HySim: A Fast Simulation Framework for Embedded Software Development

Stefan Kraemer, Lei Gao, Jan Weinstock, Rainer Leupers,
Gerd Ascheid and Heinrich Meyr
Institute for Integrated Signal Processing Systems
RWTH Aachen University, Germany
{kraemer | gao | leupers}@iss.rwth-aachen.de

ABSTRACT

Instruction Set Simulation (ISS) is widely used in system evaluation and software development for embedded processors. Despite the significant advancements in the ISS technology, it still suffers from low simulation speed compared to real hardware. Especially for embedded software developers simulation speed close to real time is important in order to efficiently develop complex software. In this paper a novel, retargetable, hybrid simulation framework (*HySim*) is presented which allows switching between native code execution and ISS-based simulation. To reach a certain state of an application as fast as possible, all platform-independent parts of the application are directly executed on the host, while the platform dependent code executes on the ISS. During the native code execution a performance estimation is conducted. A case study shows that speed-ups ranging from 7× to 72× can be achieved without compromising debugging accuracy. The performance estimation during native code execution shows an average error of 9.5%.

Categories and Subject Descriptors

I.6.7 [**Simulation and Modeling**]: Simulation Support Systems—*environments*

General Terms

Design

Keywords

HySim, Simulation, Hybrid Simulation, ISS

1. INTRODUCTION

Over the last decade the embedded system domain has been growing exponentially due to the continually increasing demands for multimedia and mobile applications. The diversity of these applications and the highly competitive market are forcing system designers to use more and more software in their design solutions. At the same time the shrinking time-to-market has made it compulsory that such software development can proceed even before a basic hardware prototype is ready. As a consequence, software developers are increasingly relying on Instruction Set Simulators (ISSs) to develop and test their applications.

CODES+ISSS'07, September 30–October 3, 2007, Salzburg, Austria.

Lately, these requirements for fast software development and debugging have triggered considerable amount of research activities to improve the simulation performance of ISSs. Different techniques have been introduced [10, 11, 13] to attain high simulation speed. Despite these improvements the simulation speed is still very low compared to real hardware. Higher simulation rates can be achieved by switching to a more abstract processor representation at the cost of reduced accuracy. Hence, the software developer can trade-off simulation performance for accuracy depending on his requirements. For example, debugging a complex application requires to simulate the application to the function where the problem is expected, and then the developer will examine the behavior of this code carefully. For the first phase the simulation speed is important whereas in the second phase the emphasis is on the simulation accuracy.

To address these issues we propose a retargetable, hybrid simulation framework, *HySim*, that offers the possibility to switch between a fast-forward mode and an ISS based simulation mode. The objectives of the proposed solution are:

Performance — High simulation speed is achieved by native code execution on the host machine. All applications with platform independent code can benefit from the native execution and can be simulated in the fast-forward mode.

Performance estimation — During the fast-forwarding a performance estimation is applied to report the approximate performance of the application to the user without the need for a detailed simulation.

Compatibility — All types of programs should be supported by the framework without modifying the binary and independent of the presence of debug information. Additionally, assembly code as well as third party libraries should be supported to make the framework adaptable to a wide range of applications.

Transparency — The framework placed on top of an existing ISS should be transparent to the ISS as well as to the other components of the simulated system, e.g. peripheral devices. This is important especially in the context of system simulation, where the processor simulator is only one building block of the entire system.

Retargetability — The HySim framework is independent of the underlying ISS and can be employed with only minor changes to a wide range of ISSs.

The HySim framework is based on the virtual coprocessor technique [4] for native execution of selected parts of the application. HySim offers an interface for extensibility. For instance, the performance estimation tool makes use of it. The remainder of this paper is structured as follows: First,

the proposed framework is compared with other available simulation technologies. Then an overview of the HySim framework is given. Afterwards, the main components of the simulator are described in detail. After the presentation of the framework, initial benchmark results are shown and discussed. The paper ends with a conclusion about the presented work.

2. RELATED WORK

A lot of different techniques have been described in the literature to alleviate the simulation time of complex software. A good overview of the available simulation techniques can be found in [19].

Sampling based simulation technologies do not simulate the complete application but only a small fraction of it and extrapolate the obtained result to the entire program.

