Software Co-Verification Based on Program Traces from Different Processors

Robledo Alencar
University of Campinas - UNICAMP
Av. Albert Einstein, 1251
Campinas, SP, Brazil
robledo.alencar@students.ic.unicamp.br

Sandro Rigo
University of Campinas - UNICAMP
Av. Albert Einstein, 1251
Campinas, SP, Brazil
sandro@ic.unicamp.br

Rodolfo Azevedo
University of Campinas - UNICAMP
Av. Albert Einstein, 1251
Campinas, SP, Brazil
rodolfo@ic.unicamp.br

ABSTRACT

High complexity and tight time-to-market have created new challenges in Multiprocessor System-on-Chip (MPSoC) designs. Virtual platforms play an important role in this scenario, by enabling design space exploration, functional verification, and IP reuse. Processors are key components on such virtual platforms, where Architecture Description Languages are usually applied to automatically generate simulators and other software tools. This paper focuses on the software verification path and its main contribution is a trace matching methodology, capable of matching traces for different architectures providing a good feedback about the correct execution of a program. We evaluated our methodology and tool using ArchC-generated instruction-set simulators (ISS) for two different processors: MIPS and PowerPC, and compared all 75 programs from acStone benchmark, 3 synthetic errors injected in the programs and in the simulator, and 10 programs with 15 variations from MiBench that matched correctly.

1. INTRODUCTION

The ever increasing complexity of electronics systems has created several new challenges regarding the design methodology. In order to deal with such complexity and a tight time-to-market, current Multiprocessor System-on-Chip (MPSoC) designers are adopting higher abstraction levels and the so called Electronic System Level (ESL) design methodology. In this scenario, it is mandatory to have languages and tools enabling hardware and software co-design, early platform verification, performance evaluation, and design reuse.

The advantages of virtual prototypes for the entire system architecture are twofold: (a) an early definition of the hardware-software interface, thus allowing a fast start of software development (no need to wait the hardware platform to be finished); and (b) an efficient approach to enable platform trade-off, analysis and debugging.

Considering that microprocessors are central modules to platform coordination and control in any SoC design, efficient microprocessor ESL models are thus relevant pieces in the design of modern virtual platforms. Such models should be very fast, and accurate enough only to the level where they allow a well-defined interface to the software boundary.

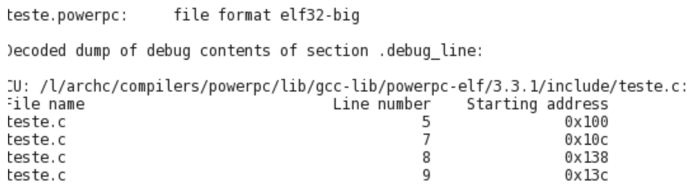
Architecture Description Languages (ADLs) have been widely used to processor model development. ADLs capture both structure, by means of hardware components, and behavior, by means of the instruction-set, of processor architectures. They are capable of generating a software tool chain, like simulators, assemblers, linkers, compilers, debuggers, etc, which make them valuable in a design automation methodology, like in ESL. Therefore, it is a natural step to ADL processor models to be included into high-level virtual platform models seeking for automation and architecture exploration.

This paper focuses on the software verification path, where several tools may be under development at the same moment: the full platform simulator, the compiler and optimizations, and the hardware abstraction layer to mention a few. The main contribution of this paper is a verification methodology and tool that matches traces from different architectures, providing a good feedback about the correct execution of a program. In this methodology, designers can use a reference platform, which simply may be a previous version of the current one, or other completely different environment even with another processor.

