Automatic Instrumentation Technique of Embedded Software for High Level Hardware/Software Co-Simulation

Aimen Bouchhima, Patrice Gerin and Frédéric Pétrot
System Level Synthesis Group, TIMA Laboratory
CNRS/Grenoble INP/UJF,
Grenoble, France
e-mail : {aimen.bouchhima, patrice.gerin, frederic.petrot}@imag.fr

Abstract— We propose an automatic instrumentation method for embedded software annotation to enable performance modeling in high level hardware/software co-simulation environments. The proposed "cross-annotation" technique consists of extending a retargetable compiler infrastructure to allow the automatic instrumentation of embedded software at the basic block level. Thus, target and annotated native binaries are guaranteed to have isomorphic control flow graphs (CFG). The proposed method takes into account the processor-specific optimizations at the compiler level and proves to be accurate with low simulation overhead.

I. Introduction

Hardware/software (HW/SW) co-simulation environments provide a valuable and cost-effective way for early HW/SW integration, software validation/debug and design space exploration. An heterogeneous multiprocessor co-simulation environment can be represented without loss of generality as shown in Fig.1. The abstraction level of the HW/SW interface determines the nature of the environment being used. The classic HW/SW co-simulation environments use instruction set simulators (ISS) as machine model and correspond to a HW/SW interface at the instruction set architecture (ISA) abstraction level. Recently, HW/SW co-simulation approaches at higher abstraction levels have been proposed. According to these approaches, embedded software no longer needs to be interpreted using an ISS. It is instead executed natively by the host machine, typically within the same (UNIX) process running the hardware simulator. Two main benefits are advocated: (1) considerable speedup (typically 3 orders of magnitude compared to a cycle accurate simulation using ISS) and (2) a straightforward integration in system-level simulation environments making it suitable for early design validation/exploration.

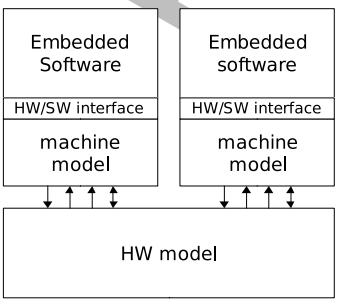

Fig. 1. Hardware Software co-simulation

Native software simulation has been successfully used for functional simulation of complex hardware/software interactions at the system level. The software part may include multithreaded applications that run on top of abstract (RT)OS simulation models. Some approaches even allow executing the real upper layer of the operating system based on a simulation model of the lower hardware-dependent part [3].

However, from a hardware point of view, native software executes atomically in zero time between two successive synchronization points (e.g I/O operations). To enable time modeling, annotations are introduced at the software code level in order to reflect the performance of this code.

The performance of a given piece of software depends on two orthogonal factors: (1) the software itself i.e the sequence of instructions composing the code and (2) the underlying hardware executing these instructions. In this paper, we exclusively focus on the first source of dependency knowing that the second one (hardware dependency) can be taken into account in a complementary way. The software dependency problem can be stated as follows: how to obtain, in native execution, the same information than the one obtained by the sequence of executed target instructions? The annotation process clearly should answer this question. Two criteria are of primary importance at this level: automation and accuracy. To the best of our knowledge, there have been no systematic methods for software annotation in existing native simulation approaches that guarantees full automation and appropriate accuracy. The main contribution of this paper is a compiler based annotation technique, namely the cross-annotation technique, that can be used in native simulation based approaches and that satisfies the two above criteria.

The rest of the paper is organized as follows. Section 2 reviews some existing works in the field of native simulation and software annotation. Section 3 introduces the annotation problem. Section 4 details the basic idea and methodology behind the proposed technique. Section 5 discusses the implementation of the cross-annotation principle within the LLVM compiler infrastructure. Section 6 exposes some experimental results obtained using the proposed method, and section 7 concludes the paper.

II. Related work

Several high level HW/SW co-simulation approaches rely on software annotation to enable performance modeling. Many of them require explicit and manual source annotation by the programmer using dedicated primitives [13, 9, 8]. This is acceptable for small test cases but quickly becomes very cumbersome and error prone for real-life applications. In [7], the authors adopt a different approach by requiring the designer to specify the overall WCET of a software task making their approach suitable for the analysis of system real-time properties but not for performance modeling.

