Accelerating Host-Compiled Simulation by Modifying IR code: Industrial application in the spatial domain

Abstract - Space applications rely on long and complex design processes, as they must deal with strict non-functional requirements such as criticality, timeliness, reliability and safety. The huge number of analysis and evaluations performed requires powerful simulations technologies combining high simulation speed and accuracy. Host-compiled simulation is a powerful approach to achieve fast, timed simulation of software running in complex embedded systems. However, in the general term, there is still the need of improving the speed and accuracy of these solutions, and there is a lack of host-compiled approaches oriented to space applications. To solve the first point, this paper presents an alternative that modifies the standard solution of adding the modeling of the cross-compiled control flow in the host computer by modifying the compiler's intermediate representation. That way, the host binary naturally follows the cross-compiled binary flow, avoiding a separate modeling, and improving simulation speed while maintaining accuracy. Additionally, the paper focuses on LEON processor, commonly used by the European Space Agency (ESA).

I Introduction

The space domain represents an important area in the world of electronic system design. On the one hand, system correctness in this domain must be completely granted, as space systems must be designed to operate in extremely difficult conditions and far away from any human being. As a result, the development of these systems typically requires long and complex design processes, in order to achieve the required figures, involving large evaluation, exploration and verification processes.

On the other hand, the conditions described above make especially important to obtain the maximum benefits from the scarce resources available while providing certain flexibility, as these capabilities will determine the possibility to overcome the problems found during a mission. However, the combination of these qualities is difficult to obtain.

Advances in hardware capabilities, such as multiprocessor or reconfigurable hardware, are being slowly adopted in the space domain, since their clear benefits clashes with the restrictions of certification processes. Additionally, there is a lack of methodologies and tools to support the exploitation of these new technologies in the scope of systems considering the peculiarities of space applications. As a result, it is important to develop tools capable of ensuring that the design process is in the right direction from the very beginning, since going back in the designs is typically very costly. The challenge is then to exploit these capabilities, considering the difficulty to cover all the cases resulting from these dynamic behaviors.

To solve this challenge, it is required to consider two ideas. First, it is mandatory to dispose of tools capable of modelling, analysing and exploring a huge set of conditions in reduced times and with high accuracy.

Secondly, to get all the benefits from these hardware architectures, the software is usually dependent on the underlying platform both in functionality and performance. Thus, HW details must be considered when the software elements are developed, allocated or dynamically allocated to the computing resources. Application models require the adequate granularity to extract the medium- and fine-grain details, such as parallelism or cache performance, that will reduce the response times of key application functionalities

Traditionally, embedded software development and verification has been performed by running the application SW on a physical prototype of the hardware platform. However, this solution is only available late in the design process and it is typically hard to use, especially as the complexity of embedded systems grows. Additionally, it can be non-adequate to analyze the design internals or to evaluate a huge number of conditions, as the evaluations are limited by the number of real platforms available.

Virtual platforms offer a powerful alternative to hardware prototypes. On them, the modeling of the processor is a key factor, since it is the element in charge of most of the system functionality. Several alternatives have been proposed to simulate the processors' operation, providing different tradeoffs in terms of accuracy vs. performance and usability.

Among them, host-compiled timing simulation has received considerable attention in the last years, as this kind of simulation maximizes the ratio between simulation speed and modeling accuracy. Thus, its improvement and adaptation to the space domain can be relevant to provide the modeling, analysis and exploration support required to adapt the next generation space domain applications to the new execution platform capabilities, while maintaining the hard constraints imposed by this application domain.

In order to improve host-compiled simulation techniques, this paper proposes an alternative to improve simulation speed while maintaining the modeling accuracy. Host-compiled techniques typically rely on annotating the impact of the target platform within the source code to analyze their effects on the host simulation. However, the code does not always behave the same in the host and in the target

processor, requiring an important overhead to solve these differences. This paper proposes a technique that, modifying the intermediate representation code, can obtain the same accuracy without this overhead.

Additionally, during this work, a host-compiled simulation infrastructure has been adapted to the space domain by modeling a LEON3-based platform, as LEON3 is a processor typically used in this domain.