We evaluated our methodology and tool using ArchC-generated instruction-set simulators (ISS) for two different processors: MIPS and PowerPC. ArchC [1, 2] is an Architecture Description Language that uses a simple processor definition as input and generates several tools as output like assemblers, linkers, debuggers, and ISS. The generated ISS is based on the SystemC [4,8] simulator, allowing easy integration with external components through a TLM interface [9]. Without lack of generality, we use the ArchC trace collection capability to implement our tool. We could use other simulators or even PIN [7] to get the same data. The trace file is a sequence of address of each executed instruction. We evaluated our results using all 75 programs from acStone benchmark, 3 synthetic errors injected in the programs and in the simulator, and 10 programs with 15 variations from MiBench that matched correctly.

This paper is organized as follows: Section 3 describes our

Figure 1: DWARF dump from `objdump`.



```
teste.powerpc:     file format elf32-big

ecoded dump of debug contents of section .debug_line:

U: /l/archc/compilers/powerpc/lib/gcc-lib/powerpc-elf/3.3.1/include/teste.c:
ile name                              Line number    Starting address
teste.c                                         5               0x100
teste.c                                         7               0x10c
teste.c                                         8               0x138
teste.c                                         9               0x13c
```

co-verification mechanism. Section 4 shows our experimental results and Section 5 presents our conclusions and future work.

2. RELATED WORK

Trace based strategies have been used before. Wild et al introduced TAPES [13], a trace based mechanism for architecture performance evaluation with SystemC to aid architecture exploration, where authors use SystemC TLM and a trace-driven simulation to capture the interaction of system resources. High-level verification in SystemC was always a concern since it is no easy task to debug a whole platform using a tool like GDB without getting into the SystemC kernel code. Rogin et al [10] presented a system do aid in this task, but no co-verification and focus on the software verification path was present. Hsieh [6] presented an infrastructure for debug and trace of a DSP system, consisting of the in-system trace interface and its methodology to optimize the compression rate of the program and data traces. The platform is used for HW/SW co-verification on a FPGA based implementation. Shimizu et al [11] describes how the verification team working on the Cell BE microprocessor benefited from trace-based techniques, in this case to determine the data flow into the platform, to uncover bugs.

3. CO-VERIFICATION MECHANISM

As mentioned before, our co-verification mechanism is based on execution trace files. Figure 2 shows a simple sample code that we are going to use throughout this section. It has a `for` loop, an `if` statement, a local function call, and a `printf` library function call. Our problem is to find a way to verify a behavior from this program executing in one machine with processor P_1, compared to its own behavior executing on another machine on a different processor, P_2.

```
1 void SayHello(int i) {
2   if (i % 2 == 0)
3     printf("Hello World!\n");
4 }
5 int main(int argc, char *argv[]) {
6   int i;
7   for (i = 0; i < 10; i ++)
8     SayHello(i);
9 }
```

Figure 2: Sample C code

We decided to use the source code lines in such a way that it becomes simple to compare two program executions. In the example case, the program starts at line 5 and goes through the `for` loop (line 7), following a function call (line 8) followed by line 1, finds an `if` (line 2), the library function call `printf` (line 3), and so on. Combining all these lines we can reach a trace from the program source code similar to this: 5, 7, 8, 1, 3, 4, 7, 8, 1, 2, 4, ... This is exactly what we need to compare two execution traces. Unfortunately, the real trace generated by the ISS is composed of executed instruction addresses (PC values), which varies between processors. So we need to convert a sequence of memory addresses to source-code-level traces and then compare both traces, and we used the debug information as a common denominator between the two versions of the same program.

3.1 Background

In this section, we briefly introduce the basic infrastructure upon which we have developed our methodology.

3.1.1 SystemC and ArchC

SystemC [4, 8, 9] appeared as a response to the need of higher abstraction levels. Traditional HDLs like VHDL and Verilog were not suitable for this task. It is also possible to apply SystemC in RTL designs, but that is not its main purpose. SystemC became one of the most used languages, both in academia and industry, to enable the so called system-level design, which evolved to what is known as Electronic System Level (ESL) design [3]. In fact, SystemC extends C++ to provide hardware modeling capabilities. The complete simulation library is distributed under an open-source license. One of its key characteristics is the smooth integration of hardware and software in the same executable model, since everything is essentially written in C++.

