A Fast and Generic Hybrid Simulation Approach Using C Virtual Machine

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

ABSTRACT

Instruction Set Simulators (ISSes) are important tools for cross-platform software development. The simulation speed is a major concern and many approaches have been proposed to improve the performance of ISSes. A prevalent technique is compiled simulation, which translates target programs into host instructions. But orders of magnitude of speed deterioration is inevitable since the difference between target and host *Instruction Set Architectures* (ISAs) can be large. An alternative is to emulate the program without sticking to binary compatibility. The performance problem is solved by using native execution. However, these emulators either require a special programming language, or a given *Application Programming Interface* (API). Last but not least, it is not trivial to integrate an emulator into a system simulator (which provides devices, external memory, etc., that the embedded programmers do care). In this paper, we propose a fast and generic hybrid simulation approach using virtualization technique to accelerate simulation and simulator-based debugging of C programs. A novel *virtual coprocessor* (VCP) is introduced as a processing element which executes C functions at high speed. This approach is C89 compliant and compatible with third party libraries and platform dependent code. It is also retargetable and can be integrated with existing ISSes. Two different ISAs are supported at present: MIPS and mAgic DSP. The average execution speed of the coprocessor is about 100 million simulated instructions per second.

Categories and Subject Descriptors

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

General Terms

Design

Keywords

Simulation, Debugging, Virtual Machine

CASES'07, September 30–October 3, 2007, Salzburg, Austria.

1. INTRODUCTION

Embedded system is a booming domain with a lot of new architectures introduced every year. Nowadays, a very short time-to-market is usually required in hardware/software co-design. Software development usually has to be started before the hardware platform and mature tools are available. This forces embedded software designers to undertake quick target-dependent developing and porting with limited tools at hand. In this situation, both compiling and debugging have to be performed by using cross-platform software development tools. Such tools can be divided into two major categories: simulation-based and emulation-based. Each has its strengths and weaknesses.

Simulation-based approaches rely on *Instruction Set Simulators* (ISSes) which interpret the behavior of target programs instruction by instruction. Fidelity and flexibility are provided by nature. For both architecture and micro-architecture design, simulators are irreplaceable for evaluation. System level simulations are also feasible when system models for buses and devices are integrated. Generally, compilers generating target binaries are also needed to support high level programming languages. The simulation-based approach is generic, but providing the compilation and simulation tools is not trivial. The number of host instructions have to be used to simulate one target instruction is orders of magnitude larger. This is an overkill for software development. Software developers do not care much about the architecture of the platform they are developing for, however they do need some target dependent features, e.g. the scratch-pad RAM and DMA. The speed of ISSes limits the work efficiency. ISS-based debugging can be extremely annoying if the speed is too slow.

A widely used alternative method is to sidestep the target ISAs, which means to compile and debug the code by using the native tool-chain of the host. The feasibility of target ISA decoupling comes from the broad existence of the software stacks, from *operating systems* (OSes) to middleware, from drivers to bytecode virtual machines. Given an abstraction layer, support routines can be provided to emulate the software. This approach is called emulation which sometimes is also known as virtualization. For example, the concept of VPU [13] abstracts processors and the OSes so that users can execute their programs before the actual hardware is ready. This enables the system architects to investigate the mapping of the application tasks with respect to space and time. Hardware abstraction and simulation environment abstraction layers [34] are proposed to build fast and accurate software simulation models by executing

the software and OS in the host machine natively. In those approaches, promising performance is achieved by using native execution on the host. But the problems arise when platform dependent code has to be supported. For instance, programs inlined with target assembly or linked with a third party library which is presented as target object code. At this high abstraction layer assembly level debugging is also suppressed, and the accuracy of performance estimation is affected for sure. Furthermore, extra efforts to integrate existing simulation components with an emulator (or virtual machine) have to be invested [13, 34].

By observing that the C programming language [14] has a dominating position in embedded software development, we propose a hybrid of the above two approaches, in which a C virtual machine is integrated with an existing ISS, providing high simulation speed and compatibility to platform dependent code at the same time. The relation between an ISS and the C virtual machine is similar to that between a processor and its coprocessor, so we call it *virtual coprocessor* (VCP). VCP fetches and processes whole functions instead of individual instructions. In other words, a function can be executed as if it were an instruction for VCP. This approach is called *Function-as-Instruction* (FaI). With FaI, hybrid simulation can be performed by switching dynamically the active processing element from ISS to VCP and vice versa. This approach is made possible mainly by two reasons. Firstly, only a very small portion of functions is target dependent. Secondly, usually users do not care about the details of all the functions during simulation and debugging. FaI can be used to step over a selected target independent function, or provide high overall simulation speed. Compatibility will not be sacrificed as platform dependent code can be processed by the original ISS. The objectives of this solution are:

- **Performance.** High simulation speed is the major goal. It is achieved by speeding up target independent C code.
- **Applicability.** A wide spectrum of existing programs should be supported by this technique. For example, third party libraries or programs using assembly code.
- **Retargetability.** It should be possible to retarget VCP to different ISAs.
- **Compatibility.** VCP can be integrated with other system simulation components in place of an ISS.
- **Usability.** Good usability should be provided for both simulation and debugging. To use this tool, additional knowledge should not be required.
- **Extensibility.** By exposing an interface, various tools can be built upon this technique.

The rest of this paper is structured as follows: The second section gives an overview and some design decisions of this approach. Section 3 describes the implementation of the instrumenter which enables VCP. We present the technical details of the bidirectional invocation for switching between ISS and VCP in section 4. In section 5, some user interfaces and extensions are given. Section 6 shows the evaluation result. After that, we compare our approach with other related works. The summary of this work is given in section 8.

2. PROPOSED APPROACH

This section describes the concept of combining the C virtual machine with an ISS to accelerate the instruction set simulation and ISS-based debugging. A conceptual hybrid architecture is proposed in which the virtual coprocessor is introduced. We also present why the current design is selected.