II State of the art

Host-compiled simulation has become an important approach for system modeling since it achieves a significant speedup without excessively compromising accuracy, compared to cycle-accurate simulations. Early works on SLS, such as [2,3], considered a unique mapping between source code and binary code, skipping the effects of compiler optimizations. However, to accurately model real embedded systems, optimizations had to be integrated.

To do so, several techniques were proposed, depending on the type of code that was to be analyzed and annotated. These techniques can be categorized [4] into binary (assembly) level simulation (BLS), IR-level simulation (IRLS) and source-level simulation (SLS), in terms of functional representation levels.

BLS typically relies on the generation of an alternative source code with the same functionality but following the binary code structure. Results obtained are quite accurate, but they have critical problems when compared with SLS or IRLS. First, there are corner cases that are difficult to model, such as indirect branches. Secondly, a virtual platform should also support software development and debugging. However, the generation of an alternative source code, completely different from the original one, disables the possibility of using the tool for debugging purposes.

Similar alternatives have been proposed using IR level. To address the mapping problems found in early SLS work, in [4] the compiler is modified to add timing information into the Intermediate Representation (IR). However, modifying the compiler takes a lot of effort. Furthermore, not all compilers can be modified.

As an alternative, Wang et al. [5], propose the generation of an annotated intermediate source code, created to model the IR code. A similar approach in also proposed in [6]. Nevertheless, these approaches have similar problems to BLS, since C code reconstructed from IR-level code is hardly readable, and also rules out source-level debugging.

Finally, the consideration of compiler optimizations has also been addressed at source-level. Most source-level works typically rely on complex analysis to relate basic blocks of the cross-compiled assembler code with the original source code. The work in [7] tries to overcome the mapping problems dividing the source code and the binary code into segments called loop levels. [8] proposes an alternative that, through a complex analysis of the binary control graph, integrates conditional annotations in the source code.

These methods have been used in other works to model HW/SW platforms [10] and networked systems for IoT[11].

In a similar way, the work in [1] proposes an approach that, taking information both from the binary flow graph and the debugging information, generates conditional timing annotations. In this case, the use of the debugging information simplifies the analysis of the binary graph, making the solution more portable. As a result, these methods have been used in other works to model GPUs [9], and also in works of other authors, such as [12]. The problem of this approach is that the accuracy and generality is obtained at the expense of a certain overhead, due to the conditional annotations.

To minimize annotation overheads, [13] proposes a solution that simplifies the execution flow graph, reducing the number of conditional annotations, and slightly increasing simulation speed. However, the problem of complex conditional annotations is not solved.

To overcome these limitations, this work presents an alternative that follows some of these ideas, but applying them at the IR level. Then, the host-compiled executable code is generated from the modified IR without requiring an additional source code generation, as is usual in IR techniques. Therefore, the code obtained is not only faster without decreasing modeling accuracy, but it can also be used for debugging purposes.

III Intermediate Level Annotation

In host-compiled simulation, functional and timing modeling of the processor is enabled by automatically annotating performance information within the original program, which is run on the host processor.

To do so, the cross-compiled binary code is analyzed, identifying its basic blocks and extracting the parameters required to model its execution in the target processor, such as the number of cycles and instructions. However, the process of finding the relationships between basic blocks of the cross-compiled binary code and the blocks of the original code requires sophisticated matching methods, due to the modifications done by compiler optimizations.

To solve it, instead of directly annotating static time values in the source code, it has been proposed to model the binary-level control flow for the target architecture together with the software functionality [1] (Fig 1).

However, the problem of this approach is that simulating the target control flow and the functionality hand-in-hand, means executing the code twice. Additionally, the modeling of the target control flow is quite artificial, requiring complex codes that increase the number of instructions to be executed during simulation. Therefore, the use of these matching algorithms leads to some simulation overhead.