SMARTS [18] is a sampling micro architecture simulation technology which utilizes a fast-forwarding and a warm-up phase to reach the detailed simulation. During the fast-forwarding phase, only the resources for the programmer's view are simulated to increase the speed. The time consuming parts like caches and branch predictors are simulated during the detailed simulation to derive the performance of the micro architecture.

Instead of using periodic sampling Sherwood *et al.* [15, 16] employ representative sampling. A combination of basic block distribution analysis and machine learning techniques are applied to cluster the application into a set of phases. One representative of each phase is simulated in detail to estimate the overall application performance. Sampling based simulation is a very interesting concept which allows fast and accurate performance estimation of an application. However, it is not suitable for software development and debugging since the user cannot control which part of the application is simulated in detail.

The concept of checkpointing allows storing the state of an ISS and restoring it at a later point of time, thereby giving the software developers the possibility to reach interesting points they want to debug in the application without having to re-simulate the entire application each time. However, if the system or the program is modified, the checkpoints may become outdated so a new simulation of the entire application is required.

The SimSnap [17] framework employs a combination of *Application Level Checkpointing* (ALC) and native execution to fast-forward the simulation. The application's source code is instrumented by the Cornell Checkpoint Compiler C^3 [2, 3] in order to save the state of the program at a given point. In the fast-forwarding mode the compiled application is natively executed on the host machine and the checkpointing capability is used to transfer the application's state into the simulator at a switching point. Currently, the proposed approach works only if the instruction set architecture (ISA) of the simulator matches the ISA of the host machine.

Ringenberg *et al.* [14] suggest to use *Intrinsic Checkpointing* to store the state of a simulation. In contrast to conventional ALC, Intrinsic Checkpointing modifies the application binary itself. Hence, this approach is independent of the availability of the application's source code.

Besides sampling techniques and checkpointing, abstract processor representations can also be used to increase the simulation performance. Both the concept of virtual architecture [6, 20] and virtual processing unit (VPU) [8] allow to simulate the processor at a very high abstraction level by providing an API for OS calls and native code execution. This approach is very suitable for early design space exploration without the need for detailed simulation.

For traditional ISS based simulation Qin *et al.* [12] report a significant performance increase by combining interpretive simulation with compiled simulation.

3. BASIC CONCEPTS OF HYSIM

This section describes the concept of our retargetable, hybrid simulation approach, which defines a hybrid processor architecture and utilizes a virtual coprocessor to accelerate the instruction set simulation.

3.1 Framework Architecture

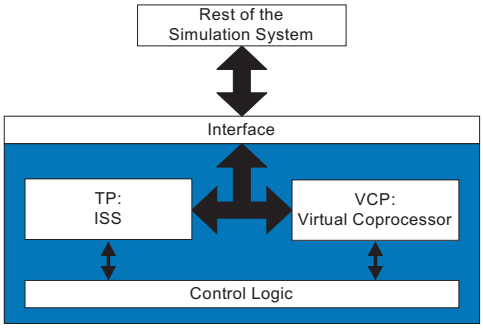

Figure 1: Architecture of the simulation system

The main idea of the HySim framework is to execute parts of the application directly on the host machine to speed up the overall simulation. From the perspective of the ISS the native code execution can be seen as a coprocessor which is transparent to all the other components of the simulation system. Since this virtual coprocessor is transparent, this concept can be applied to accelerate the ISS without modifying the rest of the simulation. Figure 1 shows the simulation system using a Virtual Coprocessor (VCP). Correspondingly, the original ISS is called Target Processor (TP). To ensure that TP and VCP can invoke each other without having to modify the TP, an external control logic is introduced. As the TP and the VCP execute in a mutually exclusive way, they can exchange internal data directly. Since the VCP is transparent to the other components of the simulation system, such as buses and peripherals, no changes are required for those components.

3.2 Software Workflow

Note that HySim is a retargetable simulator framework, where different ISSs can be plugged in. Hence, this framework can be employed to enable a fast-forwarding mode for various simulation systems. It acts as a wrapper around an existing ISS to facilitate the software design for a given target processor. The user is provided with new functionality to simulate the program up to a specific point at high speed.