2.1 Overview

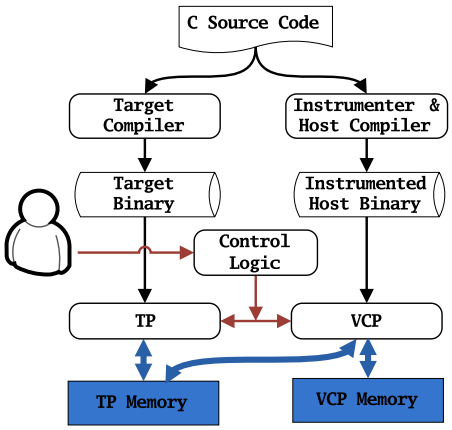


Figure 1: Architecture of the simulation system. A separate path is introduced to increase the simulation speed by using VCP.

Figure 1 shows the overall workflow of the proposed approach. The ISS is called *target processor* (TP) corresponding to VCP in this paper. The source code is compiled with the target compiler as usual. At the same time (as shown in the right path), the source code can be instrumented and compiled to a host binary for running on VCP. During the simulation, the execution of the program can be dynamically switched between TP and VCP. The switching is controlled statically or dynamically through command-line or a graphical user interface. These components are going to be introduced in this section:

- **VCP** is a coprocessor implemented as a C virtual machine. It is a fast execution engine for the target independent C functions.
- **Instrumenter** works together with the host compiler to provide inputs for VCP. The C source code is instrumented so that the functions executed at VCP behaves the same as using an ISS. VCP provides *service routines* to be used by the instrumented code.
- **TP and VCP Memories** form the memory hierarchy in this framework. TP memory stands for the original resources of TP, and VCP memory is the container for the local variables which are exclusively accessed by VCP.
- **Control Logic** is used to enable bidirectional invocation between TP and VCP. Since neither TP itself nor the target binary should be modified to keep this approach simple for retargeting, a monitoring mechanism is introduced instead.

2.2 VCP and Instrumenter

The source level instrumenter translates a target independent function into a form that can be natively executed on the host. Argument passing, value returning and global resource accesses must be modified to use service routines provided by the VCP. The calling convention defined by the target compilation system must be taken into account. Addresses of global resources at TP also need to be known. For example, if a global variable is accessed by a function to be virtually executed on VCP, then the address of that global variable at the target memory space has to be determined.

As source level instrumentation is static, the instrumenter suffers from incapability of self-modifying code supporting. But as our objective is to provide a tool for embedded software design, this does not represent a big limitation and we decided to leave it for future.

Pure static instrumentation is not sufficient. As already mentioned, some information (for example, the TP address of a global variable) of the TP program should be known. The information is not present in the source code itself, and will not be revealed until the target program is being linked. If the loader performs a relocation, the information will change again. Therefore we proposed a mixed process: instrument statically, and link the global resources at runtime.

Then it comes to how to place the virtual machine. As we want to support third party ISSes, implementing the virtual machine inside the ISS is not feasible. On the other hand, if the virtual machine is built below the ISS as a hypervisor, we will have to face the integration problem with other system components. Instead, we chose a solution that fuses another execution engine as a coprocessor in parallel.

Coprocessor techniques are widely used in modern computer systems to increase the overall performance of specific applications. For instance, coprocessors are used for floating point computation in i386 and ARM processors. Another example is for instruction set extension of application specific instruction-set processors. Usually a coprocessor works in blocking mode, which means the processor has to be paused to wait for the result. But since coprocessors are designed to enable or speedup special instructions, a performance improvement of the throughput is usually feasible.

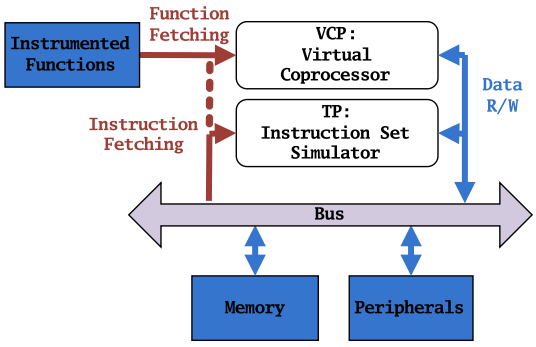


Figure 2: Relationship between TP and VCP. Functions are fetched instead of instructions by VCP.

TP and VCP execute in a mutually exclusive way in this approach. When TP is activated, the system is at *simulation mode*. TP is blocked when VCP is activated, and this is called *virtual mode*. As shown in Figure 2, TP performs instruction level simulation, while VCP consumes instrumented C functions as instructions. The candidate function for FaI must be consistent with the calling convention, and will always return to the calling point. The C source code of these functions is instrumented before being compiled by the host compiler. Therefore VCP service routines can be used to handle the target memory accesses. The instrumenter also generates stubs for bidirectional invocations which will be discussed later.

2.3 Virtual Memory Hierarchy

VCP allows instrumented code to access both native memory and TP resources. Generally, TP resources include TP data memory, TP program memory and TP registers. As TP resources exist in the memory of the ISS, we call them *TP memory* for convenience. TP memory can be accessed by both TP and VCP, so it works like shared memory. The host compiler compiles the instrumented code to a host binary, in which the native memory is used by default. The native memory cannot be accessed by TP and is considered as an affiliated local memory of VCP, so the name *VCP memory* is given for convenience. These two memories constitute a virtual memory hierarchy.

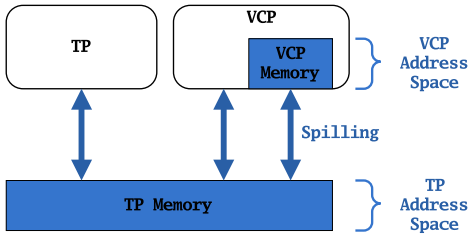


Figure 3: Virtual memory hierarchy.