Other high level co-simulation approaches provide partial solutions to automate the annotation process. In [12], the authors propose to automate the instrumentation by analyzing the target assembly code and for each assembly instruction, annotate the corresponding source code line. However, they reported the difficulty of assembly to source code mapping when enabling optimizations. Another approach [1] relies on a precharacterization and instrumentation of embedded software in a target independent way. A retargeting phase is then performed once the target architecture is fixed and the result is used to perform fast simulation and design space exploration. The authors however did not comment on how to handle processor specific optimizations. A completely different approach [10] exploits the ability to overload basic language (C++) operators to automatically inject performance counters during run-time native execution, as [4] also did. The authors reported a quite reasonable accuracy but it's not clear whether the obtained results take into account compiler target specific optimizations.

[5] proposes a method that exploits the intermediate representation (IR) resulting from the compiler front-end to apply retargetable performance analysis taking into account architecture related issues. From the modified IR, time annotated C/C++ code is regenerated and used for high level HW/SW co-simulation. However, given that the analysis task is performed on the result of the compiler front-end, it seems not to take into account the different target-specific optimizations that occur during the compiler back-end stage.

One major contribution of this paper is to clearly separate the annotation process (where and how to put annotations in the code) from the performance estimation problem (value of the delay associated with the annotations). The first problem is purely software dependent (including the effect of the target processor instruction set architecture) while the second depends on the hardware architecture (pipeline implementation, caches, bus contentions etc.). We propose a solution to the first annotation process problem that is fully automatic and accurate from this perspective.

III. Annotation basics and challenges

Given a piece of software source code (Fig.2.a), the purpose of annotation is to instrument the native version of the code to reflect the execution of the same code on the target processor. Host and target processors are different in general.

The first difficulty inherent to annotation is the data dependent behavior of the software program itself. Unlike static approaches where performance estimation is needed before program execution, we are placed in a run-time context where performance is evaluated while the program runs. One solution to

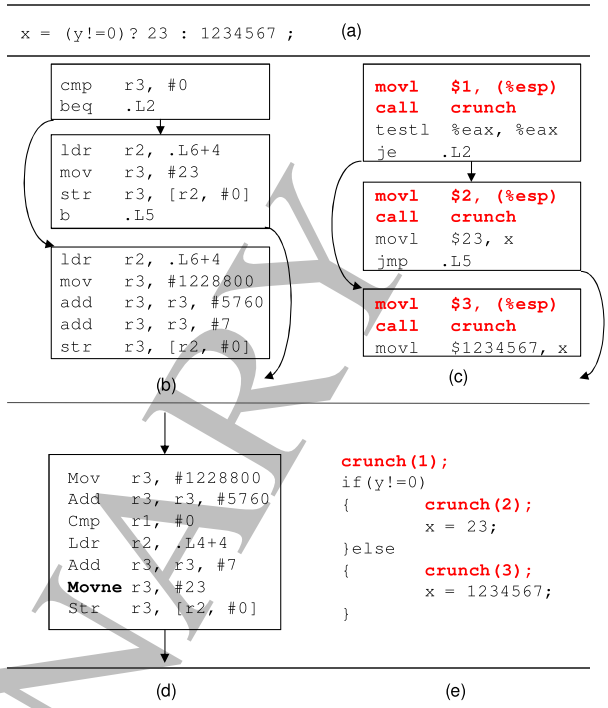

Fig. 2. Annotation techniques (a) original source code (b) non-optimized target (ARM) code (c) annotated native (x86) code (d) optimized target code (e) source level annotation.

the data-dependency problem is then to "follow" the execution control flow of the target program. Practically, this means that annotation should be performed at the basic block level of the cross-compiled program[1].

Since the simulated program is the one compiled for the host processor, the problem is then to find an instrumentation strategy working on the host compiled program while reflecting the control flow graph (CFG) of the target (cross-compiled) program.

The object-level annotation directly instruments the native object code (here x86) by inserting calls to the `crunch` function (Fig.2.c). The argument passed to `crunch` represents the identifier of the corresponding basic block in the target object code (Fig.2.b). In this example, basic blocks are numbered 1, 2 and 3. This method assumes a one-to-one mapping between the native and target CFG, which is generally not the case even when using the same compiler toolchain due to processor-specific optimizations. For example, the corresponding ARM optimized code has a completely different CFG exploiting the ARM conditional instruction (Fig.2.d).

