.393 s	7.004 s
total	113.214 s	10.285 s	102.929 s	291.109 s

TABLE I: compilation and simulation time

For each benchmark, it shows in the first column the total time, in the second column the time spent in compiling the blocks, third the execution time of the compiled code, and last the reference time of the DT2 mode.

One can see that the compiled code (DT3, using LLVM) is more than twice as fast as the simply-translated code (DT2, not using LLVM). Given that the six compiled benchmarks total up to 11,199 millions of simulated instructions, the simulated speed of the compiled code S_C is 99 Mips on average, whereas the speed of the DT2 mode S_T was 38 Mips.

However, one may notice that the runtime compiler itself is slow. Summing up the six benchmarks, 5,174 instructions in 619 basic blocks were compiled. Thus, on average, one can compile only $C = 503$ instructions per seconds. As a result, the total simulation time is smaller only for the *sorting-O3* benchmark whose binary code is highly optimized. 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 number of instructions and their compilation time, using the same benchmarks as above. It appears that a constant time of about 40 ms adds to the compilation time of any instructions, and after that the compile time is roughly linear with total instructions. Thus, we must

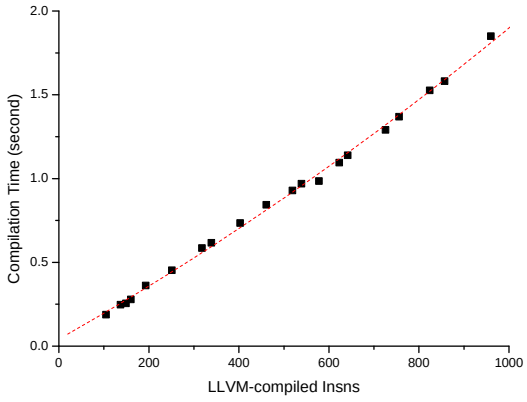

Fig. 1: Relationship between the compiled instructions and their compilation time

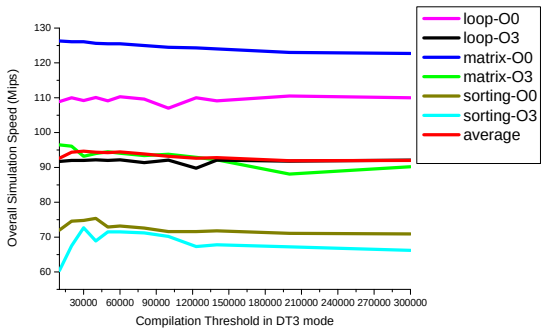

Fig. 2: Effect of the compilation threshold on the simulation speed

consider the appropriate numbers of compiled instructions due to gradually increase of the compilation time.

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 $N \times S_T^{-1}$, 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:

$$N > \frac{C^{-1}}{S_T^{-1} - S_C^{-1}} \approx \frac{1/503}{1/(38{\cdot}10^6) - 1/(99{\cdot}10^6)} \approx 123{\cdot}10^3.$$

So, we expect that the best compilation threshold value CT should be in the same order of magnitude as 123,000. For this experiment, we test 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 significant and consequently the overall simulation speed may lower than DT2 mode. When increasing the compilation threshold, less blocks get compiled and the simulation speed is going up, above the DT2 speed. However, 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 30,000. Using this value, our new DT3 mode is 63.4% faster than the previous DT2 mode. However, there are significant differences: the DT3 mode is more interesting for programs with long execution but short binary code, or at least short hot sections. For example, the DT3 mode is not interesting for the loop benchmark because their hot path are too short that there are no key improvements both for 1,000 and 300,000 as the basic block threshold. But, if one does iterations during a longer time,

	DT3 with heat				DT2
	total time	− compil. =	simul.		total.
loop-O0	31.228 s	− 0.129 s =	31.209 s		85.457 s
loop-O3	8.625 s	− 0.043 s =	8.582 s		18.661 s
matrix-O0	37.506 s	− 0.456 s =	37.050 s		111.383 s
matrix-O3	5.956 s	− 0.234 s =	5.722 s		11.629 s
sorting-O0	19.901 s	− 3.013 s =	16.888 s		43.055 s
sorting-O3	4.628 s	− 1.328 s =	3.300 s		7.004 s
total	107.844 s	− 5.203 s =	102.641 s		291.109 s

TABLE II: compilation with heat

(e.g., 100 fold iterations than before), then the DT3 mode becomes advantageous.

D. Using weight for the threshold

For the results displayed above, we used a simple counter to select the hot blocks, ignoring block size. However compiling large blocks has a better pay-off. So, we introduced a notion of *heat* to measure the *heat* of a hot block, and compile only the hottest blocks. The heat is a compound of frequency and block size, so that either very large blocks with a few executions or smaller blocks with a larger number of executions get compiled.

The measurements obtained show that we can improve performance again by selecting hot blocks with this method.

The threshold maximizing performance is reached with a heat of 20,000 instead of the former 30,000 counter value. We obtain a performance improvement compared to the mere frequency counter, and a performance gain compared to the DT2 mode.