Figure 3 shows the virtual memory hierarchy. Both TP memory and VCP memory have their own address spaces, which are called *TP address space* and *VCP address space* respectively. These two address spaces can be overlapped in value. For example, the memory address space for 32 bit MIPS processor and 32 bit i386 host processor are both from `0` to `0xFFFFFFFF`. If a pointer is not explicitly specified to which address space it points, dereferencing cannot be performed due to the ambiguity.

A pointer to VCP memory can be safely dereferenced. Further optimization by the host compiler is also feasible, if pointer analysis [2] is performed. However, any access through a pointer to TP memory has to be handled by service routines. There is a space of optimization for TP memory accesses, which is our future work.

2.4 Control Logic

FaI needs invocations from TP to VCP, and vice versa. Some bridging method is needed to mimic stacks, pass parameters and handle return values. There are two options to do this, binary instrumentation and monitoring.

As binary instrumentation will eventually increase the effort of retargeting, it is not selected. To be generic, the desired approach should not modify ISS and the target binary running on it.

An external *control logic* is introduced to enable the monitoring. The control logic monitors TP's execution and calls

the corresponding stub code to shift the execution to VCP when FaI is applied. Monitoring requires mapping information or debug information of the target program to know the TP address of each function. This information might be unsafe if the program is relocated. It is not a severe limitation for embedded software design, so we plan to address this problem in the future work.

Inverse stubs which conduct the execution from VCP to TP are injected to the instrumented functions directly by the instrumenter.

3. INSTRUMENTATION

In this section, the implementation of VCP and instrumenter is described. To enable FaI, the C source code is instrumented and compiled to a host dynamically linked library, which can be loaded by VCP at runtime. VCP acts as a service provider and the instrumenter modifies the C source code using these services to enable FaI. Since the C programming language is flexible and rich in features, the description in this section is organized by how these features are supported in virtual mode.

3.1 Local Variables

Local variables can only be accessed within the function where they are defined, if their addresses are not exposed to the outside. These variables are only visible to VCP if the functions have FaI being applied. Thus, no instrumentation is needed for them.

But there is an exception for the local static variables. Different instances of the function share the values of the local static variables. They are treated in a similar manner as global variables, and will not be located onto the stack. The instrumentation for them is similar as that of global variables.

3.2 Global and Static Variables

Global variables can be accessed both by names or through pointers. For each global variable, a unique *linking pointer* is declared by the instrumenter. A linking pointer will be dynamically linked to the corresponding global variable at TP address space when VCP loads the instrumented binary. Any access to the global variable is instrumented to use this linking pointer indirectly. Since the linking pointers point to another address space, support routines are used to handle them.

Static variables in C can be classified into external static variables and internal static variables [14], according to their scopes. Note that static variables have the same storage as global variables, therefore the same approach is applicable for them. Figure 4 gives an example of instrumenting global and static variables. Suppose `foo` is a function to be executed on VCP, then several linking pointers for the global and static variables have to be created. Then the accesses can be performed indirectly by calling service routines, e.g. `RM_int` reads an integer value from TP memory (4), (9), and `WM_int` is used for writing to TP memory (6), (7), (8).

Accessing the TP address space is slower than accessing a native global variable at the host. Therefore, unnecessary TP accesses should be avoided. For global or static variables declared with the `const` keyword, clones of the variables are created at VCP address space and can be used from VCP directly.

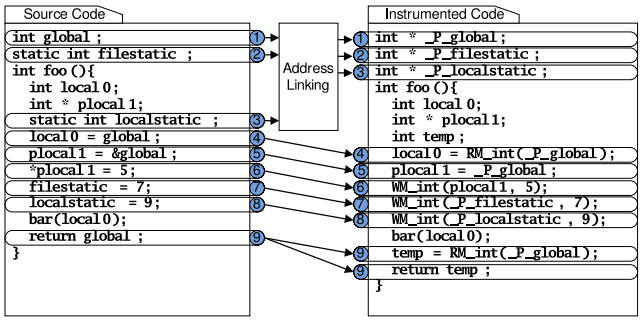


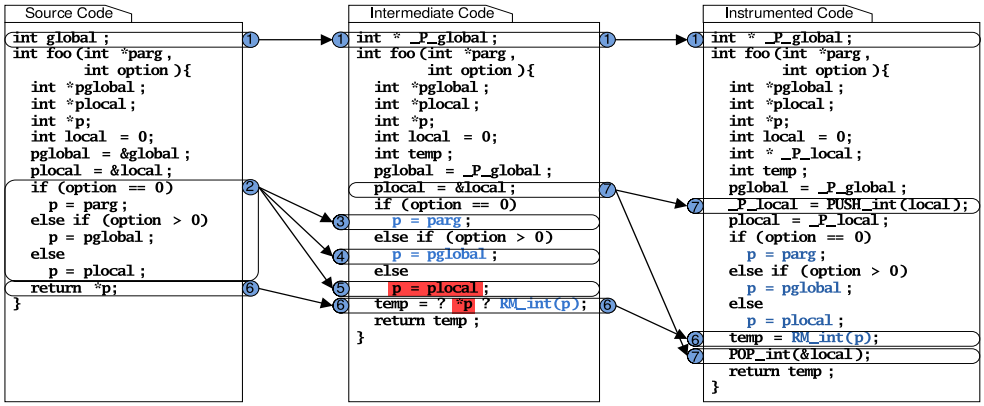
Figure 4: Example of instrumenting global and static variables. (1)(2)(3): Addresses of global and static variables are assigned to the linking variables. (5)(6): Linking variables are used to represent the TP addresses. (4)(7)(8)(9): And accesses of these variables by names are replaced by using the linking variables.