Figure 2 shows the overall workflow of our proposed framework, where two clearly separated branches, starting from the application source code, can be seen. The left branch represents the standard compilation flow for the target ISS. The right branch shows the binary generation for the virtual coprocessor. The C source code is automatically instrumented before simulation and compiled for the host machine. Additionally, the instrumenter generates a global control flow graph (GCFG), which is necessary for partitioning the application to the VCP in order to obtain maximum speed-up. From the user perspective, the framework behaves as a normal debugging environment for program debugging and simulation. Moreover, it also provides the user with the capability to set a "fast-forwarding break point". This breakpoint partitions the application into a TP-executing-part and VCP-executing-part in such a way that the breakpoint is reached as fast as possible. In order to find the optimal partitioning of the application the GCFG is used.

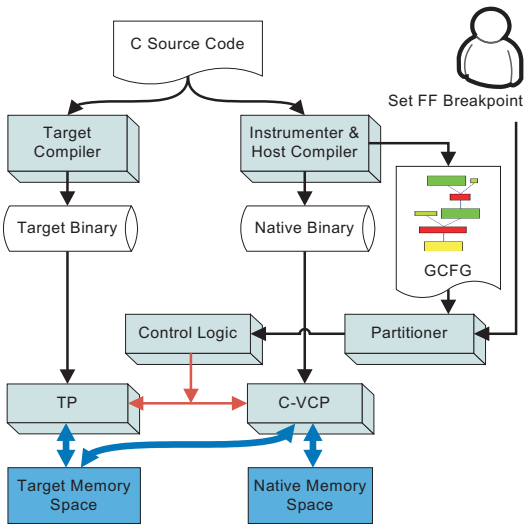

Figure 2: HySim concept

3.3 Virtual Coprocessor

The concept of the VCP allows switching between native execution and ISS based execution. This mechanism is used by the framework to improve the simulation performance, since native execution is faster than instruction set simulation. As a set of different debug formats, such as DWARF and STABS, or even no debug format at all (for processors under development), are employed in different ISSs, a more retargetable way of obtaining the information required for switching is necessary. By restricting the switching to function borders it is possible to obtain all information for the transition independent of the debug information. For a function call only the global variables, the function parameters and the return value are required. The code instrumentation ensures that the VCP code can access global variables and interpret pointers in a correct manner by accessing the target memory space as shown in Fig. 2. Since ISS and VCP are exclusively activated by the control logic, both of them can access the target memory space without the need for synchronization.

4. FRAMEWORK DETAILS

In this section the control logic for switching between both modes is discussed. Furthermore, the required code instrumentation to obtain the information needed by the control logic is explained.

4.1 Control Logic for Bidirectional Invocation

The control logic takes care of the switching between TP and VCP. The invocation of a VCP function from TP is called *forward invocation* and *inverse invocation* for the opposite direction. While loading the native binary, a link between the target function and the corresponding instrumented function is created. Based on the knowledge of the calling conventions of the target and the host compiler it is possible to build a bridge and translate function calls between both sides. The control logic is implemented using the concept of stubbing. A stub is used to translate the function parameters in case a TP function is accessed from VCP and an inverse stub performs the translation for the case of a function call to VCP. The code of the stub is directly inserted into the instrumented application. Since the target binary cannot be modified for compatibility reasons, the inverse stub is triggered by monitoring the ISS.

4.2 Instrumentation

The instrumenter is a crucial part of the HySim framework. It analyzes the application source code and preprocesses the source code for the native execution by instrumenting it. Furthermore, instrumentation can be employed to estimate the program behavior during the VCP execution. During the analysis phase, the instrumenter calculates the cost per *Basic Block* (BB) and stores the information in a separate file. A cost library is used to assign a certain weight to each operation in the code. Depending on the employed ISS, the weight represents the number of instructions or the number of cycles per operation. While executing parts of the application on the VCP a *Basic Block Vector* (BBV) is generated, which is then combined with the cost information per basic block to obtain the overall cost. Kempf *et al.* [8] use a similar approach and report that this kind of estimation can be used to quickly estimate the performance of an application. Figure 3 shows the flow of the performance estimation used in the HySim framework.

