An Integrated Performance Estimation Approach in a Hybrid Simulation Framework

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

ABSTRACT

The increasing complexities of today's embedded multimedia and wireless devices have ushered in the era of heterogeneous Multi-Processor System-on-Chip (MPSoC) architectures. This trend, in turn, have made software parallelization and optimization a subject of utmost importance for today's systems. Nowadays, providing efficient software implementations is not only mandatory for the final products, but also necessary for Design Space Exploration (DSE) of the numerous hardware choices available for an MPSoC development. Unfortunately, such co-exploration of different software solutions and hardware architectures usually requires an extraordinarily large effort due to the continually increasing gap between the speed of the real processor hardware and that of the available instruction set simulators. This problem can be greatly alleviated by using cycle approximate, but fast simulation models for early DSE where relative merits of different design solutions are more important than hundred percent cycle accuracy.

To address the issue of fast performance estimation for DSE, HySim - a hybrid simulation framework which consists of an *Instruction Set Simulator* (ISS) and a native execution engine called *Virtual CoProcessor* (VCP) has been proposed. Virtualization is performed so that parts of the execution can be shifted to the native engine without sacrificing the functional correctness of the whole application. High execution speed with performance estimation is available at the VCP side, and therefore, combined with the ISS, good accuracy can be obtained. Our previous work introduced two performance estimation approaches - *annotation based* and *dynamic profiling based*. However, some new questions are opened on how to combine these two approaches, and how to effectively partition an application to VCP and ISS. In this paper, annotation based performance estimation is used to facilitate dynamic profiling, and a preliminary sampling approach is also introduced to open the possibility of further reducing dynamic profiling overhead by inter/extrapolation.

Keywords

HySim, Hybrid Simulation, Design Space Exploration, Virtualization, Performance Annotation, Dynamic Profiling, Sampling

1. INTRODUCTION

One of the most effective methods to explore the architectural and micro-architectural design spaces is to evaluate the hardware designs with realistic applications. To tackle this issue, a wide spectrum of approaches, e.g., *cycle accurate simulation* [27, 28, 32, 33], *source-level performance annotation* [11, 12, 17, 13], *partial simulation* [32, 27, 28, 23, 20, 6, 18, 19], have been proposed to precisely obtain performance estimates for a set of given applications. In principle, all of these approaches are *performance evaluation* techniques which try to address various aspects of the following system development steps:

- **Design Space Exploration:** The architecture is modified/configured/customized to support a set of applications effectively. Depending on the modification of the architecture, DSE can be coarse grained (e.g., introducing an ASIP to the system) or more fine grained (e.g., adjusting the size of the cache).
- **Performance Optimization:** It is important to optimize the applications and/or the whole software stack to meet the design goals and provide realistic workloads for DSE.
- **Performance Verification:** Before taping out the hardware, the performance should be verified accurately.

All the above are important issues for single processor DSE. Additionally, several new issues arise when Multi-Processor system development is considered.

Nowadays, heterogeneous MPSoCs are especially attractive for application domains like communication, signal processing, multimedia, real-time gaming and so on. The key to success in those systems is to effectively exploit processing power by optimizing, partitioning and parallelizing the applications on top of them. For example, a large spectrum of communication models (shared memory, software transactional memory, message passing, etc.) presents the programmers with a dilemma of selection and migration. To make it even worse, one has to realize that such partitioning, by no means, is a one-time effort due to the continuous hardware modifications during DSE. On the other hand, to evaluate an architectural alternative in DSE, an optimized software version has to be available to provide a realistic workload. As a consequence, the architects have to continually rework the target applications before, during, and after the architectural/micro-architectural DSE. This ever tighter intertwined software/hardware design impacts the performance evaluation techniques as follows.

- Firstly, application design options increase quickly due to the diversity of the communication/synchronization hardware/middleware, computation re-parallelization among the processing elements, exploration of various compiler optimization switches, and so on. To enumerate these options, a huge number of iterations for performance evaluation are needed, with the assumption that the applications' source code is subject to change. This poses a tough challenge to all the preprocessing-based approaches (e.g. sampling [28, 32] and statistical [6] simulation).
- Secondly, precise timing information is required at runtime. Otherwise the *Multi-Processor* (MP) task execution patterns may deviate from what they are in reality [10]. Post-processing based approaches (e.g., trace-driven simulation [31]) cannot cope with this requirement easily.
- Last but not the least, optimizations have a large impact on the performance of the applications. For example, MPSoC software design without considering compiler optimizations or inline assembly coding on each single processor can hardly be advisable because of the distorted MP task execution patterns. Assuming the software is purely written

in C or is not optimized, a lot of source-level performance estimation approaches (e.g., [17]) have their practical limitations.

We analyze the current problems and believe a fast-yet-not-that-accurate simulator can be a good solution for performance optimization and coarse grained DSE, while fine grained DSE and performance verification can be resolved by the existing cycle accurate simulators. We propose a hybrid simulation framework named *HySim*, which combines a detailed *Instruction Set Simulator* (ISS) and a native execution engine called *Virtual Co-Processor* (VCP). The application is partitioned and mapped to these two simulation engines to facilitate high simulation speed with good accuracy. The concept of VCP is close to but not the same as native emulation. Program segments on VCP are transformed so that they can access resources (e.g., on-chip memory and registers) on the ISS, but at the same time perform computations natively. Therefore this program transformation is called *virtualization*.

As shown in Table 1, HySim fills the gap between source-level performance estimation and instruction accurate simulation. It can be applied to applications consisting of C-code, assembly functions or close source libraries (refer to `OBJ` in the table). RISC, DSP and VLIW architectures can be supported in terms of accurately obtaining the performance.

Techniques	Features			Major Usages			
	Speed	Accuracy	Applicability	Functional Verification	Performance Optimization	Performance Verification	DSE
Natively Compiled Execution			Portable C	●			
Source-Level Performance Estimation			Portable C Simple ISA	●	●		
HySim			C + OBJ RISC/DSP	●	●		●
Instruction Accurate Simulation			Generic	●	●		●
Cycle Accurate Simulation			Generic	●	●	●	●

Table 1: Comparison of Simulation Levels

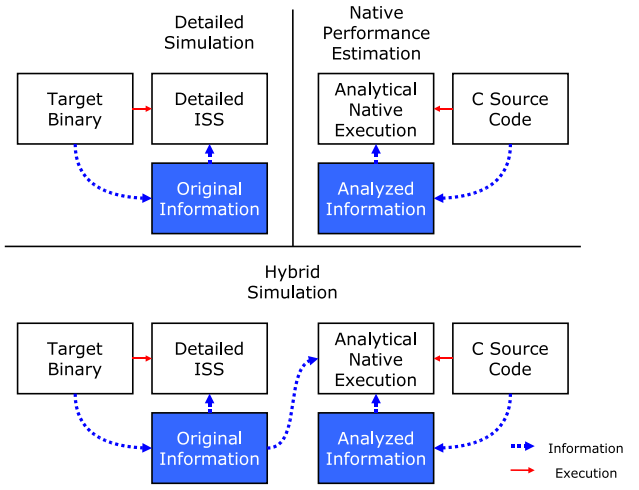


Figure 1: HySim Advantage

Performance estimation is introduced to the segments mapped to the VCP. Unlike pure native performance estimation tools (e.g., μProfiler [11]), the information contained in the target object is obtained to facilitate more accurate performance estimation. As illustrated in Figure 1, although detailed instruction set simulation processes the target binary in a relatively low speed, precise performance information can be obtained since the target binary contains the original information to reflect the program's details. Natively executing the C source code facilitates high speed, but the information analyzed from the source code is not precise/detailed enough to provide good accuracy. In hybrid simulation, the analytical native execution engine process the C source code at a high speed, And at the same time, by utilizing the original information contained at the target binary, better performance information can be provided. Concerning with the generality, hybrid simulation supports applications contain inline assembly and close source libraries, thus the practical adaptability is also improved.