The source level annotation strategy (Fig.2.e) tries to get rid of host processor dependency by instrumenting the original source code. The source code is first instrumented using information coming from target object code analysis, before being compiled to the host processor. This method is however difficult to implement in practice. In fact, finding the basic block boundaries in the source code and inserting the instrumentation call used to be a very difficult task due to the complex and rich syntax of the source code itself combined with the effect of compiler optimizations (both processor dependent and inde-

[1] A basic block is a set of consecutive instructions having exactly one entry and one exit point.

pendent ones). The same difficulty is for example behind the inaccuracy of source level debugging when dealing with compiler optimization enabled programs.

IV. Compiler-based cross annotation

A. Solution overview

The proposed solution to the annotation problem draws on the advantages of both source and object level annotation methods. The main idea is to use the compiler intermediate representation (IR) format as target to the instrumentation process. Many advantages result from this choice. (1) Like the source level approach, this allows to be host independent, since the instrumentation process is kept prior to the host processor specific back-end. (2) The annotation is independent from the used high level language (C, C++, etc). (3) The IR already contains control flow graph related information. (4) The IR generally has equivalent in-memory data structures and associated programming interfaces for easy implementation of processing algorithms.

To take into account target-specific transformations of the CFG within the back-end, we propose to extend the IR scope throughout the back-end (Fig.3.b). We define the concept of cross intermediate representation (cross-IR) to keep track of any processor specific CFG transformation while still being processor independent.

The processor specific CFG transformations can be explained by the semantic variation between the instruction set architecture (ISA) of the target processor and that of the virtual machine representing the IR (Fig.4).

B. Cross-IR construction

Given the initial IR input to the back-end stage, cross-IR is first constructed as a copy of this initial IR. In a conventional compiler back-end, the initial IR is generally transformed into more target-specific representation format(s) that undergoes a series of target-specific transformations before the ultimate binary object emission step. At this level, the control flow graph of the binary object (or equivalently the latest target-specific representation format) and that of the IR are likely to be different.

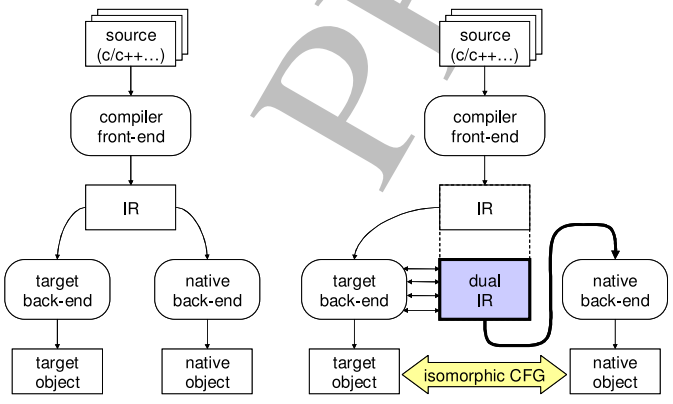

Fig. 3. Intermediate representation (IR) based annotation (a) original compiler (b) proposed extended compiler

Cross-IR is used to keep track of the transformations that affect the CFG within the different passes of the back-end. When a pass performs a transformation on the CFG of the target-specific representation format, this transformation is detected and the cross-IR CFG is updated to reflect the corresponding change in the CFG of the target-specific representation format. Of course, the cross-IR transformation should preserve the target-independent characteristic of the cross-IR. At the end of the back-end stage, a one-to-one mapping is obtained between the control flow graphs of the cross-IR and the latest target-specific representation format. This mapping is used to annotate each basic block of the cross-IR using information from the "dual" basic block in the target-specific representation.

Fig.5 shows two typical cases of CFG transformations during the back-end stage, corresponding to the two paths identified in Fig.4. The first case (Fig.5.a) exemplifies the $A \rightarrow B$ path. Here a complex IR instruction (e.g set on condition) is translated to a set of target instructions (dark area in the target CFG) inferring additional basic block and leading to the diamond-like structure. Such translation is used in case the target processor doesn't support the complex instruction and typically occurs at the starting of the back-end during the selection phase. In parallel, the cross-IR CFG is modified to reproduce the same diamond-like control structure by using instructions from the IR domain that are semantically equivalent to the used target instructions (point B is still covered by the IR instruction set in Figure 4).

