Multiprocessor Performance Estimation Using Hybrid Simulation

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

ABSTRACT

With the growing number of programmable processing elements in today's *MultiProcessor System-on-Chip* (MPSoC) designs, the synergy required for the development of the hardware architecture and the software running on them is also increasing. In MPSoC development environment, changes in the hardware architecture can bring in extensive re-partitioning or re-parallelization of the software architecture. Fast and accurate functional simulation and performance estimation techniques are needed to cope with this co-design problem at the early phases of MPSoC design space exploration. The current paper addresses this issue by introducing a framework which combines hybrid simulation, cache simulation and online trace-driven replay techniques to accurately predict performance of programmable elements in an MPSoC environment. The resulting simulation technique can easily cope with the continuous re-organizations of software architectures during an *Instruction Set Simulator* (ISS) based design process. Experimental results show that this framework can improve system simulation speed by 3-5× on average while achieving accuracy closely comparable to traditional ISSes.

Categories and Subject Descriptors

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

General Terms

Design

Keywords

HySim, Hybrid Simulation, MPSoC, Performance Estimation, Address Recovery, Cache Simulation, Cross Replay

1. INTRODUCTION

Over the past few years, the exploding complexities of multimedia, telecommunication and consumer electronic applications have prompted embedded system designers to increasingly look into MPSoC based solutions. Many embedded applications display a large amount of task-level parallelism which can be effectively exploited by such multiprocessor architectures. The continuous technology scaling trend is also fueling this paradigm shift by offering an ever increasing amount of silicon area for integration of several programmable processors on a single SoC.

The design and development of an MPSoC is a daunting proposition. It involves *simultaneous* development of the hardware architecture (i.e. selection of the proper set of processors and the communication infrastructure) and the parallelized software. This is an enormously complex task, since any change in the hardware infrastructure may require a completely new set of partitioning and parallelization schemes for the associated software. A key to successfully accomplishing this task is comprehensive design space exploration, which involves simulation of different implementation alternatives to verify functional correctness and evaluate performance. Since the speed of such an exploration process is greatly dependent on the simulation speed, fast and accurate simulators for MPSoCs are urgently needed.

This paper presents a novel MPSoC hybrid simulation framework for fast and accurate performance estimation and functional verification. The main focus is on introducing techniques to improve performance estimation accuracy of individual processors. These techniques combine novel *hybrid simulation*, memory subsystem modeling, and trace-driven replay methods to provide fairly accurate performance predictions for a wide range of processors - from simple RISCs to more complex DSPs and VLIW machines - with varying *Instruction Set Architectures* (ISA) and microarchitectures. The processor simulation techniques are bound together by a model of the communication architecture for controlling system level synchronization and providing overall performance statistics.

The rest of the paper is organized as follows. In section 2 we present a brief overview of the existing body of literature that tries to tackle different issues in MPSoC simulation. In section 3, we provide an overview of the hybrid simulation framework that forms the base of the current work. The next two sections (4 and 5) present how the simulation framework has been extended for performance estimation of simple RISC, and more complex DSP/VLIW architectures. Section 6 introduces the communication and synchronization of these PEs, and section 7 shows benchmarking result. The final section summarizes our current work and provides some directions for future work.

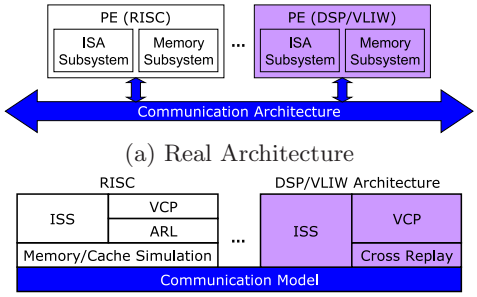
2. RELATED WORK

Figure 1.(a) shows the abstract model for an MPSoC architecture. The major components of such an architecture are the *processing elements* (PEs) and the *communication architecture*. The PEs can be of various types - dedicated hardware blocks, RISC processors, VLIWs, domain specific processors such as DSPs and Network Processing Units (NPUs) etc. The communication architecture can also be composed of a variety of components providing dedicated point-to-point or shared connections between different PEs. For our current discussion, we will only consider simulation of programmable PEs (i.e. RISCs, VLIWs, DSPs etc.).