The performance estimation has been realized previously by two different approaches [7]: *annotation based* and *dynamic profiling based*. The former analyzes the source code of the application, and annotates/instruments operation cost and memory accesses back to it. These annotations and instrumentations are evaluated at runtime, generating performance information representing processor execution cycles and memory accessing latencies respectively. The dynamic profiling based approach is based on a novel trace-replay technique named *cross replay*, this technique was originally introduced to address the cases where annotation based approach is not applicable (e.g., performance estimation for VLIW architectures). Both these approaches provide accurate timing information at runtime, so they fulfill the requirement of MPSoC architecture simulation.

In the current paper, the performance estimation work is further extended, and the contributions are as follows: This paper presents the relationship and interdependence between the virtualization and performance estimation modules. After solving some design/implementation issues, the two previous performance estimation approaches are integrated together in a single hybrid processor simulator. We also present that sampling can be used to effectively combine these approaches. Some preliminary results are shown to prove our conclusion and point out our future direction.

Like all approaches, HySim, especially its performance estimation, has certain limitations, and the most important ones are highlighted here:

- HySim is an application simulator, which means it cannot simulate a modern operating system yet. It does not support self-modifying code, but self-referential code is not a problem.
- Neither annotation based nor dynamic profiling based performance estimation works for superscalar machines. Singlescalar processor with super pipeline is also not well supported so far.
- Only one program language, C, is studied. It does cover a large portion of applications in the embedded domain, but this limits the usage in different areas.
- Another problem we have considered is how HySim can be adopted to a new architecture. Applying HySim does not require recompile the application using the target compiler. Instead, the segments executed on VCP are derived from the source code of the application, and transformations are to keep the ISS execution unaffected. Understanding the calling conventions and aggregate data structure padding used in the target compilers are necessary, but apart from that, the compilers can be completely regarded as black boxes.
- HySim also requires the interfaces of accessing/modifying registers memories of ISSes being exposed. Most of the existing ISSes fulfill this requirement.

The rest of this paper is organized as follows. Section 2 reviews the related work. Section 3 introduces the HySim framework by using an intuitive example. Before talking about the integration of the two performance estimation methods, we briefly introduce the dynamic profiling based approach at Section 4. The actual integration and the dynamic sampling profiling are presented at Section 5. The experimental result is given in Section 6, and Section 7 concludes this work.

2. RELATED WORK

Performance evaluation is an essential aspect in architectural and micro-architectural DSE. To obtain performance information effectively, various techniques are proposed in recent works. We introduce three different techniques: *fast instruction set simulation*, *partial simulation*, and *performance annotation*, and compare them with HySim.

2.1 Fast Instruction Set Simulation

Instruction set simulation can be broadly classified into interpretive, statically compiled and dynamically compiled approaches. Interpretive simulation is the basic technique which is flexible but slow. E.g., SimpleScalar [2] is a retargetable interpretive application simulator widely used in performance estimation for scientific researches [27, 28, 32].

Compiled simulation significantly elevates computation efforts (e.g., instruction fetching and decoding) from run-time to compile-time to improve the simulation speed. Normally, target binaries are transformed to native code in these statically compiled approaches [3, 34].

Static compilation based approaches have their limitation in supporting self referential/modifying code. The dynamically compiled counterparts [25, 4, 26, 22, 24] are introduced for this sake. For example, A *Just-In-Time Cache Compiled Simulation* (JIT-CCS) technique that combines retargetability, flexibility and high simulation speed is presented by Nohl *et al.* [22]. The "compilation" of target binary takes place at run time and the result is cached for reuse. The original target binary is still available for referencing and modification. If the instruction at a particular address is changed, the corresponding compilation cache is invalidated and a new compilation for this modified code is triggered on demand. A multi-processing approach [24] is proposed to benefit from the more and more popular MP host. The heart of the approach is a dual-functional simulation engine which combines interpretive and compiled simulation. To simulate an application, the simulator does not have to be paused to wait for the compilation result. Instead, in a Chip-Multi-Processor host the simulation proceeds in one processor (using interpretive engine), while another processor is undertaking the compilation.

Dynamic (also known as *Just-In-Time* (JIT)) binary translation can be regarded as a subclass of dynamically compiled simulation. It brings the simulation speed to a new height by translating the target binary into native instructions directly. For example, [30] introduces a simulator with more than 1000 MIPS peak simulation speed for the ARC processor. This extremely high speed is achieved by translating target instructions directly to native machine's instructions. However, this technique is ISA adaptable, and it is still not proved whether this approach can be applied to complex architecture (e.g., DSP) simulation. PTLsim [33] is an x86 cycle accurate simulator. The accuracy is achieved by translating the target x86 instructions to native x86 micro-instructions, i.e., the translation lowers the binary to obtain low level information. PTLsim also has a native emulation mode, in which the target x86 instructions are executed directly. In this sense, PTLsim is also a hybrid simulator. *Dynamic Binary Instrumentation* (DBI) tools, e.g., Pin [14], can also be used for simulation purpose [16]. Not like their binary translation brothers, they concentrate on instrumenting new functionalities to the binary. For Pin, the instrumentation largely keeps the original instructions when adding these functionalities, but this is not necessarily true for other DBI tools (e.g., Valgrind [21]).

2.2 Partial Simulation

As a consequence of the increasing complexity of target architectures, partial simulation techniques are proposed to obtain performance information of the whole application without having to simulate it to completion. The most famous approaches are *sampling simulation* and *statistical simulation*, which are orthogonal to fast instruction set simulation techniques.

Sampling simulation selects portions of the whole execution of the applications for detailed simulation. These portions are selected periodically or analytically.

SMARTS [32] is a periodical sampling micro-architecture simulator. Functional simulation can be used to fast-forward the execution until samples are met. After warming up, detailed simulation is performed on these samples, and the obtained performance information is used to extrapolate that of the whole application. Sherwood *et al.* [27, 28] employ analytical sampling, in which representative samples are selected by analyzing the similarity of execution traces represented by basic block vectors. A basic block vector represents the sequence of ever executed basic blocks, and it can be obtained by performing a functional simulation at the preprocessing phase. Machine learning techniques are applied to cluster the basic block vectors into a set of phases, henceforth, only one representative of each phase has to be simulated in detail to estimate the overall performance of an application. Functional simulation or checkpointing [29] facilitates fast forwarding simulation to these representatives. Recently, sampling simulation is also introduced in MP simulation domain [23, 20].