From its very beginning, ArchC [1,2] was created targeting SystemC users, so its syntax is totally based on SystemC. The main idea is to extract information from a generic SystemC processor description in order to automatically generate tools to experiment and evaluate a new *Instruction Set Architecture* (ISA). It is also published under an open-source license since 2004. As it evolved, the most common use for ArchC processor models proved to be their integration on virtual platform models, like SoCs and MPSoCs, written in SystemC. Users may design their own processor or experiment with the several existing models and tune them to their needs, even by easily specializing the ISA. In this scenario, functional verification tools are very important in order to guarantee that no error was inserted into the processor model or in the embedded software running on it. On the ArchC website [12], users can find the language tools, several processor models (including MIPS and PPC used in this paper), compilers, benchmarks, and a reference infrastructure to build virtual platforms available for download.

3.1.2 The DWARF format

The DWARF format represents the debug information of a program by mapping each source code line of the program to its program counter (PC) value. Notice that, as a function call may execute many lines of source code, a single line may be translated into many assembly instructions by the compiler, so the DWARF format maps a line of source code only into the PC value of its first assembly instruction. An example of the DWARF format, dumped with `objdump` is shown in Figure 1, where the second column is the source code line number and the third column is the mapped PC.

For instance, the line 5 of program `teste.c` starts at memory position 0x100.

3.2 Basic Trace Mapping

To map two different memory addresses traces and compare them, we compile both programs with debug information, we used the DWARF format because it is widely available but any other debug information could be used. Inside the DWARF section, there is a map between instruction addresses and their corresponding source code files and lines. So, our first step is to read all the DWARF section and convert all the instructions addresses to source code address. The problem with this step is that each different processor (P_1 and P_2) usually uses a different number of instructions to represent the same source code line and the trace is inflated with this repeated information. We solved this by simplifying the mapping such that if the source trace has the addresses A_1, A_2, A_3, A_4, and the first three addresses map to the same source code line, say line 5, it will be placed only once in the output trace.

Figure 3 shows the mapping flow for two different architectures, MIPS and PowerPC. In this specific case, we want to verify if one of the architecture toolset (compiler, libraries, and simulator) matches the result of the other by comparing the execution of the same program (source code). For each architecture, the source code is compiled (using `gcc` in our case) to produce a binary program with debug information. The debug information is extracted using `objdump` and the full instruction trace is generated by the ArchC simulator (one different instance for each processor). After that, we created the `converter` to map the *instruction trace* and the *debug information* to the *source code trace.* Both *source code traces* are compared using our `checker`.

Figure 4 shows a mapping example of a simple C source code (middle) with a MIPS (left) and a PowerPC (right) implementations. The C source code lines are numbered sequentially while the real address is used to identify each assembly code line. For instance, the `for` command at line 5 of the source code is represented by lines 0x10c–0x140 of MIPS assembly and by lines 0x10c–0x134 of PowerPC assembly (we intentionally used blank lines to align the smaller blocks to the bigger ones).

3.3 Multiple Commands On The Same Line

The basic mapping mentioned before works pretty well even for loops and conditional statements written as in Figure 2. Unfortunately, there are times where both the `for` command and the command inside it are written in the same line (like if we placed line 8 in front of 7 by removing the line break). In this case, the previous approach would compact all the address for those two commands in only one source code line. Although we cannot change the source code, we can detect that there is a loop being executed by searching for a back-edge. We updated the naive previous approach to a better one that uses the back-edges to detect the traces so that the 10 loop iterations appear 10 times in the trace file.

Alternatives solutions to multiple commands in the same line are:

- Use the DWARF column number to identify different commands in the same line, when this number is available for the processors.
- Reformat the code so that each command is placed in a different line, completely removing the problem.