A large volume of work already exists targeting complete MPSoC simulation [18, 7]. The focus of the current work, however, is on simulation of individual PEs which form a major bottleneck in achieving high system simulation speeds.

Traditionally, individual PEs have been simulated using ISS. A number of recent works have suggested various ISS acceleration techniques such as compiled [21] simulation, or just-in-time compiled [15] simulation. However, with the increasing complexity of MPSoCs, even such improvements are not enough to achieve the desired simulation speed.

DAC 2008, June 8–13, 2008, Anaheim, California, USA



(a) Real Architecture

(b) Layered Simulation Model

Figure 1: MPSoC Design

A further orthogonal improvement is *sampling* [17, 20] (or *statistical* [2]) simulation which infers the performance behavior of a PE by simulating selected *phase*s (or synthesized *trace*s) of an application (instead of simulating to completion). Recent works [16, 14] show that multiprocessor simulation can also benefit from sampling. One major drawback of these approaches is that a large amount of *pre-processing* is needed for discovering the phases (or generating the traces) of the target application. In MPSoCs, modifications in the numbers and types of PEs or the communication architecture might affect the organization of the software (e.g. the software needs to be completely re-partitioned or re-parallelized). In such cases, the time consuming pre-processing and training phase has to be repeated for each re-organization of the software.

Muttreja *et al.* proposes a hybrid simulation technique [12, 13] for tackling the performance/energy estimation problem of single processors. In their solution, some parts of an application are executed on the native host machine, whereas the rest runs on an ISS. Since native execution is much faster than ISS, significant simulation speed can be achieved if the natively executed parts are the most frequently executed pieces of code. The performance estimation for the natively executed code requires a training phase to build a procedure level performance model and therefore, is not flexible to easily account for software modifications. This limitation restricts the use of their technique in the MPSoC domain.

Another approach is *trace-driven simulation* which can be used to evaluate the performance [19], power consumption [4] or the memory access behavior [9] of computer systems. A trace is some information of interest generated during the execution of a program on a simple and fast simulation model. Later, analysis tools can process the traces offline in a detailed fashion. One problem with trace-driven simulation is that the generated traces might become excessively large. Another issue is that trace-driven simulation relies on post-processing and cannot provide performance information at runtime. The lack of such information can consequently change the execution schedule of tasks in MPSoC simulators [5].

This paper makes two major contributions on fast and accurate processor performance estimation in an MPSoC environment. Firstly, our framework can provide highly accurate performance estimates without any time-consuming pre-processing or training phases. Therefore, it can easily cope with frequent software re-organizations during an MPSoC design. Secondly, this framework is applicable for a wide variety of programmable PEs, and consequently, for various system designs.

3. OVERVIEW

This section provides an overview of our MPSoC simulation framework presented in Figure 1.(b). The framework breaks the system into simulation models for the PEs and simulation models of the communication architecture.

We consider two types of programmable PEs for performance estimation - single issue processors with RISC instruction sets, and processors with domain specific instruction sets (e.g. DSPs) or with higher degrees of static *Instruction Level Parallelism* (e.g. VLIWs). These two types cover a large subset of processors used in current embedded systems. This paper extends the hybrid simulation technique, *HySim* [3, 8], for performance estimation of both types of PEs. HySim accelerates the simulation process by executing portions of the application natively on a *Virtual Co-Processor* (VCP) while rest still on an ISS.