Instead of selecting samples from the application, a synthetic trace can be generated to represent the performance of the application. E.g., in [6], the simulation is performed once for profiling purpose. Afterward, using the obtained statistical profiles, a trace is synthesized á la Monte Carlo. This trace does not represent any functionality of the original application, and is not even an existing piece of code in the given binary. However, simulating it gives a performance profile resembling that of the original binary with high similarity.

Muttreja *et al.* propose a hybrid simulation technique [18, 19] to tackle the performance/energy estimation problem of single processors. In their solution, some parts (in fact, some functions) of a C application are executed on the native host, whereas the rest runs on an ISS. Since native execution is much faster than instruction set simulation, significant simulation speed improvement can be achieved if the natively executed parts are the hotspots of code. Power consumption and performance estimation are also available, but in order to get them an energy/performance model should be built by training with the input applications.

One major limitation of the approaches presented in this subsection is that a considerable amount of *preprocessing* is needed for various purposes (e.g., discovering the representative phases of the target application). As having been described, in MPSoC design, partitioning and parallelizing the application is a key factor in exploiting the system performance. Such kind of "software exploration" is not finalized *before* architectural exploration. Instead, they are intertwined in the lifespan of DSE. Eliminating (or at least reducing) the effort of preprocessing is desired.

2.3 Performance Annotation

An alternative to instruction set simulation is to annotate performance information into the application's source code and directly compile/execute it at the native environment. During the native execution, the previous annotated information is used to calculate the performance of the application.

One of such work is μProfiler [11]. The C source code is first lowered to a *3 Address Code Intermediate Representation* (3-AC IR) format where all the operations, including all the non-scalar variable accesses, global variable accesses and control transfer statements, are explicit. A set of machine independent optimizations, such as constant propagation, constant folding, dead code elimination etc., are performed to remove redundant operations so that the IR is closer to the realistically optimized target binary. The optimized IR is then analyzed to estimate the operation cost of each basic block. These costs are then annotated back to the IR, which in turn are natively compiled and executed to estimate the performance of the application. A memory access trace is also generated during the native execution. Cache simulation is performed to the trace afterwards. However, this approach has several weaknesses. Firstly, it assumes that the application only contains C source code. Secondly, the analysis is only performed on the IR, which does not carry enough information to represent the target binary. Last, it is only applicable for RISC like processors and does not support super-scalar or VLIW architectures.

Meyerowitz *et al.* [17] propose a performance annotation technique for heterogeneous MPSoCs. Target binaries are first simulated on cycle-accurate simulators, from which timing information is obtained and annotated back to the original C code at source-line level. Thereafter, the SystemC simulation can be performed on these annotated C code to facilitate fast whole system performance simulation. However, due to the absence (or inaccuracy) of line-to-line debugging information, optimizations on the target binaries are not allowed yet, which limits the practice of this approach.

Another method [13] is to analyze the target binary and *generate* C code out of it. Note that the generated C code is not the C source code but a much low level representation that contains precise timing information. It can be executed alone to produce the performance information, or it can even be co-executed with the original C source code. From the usability and generality viewpoints, this approach is very attractive. However, the effort of developing a binary-to-C translator should never be underestimated. The HySim framework is conceptually close to this approach, but we are attacking the same problem from an opposite direction - generating the performance information contained C code from the original C source code. Naturally, their approach can produce precise performance information, which is not a trivial task for HySim. To compare the simulation speed, we setup a simulation environment similar to them, which is a HySim simulator for MIPS-32 without cache simulation. As reported in their paper, their approach results in 13 MIPS simulation speed on a 500MHz Pentium II workstation. As a comparison, HySim runs at 70-160 MIPS with a Athlon64 X2 4600+ processor (does not benefit from the power of multi-processor), which is roughly in the same magnitude.

3. HYSIM FRAMEWORK

This section provides an overview of the HySim framework, paying especial attention to performance estimation. For virtualization purpose, HySim transforms the C source code of the applications. The transformation is performed in the HySim *frontend*, which is shown in Figure 2.

An application normally contains machine dependent parts (e.g., close source libraries, inline assembly code) and machine independent C code. The latter is passed to the instrumenter for transformation. Generated from the instrumenter, the output is still C code and can be, in turn, natively compiled and executed in the VCP. VCP - the virtual machine - is conceptually a coprocessor of the original ISS and can access ISS's memory/registers. ISS and VCP execute in a mutually exclusive way. When VCP finishes its execution, the side effect is always updated to the ISS to keep the program functionally correct. E.g., if a global variable is modified at VCP, the corresponding memory at ISS is updated.

To take a closer look at the instrumentation process, we first discuss C application virtualization, which is a key module in HySim. Afterward, the components of the annotation based performance estimation approach, operation cost annotation and memory reference instrumentation, are given. The last subsection puts everything together by giving an intuitive example and discussing the relationships/interdependence between these modules.

3.1 C Application Virtualization

C is the dominant programming language in embedded domain, and C applications are widely used to drive architectural and micro-architectural DSE. If a C application designed for one architecture can simply be recompiled to another architecture without affecting the functionality, this application is said to be portable (or target independent). Non-portable (target dependent) applications normally contain these elements but not limited to: inline assembly code, close source libraries, or irregular memory mapping.

However, from a target dependent application, one can almost always figures out target independent parts. For example, Figure 3 is an application which combines C code and assembly code (`asmpower`). The C function `slowpower` does not contain any machine dependent code, thus we expect that it can be executed

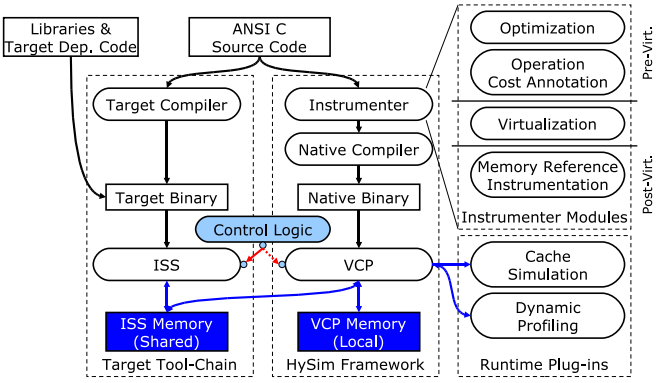


Figure 2: HySim Workflow

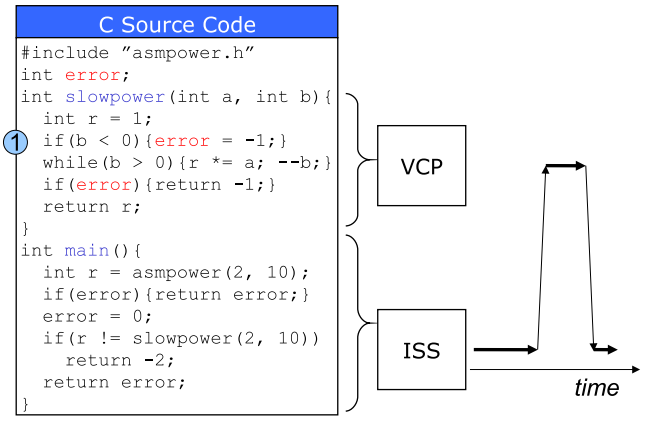


Figure 3: Example of Hybrid Simulation

natively to facilitate higher simulation speed. However, as shown in ①, a global variable `error` is accessed in this function. To synchronize the global variable between the simulated part and natively executed part, we cannot crudely cut the code and natively compile it. Moreover, input arguments and outgoing return value should also be tackled.