Mapping information reveals the addresses of global variables, so no dependency on debug information is there. Static variable linking cannot be performed without debug information. We try to resolve the addresses of static variables by using debug information. If a static variable is irresolvable, then the functions accessing it should be suppressed for FaI. Dependency on debug information will be further minimized in our future work since it is not always reliable.

3.3 Dereferencing

Variable accessing and function calling through pointers need the dereference operations. If a pointer holds a VCP address, it can be dereferenced natively. But for TP addresses, service routines have to be used to reaching TP memory, e.g. operations (4) and (6) to (9) in Figure 4. When a pointer is being dereferenced, there must be no ambiguity on whether it is holding a TP address or a VCP address. Otherwise, a *pointer hazard* will happen.

Spilling is used to remove pointer hazards. For example, pointer `p` in Figure 5 (a) has a conditional initialization (2), which may be assigned with one of the three pointers `parg`, `pglobal` or `plocal`. When FaI is applied to the function `foo`, the three pointers used to initialize pointer `p` are not consistent in address space. The caller passes `parg` (3) in by value, so it holds a TP address. Being a pointer to a global variable, `pglobal` (4) also holds a TP address. But since the local variable `local` is allocated at VCP memory, the pointer `plocal` (5) contains a VCP address. Because of that, the correct method to dereference pointer `p` cannot be decided at compile time (6). To convert the inconsistent pointers, either temporarily copying a VCP variable to TP or doing the opposite is needed. This is called a spilling. Spilling VCP variables to TP memory space is chosen, because pointer hazard can also happen when passing a pointer to a function as an argument (it means potential dereferencings may happen in the callee), in which case, the pointer has to be assumed to hold a TP address. The push / pop services `PUSH_int` and `POP_int` are used to move the data to and from the TP memory (7). So that variables in VCP memory can be migrated to TP memory temporarily to resolve this ambiguity.



(a) Original C source code (b) Pointer hazard for dereferencing (c) Resolved by spilling

Figure 5: Example of pointer hazard resolving.

3.4 Aggregate and Special Data Structures

The layouts of aggregate data like `struct` or `union` are not defined by the C standard. When a member is accessed, usually a compiler will calculate its offset in the aggregate data structure first. The offsets are decided by the members' declaration sequence and the padding between them.

This problem is sidestepped when virtualizing 32-bit little-endian MIPS ISA at 32-bit i386 host, because they have identical layouts. However, it is not the case for mAgic DSP [17]. In mAgic, 40-bit is chosen as machine word width, and the data addresses of mAgic are incremented in units of machine words instead of bytes.

This problem is hard to be resolved especially when considering aggregate data manipulated by `memcpy`-like functions, current there is no known method to copy an aggregate data to TP by VCP or vice versa. So any function which has usage of aggregate data is simply considered target dependent and forced to be non-partitionable to the VCP for mAgic.

floating point data in mAgic is also presented in a 40-bit format instead of IEEE 754 standard formats [11]. The instrumenter translates them to and from 64-bit IEEE 754 format when TP address space is accessed. The precision of the computation results are slightly different from the target processor, but this can be neglected for most of the programs.

4. HYBRIDIZATION

The hybridization controls the communication of VCP and TP. It also addresses the problem of when and how VCP and TP are invoked by each other.

4.1 Function Identification

Not all the C functions are applicable to be distributed to VCP. The instrumenter is used to identify functions for FaI. If a function is consistent to the calling convention and will always return to the point where it was called, then partitioning this function to VCP is safe, and this function is called a *Partitionable function* (P-function). The counterpart is a *Non-Partitionable function* (NP-function), which can only be executed at TP. Numerous callees can be called from one function. However, if the callees are NP-functions, applying FaI to the caller will not be prevented. This is important when assembly code or third party libraries are used.

Function invocations between TP and VCP also need to know the addresses of the functions. *Function linking* is used to share the information. For the instrumented host binary, the addresses of functions are collected when it is loaded by VCP. But for target binary, since we do not want to have a dependency on debug information, the addresses are collected from the mapping information generated by the linker of the target tools chain. Since static functions do not expose their addresses to the linker, FaI is not applicable for them. But if the caller of a static function is mapped to VCP, the static function itself will be executed on VCP also.

Both the target compiler and the host compiler may perform aggressive optimizations which may be obstacles for FaI. For example, function inlining and tail call optimization [18]. These optimizations are forbidden to the host compiler. But as one objective of the design, they cannot be suppressed for the target compilers. When a function is detected to be inlined or has been performed with other aggressive optimizations, it becomes a NP-function.

4.2 Bidirectional Invocation

The invocation of a VCP function from TP is called *forward invocation* and from VCP to TP *inverse invocation*. Since TP and VCP functions are presented in different ISAs and have different calling conventions, stub code has to be used to bridge the invocations between VCP and TP functions. The stubs are called *forward stubs* and *inverse stubs* respectively.

There are two possible methods to trigger a stub, by injecting the stub to the source code (or directly to the binary) or through a monitor. Control logic is used as a monitor for forward invocations. For inverse invocation, direct injection is performed during the instrumentation. The bodies of both types of stubs are generated by the instrumenter and reside in the native binary. When a function is mapped to VCP, the control logic monitors the TP simulator by setting a breakpoint at the entrance of the function, and when the breakpoint is met, the corresponding forward stub in the native binary is called.