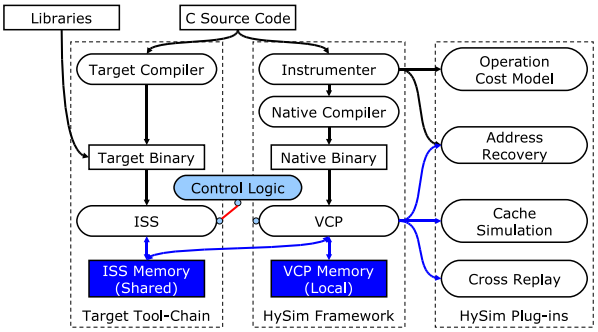


Figure 2: HySim Workflow

This work extends the performance estimation model of HySim for RISC like PEs by introducing a novel *Address Recovery Layer* (ARL) which facilitates fast and accurate memory subsystem and cache simulation. Performance estimates for RISC like PEs can be more accurately obtained through the VCP-ARL combination. For DSP/VLIW PEs, the performance estimation uses a novel technique named *Cross Replay*. To obtain the performance information, cross replay uses dynamic profiling built on trace-replay *cross* target and native ISAes.

None of these estimation techniques require training or pre-processing phases. The combined framework, as shown in Figure 1.(b), can be applied for fast functional simulation and accurate performance estimation for a large set of MPSoC architectures.

The next subsection briefly introduces the HySim framework before going into the details of the proposed extensions.

3.1 HySim: A Hybrid Simulation Framework

The workflow of the HySim framework is presented in Figure 2. HySim assumes that the application is written in a combination of C and target architecture specific assembly language. This is a reasonable assumption as far as most embedded applications are concerned.

As has been mentioned earlier, HySim combines a target architecture specific ISS execution with native code execution on the simulation host for achieving high simulation speed. Target independent sections of code (i.e. basic blocks, program statements or functions) are called *virtualizable*, since they can be executed on the native simulation host to produce the same results as ISS execution. Conversely, target dependent program elements (e.g. assembly functions) or third party libraries without source code are called *non-virtualizable* and *have to be* executed on the target ISS. Any segment of code executing on the simulation host is called *virtualized* and similarly, one executing on the ISS is called *non-virtualized*. Note that a virtualizable segment might not be virtualized (i.e. it might still be executed on the ISS), but the reverse is not true. Through hybrid simulation, the current work can easily support third party libraries and target dependent code which is a major advantage over pure source level performance evaluation approaches [6, 11].

As is shown in Figure 2, the HySim workflow combines virtual and ISS execution into one simulation framework. An entire application is compiled through the target compiler to produce a target specific binary. On the other hand, *virtualizable* portions of the application are instrumented through an instrumenter and then, compiled through the native compiler to produce a binary for the simulation host. During execution, all the non-virtualizable parts of the application are executed on ISS using the target binary. The virtualizable parts of the application can either be simulated on the ISS, or can be executed on the simulation host through the VCP. The functionalities of the instrumenter and the VCP will be shortly described. Whether a virtualizable part is executed on the ISS or the native host is decided by the *control logic* as per user preferences.

As can be seen from Figure 2, the simulation framework contains two different data memories, namely the *ISS memory* and the *VCP memory*. One major problem of virtualized execution is maintaining the consistency between these two memories so as

to present an *unified view* of the memory to the executing application. This is achieved in two steps. In the first step, the instrumenter inserts extra code in the virtualizable segments after static analysis of the application. In the second step, the VCP uses the added code fragments to maintain the consistency of the two memories, and to synchronize ISS execution and native execution so as to hide the details of virtualization from the user.

4. RISC PERFORMANCE ESTIMATION

RISC like architectures are often used in MPSoCs as controller units. They do not contain too many domain/application specific features for enhancing the data processing, but are usually equipped with memory hierarchies consisting of one or more levels of caches. The cycle count estimates for these machines can be constructed from two kinds of execution statistics : (1) the operation execution frequencies, and (2) the caching behavior.