E. Compiling with Macro-blocks

Another way to improve performance is to compile larger blocks than simple basic blocks. The method consists then in identifying strongly connected basic blocks, such as embedded loops. To that end, we need to profile, from the exit of a basic block, which are the following target basic blocks, and recursively find which target basic blocks are most frequently executed.

This profiling method leads to identifying frequent paths [19] across basic blocks, in particular to find strongly connected basic blocks such as generated by compilers for embedded loops.

When a frequent path is identified, one can construct a larger translation unit [16], named a macro-block that gathers all these basic blocks into a single unit, and compile then in a new macro-block.

To construct a macro-block start from a hot basic block, in addition to the basic block counter C, we introduce an edge counter EC. An Edge is a directed pair that denotes control flow from one basic block to its successor block. For example, the edge e from basic block a to basic block b is denoted: $e = (a \rightarrow b)$. The corresponding edge counter EC measures the execution times from a to b. If the counter EC reaches another specified threshold ET, then the successor block b should make up a macro-block. Let BB be the set of all basic blocks and E be the set of all possible edges. The set of hot successor blocks is defined as:

	DT3 with Macro-block total time − compil. = simul.	DT3 total.
loop-O0	23.225 s − 0.071 s = 23.154 s	31.254 s
loop-O3	4.072 s − 0.014 s = 4.058 s	8.561 s
matrix-O0	32.726 s − 0.440 s = 32.286 s	37.454 s
matrix-O3	3.744 s − 0.146 s = 3.598 s	5.884 s
sorting-O0	15.609 s − 2.071 s = 13.538 s	19.533 s
sorting-O3	2.444 s − 0.268 s = 2.176 s	3.856 s
total	81.820 s − 3.010 s = 78.810 s	106.542 s

TABLE III: Simulation with Macro-blocks

	DT3 with server total time − compil. = simul.	DT3 total.
loop-O0	31.262 s − 0.253 s = 31.009 s	31.254 s
loop-O3	8.761 s − 0.048 s = 8.713 s	8.561 s
matrix-O0	38.666 s − 1.698 s = 36.968 s	37.454 s
matrix-O3	6.284 s − 0.568 s = 5.716 s	5.884 s
sorting-O0	21.197 s − 2.873 s = 18.324 s	19.533 s
sorting-O3	4.332 s − 0.657 s = 3.675 s	3.856 s
total	110.706 s − 6.109 s = 104.597 s	106.542 s

TABLE IV: Simulation with compile server

$HotSucc(a) = \{b \in BB | \exists e \in E \bullet e = (a \rightarrow b) \wedge EC \geq ET\}$

From the hot basic block hbb and its edge counters, we only choose the hot successor blocks and construct our next set of hot blocks. We repeat this as follows:

$HotSet(0) = \{hbb\}$
$HotSet(1) = \{x \in BB | \forall y \in HotSet(0) \bullet x \in HotSucc(y)\}$
$HotSet(2) = \{x \in BB | \forall y \in HotSet(1) \bullet x \in HotSucc(y)\}$
...
$HotSet(n) = \{x \in BB | \forall y \in HotSet(n-1) \bullet x \in HotSucc(y)\}$

The final set of hot blocks derived from hbb will construct the macro-block set by performing the union of all the hot blocks sets:

$MacroBlock(hbb) = \bigcup_{i=1}^{n} HotSet(i)$

The upper value of n is the smallest n such that:

$HotSet(n) \subseteq \bigcup_{i=1}^{n} HotSet(i)$

This ensures that the set includes all the hot blocks; for each new hot set constructed at step n, there exists some blocks in this hot set n that are not contain in previous $n-1$ sets. However, the algorithm for finding $MacroBlock(hbb)$ is expensive to apply. Practically, $MacroBlock(hbb)$ builds gradually based on a recursive algorithm which adds basic blocks that have not been traversed and their edge counters reach ET.

Table III indicates DT3-Macro-block (the macro-block based simulation techniques) to be 23.2% faster than DT3 (basic block based dynamic binary translation) in average. In particular, performance improvement in O3 version benchmarks are much better due to that hot paths in highly-optimized codes are much compacter as a whole.

F. Using Multiprocessor Host for Parallel Compilation

A method to obtain better results and still lower the threshold consists in parallelizing simulation and compilation, using a multi-processor host. In that case, ideally, each block could be compiled in parallell of the DT2 simulation on a separate processor, replaced with the compiled block for further execution, and the result would expectedly be much faster.

However, this method introduces a synchronization overhead between the simulator and the compiler, that slows down the process, and we wanted to evaluate this as well. We have implemented to that effect a multi-threaded dynamic compiling simulator. The standard DT2 simulation runs in one thread and a compile server compiles blocks in one or more separate thread.

We choose to minimize synchronization between the two threads. The candidate blocks for compiling (those who exceed the threshold) are put into a compile queue that is not concurrently accessed. At regular intervals, the duration of which is a runtime parameter, a synchronization occurs, the simulation thread dispatches one block from the queue to the server and fetches the compiled block from previous cycle, if any.

Table IV displays the result of the benchmarks applied compile server technique.