The second case (Fig.5.b) corresponds to target dependent optimizations performed on the target CFG itself. Two situations are distinguished:

- The optimizing transformation does not introduce new instructions from the target ISA domain with no equivalent in the IR domain. This situation is handled in the same way as the previous case from a cross-IR point of view.

- The optimizing transformation takes benefit of a target-specific instruction having no direct equivalent in the IR ISA domain and leading to target CFG modification as shown in Fig.5.b. The example shown in the figure exploits the sum-of-absolute-differences (SAD) instruction available on many DSP enabled processors. Assuming this instruction has no equivalent in the IR domain, the solution is to replace it by a call to an "emulating" function in the cross-IR CFG. In this way, the cross-IR CFG

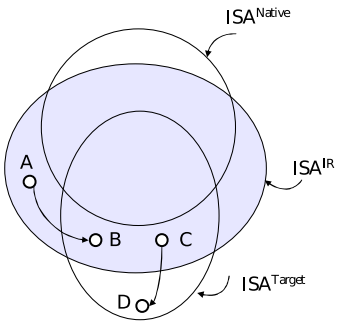

Fig. 4. ISA semantic overlap

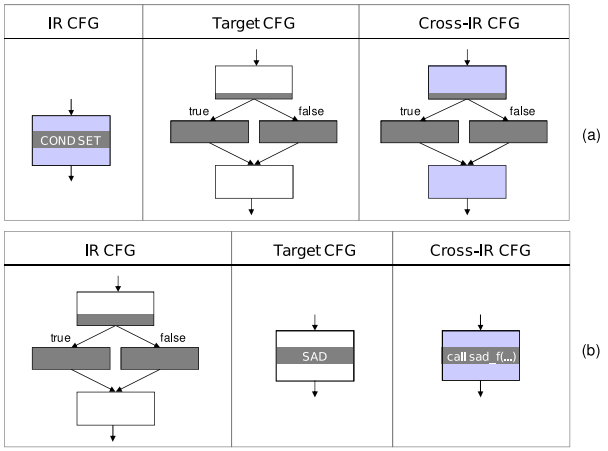

Fig. 5. CFG transformations at the back-end level

can be reduced to one basic block just like the optimized target CFG. Of course, the implementation of the function emulating the specialized target instruction should not undergo the annotation process (it will simply be linked to the annotated binary at the link stage).

V. Implementation in LLVM

A. LLVM overview

The Low Level Virtual Machine (LLVM) is an open source compiler infrastructure [6], i.e a collection of modular and reusable components for building compilers. Fig.6 shows the architecture of LLVM. The backbone of the architecture is the LLVM intermediate representation, still called LLVM that is designed to allow the implementation of low level transformations and optimizations at the middle-end level. LLVM follows the single static assignment (SSA) approach and has three equivalent formats: in-memory format, disk format for storage (bytecode) and textual human readable format.

LLVM makes use of the GNU C Compiler as front-end throughout the `llvm-gcc` tool which is actually a port of `gcc` to the LLVM ISA. It also provides the infrastructure for building processor specific back-ends through the Machine-LLVM low level in-memory representation. All processing on the LLVM and Machine-LLVM intermediate representations is organized as a set of passes managed by a pass manager. In figure 7, passes are denoted by rounded rectangles and their relative vertical position indicates the order of their execution.

B. LLVM back-end extension

Fig.7 shows an overall view of the extensions introduced at the LLVM back-end level to support the proposed cross-annotation method. The added parts are those colored in dark gray and modified parts are those colored in light gray in the figure.