Usually, most of the C operations can be implemented using one or more instructions in a RISC machine. Generally, a known number of cycles are required to execute such an operation in hardware, which is defined as the operation's *cost*. Therefore, if the costs of all the operations in a piece of C code and their respective costs are known, the cycle count for executing these operations can be easily inferred.

The number of cycles required to execute a memory reference operation can not be statically calculated. It depends on the cache behavior of the program. If a cache miss happens, extra memory access time needs to be considered. Therefore, the overall cycle count for a C application can be estimated using the following formula:

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

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 estimate the N_i for each operation i, our framework uses a technique similar to [6]. Instead of estimating the performance at the C code level, the application 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, all global variable accesses and all the control transfer statements, are explicit. To enhance the accuracy further, a set of high-level optimizations (such as constant propagation, constant folding, dead code elimination etc.) are run on the 3-AC IR to eliminate redundant operations. The instrumenter then accumulates the operation costs of each basic block, and annotates it to the code.

4.1 Online Cache Simulation

For accurate performance estimation, it is extremely important to take cache simulation into account, since the memory subsystem forms a major performance bottleneck in many modern processors. Previous works evaluate memory subsystems by analyzing the information implicit in high-level programming languages. For example, [6] generates a memory referencing trace when profiling a C application, and replays the trace by using a cache simulator. There are two major demerits of such an approach. 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 afterward. This is a major problem in MPSoCs, where inaccurate timing can bias scheduling [5] and affect the overall performance estimation adversely.

The cache simulation in this work addresses both problems. As shown in Figure 3, all the ISS generated memory references are simulated on the publicly available DineroIV [1] cache simulator when the application is only executed in completely non-virtualized mode. When virtualized segments are executed, the memory references are not directly passed to the cache simulator, but processed by an Address Recovery Layer. The ARL tries to translate a reference to an accurate (or, at least closer) memory address, which imparts better accuracy in cache simulation.

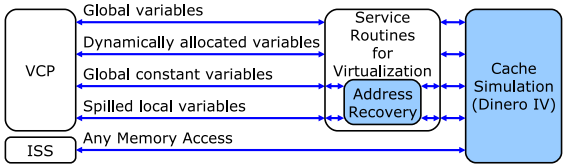


Figure 3: Cache Simulation Framework

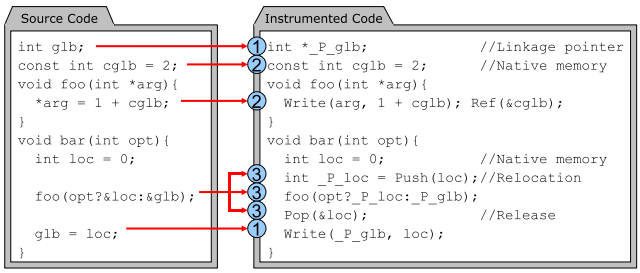


Figure 4: Example of Instrumentation for Memory Consistency

Moreover, the cache simulation is performed *in situ* to provide precise timing information during an MPSoC simulation.

To explain how address recovery works, we consider the two coexisting memory spaces (ISS memory and the VCP memory) inside HySim. The knowledge needed to recover addresses comes from both static and dynamic analysis. In Figure 4 (①), there are two virtualizable functions `foo` and `bar`. `bar` accesses a global variable `glb`, the instance of which lies at the ISS memory space. To access it, a *linkage pointer* `_P_glb`, containing the address of `glb` in the ISS memory, is created by the instrumenter. During simulation, the accesses to `glb` are handled through a set of service routines (e.g. `Write` in Figure 4) which only dereferences this linkage pointer. The precise ISS memory address for `glb` can be passed to the cache simulator through these service routines. Constant global variables (e.g. `cglb`) are handled in a slightly different manner. Since constant globals are not modified during execution, a native clone for each such variable is created inside the VCP memory for faster access. Before performing cache simulation, ARL recovers actual address of these variables using the mapping information obtained during instrumentation (②).