Each of these alternatives can complement our solution when they are available. Notice that both of them require an improved tool chain with support for column numbers or source code reformatting.

3.4 Function Calls and Libraries

The previous approach works well with function calls and libraries when the source code is exactly the same, but it is not always the case like in the *newlib* processor specific functions. In this case, some functions will have completely different traces (implementations) for the two processors, some of them with assembly implementations. To solve this, we need to skip some internal functions and we detect them by the total number of source code lines available in the debug information file. Although it is possible for two completely different functions to have exactly the same number of lines, it is not the common case and we solve this latter problem by using a list of functions to skip.

To minimize the impact on the code size, it is common that library implementers define only one function per file. This behavior make our task easier to detect different implementations by counting the number of lines. For the very low number of function with different implementations and the same number of lines, we use the aforementioned list of functions to skip.

As we will detail on section 4, our simulator uses floating point emulation and we had to exclude these functions from the trace since they use inlined assembly code. It is worth mentioning that this has been done to circumvent a simulator restriction, and not as part of our methodology.

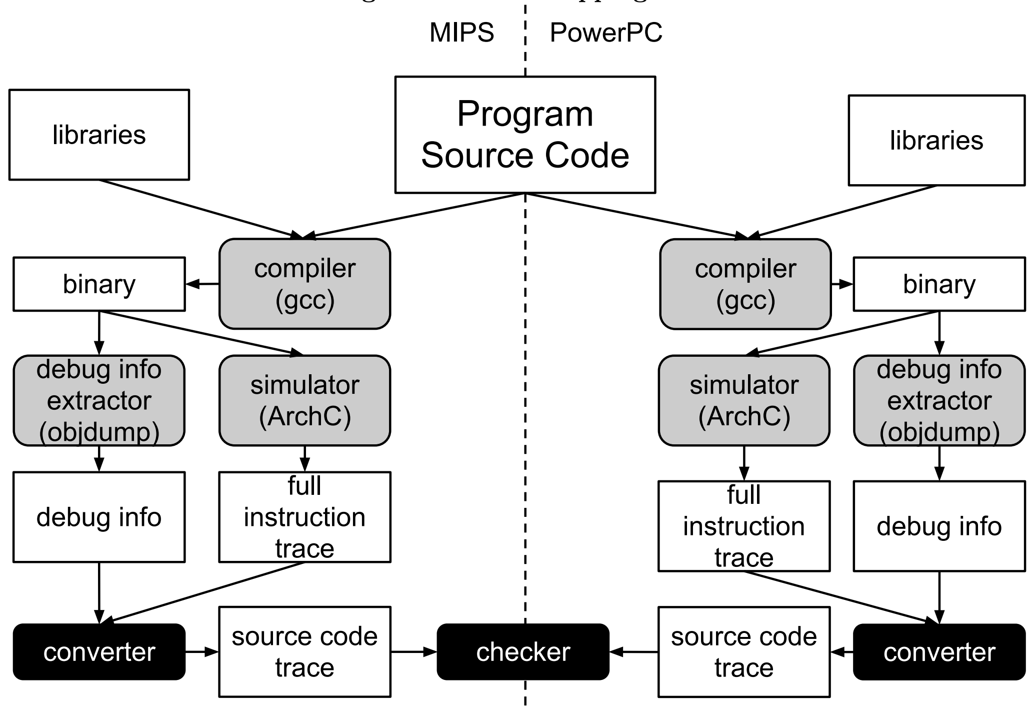
3.5 Comparing the Traces

The final step is to compare the traces generated by both executions, one from each processor. These traces should completely match but we can handle partial matches in the following way:

No library call: We remove all the library calls and only try to match the user code. This is the most relaxed comparison method and is the most easier to check due to the small number of files, although it is designed to skip the tiles that may already have been tested. Notice that the meaning of libraries can vary by considering only the standard c library as newlib and glibc, or to consider auxiliary libraries like cryptography, graphs, etc. We simplified this issue by considering the file full path as the indicative of library presence. We consider a library every file in the system library path. Notice that we must also include the cross-compiler library path here. Alternatively, we could consider only files outside the current directory and one of its subdirectories.