B.1 Cross-LLVM

The cross-IR concept is mapped to the cross-LLVM intermediate format as follows:

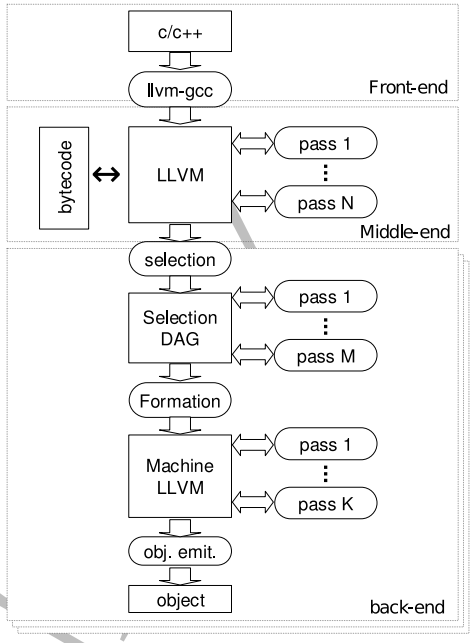

Fig. 6. LLVM architecture organization

- Initially, cross-LLVM is just a copy of the middle-end produced LLVM.
- During the code generation process (and especially the selection step), when a situation corresponding to the $A \rightarrow B$ path of Fig.4 is encountered, cross-LLVM is updated according to Fig.5.a. Example of such transformation is the translation of the LLVM select instruction for processors having no conditional execution instructions (such as ARM Thumb). The code generator translates this by inserting a diamond-like control structure. In cross-LLVM the SELECT_CC instructions is replaced by the equivalent control structure (Fig.8).
- During the processor specific optimization passes, cross-LLVM is updated according to Fig.5.b. In the particular case of the ARM back-end with the subset of supported instructions, we did not encounter such situation.

B.2 Analysis and annotation pass

Analysis and annotation is implemented as a pass that takes place at the end of the back-end, just before the ultimate object code emission pass. At this level, there is a one-to-one relation between cross-LLVM and the latest Machine-LLVM representation (that maps directly to the final target binary). The analysis and annotation pass simply performs a static analysis of each basic block in the Machine-LLVM and inserts an instrumentation stub at the beginning of the corresponding basic block of the cross-LLVM. Finally, the in-memory cross-LLVM is emitted as bytecode file to be processed by the host back-end before generating the instrumented native executable.

We implemented two kinds of instrumentation stubs. The first simply insert a call to a special function (named `crunch` in Fig.2) passing the identifier (index) of the basic block. The `crunch` function is then responsible of updating the BB trace

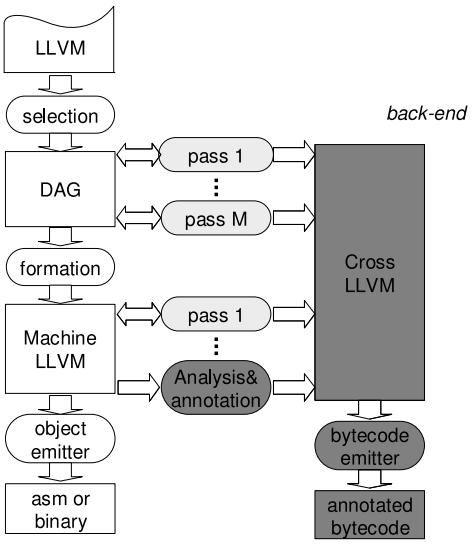

Fig. 7. Cross-annotation flow implementation in LLVM

```
x = (var > 0) ? 12 : 5;
```

```
%tmp = load i32* @var
%tmp1 = icmp sgt i32 %tmp, 0
%iftmp.0.0 = select i1 %tmp1, i32 12, i32 5
store i32 %iftmp.0.0, i32* @x
```

```
cmp r2, #0
bgt .LBB1_2
.LBB1_1:
cpy r1, r0
.LBB1_2:
ldr r2, .LCPI1_1
```

```
...
%tmp1 = icmp sgt i32 %tmp, 0
br i1 %tmp1, %c_true, %c_false
c_true:
store i32 12, i32* %iftmp.0
br label %c_next
c_false:
store i32 5, i32* %iftmp.0
...
```

Fig. 8. Translation of the LLVM select instruction: (a) source code (b) original LLVM (c) Thumb generated code (d) cross-LLVM corresponding code.

(Fig.9). Notice that we call the performance estimation function (`do_perf_estimation`) only when the trace is full. The actual implementation of this function is out of the scope of this paper. For our purpose, it's just kept empty.