Another important case, which requires dynamic analysis, concerns joining of linkage and local pointers due to control flow. In the source code of Figure 4, `foo` is called with a pointer. Depending on the value of `opt`, this pointer can either point to a local variable (`loc`), or to `glb` which results in an ambiguity when dereferencing the pointer [3]. To eliminate this ambiguity, the local variable is *spilled* (by using `Push` and `Pop`) into the ISS memory space. For this case ③, ARL translates the address of a local variable to an approximate address at the function stack at ISS memory.

To summarize, HySim can approximate the memory layout, as well as the access patterns, for a given application and a memory subsystem to provide both high simulation speed and better accuracy than the pure high-level performance annotation counterparts. Note that one limitation of this work is the absence of instruction cache simulation support, which will be addressed in our future work.

5. PERFORMANCE ESTIMATION 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 operation count based approach for RISC machines is not applicable for these PEs.

Fortunately, some assumptions about the nature of the DSP/VLIW PEs and the applications running on them can sig-

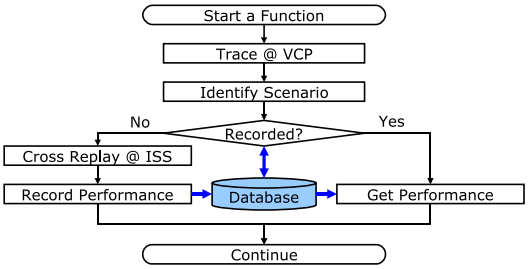


Figure 5: Cross Replay Workflow

nificantly simplify the problem. Firstly, many of such PEs have no affiliated caches. 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 PEs often have high volume of data processing and limited number of control paths (i.e. they contain limited number of if-then-else statements, loops with statically known iteration bounds etc.). So it is possible to infer the execution performance for such PEs by enumerating each control path, and then *one time* calculation of 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, a 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 performance 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 performance 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 major contribution of this approach is enabling the function level trace-replay, which is *cross-ISA* and supports optimized target binaries.

5.1 Trace Generation and Replay

While 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 variable's value), in order to replay the function in ISS some records are generated dynamically as an alternative of checkpointing.

There are 3 types of records. *Input records* are generated when a function is invoked at VCP. Each input record stores the value of one incoming function argument.

To correctly replay a virtualized function on ISS, the global states have to be reproduced. If the function reads any memory location which is not initialized in its own scope, the value or this memory location must be recorded. This is called *memory records*.

To generate memory records, a shadow memory is first created to record *clean* or *dirty* status of each address. A dirty location conceptually means that it has been changed by some outside agent than the function in consideration. A read to a dirty location must be recorded in the trace. A clean location means that it has been updated only by the virtualized function, or its values have been recorded. When a function starts executing on VCP, all memory locations are marked as *dirty*. The first read/write operation to a dirty memory changes its state to clean. Additionally, the first read on a dirty location generates a memory record which stores the value read for replay purposes. Any subsequent access (read/write) to the same location does not generate any more memory records or state changes.

Any non-virtualized subroutine invoked from a virtualized function can, *potentially*, change any global data memory. Therefore, if a global variable is read after such a function calling, the value has to be recorded again. This is handled by simply marking all memory locations as dirty after the invocation.

The third kind of records is called *return value record*. Such a record is generated when a non-virtualizable subroutine is called, and it stores the value returned by the invoked function. Note that when replaying the function, the non-virtualizable subroutine is not simulated again. The reason is not that of saving simulation time, but that of the impossibility to replay the subroutine. An example to explain this is when the subroutine modifies a not recorded global memory, the next time when it is replayed, the execution will be eventually different.

The replay phase commences for a trace miss. A dedicated ISS (namely *replay-ISS*) is created, to ensure that replaying does not affect the status of the original ISS.

5.2 Example Run of the Cross Replay

Figure 6.(a) shows a piece of virtualizable C code along with its instrumented version. For cross replay, this piece of code is first executed on the VCP. Figure 6.(b). presents the generated trace when the function `bar` is executed on VCP.