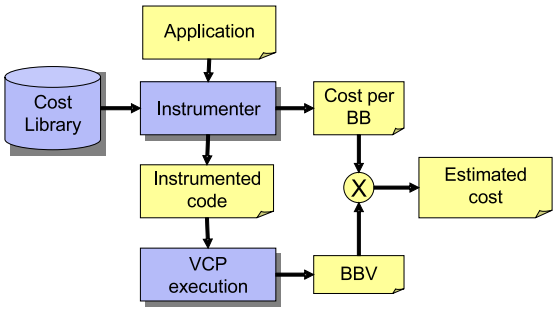

Figure 3: Performance estimation flow for the VCP

Both simulation modes maintain separate address spaces, the *TP address space* and the *VCP address space* (see Fig. 2). However, it is important that in the fast-forward mode the application can access the TP address space. Therefore, all memory accesses are automatically instrumented to ensure that they access the correct address space during execution. Global variable access and pointer access are the most common situations where the TP address space is accessed. In the following the different cases which require instrumentation are discussed:

Global variables — Global variables and local static variables [9] can be accessed by names or through pointers in C. For each of these variables a unique global pointer is declared and assigned to the address of the corresponding variable it references at TP address space. Creating a link for the global variables between both address spaces makes sure that a value change in one address space will also affect the other address space. The instrumenter replaces the access to these variables by a call to a read/write helper function, which is capable of accessing TP address space. Figure 4 gives an example of reading a global variable. When function `foo` is mapped to VCP and `bar` (1) is a TP function, `foo` will not know whether `bar` will change the value of the global variable or not. So, the access of the global variable has to be performed through a pointer to the TP address space (2), (3).

Accessing the TP address space is slower than accessing a native global variable. Therefore, the instrumentation is only performed if required. For global or static variables declared with the `const` keyword, an additional copy of the variable is created at VCP address space. The instrumenter analyzes the source code to find accesses to constant global and static variable and replaces them with the local copy.

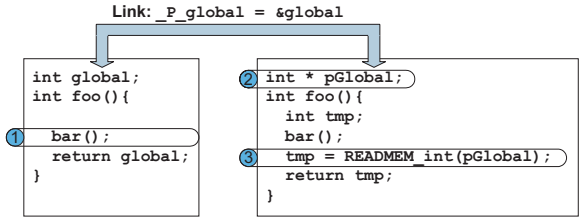

Figure 4: Global Variable Access in the VCP

Local variables — Local variables only exist in the VCP's local stack, so accessing them directly is safe. However, if they are accessed through pointers there can be pointer hazards since the pointer can point either to a global or to a local variable. This problem is solved by copying local arrays to the TP address space, thus, the pointer only points to elements residing in the TP address space.

Aggregated data structures — The layouts of aggregated data structures like `struct` or `union` in the C language are not defined in the C specification. Different compilers may produce different layouts. Currently, this problem is avoided since the target compilers employed for the case study have an identical data layout as the host.

Floating point — Floating point computation is complicated to handle by the instrumenter because it is highly platform dependent, e.g. the rounding strategy is not defined in the IEEE 754 [5] standard. Hence, by switching between both simulation modes the precision of the results might be altered. In case that exactly the same precision is required in both simulation modes, all floating point operations must be instrumented and the required precision is then emulated on the host. However, this reduces the expected speed-up of the native code execution and should only be applied if it is absolutely required.

C Standard Library — There are two possibilities to handle calls to the C standard library: map the calls to the corresponding calls on the host machine or switch back to the TP to execute the call. The latter approach is more complicated to realize, however, it is the more general solution. Mapping the function calls to the host machine might work for a set of function calls with no side effects, e.g. `fabs`. But for function calls like `malloc` this approach is not feasible, since they have side effects such as reducing the available memory on the heap. In the HySim framework all calls to the C standard library are executed on the TP.