Only first level library call: We only try to match library functions that were called by non-library functions. As an example, in Figure 2 we will try to match the `printf` function but not the `putchar` function that may be called inside `printf`. The first level library functions are usually a more general implementation that requires almost no modification between platforms. These implementations rely on underling functions to provide platform specific resources. Although

Figure 3: Trace Mapping Flow



we restricted to only the first level, the algorithm is easily extendable to more levels.

Same source functions: We match all the functions that have the same amount of source code lines detected through debug information. We can still add some exceptions that will not be compared. This approach does look only to function that matches the very first criteria of similar implementation but it proves really efficient to detect different implementations of the same function among different libraries.

All functions: We try to match all the functions. Every instruction executed by each platform will be mapped to their source code line and we will process them in the usual way. We still provide the option to exclude some functions.

3.6 Limitations

The aforementioned method has some limitations that will be addressed in future projects. The first one is that we cannot split two commands in the same line so we cannot detect if an `if` with the corresponding operation in the same line evaluates to True or False by the program trace. In the same way, we cannot detect if one of the operations inside the `for` declaration evaluated correctly. In both cases, we think that the rest of the program trace may be affected by an incorrect behavior and we may be able to detect this impact latter. Alternative implementations for these limitations include using the DWARF column name and reformatting the source code. These two alternative implementations may help when the correct tools are available, as mentioned in Section 3.3.

Program optimizations also impose a challenge to the trace generation. Some instructions may get merged together or swapped, generating a different execution order. So far we recommend that, for the functional validation, the user reduces the optimization effort in the same way that it is recommended when debugging programs.

The trace file will be extremely big if every instruction address gets written to the disk. Although we are still using the original trace file in disk, we can easily adapt our algorithm to process the address before writing them to disk reducing the required disk space and speeding up the execution. We evaluated the trace size in Section 4 where Table 1 shows the program trace sizes for the MiBench programs. Notice that we can also compress the traces and get a very good improvement in disk space utilization.

4. EXPERIMENTS

We run four sets of experiments, comparing a MIPS processor against a PowerPC one regarding the execution of a set of programs.

The first experiment was composed by the acStone benchmark, acStone is a set of kernel programs used to check processor models in increasing level of complexity, starting with an empty program and covering most of the C language constructors. For all 75 programs of acStone, we compared the execution traces generated by our tool from both processors and they matched completely.

As a second experiment, we checked our back-edge detection algorithm and its functionality in verifying the number of loop iterations executed. We compiled the program from Figure 2 to MIPS and modified the for loop to 20 iterations, as shown in Figure 6, and compiled it to PowerPC. As expected, the modified trace file mentioned the extra iterations in the second instance (PowerPC). We also mimic another error in the code generation by removing a function call in `bitcount`, that our tool was also capable of detecting. These two errors, together with acStone, try to expose cases where

```
100: addiu sp,sp,-16     1  int main() {                               100: stwu r1,-32(r1)
104: sw s8,8(sp)         2                                             104: stw r31,28(r1)
108: move s8,sp          3                                             108: mr r31,r1
                         4    int a;
10c: li v0,-5000         5      for(a = -5000; a < -2; a = a >> 1);    10c: li r0,-5000
110: sw v0,0(s8)         6                                             110: stw r0,8(r31)
114: lw v0,0(s8)         7                                             114: lwz r0,8(r31)
118: nop                 8                                             118: li r9,-2
11c: slti v0,v0,-2       9                                             11c: cmpw r0,r9
120: bnez v0,130         10                                            120: blt- 128
124: nop                 11                                            124: b 138
128: j 144               12                                            128: lwz r0,8(r31)
12c: nop                 13                                            12c: srawi r0,r0,1
130: lw v0,0(s8)         14                                            130: stw r0,8(r31)
134: nop                 15                                            134: b 114
138: sra v0,v0,0x1       16
13c: j 114               17
140: sw v0,0(s8)         18
                         19
144: lw v0,0(s8)         20   return a;                                138: lwz r0,8(r31)
                         21
148: move sp,s8          22 }                                          13c: mr r3,r0
14c: lw s8,8(sp)         23                                            140: lwz r11,0(r1)
150: jr ra               24                                            144: lwz r31,-4(r11)
154: addiu sp,sp,16      25                                            148: mr r1,r11
                         26                                            14c: blr
  // MIPS code           27 // Original C source code                    // PowerPC Code
```