As can be seen in the figure, for a single line in the source code (c[i]=a[i]*b[i];), it is necessary to call a “bb” function (Fig. 1 b), which involves multiple comparisons, and additional function calls such as the one required to evaluate

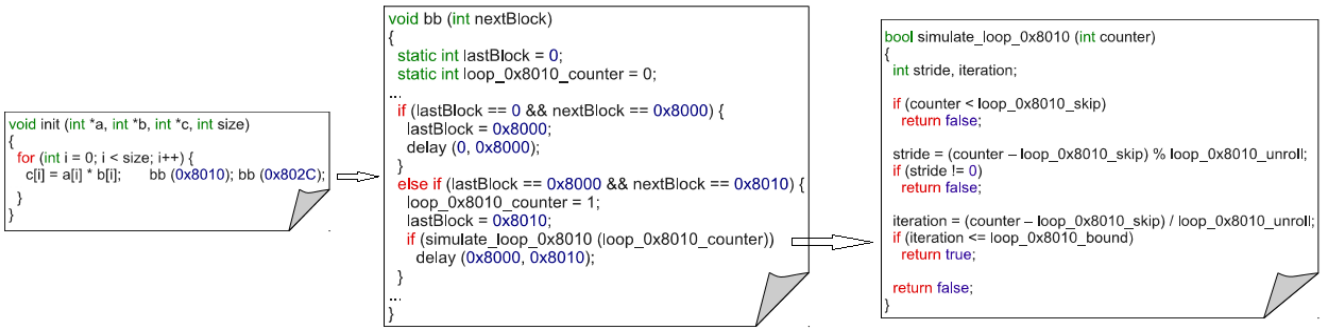

Figure 1: Annotation code, extracted from [1]

the different steps of a loop (Fig. 1 c). Thus, resulting overhead can increase simulation time up to 40 times, as can be seen in [1] results.

The proposal done in this work is that, to speed up simulation, it is necessary to find a solution capable of modeling this binary flow in a fully integrated and natural way in the host simulation. For example, we can consider a "switch" block in a C program. In the source code, there is just a basic block for each possible value of the control variable (each "case" section). Each basic block is one of the multiple jump alternatives from the "switch" instruction.

However, typical binary instruction sets do not have instructions supporting multiple jumps. Thus, in the cross-compiled binary, the "switch" can be replaced by a list of conditional braches, which means a full set of new basic blocks. If we want to model that flow in the source code, it is necessary to have a list of "if" instructions together with the original "switch" block, increasing simulation overhead.

Additionally, when compiling the source code for the host execution, the "switch" will again be transformed, due to the same reason. Thus, in the host program there will be two sets of branches: one from the native implementation of the "switch", and one from the list of blocks added to model the cross-compiled binary code. The point is that both sets have the same functionality, since they come from the same original source code. In other words, we are executing the same code twice, with the corresponding unneeded overhead.

To reduce this overhead, the paper proposes a novel alternative based on the modification of the compiler's intermediate representation (IR). The idea is to use not only the same source code for target analysis and host simulation, but also to modify the IR code used during host compilation to follow the structure of the cross-compiled binaries. Thus, basic blocks in the IR code and in the cross-compiled binary easily match, and annotations can be made without adding a duplicated execution flow, speeding up the simulation.

A. *Benefits from reusing front-end modifications*

When reusing the IR code, it is important to note that the same compiler infrastructure can generate different binary codes due to differences in the back-end step, but also in the front-end compilation step. Nevertheless, typically IR code generated by the front-end compiler for one machine can be still valid for other machines, while results obtained from the back-end are not re-targetable. Thus, the first novel idea presented by this paper is that it is not necessary to go back to source code to make the annotation, as done in previous techniques.

Moreover, it is not only possible to reuse the IR code generated for the cross-compiler to generate the simulation executable but also a simulation improvement.

Additionally, the majority of the control flow modifications made during the compilation process, are a result of front-end optimizations. Thus, if we reuse IR code, most of the differences found in [1] are no longer a problem.

B. *Annotation of back-end modifications*

The main issue resulting from the proposed approach is to handle modifications done in the back-end step. In order to modify the IR code obtained during the native compilation step to be as similar as possible to the cross-compiled binary flow, this work has focused on the operation of the LLVM compiler, since it is an open-source compiler which is gaining increasing interest [14], and in which it is easy to analyze and modify the IR code.