We address this problem by virtualizing the function (in this example, `slowpower`). Each function (or any entity in a program) consists of *computation* and *resource accessing*. In HySim, virtualization is a transformation on the functions so that all the computations are performed natively while the resource accessing is redirected to the ISS to meet the synchronization requirement. For example, global variables are accessed from the ISS by calling some *service routine*s, and stubs are created to address the argument passing and value returning problem. We will show an example to illustrate it better in the last subsection.

To summarize, the C virtualization technique can also handle the following [8]:

- Create native clones for constant variables to reduce the number of ISS resource accesses.
- Support C89 compliant pointers, including local, global, and function pointers.
- Support C standard library, except for inter-procedure jump functions (`longjmp`, `setjmp` etc.).

3.2 Operation Cost Annotation

For a singlescalar RISC machine, usually, a given number of instructions are needed for implementing a C operation. Generally, this number is known at compile time through analysis and can be defined as the *cost* of the corresponding C operation. Therefore, if the execution frequencies of all the basic operations in a piece of C code and their respective costs are known, the cycle count for executing them can be easily computed.

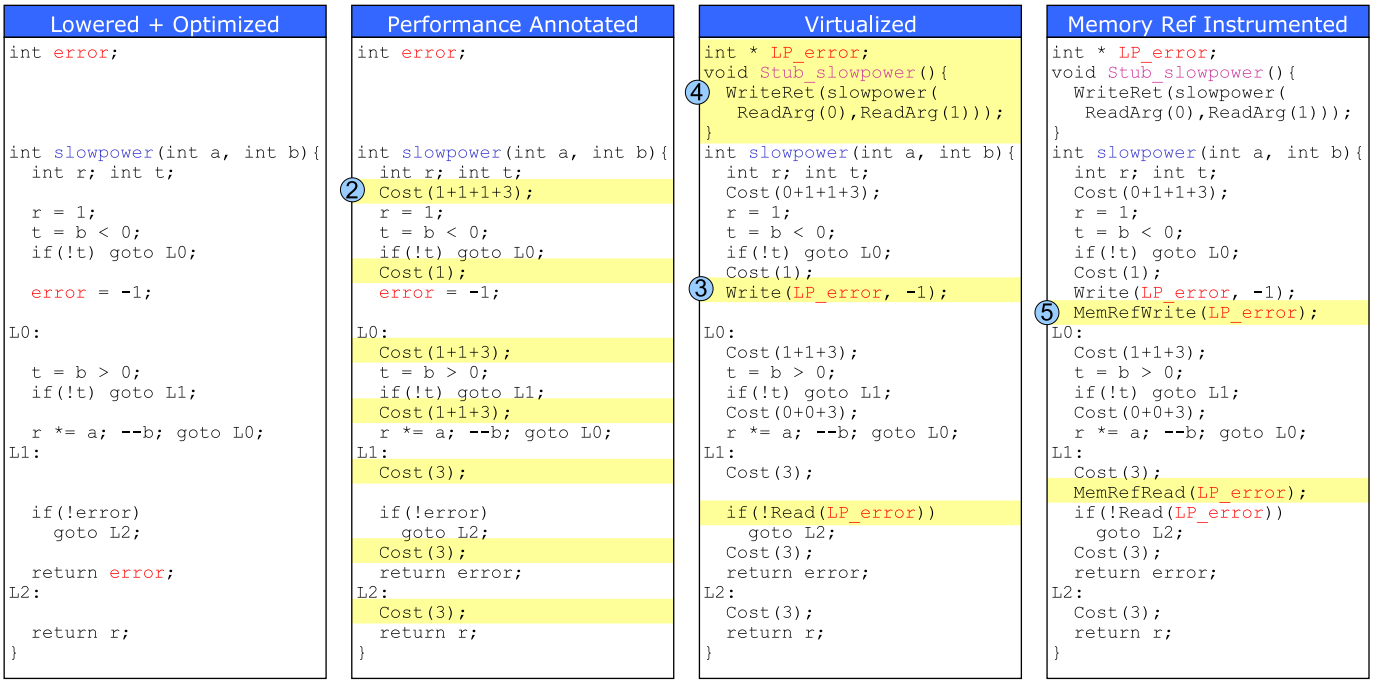


Figure 4: Example of HySim Transformation

We define a *cost file* for the target architecture, which is a table indexed by C operations and the types of the operands. For example, an integer `plus` operation takes 1 cycle and a conditional jump operation 3 cycles (although instructions can be scheduled to the delay slot, our experiments show 3 is still a better estimation than 2, 1 or 0) for the MIPS 4K architecture.

The C source code is first lowered to convert the high level control structures (e.g., `if-else` statements) and data structures (e.g., `struct`) to low level forms (e.g., `goto` statements, which can be one-to-one mapped to machine assembly code; primary built-in data types, for which operation costs can be located from the cost file). Machine independent optimizations are applied to the lowered IR to get similar performance estimation as realistic target applications. For each basic block, the operation costs are accumulated and annotated back to the IR. At execution time, the annotated costs can be collected for performance estimation. The lowering, optimizations and operation cost annotation are performed before virtualization, since virtualization introduces new statements which should not be considered as in operation cost annotation. As shown in Figure 2, these processes are executed at *pre-virtualization* phase.

3.3 Data Cache Simulation

For the sake of accurate performance estimation, it is extremely important to take cache simulation into account, since the memory subsystem is a major performance bottleneck in many modern processors. Previous works (e.g., [11]) evaluate the memory subsystems by analyzing the memory access information at the source code level, generating memory traces and simulating them using cache simulators. There are two major demerits of this method. Firstly, native addresses of the variables are used for cache simulation. These addresses only reflect the collisions in memory referencing, but not the actual memory layout which is also an important factor in cache simulation w.r.t. cache-line fetching and association. Secondly, performance estimation is only possible offline by replaying the memory trace afterwards. This is a major problem in MPSoC simulation, where inaccurate timing can deviate task scheduling [10] and affect the overall performance estimation adversely.

The cache simulation in this work addresses the latter problem by simulating the memory references at runtime (As shown in Figure 2 cache simulation is a runtime plug-in for the VCP).

The solution of the former issue deserves more explanation. As shown in Figure 2, memory reference instrumentation is a *post-virtualization* pass. Thus, we can reuse one feature provided by the virtualization, that global variables are accessed from the ISS by using the service routines. Therefore, the actual (instead of native) addresses can be obtained and used for cache simulation. Additionally, an *Address Recovery Layer* (ARL) [7] is needed to translate some addresses that cannot be used in cache simulation directly. For example, for constant global variables, since native clones are created for fast access, before passing addresses to the cache simulator ARL translates the native clones' addresses to the actual ones.

3.4 Putting Them All Together

To help the understanding of HySim framework, this subsection gives an example of applying instrumentation to the function `slowpower` in Figure 3.

As presented in Figure 4, the source code of `slowpower` is lowered and optimized. Then, for each basic block the corresponding operation cost is looked up from the cost file, accumulated and back annotated. For example, ② is the cost annotation for the first basic block, being reckoned as 6. Note that a better way is to define the cost to 5 because the instruction `r = 1;` can be scheduled to the delay slot after a branch instruction. This is not implemented in HySim but we plan to address it in future.

