Emulating Operating System Calls in Retargetable ISA Simulators

Marcus Bartholomeu *Sandro Rigo*
Rodolfo Azevedo *Guido Araujo*

Technical Report - IC-03-029 - Relatório Técnico

December - 2003 - Dezembro

Emulating Operating System Calls in Retargetable ISA Simulators

Marcus Bartholomeu* Sandro Rigo† Rodolfo Azevedo Guido Araujo

Abstract

In this paper, we propose a method that enables operating system calls from inside architecture simulators. The proposed framework provides support to file I/O and dynamic memory systems, and can be incorporated into any ISA simulator, at the instruction or cycle-accurate levels. It enables calls to POSIX-compatible system routines in a simulated application, without requiring any change to the application code. Since file I/O is performed transparently, the input and output data for the application program is read/written directly from/to the host filesystem, and all console operations are redirected to the host console. This framework was tested in ISA simulators synthesized from models written in the ArchC Architecture Description Language, but it can be incorporated into any ADL or hand-coded simulator. Instruction and cycle-accurate models, for both MIPS I and SPARC V8 architectures have been thoroughly evaluated by successfully running programs from the Mibench and MediaBench benchmarks.

1 Introduction

With the advent of System-on-Chip (SoC) designs, it is becoming possible to design an entire embedded system onto a single chip. As we approach the availability of 100 Million gates, for the upcoming 90nm VLSI technology, a new scenario is been drawn in which SoCs could be composed of a number of specialized processors interconnected by an elaborate *Network-on-a-Chip* (NoC). In such scenario, methods that are capable of fast tailoring a processor to some specific task can become an important design tool.

Moreover, the increase in the complexity of the applications running on such systems has considerably increased the amount of data stored. Simulating such complex systems without a file system support will certainly become a verification nightmare. Even highly constrained devices, like handhelds and palmtops, are gradually been designed with non-volatile memory devices that support file system implementations.

As a result, models that can provide system-level processor simulation and OS support are becoming essential verification and performance analysis tools. Researchers have been able to automatically synthesize simulators starting from models written in Architectural Description Languages (ADLs) like EXPRESSION [6], LISA [14], nML [1], ISDL [5] and

*Research supported by FAPESP (grant 01/09424-0)

†Research supported by FAPESP (grant 00/14376-2)

others. Although much is heard about RTOS for embedded systems, we have not been able, to the best of your knowledge, to identify any ADLs that can automatically generate simulators capable of supporting OS calls. Typically, ADLs solution to this problem is to change the program to be simulated so it incorporates all I/O data as program data. Unfortunately, this approach requires the program to be recompiled whenever a new I/O data is required.

The approach proposed in this paper permits not only to run programs without any modification, but it also allows I/O operations to occur directly with files or terminals, as it is expected from the program behavior when it runs in real systems. We already have a totally functional implementation of such technique, for both MIPS I [8] and SPARC V8 [10] architectures, though there is still space for improvements, like the support to multi-task and real-time scheduling (so it can be called an RTOS).

This paper is divided as follows. Section 2 lists the work available in the literature. Section 3 describes the central ideas behind our approach to ADL system call emulation and shows how our approach can be integrated into any ADL or processor simulation platform. Section 4 describes our experimental results and Section 5 provides the main conclusions and proposes future extensions.

2 Related Work

There is not enough information in the literature to support the claim that system call emulation is implemented in the currently available ADLs. Halambi et al. [7], in his work on EXPRESSION, notes that more work is needed in the support for operating systems for future SoCs that will contain heterogeneous multiple processor architectures.

Simplescalar [2] is a well-known instruction set simulator that has good performance, but limits the retargetability to MIPS-like architectures. Complex DSP processors like Texas Instruments TMS320C62x, for example, cannot be correctly described. Simplescalar supports system calls by the same mechanism of the MIPS instruction set. System calls are initiated with the `SYSCALL` instruction. Prior to execution of a `SYSCALL` instruction, register `$v0` should be loaded with the system call code. The arguments of the system call interface prototype should be loaded into registers `$a0-$a3`, respecting the prototype order.

The FastSim [13] is a memoizing (an optimization commonly used in functional programming languages) micro-architecture simulation. Version 2 of FastSim addresses the primary weakness of the original FastSim simulator, which is the lack of flexibility. Different from its initial version, which simulates a static out-of-order processor, the second version is now a framework for generating architecture simulation. A simulator specification is written in Facile [12], which is described as an Architecture Simulation Language. Interaction with the operating system does not seem to be available.