Typically, differences between the IR and the binary code can be catalogued in two different groups. First, there are differences stemming from instructions that cannot be directly transformed into assembler code. They result in cases such as in the "switch" commented above, where modeling the binary flow leads to unnecessary code duplicity.

A second group of differences appears because the back-end compilation step may modify some of the decisions taken by the front-end to generate the IR code. For example, the back-end compiler may change the order of "then" and "else" clauses in an "if" structure found in the IR. It can also modify the operations done in comparisons, for example to force a comparison with '0', if the compiler considers that the resulting improvement in performance merits the change.

It is important to note that only the changes that involve the creation or destruction of basic blocks are relevant for the IR modification during the annotation process. Other

changes, such as a reordering of the basic blocks, are no critical for that process.

To handle both cases, the proposed process for the IR modification has the following steps:

- Generate flow graphs with the basic blocks, both in the IR and in the binary codes and find the differences, matching basic-block marks (Figure 2).
- If there are different basic blocks, cross-check the IR code to know the IR instruction that caused the change, searching for one of the elements described below.
- If so, modify the IR to match the binary following a template that depends on the responsible instruction.
- Adapt the order of the basic blocks of the template to the blocks in the binary.
- If no solution is found to adapt the template to the binary flow or it is not caused by an identified element, the solution applied in [1] is applied.

The last step has been added for completeness, but in our experiments, the previous steps have solved all the differences without requiring this last one.

Thus, the most critical point is to identify the elements in the intermediate code that provoke the modifications and generate the templates used to modify the IR code.

Considering LLVM IR code, the main language constructs provoking differences between IR and binary code are switch and return clauses, complex logic conditions and select and data extension instructions.

Switch clauses do not have a direct mapping into assembler instructions, since assembler branches do not permit multiple jumps. As a result, the compiler can replace the switch by a list of conditional branches or a branch table.

Return instructions in the IR code is not followed by any basic block. As a result, the compiler tries to optimize the execution graph of the function and the "return" basic blocks disappears. To model this, the IR code is modified by analyzing the binary code graph and moving codes and removing jumps until both fit.

When a condition is the result of the ***composition of multiple simple conditions*** (e.g. "if(a>0 && b<1 && c==0) {…}"), binary and IR codes differ. In LLVM, the IR code considers the logical conditions as any other mathematical operation. Thus, it obtains the result of the composition first, and then applies it to a single "if" statement. However, for a standard processor, this procedure is inefficient. Instead, the back-end compiler changes the single "if" of a compositional condition for several "if" clauses with single conditions (e.g. "if (a==0) { if (b<1) { if (c>2) {…} } }").

As a consequence, the approach proposed modifies the IR code taking the simple conditions that form the composite conditions and creating as many "if" clauses as required, following the order found in the cross-compiled binary code.

So, for example, in Fig 2, it can be shown how the branch at the end of the "entry" basic block depends on a composite condition, and how a new "if" struct is added to use simple conditions instead of composite conditions, as the compiler back-end does when generating the binary code. Furthermore, "uc_add_time" functions are added to annotate the number of cycles and instructions in the basic blocks.

```
define i32 @func(i32 %a, i32 %b) #0 {
entry:
  %cmp = icmp sgt i32 %a, 2
  %cmp1 = icmp slt i32 %b, 1
  %or.cond = and i1 %cmp, %cmp1
  %0 = call i32 @uc_add_time(i32 4, i32 4)
  br i1 %cmp, label %my_entry.newIfthen, label %if.else

my_entry.newIfthen:               ; preds = %entry
  %2 = call i32 @uc_add_time(i32 3, i32 3)
  br i1 %cmp1, label %if.then, label %if.else

if.then:                          ; preds = %my_entry.newIfthen
  %add = add i32 %a, 1
  %add2 = add i32 %add, %b
  %4 = call i32 @uc_add_time(i32 5, i32 3)
  br label %if.end

if.else:                          ; preds = %my_entry.newIfthen, %entry
  %sub = sub nsw i32 %a, %b
  %5 = call i32 @uc_add_time(i32 5, i32 3)
  br label %if.end

if.end:                           ; preds = %if.else, %if.then
  %var.0 = phi i32 [ %add2, %if.then ], [ %sub, %if.else ]
  ret i32 %var.0
}
```

Figure 2: Annotated IR code

Select instructions are included by the LLVM compiler for short conditional blocks with a single instruction, replacing the "if" clause. If there is a divergence between the basic blocks of the IR and the binary due to a "select" clause, the IR is modified by replacing the "select" with an explicit "if", where timing annotations can be added.