The second implementation variant is to directly inline the content of the `crunch` function at the beginning of each basic block to avoid the runtime cost of the call instruction. This will be analyzed with more details in the experimentation section.

```
void crunch(unsigned int index) {
   if (bb_count >= trace_size){
       bb_count = 0; do_perf_estimation();
   }
   trace[bb_count] = index;
}
```

Fig. 9. `crunch` function behavior

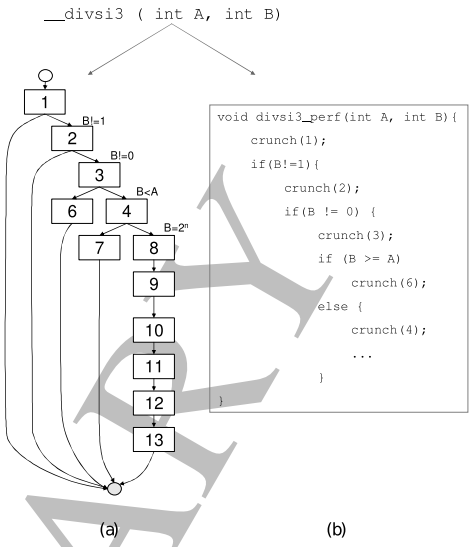

Fig. 10. characterisation of the `__divsi3` for ARM

B.3 "Black box" code characterization

Embedded software often includes parts that are available only under the object format. This is particularly true for hand optimized performance critical algorithms. The performance of such parts cannot be evaluated by the proposed cross-annotation technique. We rely on off-line manual performance characterization of such "black box" software. This characterization has to be made only once.

Examples of such black box software include some calls to the C standard library which are implicitly introduced by the compiler back-end. For instance, if the target processor doesn't support integer division, the compiler back-end replaces the unsupported instruction by a call to the software routine (`__divsi3`) emulating the integer division and implemented in assembly language. In this case, the extended LLVM back-end automatically inserts another call to the characterizing function (called `divsi3_perf` in Fig.10). Another possible approach would be to take benefit of recent advances in the decompilation domain to automatically recover the intermediate representation from the target assembly [11].

VI. Experimentation

To validate our approach, we applied the cross-annotation flow on a set of test applications taken mostly from the MiBench embedded testbench collection [2]. Fig.11 depicts the environment used for the experiments. We considered ARM as target processor. The bolded parts in the figure highlight the features enabled by the cross-annotation approach.

To assess for the accuracy of the proposed method, we compared the basic block traces generated by the ISS based simulation and the annotated native execution. To enable BB trace generation at the ISS level, the ISS is modified to accept a watchpoint configuration file containing addresses of the first instruction of each BB in the target binary along with the associated BB index. This file is generated by the analysis and annotation pass within the extended LLVM. For all applications in our testbench, we obtained a perfect match between

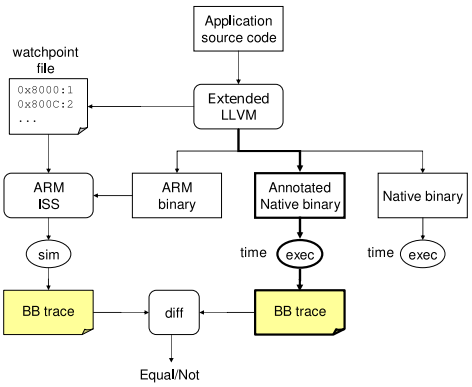

Fig. 11. experimentaion environment setup

TABLE I
PERFORMANCE RESULTS OF THE PROPOSED APPROACH

Bench	BB	Slowdown (compared to bare native execution)		Speedup (compared to ISS)		Program memory overhead	
name	size	call	inline	call	inline	call	inline
aac	9.3	2.6	1.2	1408	3087	21%	64%
qsort	4.3	7.4	2.6	526	1312	46%	139%
bitcount	5.8	6.0	2.3	609	1634	34%	103%
strsearch	6.1	5.2	2.0	765	1898	33%	101%
rijndael	107.5	1.1	1.0	3486	3567	1.8%	5%
dijkstra	5.8	5.6	2.1	743	1792	34%	103%
crc32	5.3	6.3	2.4	654	1564	37%	113%
gsm	5.46	6.0	2.2	667	1654	36%	109%
adpcm	7.9	3.4	1.3	1209	2752	25%	76%

the two BB traces proving the 100% accuracy of the proposed method. Of course this accuracy is only with respect to the annotation problem, i.e where to put instrumentation in the code to reproduce the same target BB trace. Using this trace, dynamic performance estimation can be performed with more or less accuracy given details on the target hardware architecture (outside the scope of this paper).