Figure 4: C source code with the MIPS (left) and PowerPC (right) versions. The assembly code blocks start at the corresponding C source code lines

```
1 //!Instruction sra behavior method.
2 void ac_behavior( sra )
3 {
4   RB[rd] = (ac_Sword) RB[rt] >> shamt;
5 }
```

```
1 //!Instruction srl behavior method.
2 void ac_behavior( srl )
3 {
4   RB[rd] = RB[rt] >> shamt;
5 }
```

Figure 5: ArchC MIPS model behavior for instructions `sra` and `srl`

the under development compiler may have bugs.

```
void SayHello(int i) {
  if (i % 2 == 0)
    printf("Hello World!\n");
}
int main(int argc, char *argv[]) {
  int i;
  for (i = 0; i < 20; i ++)
    SayHello(i);
}
```

Figure 6: Code from Figure 2 with 20 iterations instead of 10.

The third experiment was the injection of errors in the simulator. The error was injected on the MIPS simulator and the PowerPC was used to detect the result. We injected a very simple, although difficult do detect, error on the shift right instructions. In fact, this error was a bug in the very first version of ArchC MIPS model and took two weeks for the developers to find it by debugging the printf function. MIPS processors have two types of shift right instruction, `sra` for arithmetic version and `srl` for logic version. The **correct** implementation of both is shown in Figure 5. We removed the typecast of `sra` (Line 4) turning it equal to `srl`. This makes the MIPS version of the loop to run only one iteration of the `for`, which is detected by our toolset.

The last experiment was the execution of 10 programs with 15 variations from the MiBench Benchmark Suite [5]. Table 1 shows the evaluated programs with their trace size, the reduced trace size, and also the amount of debug information extracted from the binary files. The reduced trace size can be made even smaller if compression is applied. For example, the MIPS Bitcount program trace file originally has 464 MB that are reduced to 45M, a 10 times reduction, but can get as small as 20KB if compressed with bzip2. After our algorithms, they are reduced to 45 MB for MIPS and 45 MB for PowerPC. All the resulting traces matched as expected. Some of the programs, like Basicmath, are based on floating point operations. Since both MIPS and PowerPC ArchC models does not include floating point instructions, they were emulated in software, which generated very big traces. The floating point emulation libraries have different implementations for MIPS and PowerPC, so they were ignored in the trace comparison, generating reduced traces much smaller if compared to the original ones.

5. CONCLUSIONS AND FUTURE WORK

This paper presented a new approach to co-verify a program execution by matching the execution trace from the

Program	**Original Traces**		**Output Traces**		**Debug**	
	MIPS	PowerPC	MIPS	PowerPC	MIPS	PowerPC
ADPCM Enc	396	313	73	73	224	252
ADPCM Dec	304	242	54	54	224	252
Basicmath	6.4G	7.1G	2.1	2.1	284	316
Bitcount	464	389	45	45	244	272
CRC	268	210	16	16	236	264
Dijkstra	462	379	24	24	288	332
JPEG Enc	155	123	26	33	572	620
JPEG Dec	54	32	6.4	6.8	572	620
Quicksort	114	116	1012K	1012K	292	336
Susan Smooth	454	2.8G	33	33	364	400
Susan Edge	59	51	2.1	2.1	364	400
Susan Corners	29	25	2.1	2.1	364	400
FFT 4096	3.6G	3.9G	2.3	2.3	268	300
FFT 8192	8.6G	9.4G	5	5	268	300
SHA	154	117	4.9	4.9	224	252