Zero extension instructions are used to convert data types. It is typically used when saving values from boolean operations. In that cases require conditional branches are used for their implementation. If so, the instruction is modified adding an explicit "if" clause in the IR code, as in the "select" instructions case.

C. Overall proposed procedure

To implement the annotation process described before, a complete compilation flow is proposed (figure 3). In this flow, the first step is to compile the original code for the target processor, extracting the corresponding IR code. Then, the IR code is cross-compiled into assembler code and analyzed to obtain the basic-block information to be annotated. After that, the IR code is modified following the steps described in the previous section, in order to generate an annotated IR code with minimal overhead. Finally, the code is compiled for the host computer and simulated.

This approach has several advantages with respect to the approaches found in the bibliography so far. First, the IR code is not transformed into a new C code, but directly reused. Thus, the debugging information found in the final host executable corresponds to the original source code, and thus, debugging is made possible.

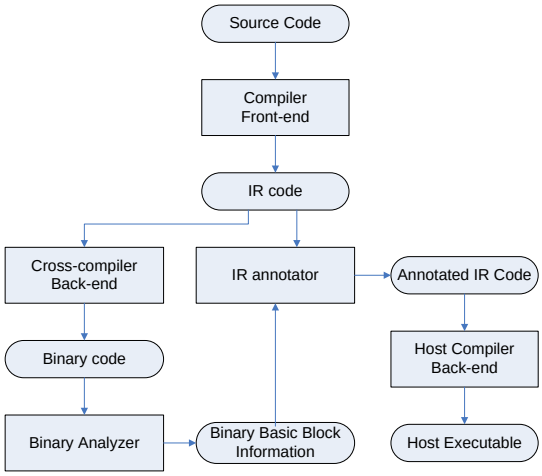

Figure 3: Annotation flow

Secondly, the modifications proposed are quite generic, so they can be valid for a large set of processors. In the same way, modifications of the IR code are not done within the compiler, but by an external tool. Since this tool is processor-independent, the approach can be widely used.

Finally, timing annotation has been slightly modified with respect to the proposal presented in [1]. Instead of adding a "wait" statement to model the timing delay in every basic block, the proposed annotation function only accumulates the number of cycles and instruction of each basic block in a global variable. The "wait" statements are added only at OS function calls, and at specific basic blocks in the code. In this way, the simulation speed-up is also increased.

IV Results

To evaluate the accuracy and speed of the technique proposed, it has been applied to obtain execution time estimations of several examples of a generic test suite. All tests have been compiled with LLVM compiler with –O3 optimizations degree. Results obtained have been compared with the simulation techniques in [1], and with the execution in a real platform and in the host computer.

To obtain the estimation and measures, a XILINX ML506 platform board has been used. In this board, its FGPA (Virtex 5) has been configured to integrate a platform with a LEON3 core [16]. To obtain information about the number of instructions required to execute the examples in the real board, the LEON3 peripheral "l3stat", which provides several HW performance counters has been used.

The examples provided in the Mälaardalen benchmark suite [15] have been executed and evaluated, as proposed in [1]. The results obtained can be seen in Table 1. To obtain accurate time measurements, and minimize the variability added by the Linux OS, the examples have been executed in a loop repeating them 1 million times.

This table first compares the number of instructions required to execute the examples in the real board and the estimation obtained with the proposed technique. As it can be shown, errors are typically around 1%, with a maximum value of 7%. First analysis of the errors indicate that most the instructions not considered by the estimation tool are related to traps in the real board. This problem is under evaluation and will be analyzed in future works.