Although performance of compile server is a little less than DT3, we believe this may be improved by larger benchmarks. Furthermore, the heat threshold can be lower in larger benchmarks.

V. Conclusion

In this paper we presented a new dynamic translation modes within the SimSoC simulator based on LLVM. This approach compiles target instructions into LLVM bitcode, followed by several optimization passes invoked, ending with a call to the LLVM Just In Time compiler to generate native code. We have tested six benchmark programs on this simulation mode, and the results demonstrate that the execution time is faster than the non native translation mode.

However, as the translation time increases to the detriment of the overall simulation time, we have implemented a mixed mode to selectively compile only frequently executed code, over a threshold T. This mixed mode yields better results.

To further accelerate simulation, it is necessary to decrease the value of that threshold. For that purpose, we have explored two orthogonal alternatives, one to compile larger units of code, called macro blocks, and one to distribute dynamic translation to multi-processors through a compile server.

Our future work will continue in same direction to further decrease the threshold. We think it is possible although the gains will not be as significant.

Acknowledgments

This work is supported jointly by France National Research Agency and China National Science Foundation SIVES project.

References

[1] INRIA, "Simsoc open source software." [Online]. Available: http://gforge.inria.fr/projects/simsoc

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

[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] L. Cai and D. Gajski, "Transaction level modeling: an overview," in *CODES+ISSS '03: Proceedings of the 1st IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*. New York, NY, USA: ACM Press, 2003, pp. 19–24.

[5] A. Donlin, "Transaction level modeling: flows and use models," in *CODES+ISSS '04: Proceedings of the 2nd IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*. New York, NY, USA: ACM, 2004, pp. 75–80.

[6] F. Ghenassia, Ed., *Transaction-Level Modeling with SystemC. TLM Concepts and Applications for Embedded Systems*. Springer, June 2005, iSBN 0-387-26232-6.

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

[8] S. Sutarwala, P. G. Paulin, and Y. Kumar, "Insulin: An instruction set simulation environment," in *CHDL '93: Proceedings of the 11th IFIP WG10.2 International Conference sponsored by IFIP WG10.2 and in cooperation with IEEE COMPSOC on Computer Hardware Description Languages and their Applications*. Amsterdam, The Netherlands, The Netherlands: North-Holland Publishing Co., 1993, pp. 369–376.

[9] B. Cmelik and D. Keppel, "Shade: A fast instruction-set simulator for execution profiling," in *SIGMETRICS '94: Proceedings of the 1994 ACM SIGMETRICS conference on Measurement and modeling of computer systems*. New York, NY, USA: ACM, May 1994, pp. 128–137.

[10] E. Witchel and M. Rosenblum, "Embra: fast and flexible machine simulation," in *SIGMETRICS '96: Proceedings of the 1996 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*. New York, NY, USA: ACM, 1996, pp. 68–79.

[11] M. Reshadi, P. Mishra, and N. Dutt, "Instruction set compiled simulation: a technique for fast and flexible instruction set simulation," in *Design Automation Conference, 2003. Proceedings*, 2003, pp. 758–763.

[12] 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 *DAC '02: Proceedings of the 39th conference on Design automation*, ser. DAC '02. New York, NY, USA: ACM, 2002, pp. 22–27. [Online]. Available: http://doi.acm.org/10.1145/513918.513927

[13] K. Scott, N. Kumar, S. Velusamy, B. Childers, J. W. Davidson, and M. L. Soffa, "Retargetable and reconfigurable software dynamic translation," in *Proceedings of the International Symposium on Code Generation and Optimization (CGO03)*, 2003.

[14] H. Shi, Y. Wang, H. Guan, and A. Liang, "An intermediate language level optimization framework for dynamic binary translation," *SIGPLAN Not.*, vol. 42, no. 5, pp. 3–9, 2007.

[15] 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.

[16] D. Jones and N. Topham, "High speed cpu simulation using ltu dynamic binary translation," in *Proceedings of the 4th International Conference on High Performance Embedded Architectures and Compilers*, ser. HiPEAC '09. Berlin, Heidelberg: Springer-Verlag, 2009, pp. 50–64. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-92990-1_6

[17] 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.

[18] B. Delsart, V. Joloboff, and E. Paire, "JCOD: A lightweight modular compilation technology for embedded Java," *Lectures Notes in Computer Science*, vol. 2491, pp. 197–212, 2002.

[19] D. Ung and C. Cifuentes, "Optimising hot paths in a dynamic binary translator," *SIGARCH Comput. Archit. News*, vol. 29, pp. 55–65, March 2001. [Online]. Available: http://doi.acm.org/10.1145/373574.373590

[20] 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.

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

[22] F. Blanqui, C. Helmstetter, V. Joloboff, J.-F. Monin, and X. Shi, "Designing a CPU model: from a pseudo-formal document to fast code," in *3rd Workshop on: Rapid Simulation and Performance Evaluation: Methods and Tools (RAPIDO)*, Heraklion, Crete, Greece, jan 2011, best paper award, http://www-rocq.inria.fr/~blanqui/rapido11-pdf.html.

[23] 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.