The Sun's Shade [3] simulator is another instruction set simulator that allows hand-made retargets for MIPS I, SPARC V8 and SPARC V9 architectures. It uses the a dynamic cross-compilation technique to achieve a good performance. The main purpose of Shade is to instrument code so as to extract exact cycle counts. Although operating system calls are supported, model synthesis is manual and specialized to RISC architectures.

The GNU Project has a GNU simulator that is generated when cross-compiling the GNU Debug[1] package. It is seamless integrated to the debugger if called by the `target sim` command. This simulator has operating system support but it allows only a fixed group of architectures for retargetability. A missing functionality is the ability to get command line arguments, so the needed arguments have to be manually encoded and compiled with the application and thus become fixed.

We have not been able to identify, from the literature available for ADLs like LISA [14], nML [1], ISDL [5] and others, any indication that support the existence in such languages of mechanism to allow automatic system-calls mapping.

Although this work is based on simulators that are automatically generated from the ArchC [11] architecture description language, the ideas exposed here can be implemented into any ADL or even any processor simulator.

3 A Flexible OS Emulation Technique

The majority of the simulators described in Section 2 cannot handle OS calls properly. Their major goal is to simulate the instruction behavior in the target architecture.

On the other hand, real embedded applications deal with great amounts of input data and if the simulator does not have I/O system support (by an operating system), input data cannot be dynamically provided to the processor as it is expected in a real program run. A typical approach to this problem is to assign a big vector to the data input, hard-coded into the application code, and compile it together with the application. Another vector is allocated to the processed output. This means that correctness can only be measured by comparing the traces generated by the simulator with the traces from the real hardware or another simulator or by using another vector to include the correct output into the program, requiring even more changes in the benchmark source code.

The motivation for this work is to implement a general way to couple an I/O system and a file system functionality to any architecture simulator. This makes verification easier because the simulated application can read its input from a file (or even the keyboard) in the host machine, and write the output to another file. Debug and error messages can be directed to the host screen for fast error debugging. The existing benchmarks sources need not to be changed in any way, and any additional option to the simulated program can be given as command line arguments, that are copied from the simulator command line.

In this section we present the main ideas behind our approach to simulation OS support. Note that if we include an operating system in the model for the target architecture, the host operating system needs also to be considered, as it is the one that will perform the real job. Thus, a communication interface between the application OS call and the native OS is required.

Figure 1 shows the levels of abstraction considered in our framework. We model the host machine by using three abstraction levels. The lowest level is the hardware, the processor that executes the instruction flow. The next level is the operating system, which provides some common functionalities used in programs, like I/O, file system and memory

[1]GNU Debug homepage: `http://www.gnu.org/software/gdb`

management. Then there is the application level, where the user program is in. The system simulator runs as an application in the host machine, and simulates the two lower levels of the target machine.

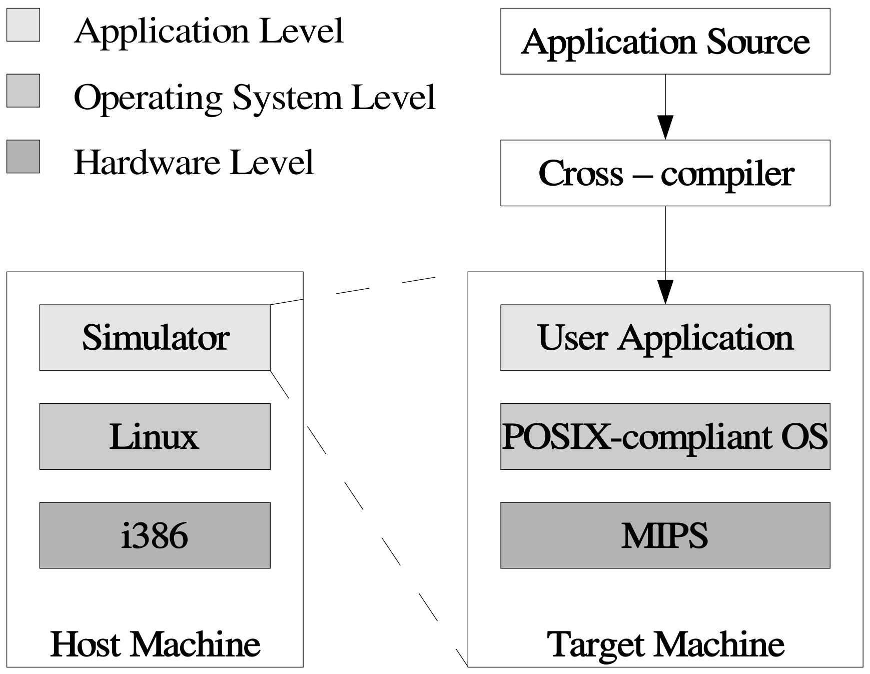


Figure 1: Levels of abstraction

The source application is compiled with a cross-compiler for the target, and the resulting binary program is the input to the simulator, completing the application level for the target virtual machine. These abstraction levels facilitate changes in processor and operating systems used in the execution environment.