To evaluate the performance of the cross-annotation approach, we computed the slowdown as a ratio of the annotated binary execution speed by the non-annotated version. This slowdown is computed both for inlined and non-inlined instrumentation techniques. The memory overhead introduced by the instrumentation is also evaluated in both cases.

From table 1, we see that the simulation slowdown depends on the average size of all executed basic blocks. When this size increases, the relative overhead introduced by the instrumentation stub in each basic block decreases (almost null for rijndael). In all cases, this overhead is smaller using the inlined instrumentation stub that avoids the function call cost. This comes with an increase in the program memory size, which is not critical from a host point of view. For the average case of BB size ($4 \sim 6$), the annotated native binary executes about two times slower than the non-annotated version (for the inlined case) which is very interesting from a performance enable high level HW/SW co-simulation point of view. This slowdown factor has to be compared with that of an ISS. In our case, the execution of the annotated native binary is almost 3 orders of magnitude (x1000) faster than ISS.

VII. CONCLUSION

We presented a compiler-based approach for automatic and accurate annotation of embedded software that targets performance modeling in high level hardware/software cosimulation approaches. While not restricted to a particular compiler, the proposed method has been implemented within the LLVM compiler infrastructure that offers many attractive features simplifying such implementation. Experiments on various test applications showed the effectiveness of the proposed method both in terms of accuracy and simulation overhead.

REFERENCES

[1] L. Cai, A. Gerstlauer, and D. Gajski. Retargetable profiling for rapid, early system-level design space exploration. In *Proceedings of the Design automation Conference*, 2004.

[2] D. Ernst and al. Mibench: A free, commercially representative embedded benchmark suite. In *IEEE International Workshop on the Workload Characterization*, 2001.

[3] P. Gerin, Hao Shen, A. Chureau, A. Bouchhima, and A. Jerraya. Flexible and executable hardware/software interface modeling for multiprocessor soc design using systemc. In *Proceedings of the Asia and South Pacific Design Automation Conference*, 2007.

[4] P. Giusto, G. Martin, and E. Harcourt. Reliable estimation of execution time of embedded software. In *Proceedings of the conference on Design, automation and test in Europe*, 2001.

[5] Y. Hwang, S. Abdi, and D. Gajski. Cycle-approximate retargetable performance estimation at the transaction level. In *Proceedings of the conference on Design, automation and test in Europe*, 2008.

[6] C. Lattner and V. Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the International Symposium on Code Generation and Optimization*, 2004.

[7] J. Madsen, K. Virk, and M. J. Gonzalez. Abstract RTOS modelling for multiprocessor system-on-chip. In *International Symposium on System-on-Chip*. IEEE, 2003.

[8] R. Le Moigne, O. Pasquier, and J-P. Calvez. A generic rtos model for real-time systems simulation with systemc. In *Proceedings of the conference on Design, automation and test in Europe*, 2004.

[9] J. J. Pieper, A. Mellan, J. M. Paul, D. E. Thomas, and F. Karim. High level cache simulation for heterogeneous multiprocessors. In *Proceedings of the Design Automation Conference*, 2004.

[10] H. Posadas, F. Herrera, P. Sánchez, E. Villar, and F. Blasco. System-level performance analysis in systemc. In *Proceedings of the conference on Design, automation and test in Europe*, 2004.

[11] T. Reps, G. Balakrishnan, and J. Lim. Intermediate-representation recovery from low-level code. In *Proceedings of the ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation*, 2006.

[12] S. Yoo, G. Nicolescu, L. Gauthier, and A. Jerraya. Automatic generation of fast timed simulation models for operating systems in soc design. In *Proceedings of the conference on Design, automation and test in Europe*, 2002.

[13] H Yu, A. Gerstlauer, and D. Gajski. Rtos scheduling in transaction level models. In *Proceedings of the international conference on Hardware/software codesign and system synthesis*, 2003.