Table 1: Trace Sizes for the Mibench Suite. Values are in MB unless otherwise mentioned.

current architecture against another one that has already a correct execution environment. We developed a conversion algorithm based on the source code debug information and apply this algorithm for both programs, verifying the correct execution by comparing the result trace.

We evaluated our results using all 75 programs from acStone benchmark, 3 synthetic errors injected in the programs and in the simulator, and 10 programs with 15 variations from MiBench that matched correctly.

As the future work, we intend to improve the detection algorithm to provide a better feedback to the developer in cases where only part of the debug information is available. In this case, we may allow the user to skip part of the trace, as we do now, but also to use some code detection features to find hidden loops and function calls, and trying to match them on the other platform.

6. ACKNOWLEDGEMENTS

This work was partially supported by CNPq, FAPESP and FAEPEX. We would like to thank the anonymous reviewers for their feedback on the submitted paper.

7. REFERENCES

[1] G. Araujo, S. Rigo, and R. Azevedo. *Processor Description Languages*, chapter Processor Design with ArchC, pages 275–293. Morgan Kaufmann, 2008.

[2] R. Azevedo, S. Rigo, M. Bartholomeu, G. Araujo, C. Araujo, and E. Barros. The archc architecture description language and tools. *International Journal of Parallel Programming*, 33(5):453–484, 2005.

[3] B. Bailey, G. Martin, and A. Piziali. *ESL Design and Verification: A Prescription for Electronic System Level Methodology*. Morgan Kaufmann, 2007.

[4] D.A.S Committee. *IEEE Std 1666-2005 IEEE standard SystemC language reference manual*, 2006.

[5] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown. Mibench: A free, commercially representative embedded benchmark suite. In *Proceedings of the Workload Characterization, 2001. WWC-4. 2001 IEEE International Workshop*, pages 3–14, Washington, DC, USA, 2001. IEEE Computer Society.

[6] Ming-Chang Hsieh and Chih-Tsun Huang. An embedded infrastructure of debug and trace interface for the dsp platform. In *Design Automation Conference, 2008. DAC 2008. 45th ACM/IEEE*, pages 866 –871, 2008.

[7] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '05, pages 190–200, New York, NY, USA, 2005. ACM.

[8] OSCI. Systemc library. WebSite, March 2011. http://www.systemc.org.

[9] OSCI. Tlm-2.0 standard. WebSite, March 2011. http://www.systemc.org.

[10] Frank Rogin, Christian Genz, Rolf Drechsler, and Steffen Rülke. An integrated systemc debugging environment. In Eugenio Villar, editor, *Embedded Systems Specification and Design Languages*, volume 10 of *Lecture Notes in Electrical Engineering*, pages 59–71. Springer Netherlands, 2008. 10.1007/978-1-4020-8297-9-5.

[11] Kanna Shimizu, Sanjay Gupta, Tatsuya Koyama, Takashi Omizo, Jamee Abdulhafiz, Larry McConville, and Todd Swanson. Verification of the cell broadband engine; processor. In *Proceedings of the 43rd annual Design Automation Conference*, DAC '06, pages 338–343, New York, NY, USA, 2006. ACM.

[12] ArchC Team. Archc architecture description language. WebSite, March 2011. http://www.archc.org.

[13] Thomas Wild, Andreas Herkersdorf, and Gyoo-Yeong Lee. Tapes—trace-based architecture performance evaluation with systemc. *Design Automation for Embedded Systems*, 10:157–179, 2005. 10.1007/s10617-006-9589-4.