Afterward, the IR is virtualized. Global variable `error` is accessed through calling of a service routine (③). Note that the pointer `LP_error` (literally *Linkage Pointer* of `error`) is created by the virtualization module and actual address of `error` is assigned to it at load time.

To pass the arguments and to get the return value, a stub is created (④). While switching from simulation mode to the invocation of `slowpower` at VCP, `Stub_slowpower` is first executed, which tackles the actual calling of the virtualized `slowpower`. Stubbing makes sure that the virtualized functions can not only be called from ISS but also by other virtualized functions.

Memory reference instrumentation annotates the referencing of the global variables using the actual addresses (or recovered addresses [7]). For example, (⑤) in the figure is an instrumentation

of memory access. Local variables are always considered to be allocated to registers. This assumption is overoptimistic but we still got fair results in experiments.

While running the simulation, VCP performs performance estimation by utilizing both operation cost annotation and memory reference instrumentation. The cache simulation indicates hit or miss for each memory access. The following formula is thereby used to compute the total estimated cycles:

$$Cycles = \Sigma_{i=1}^{n} N_i \times C_i + N_{hit} \times C_{hit} + N_{miss} \times C_{miss}$$

where N_i and C_i are the execution count and cost, respectively, for C operation i. N_{hit} and N_{miss} are the estimated cache hits and misses, while C_{hit} is the cost of a hit, and C_{miss} is the penalty of a miss.

To summarize, virtualization distorts operation cost annotation but it is a prerequisite for memory reference instrumentation, therefore the calling sequence of these modules is selected as presented.

4. DYNAMIC PROFILING

Dynamic profiling is another performance estimation approach proposed in HySim. It was originally introduced [7] to address the performance estimation issue for DSP/VLIW architectures.

Architectures with domain specific features (DSPs, NPUs, VLIWs) are often used in MPSoCs for speeding up the computation intensive parts of an application. For such architectures, the quality of the code heavily depends on the target dependent optimizations of the target compilers. Unless the whole compiler back-end is re-implemented, these optimizations cannot be imitated. As a consequence, the annotation based approach is not applicable for these processing elements.

Fortunately, some assumptions about the nature of the DSP/VLIW architectures and the applications running on them can significantly simplify the problem. Firstly, many of such architectures have no affiliated cache. Therefore, the execution time of a specific control path in such architectures is always the same (i.e. it does not depend on the memory access patterns). Secondly, the code segments running on such architectures often have high data workloads but limited number of control paths (i.e. they contain limited number of `if-else` statements, loops with statically known iteration bounds etc.). So it is possible to infer the execution performance for such architectures by enumerating each control path, and then calculating *once* the cost of each control path. This is implemented using a dynamic profiling technique called *cross replay*.

The overall workflow for cross replay is presented in Figure 5 which shows the execution of a virtualized function on the VCP. While simulating a function on the VCP, an execution trace is generated which uniquely enumerates the control path (referred to as a *scenario*) taken during execution. Once the execution of the virtualized function finishes, the scenario is searched into a *database*. If the scenario is not found in the database (a miss), then the part of the function that has been executed in virtual mode is *replay*ed on the ISS to obtain and record its cycle cost in the scenario database. If the scenario is already in the database (a hit), it means it has been previously simulated on the ISS and its cost has been recorded. In such a case, the performance record is retrieved from the database. Since the application tracing is done for each function *on-the-fly*, the total trace size is manageable.

The key of this approach is the *cross-ISA* trace-replay, which differentiates HySim with other trace-replay based simulation tools (e.g., Nirvana [1]). Details of cross replay were presented in [7], but to facilitate the later discussion, an example is presented.

4.1 Trace Generation

When a virtualized function is executed on the VCP, a execution trace is generated to represent the scenario. Additionally, since the virtual execution also has side effect (e.g. changing global variables' values), in order to replay the function in ISS, some records are generated dynamically as an alternative of checkpointing.

In Figure 6, when `slowpower` is invoked at VCP, the incoming

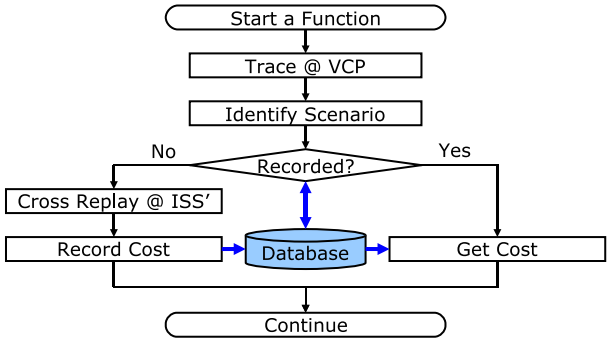


Figure 5: Cross Replay Workflow

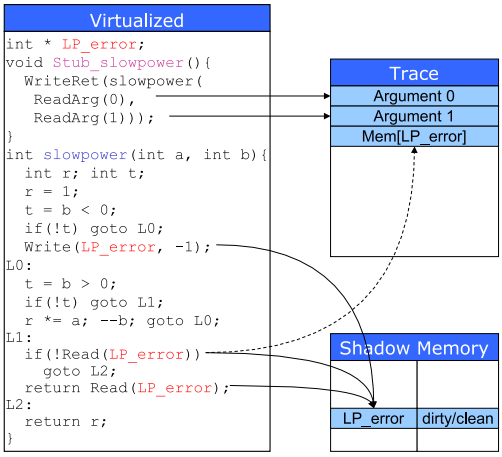


Figure 6: Cross Replay Example

arguments are recorded into the trace. We know that the execution of a function not only depends on the incoming arguments, but also on the global variables it accesses. Therefore, as the function (in this case, `slowpower`) is executed on VCP, accesses to global variables are recorded, unless the values of these variables are reproducible (e.g., generated by this function itself). A shadow memory is used to indicate the reproducibility for each word in the memory.

In the example, if `b` is less than zero, then the global variable `error` is written by the service routine `Write(LP_error, -1)`. `Write` will check the shadow memory for this address, which is at the initial status - *dirty*. Since `Write` constructs the value of this global variable, it will change the corresponding shadow memory to *clean*. Later, `error` is read twice by `Read(LP_error)`. But as the shadow memory for this global variable is clean (means the value is produced by this function itself), no trace is generated. For another situation that `b` is larger than or equal to zero, `error` will not be written, thus when the value is read at `if(!Read(LP_error))` a trace is generated to record the value of `error` (which is not produced by this function itself), and the corresponding shadow memory is marked to *clean* to indicate that the memory location has now been traced. The next `Read` will not trigger another trace generation.

4.2 Dynamic Replay

According to Figure 5, if the execution scenario of the function at VCP is not found in the database, dynamic profiling is triggered to fill the performance information into the database.