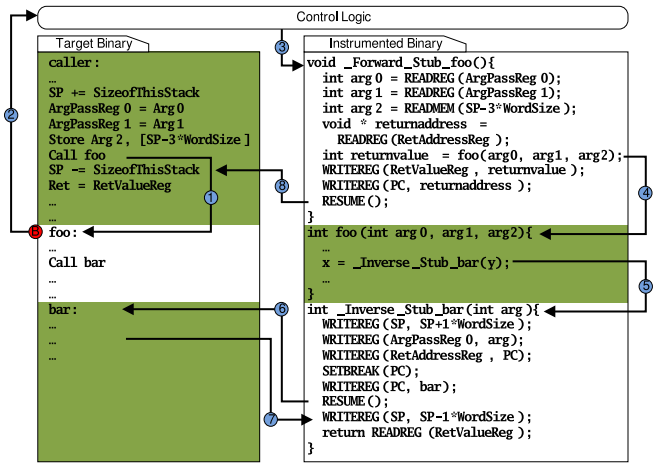


Figure 6: Example of bidirectional invocation. The forward stub is implemented by setting and monitoring a breakpoint. Direct injection is used for inverse stub.

Figure 6 shows an example of invocations for both sides. The shadows show that the VCP function `foo` is called from a TP function `caller` and is distributed to VCP. Function `foo` will further call a function named bar at TP. The control logic sets a breakpoint at the entrance of `foo` (2). When TP breaks there, the control logic calls the forward stub of `foo` (3). The forward stub bridges the execution of `foo` to the host execution (4). Function `bar` is invoked from `foo` by directly injected inverse stub (5). The inverse stub of `bar` invokes bar at TP by mimicking the stack for calling and returning (6), (7). When `foo` is finished at VCP the rest of the execution is resumed from `caller` (8).

4.3 Library Support

When a function from a third party library is called in the program, it can be easily identified that there is a function without defination. Therefore, this function will be executed at TP even if the caller is executed at VCP. But if the behavior of a library function is known, a VCP clone can be used to replace it. Pure functions (with no side effect) can be cloned for sure. Functions have defined side effect are also applicable for cloning. For example, `memcpy` is a function in Standard C Library, which has defined side effect. A VCP clone can be used as an alternative safely. Note that the native `memcpy` on the host cannot be used, because the VCP clone accepts argument pointers holding TP address. On the other hand, `malloc` is not applicable as it has side effect on internal data structures. A set of Standard C Library functions with defined side effect are supported in VCP as services.

There are some library functions e.g. `longjmp`, `setjmp` which manipulate the stack in a nonstandard manner. Consequentially, not only these functions themselves, but also the direct and indirect callers of them cannot be executed at VCP. Thus, they are annotated as exceptional cases of the NP-functions, named *Recursively-Non-Partitionable functions* (RNP-functions). An RNP-function recursively suppresses the partition of the callers. So if a function calls any RNP-function direct or indirectly, it has to be executed at TP.

Generally, functions implemented in assembly or containing inline assembly are regarded as library functions. As RNP is a very strong restriction, the default attribute for them is NP. If there is any exception, the user has to set the function's attribute manually.

5. USER INTERFACES

By using the technique presented here, a program can be partially executed at VCP. This is called partitioning. Function is the basic unit of partitioning. But it is insufficient to only partition one single function. As a function can call other functions, they can be grouped together. A cluster is a group of functions with a root function. Starting from the root, all the partitionable callees are included into this group. The calling can be done natively and there is no invocation overhead (But if there is any pointer hazard, they do trigger a spilling).

To facilitate using VCP, both static and dynamic user interfaces are defined, which can be further extended.

5.1 Static Partitioning

The user can set the functions to be partitioned to VCP statically before the simulation proceeds. When the user wants to run some regression test or benchmark some part of the program regularly, he or she can define a partitioning of the program manually or with the assistance of the tool named HySim, as will be described below.

Another use case is to provide a pre-partitioned library using this technique. The provider only needs to hand over the target binary, headers, plus the host binary for FaI and the pre-partitioning to the users. Then, the users can benefit from VCP without any interference and awareness.

5.2 Fast Step Over

The basic dynamic partitioning use case is fast step over when debugging. Step over is one of the most frequently used command in debugging. With VCP technique, a user instruct a fast stepping over on a function. Thus, the function (actually the cluster) will be executed in FaI mode.

5.3 Fast Breakpoint

A tool named HySim has been developed based on this technique by introducing a new user command named *fast-forwarding* (FF) breakpoint to an existing debugger. It facilitates users to get an automatic partitioning in order to reach a given point of program in a fast execution speed. The user can simply set a FF breakpoint instead of a normal breakpoint to trigger the automatic partitioning. At this time, inter-procedural control flow is analyzed first to get the knowledge about which functions may be executed before reaching the specified FF breakpoint. Then the partitioning is performed based on the partition properties of those functions.

6. EVALUATION RESULT

This section describes the experimental results, which have been measured on a computer with an Athlon64 X2 4600+ processor and 4 GB of memory, running Fedora Core 4 distribution of Linux operating system. The cross compiler for MIPS is GCC 2.96 for 32-bit little-endian MIPS, and Target Compiler [31] is used for mAgic. The instrumented source code for VCP is compiled with GCC 4.0.2.

6.1 Application Performance Improvement

To benchmark VCP technique for MIPS simulation, 5 algorithms are chosen: encryption (`DES`), message digest (`MD5`), image edge detection (`Susan`), audio codec (`G721`), and image decoding (`JPEG_Dec`). Assuming that the user is not an expert, only the evident hotspot functions (1 to 2 functions for each case) are manually partitioned to VCP. For instance, `des_crypt` and `des3_crypt` for `DES`, and `idct_islow` for `JPEG_Dec`. The overall simulation speed has a significant increase after applying FaI to these functions. Figure 7 shows the percentage of execution benefit from partitioning the hotspot functions. As shown in Figure 8, the *Original Speed* of the MIPS ISS is around 3 million *Instruction Per Second* (IPS). After manual partitioning, the *Overall Speeds* of these applications increase from 3.3 to 94.0 million IPS with speedups from 1.3 to 34.5, related to the result of the partitioning. The pure *VCP Speed* is about 110 million IPS.