System calls are implemented by reserving a block of program memory addresses and associating each address to a system call that needs host system interaction. Not all system routines need host interaction in this implementation due to some incompatibilities, as shown latter. We believe that this approach is very general and can be implemented into any simulator.

As a jump to one of the special addresses behave like a system call routine, some additional information from the architecture, which is not commonly present in ADLs, are needed. This information is present in the Application Binary Interface (ABI) [15, 16] manual of the architecture. The next sections explain the changes that we have made in each block of Figure 1 to address the main problems associated to this implementation.

3.1 Modeling the OS Abstraction Layer

Programming in a high level language like C/C++ is a necessity for the today development flow. The majority of applications are designed using C/C++ together with some hand-tunned assembly code to improve performance of critical inner-loops.

Any non-trivial C/C++ program uses the standard C library (`stdlib`), which provides the most essential program routines for I/O, dynamic memory allocation and other utilities. There is a widely used implementation of `stdlib` from GNU, the GNU C Library, used in most Linux distributions, but the most suited implementation for embedded systems is the Cygnus Newlib, due to its light implementation.

Newlib libraries implement system interaction routines, like input and output (to/from terminal or files), memory management, time information, etc. Many of these routines do not call the system directly, but use a small set of wrapper functions for this task. All input/output routines in C (like `printf()` family, `fwrite()`, etc.) use only 4 system-calls: `open()`, `close()`, `read()`, `write()`. Memory management uses only `sbrk()` to enlarge/shrink the stack.

We decided not to implement this functionality through software interrupts because each architecture may treat their interrupts in their own particular way. The idea used for the implementation is similar to the one used in some system board simulators. An address space is reserved so that when the program counter points to one instruction from this space, the simulator executes a special behavior associated to the specific address. Hence, the real instruction that is in this address is meaningless (it is normally an invalid instruction, anyway), in the sense that it does not pass through the normal fetch and decode process. The simulator calls the host routine equivalent to the one mapped to that specific address.

The arguments passed to the simulated routine need to be converted into arguments to the real host routine that will do the job. The way to set these arguments vary from one processor to another, and can normally be found in the ABI (Application Binary Interface) Manual for the architecture.

As the designer of a new architecture, one is able to tell from which storage elements (memory, register bank, etc) the arguments to the system call will come from. A simple way to do that is by writing interface functions that provide the required information to the ADL compiler. Typical information that is required by most of the operating system calls are:

- How to get the first three arguments provided by a function call, and more than that, how to distinguish the type of the arguments from integer numbers, pointers or strings (note that in some cases an endianness conversion may be required);
- How to save the return value as an integer or pointer (may also need conversion);
- How to return from the system call. Normally this is just a jump instruction that uses the register which contains the return address;
- How to store strings so the command line arguments can be stored into memory before simulation begin.

All operating system functionality is implemented as a new class in ArchC, named `ac_syscall`, which has virtual methods that needs to be specialized for each new processor to provide the correct ABI implementation for the architecture.

3.2 Example: the `open()` function

We now give an example of the `open()` low-level system call routine. Other routines were implemented in a similar fashion. The POSIX prototype for `open()` is

```
int
open(const char *pathname, int flags, mode_t mode);
```

We implemented a wrapper function in the simulator to interface the target call and the real host system call for `open()`. This wrapper can be seen in Figure 2.

```
void ac_syscall::open()
{
  unsigned char pathname[100];
  get_buffer(0, pathname, 100);
  int flags = get_int(1); correct_flags(&flags);
  int mode = get_int(2);
  int ret = ::open((char*)pathname, flags, mode);
  if (ret == -1) {
    RUN_ERR("System Call open (file %s): %s\n",
            pathname, strerror(errno));
    exit(EXIT_FAILURE);
  }
  set_int(0, ret);
  return_from_syscall();
}
```

Figure 2: Interface function for `open()`

As it can be seen from the above function prototype, the first argument to `open()` is a string representing the file name to be opened. The function `get_buffer()` is used to get the string. This function is responsible for finding the char pointer argument, read the target memory, make any endianness conversion from the target to the host architecture, if necessary, and return the string as a char pointer to the host memory space.