4.3 Application Partitioning

The HySim framework offers the user two possibilities of partitioning: manual partitioning and automatic partitioning. In the case of manual partitioning the user can partition certain functions of the application to the VCP. However, no sanity checking is performed if the partitioning is feasible or not. In case of the automatic partitioning the user only specifies a special breakpoint called fast-forward breakpoint (FF breakpoint) and the application is then automatically partitioned to reach this point as fast as possible. Executing the complete application on the VCP would be the fastest solution. However, it is not feasible to move all types of functions to the VCP without compromising the switching capability. For instance all functions contributing to the actual call stack must remain on the TP otherwise it is not possible to continue the execution after returning from the VCP.

Global Control Flow Graph — The Global Control Flow Graph $GCFG = (N, E)$ is a *Directed Graph* where each node $n \in N$ represents a basic block (BB), each edge

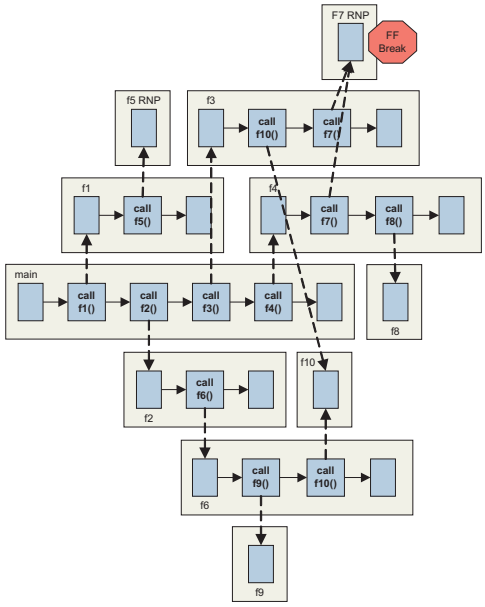

Figure 5: Example of a GCFG with a FF breakpoint

$(n_i, n_j) \in E$ depicts a control flow transition from BB n_i to BB n_j. The GCFG is essential for program partitioning since it contains all information about the function execution sequence. The GCFG can be seen as an extension to the traditional Control Flow Graph (CFG) [1] which represents the control flow between basic blocks at function level. The GCFG is obtained by merging all local CFGs together, so that:

$$V = \bigcup_{i=1}^{K} N_{CFG_i} \qquad E = \left[\bigcup_{i=1}^{K} E_{CFG_i} \right] \cup E_{call}$$

where E_{call} denotes the set of edges from the function call to the entry node of the callee and K is the number of functions in the application. In Fig. 5 the GCFG generation is exemplified using a simplified version of the GCFG. The GCFG comprises eleven CFGs, one for each function, and starts with the `main` function. Inside the `main` function four function calls can be seen, each function call points to the CFG of the corresponding function. Hence, the GCFG provides information about the calling sequence of functions and about the call stack for a given function.

In the GCFG each function can be marked with two different flags: *Non-Partitionable* (NP) and *Recursive-Non-Partitionable* (RNP). Both flags indicate to the partitioner that all functions with these flags need to be executed on TP. The difference between NP and RNP is that in case of RNP not only the marked function is executed on TP but all functions contributing to the actual call stack. In the following, the different cases which require special treatment during GCFG generation are discussed:

- **Static functions** — Static functions are not visible at linking phase. Hence, there may be a lot of `static` functions with the same name inside a program. To resolve ambiguity the merging is performed in two steps: First, merge functions inside a compilation unit, and then merge the compilation unit level CFGs into the GCFG.

- **Third party libraries** — Third party libraries normally only expose the definition of their functions. A

function cannot be executed in VCP without the availability of the complete C source code, so a pseudo body is added for each incomplete function and annotated as NP.

- **Function pointer** — The C language allows function calls through pointers for dynamic behavior. Most function pointers are irresolvable at compile time. All the functions containing function calls by pointer are annotated as RNP.

- **Direct stack manipulation** — The C language also supports a set of functions like `setjmp` and `longjmp`. They modify the stack directly and change the control flow. Such kinds of functions are marked as RNP.