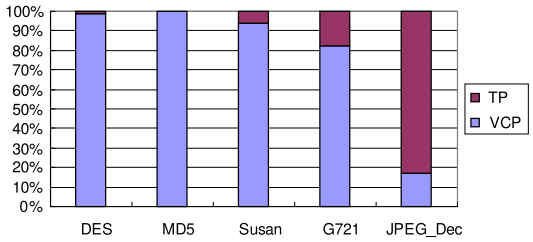


Figure 7: Percentages of runtime execution count for manual partitioning.

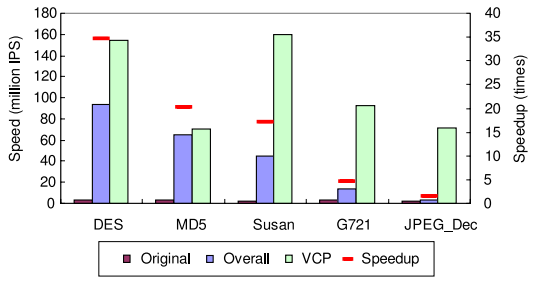


Figure 8: Performance and speedup.

6.2 Function Level Speedup

As described, another use case of this approach is to improve *step over* speed in debugging. To evaluate the result, some functions are selected from the above 5 cases. Not only the hotspot functions used in application level benchmarking, but also functions with both large and small runtime execution counts are considered. Table 1 shows the individual function level results for the MIPS processor, sorted by time spent on original simulation.

It can be obtained that this approach produces a good performance improvement for large functions and a fair speedup for small functions which are already less time consuming. There are two major impact factors on the speed of FaI:

- **TP Execution.** Library routines for which source code is not available and NP-functions can only be simulated by TP. As an example, although the function `jpeg_make_d_derived_tbl` has a quite large runtime execution count, the speedup after applying FaI is only 1.9 times. The reason is this function calls several C library functions, which are simulated at TP by inverse invocations.
- **TP Memory Access.** For instance, both `susan_thin` and `des3_crypt` are completely executed on VCP, but the speedups of them have a significant difference. Because the former mainly operates on pointers which are instrumented to use TP memory accesses, but the later uses global constant variables which have a clone at VCP memory space.

Function Name	Time Spent on ISS (millisecond)	Time Spent on FaI (millisecond)	Speedup (times)
`susan_edges`	129019.029	2040.352	63.2
`susan_thin`	3217.940	83.147	38.7
`jpeg_make_d_derived_tbl`	7.314	3.783	1.9
`md5_csum`	4.926	0.392	12.5
`md5_finish`	4.663	0.384	12.1
`ycc_rgb_convert`	4.323	0.151	28.6
`h2v2_fancy_upsample`	4.125	0.068	60.3
`md5_process`	3.904	0.270	14.5
`jpeg_idct_islow`	3.242	0.058	56.3
`decode_mcu`	2.335	0.235	9.9
`des_set_key`	1.972	0.020	97.3
`g721_encoder`	1.759	0.043	41.1
`g721_decoder`	1.570	0.041	38.3
`step_size`	1.007	0.015	65.8
`des3_crypt`	0.837	0.006	138.1
`reconstruct`	0.810	0.014	59.8
`update`	0.675	0.025	27.4
`predictor_zero`	0.417	0.015	28.4
`jpeg_fill_bit_buffer`	0.283	0.018	15.5
`des_crypt`	0.247	0.003	83.0
`jpeg_huff_decode`	0.240	0.018	13.4
`predictor_pole`	0.102	0.013	7.6
`g72x_init_state`	0.096	0.010	9.7
`md5_update`	0.076	0.011	6.7
`fullsize_upsample`	0.052	0.015	3.5

Table 1: Individual Function Level Results.

6.3 Retargetability

To verify the retargetability, the integration with ISS for mAgic DSP is performed. mAgic is a floating point DSP with *Very Long Instruction Word* (VLIW) architecture. This architecture is extremely fast, e.g. it can finish a whole butterfly of floating point *Fast Fourier Transform* (FFT) in a single cycle. But for us, this architecture is extremely slow in simulation. Since functions using aggregate data structures are not supported for mAgic, only 3 cases are selected to show the preliminary results. Floating point operations are used to implement the cases `FIR` and `FFT`, and `Edge_detection` is a fixed point case. As 40-bit floating point format is used in the mAgic ISS, the original simulation speed of floating point cases is slower than the fixed point one. After partitioning the hotspot functions to VCP, results are generated for these applications in Table 2. The execution speed of the VCP is quite close to the one for MIPS simulation.

Case Name	Operation Type	Number of Instructions	Original ISS Speed (million IPS)	Overall Speed using VCP (million IPS)	Speedup (times)
`Edge_detection`	fixed point	682314	0.699	110.765	158.5
`FIR`	floating point	215732	0.395	67.628	171.1
`FFT`	floating point	227579	0.320	28.698	89.7

Table 2: Preliminary Application Level Results for mAgic DSP.

7. RELATED WORK

Three kinds of related work are discussed in this section. Firstly, previous techniques to improve the speed of ISSes are described. Thereafter, the existing instrumentation techniques are discussed. Finally, some interesting execution switching approaches are given, though some of them are used in other domains rather than simulation.

7.1 Fast Simulation Techniques

Simulation techniques can be coarsely divided into interpretive, statically compiled and dynamically compiled approaches. Interpretive simulation is the basic technique which is flexible but slow. Simplescalar [6] is a retargetable interpretive simulator widely used for performance estimation and scientific research [26, 27, 32].

Compiled simulation moves some computation such as fetching and decoding from run-time to compile-time to improve the simulation speed. In statically compiled approaches [7, 35] dynamic decoding of target binary code is avoided. For example, SyntSim [7] is a simulator synthesizer, which generates statically compiled/interpretive mixed simulators. In their approach, target binary can be translated into C language and further compiled to host instructions. Rather than using C language as a code generation interface, an improvement [35] is introduced, in which an aggressive low level code generation interface is used.