The next argument to `open()` is an integer number that encodes the option flags. This argument is caught from the target by using function `get_int(1)`, which is able to find the argument `1` (the second one) in an architecture dependent way, and to return it as an integer. The third `open()` argument is also an integer and is treated similarly.

All three arguments are collected from the application, and this is followed by the execution of the `open()` function from the native host. The return value, which is an integer for the `open()` call, must be returned to the application. Function `set_int()` knows where the target expects to receive the return value, and thus it takes the appropriate actions.

Yet another result must also be communicated upwards to the application. It is a global variable called `errno`, used by every function in the C Library implementations as a complementary error return value. The problem is that each simulated application will have this variable in a possibly different target memory position, and there is no way to know its position if the application loaded into the simulator does not carry symbol information. We chose to treat C Library errors like these by showing the error message and exiting simulation with a failure code (as shown in Figure 2). This approach can be improved, but since the benchmark programs do expect their I/O functions to behave correct, we choose to stop the simulation at errors.

3.3 The Application Binary Interface (ABI)

Functions that know how to get parameters from the target system calls are an example of interface functions that must be implemented for each target architecture. Functions in this group are normally very small, with less then five lines. The information required to implement them is taken from the processor Application Binary Interface manual. As an example, Figure 3 shows some functions for the MIPS architecture and how they are encoded using ArchC to access the register file and memory. Notice from that example, that the only information needed was the number of registers that the architecture used as arguments, the return value, and the return instruction.

The methods presented in Figure 3 are just specialization of some methods of the base ArchC system call class, called `ac_syscall`, which has 265 source code lines and do not require any change between architectures. The MIPS I system call methods required only 51 extra lines in the source code file to implement the specialized methods. The SPARC V8 also required the same number of extra lines since the difference between the MIPS-specific and SPARC-specific files are only the register numbers. The small number of lines in the specific files show that it is very easy to port this method to new architectures.

3.4 Compiler Requirements

As mentioned before, system calls for the target architecture are mapped to a reserved block in program memory where each address was associated to a system call that needs host system interaction. As one might expect, this will require linker support.

We chose the GNU Compiler Collection (GCC) as the compiler for our evaluation due its high reconfigurability and availability of many ports to various target processors. In the following, we list the main changes required to implement our system-call approach into an existing GCC cross-compiler[2]:

Create a new spec file. The specification file is the main source of configuration for GCC. If a new spec file is specified at the command line, its contents are merged with the original options. Hence, only the changed options need to be in the new spec file. Figure 4 shows these changes: a new start file (instead of the original C Runtime functions in `crt0.o`), no end file, a new linker script and an option to use the C Library and the System Calls Library.

[2]CrossGCC FAQ: `http://www.objsw.com/CrossGCC`

```
void mips1_syscall::get_buffer(int argn,
                               unsigned char* buf,
                               unsigned int size)
{
  unsigned int addr = RB.read(4+argn);

  for (unsigned int i = 0; i<size; i++, addr++) {
    buf[i] = MEM.read_byte(addr);
  }
}

int mips1_syscall::get_int(int argn)
{
   return RB.read(4+argn);
}

void mips1_syscall::set_int(int argn, int val)
{
   RB.write(2+argn, val);
}

void mips1_syscall::return_from_syscall()
{
  ac_pc = RB.read(31);
}
```

Figure 3: Interface functions for the MIPS Architecture

Create a new start file. This is an architecture dependent initiation routine in assembly format. Stack register and some other registers may be initialized here. This is the entry point for the simulation. It is usually only necessary to make a few changes in the original start file for the architecture used to run into the simulator.

Create a new linker script file. The linker script file informs the linker how to merge the object files. In this file we specify the entry point (we chose address 0 for simplicity) and indicate that the start file is followed by the block of addresses reserved to the system functions.

3.5 Simulator Requirements

A few changes are also required to be performed to the simulator so it can understand that addresses inside the reserved block are to be treated differently. Changes are dependent upon the simulation engine. We highlight the required changes in the next paragraphs:

Pass command line options. Before the start of the simulation, a special interface routine is called to initialize arguments that will be passed to the `main()` function of the target program. These arguments come from the simulator command line options after removing the simulator specific options.