GCFG Partitioning — The partitioning of the GCFG is steered by the user through defining a fast-forwarding breakpoint. A FF breakpoint can be set in any function independently of the availability of the C source code. The function with the FF breakpoint is called *FF function*. The goal of the partitioner is to find a good partitioning, considering the fact that not all functions can be executed on the VCP. Figure 6 shows the partitioning algorithm in pseudo code.

```
begin
01  //Step 1
02  for (all Pred(FF function)) do
03  if node = function call then
04     add to set_calls
05  if node = function start then
06     add to set_executed;
07  end for
08  set_candidates = set_calls − set_executed
09  //Step 2
10  for (all Pred(RNP functions)) do
11  if node = function header then
12     add to RNP_executed;
13  end for
14  //Step 3
15  switch_func = set_candidates − RNP_executed
end
```

Figure 6: Partitioning Algorithm

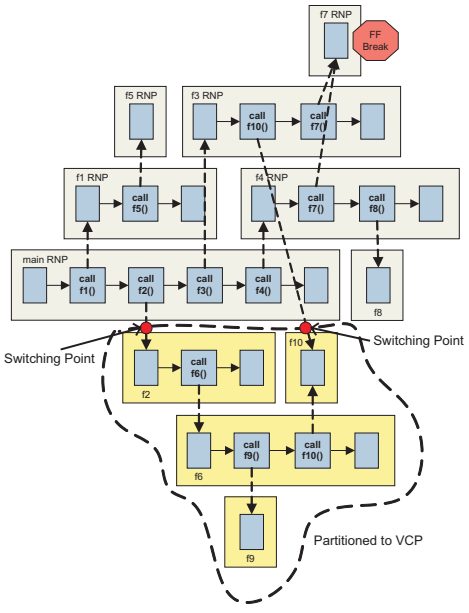

Figure 7: Example of a partitioned GCFG.

In a first step, all functions that are *alive* as well as all functions possibly executed before the *FF function* are stored. An instance of a function is considered *alive* if the execution has started but not yet ended. By subtracting the set of alive functions from the set of functions possibly executed before the *FF function*, the set of functions (*set_candidates*) is obtained which are not alive. In a second step all nodes marked as RNP are handled. For these nodes the functions which are alive are recorded (*RNP_executed*). In the last step the difference of the set *set_candidates* and the set *RNP_executed* is computed to obtain all functions that are possibly executed before the *FF function* and are not alive. The partitioning is exemplified in Fig. 5. Suppose, `f5` performs a function call by using a function pointer. Since a function pointer points potentially to all functions, this function is marked as RNP. The user sets a FF breakpoint within `f7`, so `f7` is marked as RNP to maintain the stack for later simulation on the TP. Starting at function `f7` the set *set_call* = {`f3`, `f4`, `f2`, `f1`, `f10`} and the set *set_executed* = {`f3`, `f4`} is calculated. By computing the difference between both sets the function that are not alive are identified *set_candidate* = {`f1`, `f2`, `f10`}. The functions `f5` and `f1` are executed in order to execute the function `f5` and thus cannot be partitioned to the VCP. Thus, only at the border of the functions `f2` and `f10` it is possible to switch from the TP simulation to the VCP simulation. Hence, the functions `f2`, `f6`, `f9` and `f10` can be executed in VCP. Figure 7 shows the obtained partition.

5. RESULTS

In this section the initial results obtained for a set of benchmarks are discussed. The benchmarking is performed on a PC with an Athlon64 X2 4600+ processor, 4 GB of memory and Linux Fedora Core 4 as operating system. An instruction accurate MIPS32 ISS is employed together with the HySim framework. For the MIPS32 ISS the *sde-gcc 2.96* is used as cross compiler. On the host side, the instrumented source code for the VCP is compiled using *gcc 4.0.2*.

As benchmark a set of four different embedded applications was selected, covering different application domains: encryption (*des*), message digest (*md5*), edge detection (*susan*) and image decoding (*jpeg*).

In a first experiment, the HySim framework is used to execute the application as fast as possible by partitioning the maximum of functions to the VCP. Clearly, this is not the typical use case of this framework but it allows estimating the maximum achievable acceleration. Table 5 shows the detailed numbers obtained from the different benchmarks for the MIPS32 ISS. For all applications a significant speed-up ranging from range from 7× to 72× with an average of 33.6× can be seen. The *jpeg* application shows the lowest speed-up of all applications because the frequent use of function pointers makes it hard to find a good partitioning. Therefore, the fraction of code executed on the VCP is much lower than in all the other benchmarks.