Statically compilation based approaches are limited, since self modifying code cannot be supported. Dynamically compiled simulation approaches are proposed in [24, 9, 25, 22, 23]. A *Just-In-Time Cache Compiled Simulation* (JIT-CCS) technique that combines retargetability, flexibility and high simulation performance is presented by Nohl *et al.* [22]. The compilation of target program takes place at run time but the compilation result is cached for reuse. Once the program at a given address is changed, the cache is invalidated and a new compilation for the modified code can triggered on demand. A multiprocessing approach [23] is proposed to benefit from multiple processing units. The heart of the approach is a simulation engine capable of mixed interpretative and compiled simulation. Instead of pausing the simulation for compilation, frequently executed blocks can be dynamically compiled using assistant processors. This multiprocessing approach offers higher performance and improved speed consistency.

Dynamically compiled simulation is close to *Dynamic Binary Instrumentation* (DBI), in the sense that both generate and execute native code. But unfortunately, since high level information is irreversiblly lost in the target binary, the performance of the generated native code is much lower than the natively compiled one from the source code. This drawback is extremely significant for complex architectures.

On the other hand, the VCP approach described in this paper does not mine information from target binaries but from target independent part of the source code. There are two major advantages, utilizing high level information to get higher speed, and being easy to retarget.

7.2 Instrumentation Techniques

As the core of our approach is a source level instrumenter, some reviews and comparison of the instrumentation techniques are presented here.

Source level instrumentation techniques are used for source code simplification [19], error detecting [20] and fine grain application profiling [12]. Source level instrumentation is processed at compile time in nature.

The technique presented in this paper is based on a variance of source level instrumenter, but the instrumented code behaves like a target function (in the sense of compatibility with an ISS) instead of a native function.

Numerous DBI tools/frameworks [29, 28, 21, 16, 3, 33] are designed for application profiling, error detection, virtualization etc. DBI tools are powerful as they can support self-modifying code and even privileged instruction emulation.

Probe based dynamic binary instrumenters [29, 28] inject the instrumentation code into the executable by some means, and the instrumentation code can be invoked consequentially.

A more flexible approach is to use binary translation, which can be further classified into copy and annotate [16, 5, 3, 33], copy and modify [1], or Just-In-Time (JIT) compilation [21] based approaches. In binary translation based instrumenters, the original executable is not executed but only the translation. More functionality and consolidation can be achieved by these approaches.

The hybridization approach in this paper uses breakpoints to monitor the execution of target binary. It can be seen as a combination of using probe based approach to facilitate source level instrumentation.

7.3 Execution Switching Approaches

For architecture design space exploration, mixed level simulation is used to fast forward the program to a specific point by functional simulation and continue from there with detailed simulation [27, 32]. This hybrid approach sacrifices some accuracy at irrelevant pieces of code to get an overall performance improvement of the entire simulation.

The SimSnap [30] framework utilizes application level checkpointing to perform the switching from host execution to simulation. The application's source code is instrumented by the Cornell Checkpoint Compiler [4] in order to save the state of the program at a given point. The target binary to be simulated also needs a corresponding instrumentation so that the execution can be resumed from that point. Cur-

rently, the proposed approach works only if the ISA of the simulator is compatible with the ISA of the host machine. However, the authors have pointed out it is possible to use this approach if the host ISA is different from the ISA to be simulated.

C virtual machines are used as log recorders for reversible debuggers [10, 15]. An interesting approach of switching between native execution (called *native mode*) and C virtual machine execution (*virtual mode*) is proposed by [15]. Virtual mode is slower but reverse execution information is logged, and native mode do not have this overhead. Users can dynamically switch the debugging between each mode based on their requirements. The dynamic switching between native mode and virtual mode is enabled by utilizing the debug information.

One interesting instrumentation tool BitRaker Anvil [8] is proposed by Calder *et al.*. The target binary to be simulated is instrumented by invoking host native annotation code. Thus better performance can be achieved. In contrast, VCP technique allows interaction by using bidirectional invocation.

8. SUMMARY

We proposed a novel hybrid simulation approach which uses a virtual coprocessor to accelerate the instruction set simulation and ISS-based debugging.

The VCP is a C virtual machine which leverages the simulator for target processor by executing some C functions in virtual mode as if they were single instructions for the VCP. The coprocessor can achieve a speed of more than 100 million IPS for both RISC and a quite complicated VLIW DSP. A wide spectrum of C89 compliant programs can be supported. Good compatibility with platform dependent code is provided. Assembly and third party libraries can also be used.

The introduction of VCP is seamless for the existing simulation systems. The interfaces between the orignal ISS and other simulation components are untouched.

Various user interfaces are provided, from which embedded software development, debugging and testing can benefit. Library providers can also use this technique to package a FaI enabled library without leaking source code information. By using this technique, various tools can be developed, e.g. a performance estimator is built as an extension.

The tool is retargetable and supports two different target processors at present: the MIPS processor and the mAgic DSP. To support a new ISA, integration with an existing ISS is needed and the retargeting should be performed according to the target ISA and the calling convention of the target compiler. Limited efforts need to be invested to retarget VCP to different ISAs.

This work is only the first step, and there are several known weaknesses. The aggregate data structures cannot be generically supported. The spilling method for dereferencing is brute force. Target independent code detection is not smart enough. Dynamically relocatable code (e.g. dynamically linked libraries) is not supported. Self modifying code is not supported. The dependency on debug information is not completely removed. We plan to address them in the future work. Although the current approach has some shortcomings, it is still applicable for DSP or embedded processor simulation.

9. ACKNOWLEDGMENTS

This work is part of the European project SHAPES (`shapes-p.org`). We would like to thank Pier Paolucci, Alberto Dell'Olio and Stefano Fasciani of Atmel for providing us mAgic hardware information to develop the mAgic simulator. Thanks to Torsten Kempf, Jianjiang Ceng and Jeronimo Castrillon for assisting the mAgic simulator development. And thanks to the anonymous reviewers for their valuable comments.