Catch system calls. When the simulated Program Counter (PC) reaches a reserved

```
*link:
-L/l/archc/compilers/ac_specs/mips1 -Tac_link.ld

*startfile:
ac_start.o

*endfile:


*lib:
-lc -lac_sysc
```

Figure 4: New spec file for GCC

address, instead of passing through normal fetching, decoding and execution, the simulator calls a wrapper function corresponding to the respective address. This wrapper will do the following steps:

- Get the necessary arguments, making any conversion necessary from target to host (e.g. endianness);
- Execute the system call in the host;
- Return the result to the target, making any conversion necessary from host to target;
- Return to the function that called the system routine in the target;

We also tested this approach using the compiled simulator generated from the ArchC description for both MIPS I and SPARC V8. In this case, it is even simpler to call the wrapper inside the simulator since the fetch and decode stages are pre-executed when ArchC generates the compiled simulator. This is done by placing the wrapper function calls into the correct positions inside the generated C++ program.

Return exit code to the system. After the end of simulation, the simulator needs to call the special interface routine `get_exitcode()` and exit the simulator with the error code provided by the target program.

The interface functions shown in this paper as example are for instruction-level simulators. For cycle-accurate simulators, to use the same functions without modifications, the simulator must be instructed to block new instructions from entering in the pipeline (if there is one present) and execute the system call only when the pipeline is empty, so we guarantee that the registers contains valid values and no bypassing is necessary.

3.6 The Retargetable System Call Library

We have grouped the system call routines in a retargetable library written in the C language, so only a recompilation is needed to port it to any other target. Routines that need host

Group	Function	Host interaction
I/O	open	√
	creat	√
	close	√
	read	√
	write	√
	isatty	√
	lseek	√
	fstat	√
Control	_exit	√
	chmod	
	chown	
	stat	
	getpid	
	kill	
	unlink	
Time	time	
	times	
	gettimeofday	
Memory	sbrk	√

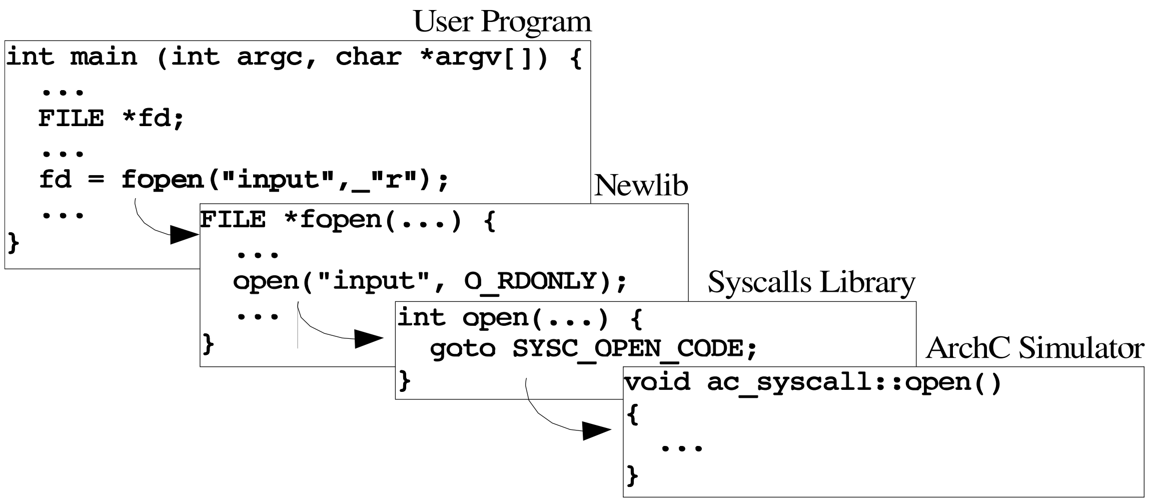
Table 1: Supported system calls

interaction and thus have special addresses are implemented in C by a `goto` statement to this special address. This library needed to be linked to the benchmark programs.

Table 1 shows all system calls that we have implemented inside the ArchC compiler. Routines that interact with the host are checked in the Host interaction column.

Not all routines need host interaction. That is due to incompatibilities between the host and target OS. For example, a few programs in our benchmarks used time functions. These functions contain structures that are incompatible between different architectures, due to alignment problems in the structure fields (alignment for char and short types). We solved this problem for now, by returning a zeroed structure, so that programs run correctly, but cannot count time. Another possible approach is to copy the structure field-by-field correcting them as necessary. But this approach needs more user intervention than the current one.