The overall speed of the HySim is dependent on three factors, the speed of the ISS, the VCP speed and the partitioning of the program. The ISS speed can be considered constant for a given ISS, but faster ISS will result in a faster overall simulation. The pure VCP speed is 155 MIPS on average compared to the approximately 4800 MIPS of the host machine, this slowdown is coming from the instrumentation overhead. The amount of instrumentation heavily depends on the application type. If global variables and pointers are rarely used, then the instrumentation overhead is reduced. For the *des* application the highest simulation rate has been measured with 185 MIPS on average. The reason for the high simulation performance is that *des* heavily accesses global but constant variables, which do not need to

	MIPS32 ISS			HySim					speed up	estim.
App.	instr.	time [s]	MIPS	time [s]	TP [s]	VCP [s]	avg. MIPS	VCP MIPS	factor	error
des	128 M	50.48	2.53	0.69	0.31	0.38	185	330.36	72×	2.45 %
jpeg	116 M	40.57	2.87	5.71	4.8	0.91	20.4	112.84	7.1×	17.62 %
md5	63 M	21	3.01	0.939	0.038	0.901	67.3	69.93	22.34×	14.61 %
susan	462 M	184	2.5	5.68	0.78	4.89	81.5	92.0	32.4×	3.27 %

Table 1: Benchmarks executed on MIPS ISS and HySim

be instrumented. This shows that it is possible to increase the performance of the HySim framework by carefully optimizing the instrumentation strategy.

The program partitioning mainly depends on two factors: the position of the FF breakpoint and how the program is written. As mentioned earlier, the usage of function pointers and stack manipulating functions, e.g. `longjmp`, lead to a suboptimal partitioning.

The error of the performance estimation during the native execution ranges from 2.45% to 17.62% with an average error of 9.5%. For this case study the number of instructions executed was used to estimate performance, since the ISS itself is also only instruction accurate. The average error of 9.5% clearly shows that it is possible to provide the user with some early performance estimation during the native execution. The performance estimation can be refined to estimate the number of cycles as shown in [7], with an average error of 11%. Hence, in conjunction with a cycle accurate ISS, the HySim framework can also be used to obtain reasonably accurate cycle estimations.

6. CONCLUSION

In this paper a retargetable, hybrid simulation framework is presented that gives the software developer the possibility to switch between native code execution and ISS based simulation. With the HySim framework it is possible to reach the desired state of an application quickly and then switch to a slow but detailed simulation. The partitioner takes care of moving as many functions as possible to the VCP for fastest possible execution. However, target dependent code like assembly will be executed on the ISS to ensure correct behavior. The fact that the used ISS is not aware of the VCP and does not need to be modified, makes the HySim framework a versatile instrument which can be employed for a wide range of ISSs. The obtained results show that this concept is capable of achieving high simulation speed up to 185 MIPS. In order to further speed up the native execution, more research is required to reduce the amount of instrumentation without losing the capability to switch between both simulation modes. Furthermore, techniques like *cache warming* should be incorporated into the framework to obtain precise simulation results for caches.

7. ACKNOWLEDGMENTS

This work is part of the European project SHAPES. For more information visit `www.shapes-p.org`.

8. REFERENCES

[1] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Addison Wesley, August 2006.

[2] G. Bronevetsky, D. Marques, K. Pingali, and P. Stodghill. Automated Application-Level Checkpointing of MPI Programs. In *PPoPP '03: Principles and Practice of Parallel Programming*, New York, NY, USA, 2003. ACM Press.

[3] G. Bronevetsky, D. Marques, K. Pingali, and P. Stodghill. Collective Operations in Application-Level Fault-Tolerant MPI. In *ICS '03: Proceedings of the 17th annual International Conference on Supercomputing*, New York, NY, USA, 2003. ACM Press.

[4] L. Gao, S. Kraemer, R. Leupers, G. Ascheid, and H. Meyr. A Fast and Generic Hybrid Simulation Approach Using C Virtual Machine. In *CASES '07: Compilers, Architecture and Synthesis for Embedded Systems*, New York, NY, USA, 2007. ACM Press.