Table 1**:** Estimation accuracy and simulation speed

	Number of instructions			Execution time (msec per 1Mill exec.)		
Test	Board	Estimation	Error %	Native	Estimation	Factor
adpcm	83904	83685	0.26	45451	45744	1.01
bs	64	67	4.69	14	24	2.00
cnt	2075	2073	0.10	782	973	1.24
compress	4894	4930	0.74	724	1322	1.83
cover	934	936	0.21	125	135	1.08
crc	20851	20850	0.00	207	264	1.30
duff	518	518	0.00	72	81	1.14
edn	46941	47036	0.20	2943	3134	1.06
expint	1704	1703	0.06	541	555	1.02
fac	146	145	0.68	34	62	2.00
fdct	1586	1586	0.00	166	231	1.44
fft	1505	1393	7.44	245	272	1.13
fir	282245	279499	0.97	22292	74774	3.35
insertsort	480	478	0.42	75	135	1.86
jfdctint	2923	2935	0.41	452	516	1.13
lcdnum	185	184	0.54	11	30	3.00
lms	211464	208788	1.27	44524	47134	1.06
ludcmp	1970	1930	2.03	233	274	1.17
matmult	91577	91577	0.00	10422	10502	1.01
ns	6939	6945	0.09	601	734	1.22
nsichneu	6765	6764	0.01	702	802	1.14
prime	6456	6884	6.63	1696	2011	1.19
qsort	1286	1319	2.57	85	111	1.38
qurt	736	743	0.95	221	220	1.00
st	91132	87087	4.44	13358	13364	1.00
statemate	1025	1038	1.27	90	212	2.33
ud	1909	1893	0.84	290	301	1.03

Additionally, overhead results in table 1 shows a mean factor value (time of the proposed approach / time to execute the original code) of 1.45 with worst overhead of 3.35%.

To analyze these results with respect to other previous approaches in the state of the art, the comparison with the results reported in [1] can be found in table 2. Although the values reported in this previous approach were obtained for an ARM-based platform and using a gcc compiler, we consider that the ratios of accuracy and overhead can be compared with the new ones, in order to evaluate the benefits of the proposed approach.

As the table shows, the error of the developed tool for the benchmarks selected in [1], is lower than 1.3, while in the previous approach was up to 16%. Additionally, the overhead factor of the proposed approach is lower than 2.4 times, while in the previous approach the overhead required

execution times up to 40 times bigger than the native execution of the original code for the worst example.

Table 2**:** Comparison with the approach in [1]

	Proposed Approach		Paper [1]	
	Estimation Error %	Overhead Factor	Estimation Error %	Overhead Factor
crc	0.00	1.29	0.00	3.22
edn	0.20	1.06	16.66	10.71
matmult	0.00	1.01	0.14	5.38
nsichneu	0.01	1.14	0.00	40.04
statemate	1.27	2.35	-2.15	10.00

Thus, results demonstrate that the proposed approach can provide a very good ratio accuracy vs. speed, with very high accuracy and very low overhead for the examples proposed.

This technology has been also applied to a larger application, more common in the space domain: a CCSDS 122 use case. For this use case, operations with one core and two cores have been evaluated, using one and two threads to run the code. When moving to a dual-core platform, the mapping of tasks to cores change on each execution, so instructions and cycles on each processor cannot be compared separately. Thus, comparison is only possible analyzing the overall execution time. Results obtained can be found in table 3.

Table 3**:** Estimation of a CCSDS 112 use case

Cores / Threads		Board		Estimation		Error
		CPU0	CPU1	CPU 0	CPU 1	%
1 core	Instruc.	396363906		392061803		1,1
	Time(ms)	13202,6		12407,5		6,0
2 cores	Instruc.	387605066	39716098	413141813	32904	3,3
1 thread	Time(ms)	13786,4	13822,4	12737,65	12737,65	7,8
2 cores	Instruc.	191801968	232804017	214835297	198339547	2,7
2 threads	Time(ms)	9064,5	9104,6	7399,92	7399,92	18,7

As it can be seen, the parallelization integrated in the code can reduce the execution time taking advantage of the dual core. As the table shows, when considering the example, the accuracy of the estimation tool is still quite good since all errors are below 20%.