Figure 5 shows the big picture of our approach, summarizing how the control flow is transfered from the user program running inside the ArchC simulator to the native host OS routine. First, the user program calls original Newlib's `fopen` function, as usual. Next, `fopen` calls the `open` function provided by our Syscall Library which was linked to the user program. After that, the `open` function calls the ArchC Simulator wrapper by jumping into the `open` reserved address. Finally, the simulator wrapper uses the appropriate interface functions to create the link between target and host architectures and calls the host OS `open` function which take the appropriate action.



```
int main (int argc, char *argv[]) {
  ...
  FILE *fd;
  ...
  fd = fopen("input",_"r");
  ...
}
```

```
FILE *fopen(...) {
  ...
  open("input", O_RDONLY);
  ...
}
```

```
int open(...) {
  goto SYSC_OPEN_CODE;
}
```

```
void ac_syscall::open()
{
  ...
}
```

Figure 5: Sequence for calling the host OS

4 Results

We implemented our technique into the ArchC Architecture Description Language toolkit. Retargetable simulators, with support to operating system calls, have been synthesized from ArchC descriptions for the MIPS I architecture, both at the instruction-level and cycle-accurate level, and SPARC V8 at instruction-level.[3] There was no need for modifications in the application binary for the MIPS I instruction-level and cycle-accurate models. We expect no differences for the SPARC V8 at both levels as well.

For the sake of curiosity, we instrumented the MIPS I and SPARC V8 simulators, at the instruction-level, to count each operating system call that interacts with the simulator. The results of this experiment for the MIPS I and SPARC V8 compiled simulator (created automatically from the ArchC model), are shown in Tables 2 and 3, respectively. We performed these experiments on a Pentium 4 2.4GHz running Linux. In this environment we can simulate 13 millions of instructions per second on average.

The first two columns of Tables 2 and 3 show some of the programs executed and the benchmark they are from. We experimented with two benchmarks: Mediabench [9] and Mibench [4]. Programs in the Mibench benchmark are divided into two versions which process different amount of data (*small* and *large*). In the tables, we use underscores to distinguish between *small* and *large* versions, and parenthesis to highlight when the source code is the same but the input files are different. Some programs, like `pegwit`, require command line parameters that we also put between parenthesis (we ran pegwit twice, one with *encrypt* option and another with *decrypt*).

The next columns of Tables 2 and 3 show the number of times each system routine

[3]The cycle-accurate level model for this processor is still on progress.

Benchmark	Program	Size (instructions)	open	close	read	write	lseek	fstat	sbrk	Executed instructions
Mediabench	adpcm rawcaudio	7,008			149	149				7,491,406
	adpcm rawdaudio	7,001			149	149				5,902,037
	jpeg cjpeg	33,896	2	2	100	6		2	12	16,776,931
	jpeg djpeg	36,003	2	2	7	100		2	6	5,902,037
	mpeg2 mpeg2encode	32,685	15	15	503	161		15	24	11,699,579,115
	mpeg2 mpeg2decode	22,483	13	13	20	144	2		12	3,858,000,780
	pegwit (encrypt)	21,482	3		186	90	2	4	5	31,027,254
	pegwit (decrypt)	21,482	2		107	90	1	2	4	17,497,043
Mibench	basicmath_small	13,839				19,733		1	2	1,386,360,829
	basicmath_large	14,057				492,999		1	2	22,469,864,764
	bitcount (small)	9,551				12		1	2	45,593,411
	bitcount (large)	9,551				12		1	2	684,250,119
	qsort_small	12,330	1		54	10,003		2	3	14,359,301
	qsort_large	14,435	1		1,537	50,003		2	2	991,774,387
	susan (smooth. small)	18,417	2	2	8	8		2	5	35,318,139
	susan (smooth. large)	18,417	2	2	109	109		2	5	423,335,580
	dijkstra_small	12,190	1		29	222		2	4	59,276,836
	dijkstra_large	12,190	1		29	1,177		2	4	284,841,999
	crc (small)	9,165	1	1	1,338	1		2	2	31,642,843
	crc (large)	9,165	1		25,989	1		2	2	588,282,883
	fft (small)	11,896				116		1	5	760,568,610
	fft (large)	11,896				952		1	5	15,646,419,915
	sha (small)	7,556	1	1	306	1		2	2	13,036,286
	sha (large)	7,556	1	1	3,173	1		2	2	137,088,164
	string search_small	8,528				57		1	2	275,212
	string search_large	8,524				1,332		1	2	6,844,944

Table 2: System calls counts for MIPS I

was called in the execution of the programs (the empty cells indicate that a function was not called by that particular program). The MediaBench `mpeg2encode` program opens 15 files, from which 12 are different input files used to produce its results. In a similar way, `mpeg2decode` program opens 13 files, from which 12 are output. The Mibench `bitcount` program also called function `times` 14 times, although this function is empty, the final result was correct. The majority of the operating system functions are read/write to input/output files.