Dynamic profiling is done by *cross replay*ing the function at an instruction set simulator. In HySim the simulator for replaying is a dedicated instance of the target ISS, which is isolated from the rest of the system. The former generated trace is used to drive the replay. In the example, incoming arguments (and the value of `error`, if recorded) are loaded at the beginning of the replay. The dedicated *replay ISS* (`ISS'` in the figure) runs from the starting

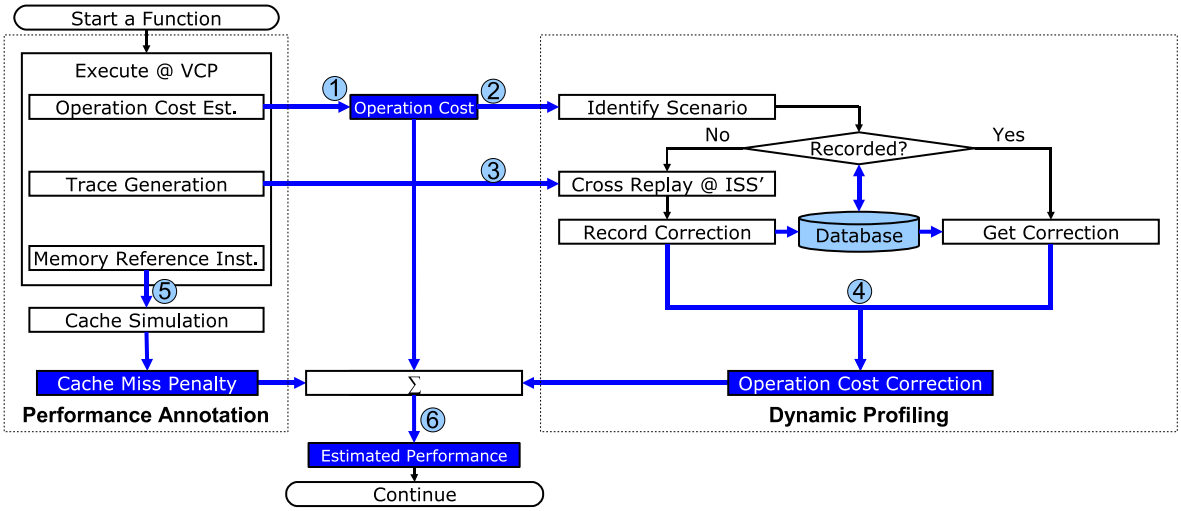


Figure 7: Integrated Approach

point of the function `slowpower` to its end, and the number of the consumed cycles is recorded thereby.

4.3 Implementation

Cross replay is realized as a runtime plug-in (Figure 2). To implement it, the service routines for virtualization are extended internally, and there is no extra instrumentation module needed.

For a floating-point clustered VLIW DSP (mAgic [15]) dynamic profiling based performance estimation is implemented [7], and the experiment indicates that the overhead of cross replay is quite marginal, by introducing 0.6% to 9.0% to the total execution time).

5. INTEGRATED PERFORMANCE ESTIMATION APPROACH

One of the focuses of the current paper is to discuss the relationships between virtualization and the two performance estimation approaches. As having been shown, both annotation based and dynamic profiling based approaches gain advantage from virtualization. In this section, we introduce the integration of these two approaches, discuss the issues in design and implementation, and introduce a simple sampling policy to the dynamic profiling approach.

5.1 Motivation

Both annotation based and profiling based approaches have their advantages and disadvantages. We compare them on the basis of two factors: runtime overhead and estimation accuracy.

Performance estimation approaches introduce overhead to the native execution. Experiments show that calculating operation cost at runtime has barely zero overhead. It can be easily understood that an operation cost accumulation operation can be inlined thereby costs only one integer addition operation. Cache simulation introduces most of the performance estimation overhead for the annotation based approach. For MIPS processor, the benchmark shows a speed reduction of 3x to 6x. Note that the overhead comes from both the address recovery layer and the cache simulator itself. Dynamic profiling has a marginal overhead for mAgic simulation, but for other ISAes, it is hard to tell how much overhead will be introduced before the implementation.

Estimation accuracy is another important factor. It not only depends on the approaches themselves but also the target ISAes. Since out-of-order execution is not considered at instrumentation time, operation cost annotation only works for the single-issue architectures in principle. However, some of these architectures (e.g., ARM11) have deep pipeline and complex interlocking mechanism, unless pervasive micro-architecture dependent analysis is performed, operation cost annotation cannot be practically adopted. Moreover, the performance of realistic applications relies on the optimizations by target compilers. Our machine independent optimizers can imitate the optimizations of most RISC compilers, but is not sufficient for those of DSP/VLIW compilers. Memory reference instrumentation also has its limits. The current assumption is that all the local variables are allocated to the registers. Register spilling, heterogeneity of register file, prologues/epilogues of functions are not taken into account when instrumenting the memory accesses. The dynamic profiling based approach shows good results on a single-issue DSP simulation (For mAgic processor, there is no error at all), but since the replay itself is performed at the dedicated replay ISS, the status of performance related micro-architectural resources (e.g., cache, branch predicator), if there is, of the original target ISS is not considered, i.e., the profiling is performed in an *unwarmed* simulator.

application	actual performance (cycles)	estimated performance (cycles)	error rate (%)
`DES`	281714740	282687692	+0.35
`MD5`	67939086	70051332	+3.11
`G721 Enc`	371733857	404845180	+8.91
`G721 Dec`	329697641	331173136	+0.45
`JPEG Dec`	24146154	21872488	-9.42
mean	1075231478	1110629828	+3.29

Table 2: Operation Cost Estimation for MIPS

On the basis of some observations, we are motivated to integrate these two imperfect approaches to a unified and better one. The following discussion is based on experiments for MIPS processor simulation. Table 2 shows the operation cost estimation result. We analyzed the applications `G721 Enc` and `JPEG Dec` as they have relative large estimation error rates. The kernel operations in `G721 Enc` are control intensive, where most of the basic blocks are rather small. I.e., the portion of branch instructions is rather high. The estimation error (due to the absence of branch delay slot scheduling) for branch operation impacts the final result significantly. As described before, the scheduling for branch delay slot is not considered, which does happen at the target compiler side, therefore it is the major source of the deviation. For another application (`JPEG Dec`), the key functions are data intensive and a lot of intermediate computation results are stored in the local variables. This causes a large number of local variables to be spilled into memory. Therefore the estimation is overoptimistic.

Although these problems stem from the imperfect annotation/instrumentation, further improvements on these modules re-

quires machine dependent optimizations, which cannot be reused easily. These improvements are possible future work, but more generic approaches are desired at present. Therefore, to address these problems, we propose introducing dynamic profiling to RISC like architectures to increase the estimation accuracy of the frequently invoked key routines.

5.2 Integration

There are several practical issues in the integration. Our primary result has shown that the overhead of cross replay is marginal for DSP/VLIW architecture simulation. But when we directly integrate this approach to RISC-like processor simulators, the speed is decreased significantly (sometimes even slower than the detailed simulation). Profiling indicates that there are two bottlenecks in the cross replay for RISC-like architectures. Firstly, the size of the shadow memory (which equals to the size of the on-chip memory) is increased by two orders of magnitude, thereby the effort of clearing it each time before tracing is much more costly. Secondly, the execution trace for scenario identification can be much larger than what it is for DSP/VLIW architectures, because more control-intensive applications are normally used in RISC-like architectures.

The first problem is solved by using a hierarchical shadow memory. An page table is created, which needs to be cleared each time a function is traced. The memory pages are indexed by the page table, and are cleared on demand when accessed. By doing this, the overhead of clearing shadow memory is reduced significantly.