[5] IEEE Standards Committee 754. *IEEE Standard for Binary Floating-Point Arithmetic, ANSI/IEEE Standard 754-1985*. Institute of Electrical and Electronics Engineers, New York, 1985. Reprinted in ACM SIGPLAN Notices, 22(2):9-25, 1987.

[6] A. A. Jerraya, A. Bouchhima, and F. Pétrot. Programming Models and HW-SW Interfaces Abstraction for Multi-Processor SoC. In *DAC '06: Conference on Design Automation*, New York, NY, USA, 2006. ACM Press.

[7] Karuri, K., Al Faruque, M.A., Kraemer, S., Leupers, R., Ascheid, G. and Meyr, H. Fine-grained Application Source Code Profiling for ASIP Design. In *42nd Design Automation Conference*, Anaheim, California, USA, June 2005.

[8] T. Kempf, K. Karuri, S. Wallentowitz, G. Ascheid, R. Leupers, and H. Meyr. A SW Performance Estimation Framework for Early System-Level-Design using Fine-Grained Instrumentation. In *DATE '06: Conference on Design, Automation and Test in Europe*, 3001 Leuven, Belgium, Belgium, 2006. European Design and Automation Association.

[9] B. W. Kernighan and D. Ritchie. *The C Programming Language (2nd Edition)*. Prentice Hall PTR, March 1988.

[10] A. Nohl, G. Braun, O. Schliebusch, R. Leupers, H. Meyr, and A. Hoffmann. A Universal Technique for Fast and Flexible Instruction-Set Architecture Simulation. In *DAC '02: Conference on Design automation*, New York, NY, USA, 2002. ACM Press.

[11] M. Poncino and J. Zhu. DynamoSim: A Trace-based Dynamically Compiled Instruction Set Simulator. In *ICCAD '04: Proceedings of the 2004 IEEE/ACM International conference on Computer-aided design*, Washington, DC, USA, 2004. IEEE Computer Society.

[12] W. Qin, J. D'Errico, and X. Zhu. A Multiprocessing Approach to Accelerate Retargetable and Portable Dynamic-compiled Instruction-set Simulation. In *CODES+ISSS '06: Conference on Hardware/Software Codesign and System Synthesis*, New York, NY, USA, 2006. ACM Press.

[13] M. Reshadi, P. Mishra, and N. Dutt. Instruction Set Compiled Simulation: A Technique for Fast and Flexible Instruction Set Simulation. In *DAC '03: Conference on Design Automation*, New York, NY, USA, 2003. ACM Press.

[14] J. Ringenberg, C. Pelosi, D. Oehmke, and T. Mudge. Intrinsic Checkpointing: A Methodology for Decreasing Simulation Time Through Binary Modification. *Performance Analysis of Systems and Software*, March 2005.

[15] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder. Automatically Characterizing Large Scale Program Behavior. In *ASPLOS-X: Proceedings of the 10th international conference on Architectural Support for Programming Languages and Operating Systems*, New York, NY, USA, 2002. ACM Press.

[16] T. Sherwood, E. Perelman, G. Hamerly, S. Sair, and B. Calder. Discovering and Exploiting Program Phases. *IEEE Micro*, December 2003.

[17] P. K. Szwed, D. Marques, R. M. Buels, S. A. McKee, and M. Schulz. SimSnap: Fast-Forwarding via Native Execution and Application-Level Checkpointing. *interact*, 00, 2004.

[18] R. Wunderlich, T. Wenisch, B. Falsafi, and J. Hoe. SMARTS: Accelerating Microarchitecture Simulation via Rigorous Statistical Sampling. In *30th Annual International Symposium on Computer Architecture*, June 2003.

[19] J. J. Yi and D. J. Lilja. Simulation of Computer Architectures: Simulators, Benchmarks, Methodologies, and Recommendations. *IEEE Trans. Comput.*, 55(3), 2006.

[20] S. Yoo, I. Bacivarov, A. Bouchhima, Y. Paviot, and A. A. Jerraya. Building Fast and Accurate SW Simulation Models Based on Hardware Abstraction Layer and Simulation Environment Abstraction Layer. In *DATE '03: Conference on Design, Automation and Test in Europe*, Washington, DC, USA, 2003. IEEE Computer Society.