The last column of Tables 2 and 3 presents the total number of instructions executed for the respective MIPS or SPARC binary program. We observe in these tables that GCC produces better code for SPARC V8 than for MIPS I, both in code size and instructions executed. The only differences in system calls counts showed in Tables 2 and 3 is restricted to the `sbrk` function. This function is for memory management, so GCC deals differently with memory for MIPS I and SPARC V8.

We believe that the huge amount of instructions executed in all benchmarks, producing correct results, strongly supports our operating system emulation library, the changes we made into the simulator and the ArchC models that we designed for both the MIPS I and SPARC V8 architectures.

Table 4 shows the size of the input and output files for each of the used programs (cells

Benchmark	Program	Size (instructions)	open	close	read	write	lseek	fstat	sbrk	Executed instructions
Mediabench	adpcm rawcaudio	6,911			149	149				7,256,632
	adpcm rawdaudio	6,906			149	149				6,261,168
	jpeg cjpeg	24,906	2	2	100	6		2	12	14,288,756
	jpeg djpeg	29,638	2	2	7	100		2	5	4,187,862
	mpeg2 mpeg2encode	29,438	15	15	503	161		15	24	10,834,241,285
	mpeg2 mpeg2decode	21,113	13	13	20	144	2		12	3,451,835,100
	pegwit (encrypt)	19,590	3		186	90	2	4	5	30,897,936
	pegwit (decrypt)	19,590	2		107	90	1	2	4	16,863,601
Mibench	basicmath_small	12,658				19,733		1	2	1,305,099,573
	basicmath_large	12,825				492,999		1	2	21,365,365,697
	bitcount (small)	9,159				12		1	2	49,862,749
	bitcount (large)	9,159				12		1	2	748,368,498
	qsort_small	11,782	1		54	10,003		2	3	14,137,667
	qsort_large	13,549	1		1,537	50,003		2	3	792,301,298
	susan (smooth. small)	16,937	2	2	8	8		2	4	30,084,780
	susan (smooth. large)	16,937	2	2	109	109		2	4	349,096,325
	dijkstra_small	11,627	1		29	222		2	5	50,927,674
	dijkstra_large	11,627	1		29	1,177		2	5	244,328,853
	crc (small)	7,438	1	1	1,338	1		2	2	30,235,435
	crc (large)	7,438	1		25,989	1		2	2	587,711,536
	fft (small)	11,036				116		1	5	715,774,279
	fft (large)	11,036				952		1	5	14,400,682,487
	sha (small)	8,710	1	1	306	1		2	3	13,254,951
	sha (large)	8,710	1	1	3,173	1		2	3	137,975,708
	string search_small	7,139				57		1	2	269,813
	string search_large	7,154				1,332		1	2	6,689,371

Table 3: System calls counts for SPARC V8

marked with a dash in the table represent programs which do not have input, only command line parameters are required). Putting all inputs into the source code files as vectors is a practical approach only when input files are very small. As it can be seen from the table, this is not the case for most of the programs we used. The most impressive case is for the `crc` program. Its binary has only 9,165 instructions and its *large* input file has 26,611,200 bytes (725 times bigger). Cases like this are reproduced, in a small proportion, in the other benchmarks.

We verified the correct execution of the benchmarks by comparing the output files produced by the MIPS simulator with the files produced by an i386 native execution. Despite a few endianness problems in some comparisons, all the produced results were correct.

5 Conclusions and Future Work

In this paper we highlight the need for operating system support in SoC simulation, as it facilitates the use of existing benchmarks (just recompiling without changing source code) and improves testing (I/O operations behave like in a real execution). We also describe a general framework to include operating system functionality into any architecture simulator,

Benchmark	Programs	Input (bytes)	Output (bytes)
MediaBench	adpcm rawcaudio	295,040	73,760
	adpcm rawdaudio	73,760	295,040
	jpeg cjpeg	101,484	5,645
	jpeg djpeg	5,756	101,484
	mpeg2 mpeg2encode	506,880	76,320
	mpeg2 mpeg2decode	34,906	506,880
	pegwit (encrypt)	91,503	91,537
	pegwit (decrypt)	91,537	91,503
Mibench	basicmath_small	–	426,600
	basicmath_large	–	16,465,695
	bitcount (small)	–	625
	bitcount (large)	–	634
	qsort_small	53,437	53,463
	qsort_large	1,572,431	1,572,490
	susan (smooth. small)	7,292	7,233
	susan (smooth. large)	110,666	110,607
	dijkstra_small	29,144	1,342
	dijkstra_large	29,144	6,931
	crc (small)	1,368,864	42
	crc (large)	26,611,200	42
	fft (small)	–	116,210
	fft (large)	–	970,409
	sha (small)	311,824	45
	sha (large)	3,247,552	45
	string search_small	–	3,197
	string search_large	–	92,672

Table 4: Input and output file size for the benchmark programs

no matter if it has been synthesized by some Architectural Description Languages or hand-coded.