The second issue is addressed by reusing the estimated operation cost for scenario identification (and therefore performance lookup). The integrated approach is presented in Figure 7. When a function is executed in the virtual mode, annotation based estimation is performed to get a rough evaluation on the *operation cost* (① in Figure 7). Instead of using the costly execution trace, the operation cost is used to facilitate the identification of different scenarios (②). Note that scenarios and operation costs are not one-to-one mapped (i.e., different scenarios can produce identical operation costs), so there might be errors in the identification. In some of our testcases, these errors do happen, but since for each function the variation of ratios of corrections and estimated operation costs is correlated, an error identification does not harm a lot on the performance estimation. The right hand side of Figure 7 resembles Figure 5, except that *cost* is replaced by *correction* (④). Correction is the difference of the estimated costs from dynamic profiling and annotation based approaches. The reason of using correction is that dynamic sampling profiling can be introduced easily. We will come back to this topic in the next subsection. As usual, the memory reference instrumentation enables cache simulation (⑤). The estimated performance of each function comes from the sum of the cache miss penalty, estimated operation cost and operation cost correction (⑥).

5.3 Dynamic Sampling Profiling

Dynamic profiling addresses the accuracy issue of operation cost estimation with a compromise on simulation speed. To balance these two important factors, we introduce sampling in the dynamic profiling.

In dynamic sampling profiling, not all the scenarios have to be profiled. To profile a scenario, some tracing overhead is put upon the VCP execution to generate the estimated performance (*estimation*), and the correction. When a number of samples are profiled, there can be a lookup table for estimation and correction. One not-sampled execution can also results the estimation without extra overhead of tracing. This estimation is in turn used to lookup the correction. If the estimation is not found in the table, the correction is inter/extrapolated using neighboring samples.

The inter/extrapolation works because the correction comes from the inaccuracy of annotation based estimation, which in turn is the consequence of predication errors of register allocation and scheduling. The nature effective scope of these factors is at function level, so for each function inter/extrapolation works.

The policy of sample selection can be various. For example, in this paper, we first profile a given number of executions for each function, later Monte Carlo approach is used to randomly select executions and profile them, after an upper limit is met, no execution is sampled anymore. The configuration of this policy is described in the result section, and we will see that the performance estimation with this simple strategy provides a fair accuracy. Better sampling technique without involving too much preprocessing/runtime overhead is one of the topic we would like to address in future.

6. EXPERIMENTAL RESULT

6.1 Experiment Setup

All the experiments have been performed on a simulation host with a Athlon64 X2 5200+ processor and 4 GB of memory, running Fedora Core version 4. The target architecture of simulation is a MIPS 4K core, implemented using LISA tools [9] (now known as CoWare Processor Designer). The pipeline of this architecture is not modeled, so it is an instruction accurate model. But both date cache and instruction cache are simulated. The instruction cache is also modeled in LISA, and is only driven by the ISS. The data cache is customized from DineroIV [5], an open source cache simulator, and is driven by both the ISS and VCP.

We choose 4 applications to evaluate the current work. `DES` is an en(de)cryption algorithm optimized for embedded system. `MD5` is an application performing checksums. `JPEG Dec` is the decoder for multiple JPEG frames (The input/output pictures are stored at native file system). And `Livermore` is a benchmark for ISA and compiler optimization. We choose it because evaluating ISA extension and compiler optimization are also important scenes of using simulators.

To simulate these applications in the hybrid way. Part of the application should be mapped to VCP for native execution. The partitioning is performed manually in this work. The names of the functions to be mapped to VCP are specified by the user in an XML description file. This approach can be inadequate for ingenuity users and we plan to address it in future. For `DES`, `MD5`, and `Livermore` the policy of partitioning is to select the computational hotspots to be executed on VCP. `JPEG Dec` is a relative large application, and we map the *Inverse DCT* (IDCT) procedure to VCP. Note that to simulate `JPEG Dec` application the performance bottleneck is at file system access, but the corresponding functions (`fread`, `fwrite`) are non-virtualizable [8] thereby cannot be mapped to VCP.

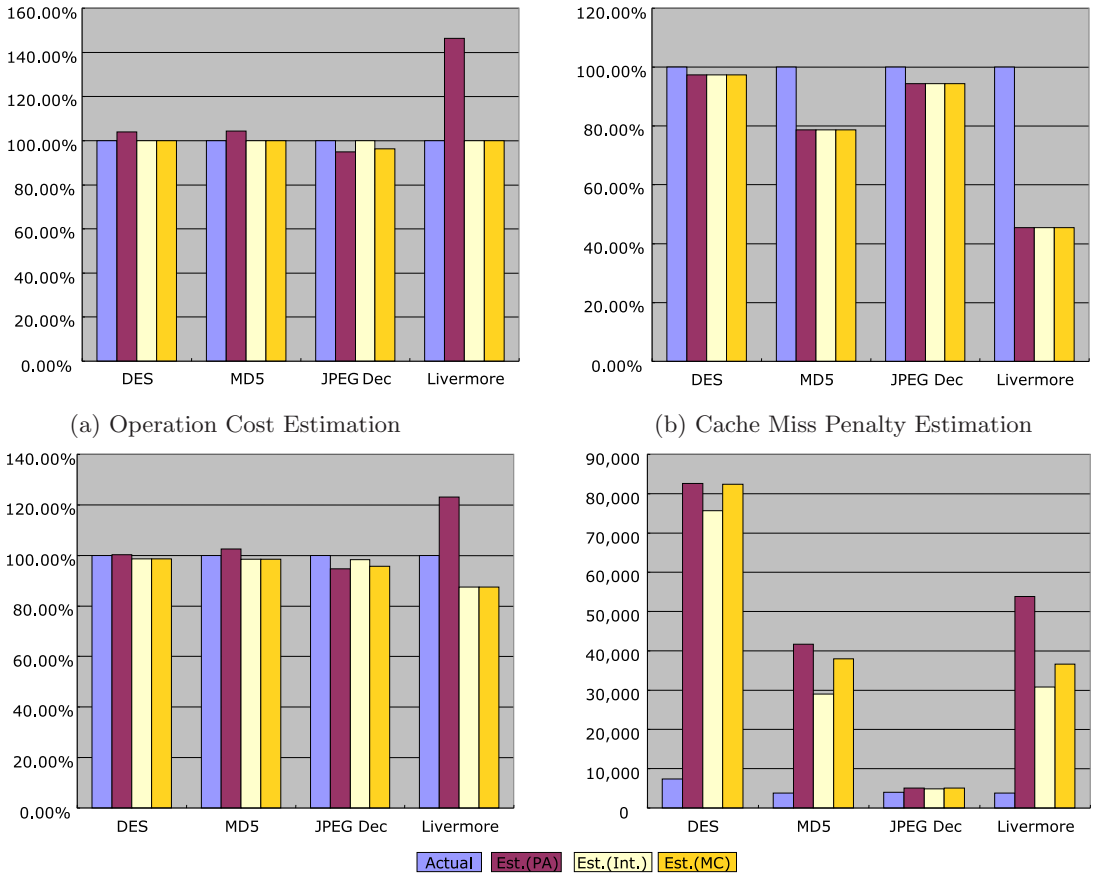
The Monte Carlo sampling is configurable. In this experiment, the sample rate is 1/100. Initially, 5 executions are profiled, and there is a sampling upper limit (equals to 100) for each function.