V Conclusions

Host compiled simulation has demonstrated during last years to be a very attractive solution since it enables obtaining accurate performance estimation times with high simulation speed. Additionally, it provides an early way to create virtual platforms where application SW can be developed, evaluated and debugged considering timing parameters. As a result, it can be adapted to solve the problems found when adapting modern HW platforms to the space domain.

Additionally, results obtained by previous techniques still can be improved in terms both of accuracy and overhead. To solve the discrepancies between the host binary and the target binary, the proposed approach presents a technique that modifies the IR code generated by the compiler for the target platform, in order to model the details of its execution in the target board. Then, this IR code is compiled with the host compiler back-end and executed in the host computer. The modifications done in the IR code have two goals. First, the basic block structure is modified to replicate the structure of the cross-compiled binary. Then, IR basic blocks are annotated with the performance information required to model their execution in the target board when run in the host computer. These modifications of the basic blocks enable to apply simple annotation mechanisms and avoids the overhead found in previous approaches.

The technique has been adapted to a typical Space platform, based on a LEON3 processor, and evaluated with a benchmark suite and a typical CCSDS112 space application.

Results obtained show that the proposed approach overcomes most of the limitations found in previous approaches, providing not only fast and accurate SW simulation and performance analysis but also SW debugging.

REFERENCES

[1] S. Stattelmann, O. Bringmann, and W. Rosenstiel, “Fast and Accurate Source-Level Simulation of Software Timing Considering Complex Code Optimizations,” in Proceedings of DAC’11, San Diego, California, 2011

[2] Y. Hwang, S. Abdi, and D. Gajski, “Cycle-approximate retargetable performance estimation at the transaction level,” in Proc. of DATE, 2008.

[3] T. Meyerowitz et all, “Source-level timing annotation and simulation for a heterogeneous multiprocessor,” in Proceedings of DATE, 2008.

[4] O. Matoussi and F. Petrot, “IR-level annotation strategy dealing with aggressive loop optimizations for performance estimation in native simulation,” in Proceedings of CODES+ISSS, 2017.

[5] Z. Wang and A. Herkersdorf, “Software performance simulation strategies for high-level embedded system design,” Performance Evaluation, vol. 67, no. 8, pp. 717–739, 2010.

[6] X. Zheng, L. K. John, A. Gerstlauer , “Accurate Phase-Level Cross-Platform Power and Performance Estimation”, Proc. of DAC 2016.

[7] Z. Wang, K. Lu, and A. Herkersdorf, “An approach to improve accuracy of source-level TLMs of embedded software”, Proc. DATE, 2011

[8] D. Mueller-Gritschneder, Kun Lu and Ulf Schlichtmann, “Control-flow-driven Source Level Timing Annotation for Embedded Software Models on Transaction Level”, Proc. of DSD 2011.

[9] C. Gerum, O. Bringmann & W. Rosenstiel, “Source level performance simulation of GPU cores”, Proc. of DATE, 2015

[10] R. Stahl, D. Mueller-Gritschneder, U. Schlichtmann, “Automated Redirection of Hardware Accesses for Host-Compiled Software Simulation”, Proc. of FDL, 2018

[11] P Penil, A Díaz, H Posadas, J Medina, P Sánchez, “High-level design of Wireless sensor networks for performance optimization under security hazards”, ACM Transactions on Sensor Networks (TOSN), 2017

[12] O. Bringmann, W. Ecker, A. Gerstlauer, et al., “The Next Generation of Virtual Prototyping: Ultra-fast Yet Accurate Simulation of HW/SW Systems”, Proc. of DATE 2015

[13] S. Schulz, O. Bringmann, “Accelerating Source-Level Timing Simulation”, Proc. of DATE, 2016.

[14] Nick Flaherty, “ARM Moves to LLVM Open-Source for Future Compilers”, http://www.eetimes.com/document.asp?doc_id= 1321853, EETimes, 2014.

[15] Mälardalen WCET research group WCET Benchmark Suite. http://www.mrtc.mdh.se/projects/wcet

[16] Cobham Gaisler AB, Download Cross Compiler System, http://www.gaisler.com/index.php/downloads/compilers