The operating system functionality can also be used to check the processor models described using ADLs so that a huge amount of input can be used to test every instruction. The method proposed here can be used to include even bigger benchmarks into ADL models development road-map.

Although the approach we presented here is simple, it handles correctly all the functions it proposes to do guaranteeing correct results from the benchmark programs. This abstraction level implementation also turns it very easy to port from one processor to another, requiring just the specialization of a few methods of the `ac_syscall` class. To the best of our knowledge there is not any other ADL which implement such feature.

The way we emulate operating system calls can also be used to simulate hardware or software modules that are not available at the moment by creating wrapper functions to them and placing these functions together with the OS wrappers.

Routines that deal with multi-tasking, like `fork()`, have not been implemented yet. Multi-tasking support lays on the road-map of the ArchC project. As it is not currently available in our library, a compilation error is flag if the application uses one of those instructions.

References

[1] A. Fault, J. Van Praet e M. Freericks. Describing Instruction Set Processors using nML. In *in Proc. European Design and Test Conf., Paris*, pages 503–507, March 1995.

[2] D. Burger and T. M. Austin. The Simplescalar Tool Set Version 2.0. Technical Report 1342, Computer Sciences Department, University of Wisconsin, 1997.

[3] Robert F. Cmelik and David Keppel. Shade: A fast instruction-set simulator for execution profiling. Technical report, University of Washington, 1993.

[4] Matthew R. Guthaus, Jeffrey S. Ringenberg, Dan Ernst, Todd M. Austin, Trevor Mudge, and Richard B. Brown. Mibench: A free, commercially representative embedded benchmark suite. In *IEEE 4th Annual Workshop on Workload Characterization*, Austin, TX, December 2001.

[5] George Hadjiyiannis, Silvina Hanono, and Srinivas Devadas. ISDL: An instruction set description language for retargetability. In *Design Automation Conference*, pages 299–302, 1997.

[6] A. Halambi, P.Grun, V.Ganesh, A.Khare, N.Dutt, and A.Nicolau. EXPRESSION: A language for architecture exploration through compiler/simulator retargetability. In *in Proc. European Conference on Design, Automation and Test(DATE)*, March 1999.

[7] Ashok Halambi, Peter Grun, Hiroyuki Tomiyama, Nikil Dutt, and Alex Nicolau. Automatic Software Toolkit Generation for Embedded Systems-On-Chip. In *Procedings of the IEEE International Conference on VLSI and CAD*, 1999.

[8] Gerry Kane and Joe Heinrich. *MIPS RISC Architecture*. Prentice Hall, New Jersey, 1992.

[9] Chunho Lee, Miodrag Potkonjak, and Willian H. Mangione-Smith. Mediabench: A tool fo evaluating and synthesizing multimedia and communications systems. In *Proc. of 30th Annual International Symposium on Microarchitecture*, December 1997.

[10] Richard P. Paul. *SPARC Architecture, Assembly Language Programing, and C*. Prentice Hall, 2000.

[11] Sandro Rigo, Rodolfo J. Azevedo, and Guido Araujo. The ArchC architecture description language. Technical Report IC-03-15, Institute of Computing, University of Campinas, June 2003.

[12] Eric Schnarr, Mark Hill, and James Larus. Facile: A Language and Compiler For High-Performance Processor Simulators. In *ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation (PLDI01)*, Snowbird, Utah, June 2001.

[13] Eric Schnarr and James Larus. Fast Out-Of-Order Processor Simulation Using Memoization. In *Eighth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VIII)*, San Jose, California, October 1998.

[14] Vojin Zivojnovic e Heinrich Meyr Stefan Pees, Andreas Hoffmann. LISA - Machine Description Language for Cycle-Accurate Models of Programmable DSP Architectures. In *in Proc. of ACM Design Automation Conference(DAC)., New Orleans*, 1999.

[15] Inc. The Santa Cruz Operation, editor. *System V Application Binary Interface – MIPS Processor Supplement.* The Santa Cruz Operation, Inc., 1996.

[16] Inc. The Santa Cruz Operation, editor. *System V Application Binary Interface – SPARC Processor Supplement.* The Santa Cruz Operation, Inc., 1996.