10. REFERENCES

[1] K. Adams and O. Agesen. A comparison of software and hardware techniques for x86 virtualization. In *ASPLOS-XII: Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 2–13, New York, NY, USA, 2006. ACM Press.

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

[3] S. Bhansali, W.-K. Chen, S. de Jong, A. Edwards, R. Murray, M. Drinić, D. Mihočka, and J. Chau. Framework for instruction-level tracing and analysis of program executions. In *VEE '06: Proceedings of the 2nd International Conference on Virtual Execution Environments*, pages 154–163, New York, NY, USA, 2006. ACM Press.

[4] 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.

[5] P. P. Bungale and C.-K. Luk. PinOS: a programmable framework for whole-system dynamic instrumentation. In *VEE '07: Proceedings of the 3rd International Conference on Virtual Execution Environments*, pages 137–147, New York, NY, USA, 2007. ACM Press.

[6] D. C. Burger and T. M. Austin. The simplescalar tool set, version 2.0. Technical Report CS-TR-1997-1342, 1997.

[7] M. Burtscher and I. Ganusov. Automatic synthesis of high-speed processor simulators. In *MICRO 37: Proceedings of IEEE/ACM International Symposium on Microarchitecture*, pages 55–66, Washington, DC, USA, 2004. IEEE Computer Society.

[8] B. Calder, T. Austin, D. Yang, T. Sherwood, S. Sair, D. Newquist, and T. Cusac. Bitraker Anvil: Binary instrumentation for rapid creation of simulation and workload analysis tools. In *Proceedings of Global Signal Processing (GSPx) Conference*, 2004.

[9] B. Cmelik and D. Keppel. Shade: A fast instruction-set simulator for execution profiling. *ACM SIGMETRICS Performance Evaluation Review*, 22(1):128–137, May 1994.

[10] C. Demetrescu and I. Finocchi. A portable virtual machine for program debugging and directing. In *SAC '04: Proceedings of the 2004 ACM Symposium on Applied Aomputing*, pages 1524–1530, New York, NY, USA, 2004. ACM Press.

[11] 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.

[12] K. Karuri, M. A. A. Faruque, S. Kraemer, R. Leupers, G. Ascheid, and H. Meyr. Fine-grained application source code profiling for ASIP design. In *DAC '05: Proceedings of the 42nd Annual Conference on Design Automation*, pages 329–334, New York, NY, USA, 2005. ACM Press.

[13] T. Kempf, M. Doerper, R. Leupers, G. Ascheid, H. Meyr, T. Kogel, and B. Vanthournout. A modular simulation framework for spatial and temporal task mapping onto multi-processor SoC platforms. In *DATE '05: Conference on Design, Automation and Test in Europe*, Washington, DC, USA, 2005. IEEE Computer Society.

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

[15] T. Koju, S. Takada, and N. Doi. An efficient and generic reversible debugger using the virtual machine based approach. In *VEE '05: Proceedings of the 1st ACM/USENIX International Conference on Virtual Execution Environments*, pages 79–88, New York, NY, USA, 2005. ACM Press.

[16] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *PLDI '05: Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 190–200, New York, NY, USA, 2005. ACM Press.

[17] mAgic DSP. `www.atmel.com`.

[18] S. S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1997.

[19] G. C. Necula, S. McPeak, S. P. Rahul, and W. Weimer. CIL: Intermediate language and tools for analysis and transformation of C programs. In *Computational Complexity*, pages 213–228, 2002.

[20] G. C. Necula, S. McPeak, and W. Weimer. CCured: Type-safe retrofitting of legacy code. In *POPL '02: Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 128–139, New York, NY, USA, 2002. ACM Press.

[21] N. Nethercote and J. Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *PLDI '07: Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 89–100, New York, NY, USA, 2007. ACM Press.

[22] 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.

[23] 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.

[24] M. Reshadi, P. Mishra, and N. Dutt. Instruction set compiled simulation: A technique for fast and flexible instruction set simulation. In *DAC '03: Proceedings of the Conference on Design Automation*, New York, NY, USA, 2003. ACM Press.

[25] E. Schnarr and J. R. Larus. Fast out-of-order processor simulation using memoization. In *ASPLOS-VIII: Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 283–294, New York, NY, USA, 1998. ACM Press.

[26] 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.

[27] T. Sherwood, E. Perelman, G. Hamerly, S. Sair, and B. Calder. Discovering and exploiting program phases. *IEEE Micro*, December 2003.

[28] A. Srivastava, A. Edwards, and H. Vo. Vulcan: Binary Transformation in a Distributed Environment. Technical Report MSR-TR-2001-50, 2001.

[29] A. Srivastava and A. Eustace. ATOM: A system for building customized program analysis tools. In *PLDI '94: Proceedings of the ACM SIGPLAN 1994 Conference on Programming Language Design and Implementation*, pages 196–205, New York, NY, USA, 1994. ACM Press.

[30] P. K. Szwed, D. Marques, R. M. Buels, S. A. McKee, and M. Schulz. SimSnap: Fast-forwarding via native execution and application-level checkpointing. *8th Workshop on Interaction between Compilers and Computer Architectures*, 00, 2004.

[31] Target Compiler Technologies. `www.retarget.com`.

[32] 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.

[33] J. Yang, S. Zhou, and M. L. Soffa. Dimension: an instrumentation tool for virtual execution environments. In *VEE '06: Proceedings of the 2nd International Conference on Virtual Execution Environments*, pages 164–174, New York, NY, USA, 2006. ACM Press.

[34] 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.

[35] J. Zhu and D. D. Gajski. A retargetable, ultra-fast instruction set simulator. In *Proceedings Design, Automation and Test Europe Conference and Exhibition*, pages 298–302, 1999.