6.2 Result

We evaluate 4 configurations of simulation for each application: The original detailed ISS (`Actual` in those figures), annotation based performance estimation in HySim (`Est.(PA)`), the integrated performance estimation (`Est.(Int.)`), and the performance estimation using Monte Carlo dynamic sampling profiling (`Est.(MC)`). The results for operation cost, cache penalty, overall performance (the sum of the former two), and simulation speed are presented in Figure 8.(a) to (d) separately.

From Figure 8.(a), we can see the original annotation based performance estimation is very accurate for the first two applications. For `JPEG Dec`, although it looks like the estimation is accurate, the close estimation result is a consequence of relatively small portion of code being mapped to VCP. And for `Livermore`, since it is a dedicated benchmark for evaluating ISA and compiler optimization, there are a lot of opportunities for machine dependent optimizations (e.g., register allocation), and the estimated performance by using annotation based approach with machine independent optimizations has a rather significant deviation.

By integrating dynamic profiling to the estimation framework, the accuracy for operation cost is improved dramatically, but the simulation speed is sacrificed. As shown in Figure 8.(d), the speed drawback is from 5.0% to 42.7%. From the same figure, we can see that sampling improves simulation speed for all these cases, and after adopting sampling, the speed slowdown becomes from 0.3% to 31.9%.



(a) Operation Cost Estimation

(b) Cache Miss Penalty Estimation

(c) Overall Estimated Performance

(d) Simulation Speed (KIPS)

Figure 8: Experimental Result

The introduction of sampling does not affect operation cost estimation accuracy significantly except for `JPEG Dec`. The estimation error rate of this application increases from 0.0% to 3.8% when sampling is used, but still less than pure annotation based approach, which gives an error rate of 5.1%. The reason is the first 5 samples (recall our upper limit) happen to process blocks with a lot of zeroes, IDCT for which is faster than average. We expect better sampling policy can be the solution to attack this problem.

Another issue reflected from the results is the significant error rate on estimating cache misses for `Livermore`. After analysis, we discover that the error also comes from register allocation. A large amount of local variables are spilled to memory, causing a lot of cache misses. The integration of dynamic profiling to annotation based performance annotation only improves the operation cost accuracy, but cache simulation accuracy is not touched. We would like to address this problem in future.

7. CONCLUSION

Hybrid simulation improves traditional instruction set simulation by mapping part of the application to a native execution engine to be executed at virtual mode. Performance estimation at the virtual mode utilizes information analyzed from the target binary to produce accurate results. Two primary performance estimation approaches: annotation based and dynamic profiling based, are discussed in this paper, and their relationships with virtualization are described. This work integrates these two approaches to further improve the operation cost estimation accuracy for RISC-like architecture simulation. We address both design and implementation issues during the integration.

The integrated approach is slower compared to the pure annotation based one. Dynamic sampling profiling is introduced to provide a balance of the speed slowdown and accuracy improvement. Experimental result shows that for the integrated approach with dynamic sampling profiling, a 0.1% average error rate on operation cost is achieved, while the simulation speed is improved by 8 times compared with the detailed simulation.

8. ACKNOWLEDGMENTS

This work is supported by the European project SHAPES (`www.shapes-p.org`) and the HiPEAC Network (`www.hipeac.net`).

9. REFERENCES

[1] 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.

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

[3] 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.

[4] 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.

[5] J. Edler and M. D. Hill. Dinero IV Trace-Driven Uniprocessor Cache Simulator "http://www.cs.wisc.edu/ markhill/DineroIV/".

[6] L. Eeckhout, K. de Bosschere, and H. Neefs. Performance analysis through synthetic trace generation. In *ISPASS '00: IEEE International Symposium on Performance Analysis of Systems and Software*, 2000.

[7] L. Gao, K. Karuri, S. Kraemer, R. Leupers, G. Ascheid, and H. Meyr. Multiprocessor performance estimation using hybrid simulation. In *DAC '08: Conference on Design Automation*, 2008.

[8] 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*, 2007.

[9] A. Hoffmann, O. Schliebusch, A. Nohl, G. Braun, O. Wahlen, and H. Meyr. A methodology for the design of application specific instruction set processors (ASIP) using the machine description language lisa. In *ICCAD '01: Proceedings of the 2001 IEEE/ACM international conference on Computer-aided design*, 2001.

[10] J. Jung, S. Yoo, and K. Choi. Fast cycle-approximate MPSoC simulation based on synchronization time-point prediction. *Design Automation for Embedded Systems*, 11(4):223–247, December 2007.

[11] 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.

[12] 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.

[13] M. T. Lazarescu, J. R. Bammi, E. Harcourt, L. Lavagno, and M. Lajolo. Compilation-based software performance estimation for system level design. In *HLDVT '00: Proceedings of the IEEE International High-Level Validation and Test Workshop (HLDVT'00)*, page 167, Washington, DC, USA, 2000. IEEE Computer Society.

[14] 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.

[15] mAgic DSP. www.atmel.com.

[16] C. McCurdy and C. Fischer. Using pin as a memory reference generator for multiprocessor simulation. *SIGARCH Computer Architecture News*, pages 39–44, 2005.

[17] T. Meyerowitz, M. Sauermann, D. Langen, and A. Sangiovanni-Vincentelli. Source-Level timing annotation and simulation for a heterogeneous multiprocessor. In *DATE '08: Conference on Design, Automation and Test in Europe*, 2008.

[18] A. Muttreja, A. Raghunathan, S. Ravi, and N. K. Jha. Hybrid simulation for embedded software energy estimation. In *DAC '05: Proceedings of the 42nd annual conference on Design automation*, pages 23–26, New York, NY, USA, 2005. ACM.

[19] A. Muttreja, A. Raghunathan, S. Ravi, and N. K. Jha. Hybrid Simulation for Energy Estimation of Embedded Software. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2007.

[20] J. Namkung, D. Kim, R. Gupta, I. Kozintsev, J.-Y. Bouget, and C. Dulong. Phase guided sampling for efficient parallel application simulation. In *CODES+ISSS '06: Proceedings of the 4th international conference on Hardware/software codesign and system synthesis*, pages 187–192, New York, NY, USA, 2006. ACM.

[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] E. Perelman, M. Polito, J.-Y. Bouguet, J. Sampson, B. Calder, and C. Dulong. Detecting Phases in Parallel Applications on Shared Memory Architectures. In *IEEE International Parallel and Distributed Processing Symposium*, June 2006.

[24] 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.

[25] 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.

[26] 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.

[27] 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.

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

[29] 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.

[30] N. Topham and D. Jones. High speed CPU simulation using JIT binary translation. In *MoBS '07: Workshop on Modeling, Benchmarking and Simulation*, 2007.

[31] T. Wild, A. Herkersdorf, and R. Ohlendorf. Performance evaluation for system-on-chip architectures using trace-based transaction level simulation. In *DATE '06: Proceedings of the conference on Design, automation and test in Europe*, pages 248–253, 3001 Leuven, Belgium, Belgium, 2006. European Design and Automation Association.

[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] M. T. Yourst. Ptlsim: A cycle accurate full system x86-64 microarchitectural simulator. In *ISPASS '07: IEEE International Symposium on Performance Analysis of Systems and Software*, 2007.

[34] 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.