Source Code

```
int glb;
int foo(int *arg){
  return *arg + 1;
}
void bar(int opt){
  int loc = ext(glb);
  int *p = malloc(4);

  if(opt){
    *p = foo(&glb);
  }else{

    *p = foo(&loc);

  }
  glb = loc;
}
```

Instrumented Code

```
int *_P_glb;
int foo(int *arg){
  return Read(arg) + 1;        //RecordMem(arg)
}
void bar(int opt){             //RecordInput(opt)
  int loc = inv_ext(glb);      //RecordRet(loc)
  int *p = inv_malloc(4);      //RecordRet(p)
                               //SetDirtyAll()
  if(opt){
    Write(p, foo(_P_glb));     //SetClean(p)
  }else{
    int _P_loc = Push(loc);    //SetClean(_P_loc)
    Write(p, foo(_P_loc));     //SetClean(p)
    Pop(&loc);                 //SetDirty(_P_loc)
  }
  Write(_P_glb, loc);          //SetDirty(_P_glb)
}
```

(a) Trace generation

Scenario 1: `opt` is true

index	dep	type	info	width	value
1	N.A.	input	NO. 0	4	arg:1
2	N.A.	return	ext	4	Ret_of ext
3	N.A.	return	malloc	4	Ret_of malloc
4	3	memory	arg	4	*arg

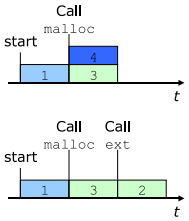
Scenario 2: `opt` is false

index	dep	type	info	width	value
1	N.A.	input	NO. 0	4	arg:0
2	N.A.	return	ext	4	Ret_of ext
3	N.A.	return	malloc	4	Ret_of malloc

(b) Generated records

Target Pseudo Code

```
int glb;
void bar(int opt){
  int *p = malloc(4);
  if(opt){
    *p = glb + 1;
  }else{
    int loc = ext(glb);
    *p = loc + 1;
  }
  glb = loc;
}
```



(c) Cross replay

Figure 6: Example of Cross Replay

The trace (in Figure 6.(b)) shows two scenarios - when the variable `opt` is true and false. Firstly let us consider the scenario where `opt` is true. The first record in the trace is an *input record* which stores the value of the argument, `opt`, to the function `bar`. The first two statements in `bar` are invocations of two non-virtualized functions `ext` (an external function whose source code is absent) and `malloc` (a standard library routine which has side effect). Therefore, the next two records are *return value record*s correspondingly. The last record is a memory record. When `opt` is true, `bar` calls `foo` with the address of the global `glb`. Reading this global inside `foo` creates the memory record which stores the value of `glb`. Only this memory record has a *dep* field whose significance will be described shortly. In the instrumented code, these recording tasks are accomplished by calling functions, such

as `RecordInput, RecordRet, RecordMem` from the corresponding service routines. The traces generated for another scenario can be understood similarly. Note that there is no memory record since `loc` is initialized in the scope of function `bar`.

Figure 6.(c) shows the replaying of these traces on the replay-ISS. The target compiler can perform a number of aggressive optimizations on the code. One possible example of the transformed code is shown in the *target pseudo code* where the function `foo` has been completely inlined. However, the replay can still work with this optimized code.

HySim can only load trace records at some specific points of replay. These points are - the beginning of the replayed function, and after the call to a non-virtualized function. The input record is loaded at the beginning of `bar`, to initialize the variable `opt`. Since `malloc` and `ext` are simulated at ISS when `bar` is executed at virtual mode, their performance has already been recorded. Thus, replay-ISS directly get the return values from the return value records, instead of simulating them again.

The *dep* fields indicate loading dependencies. For example, since `malloc` has side effects, its invocation *might* change the value of `glb`, thus the memory record for `arg` can only be loaded after the call to `malloc` returns.

Last but not the least, target compiler may exchange the calling sequence of `malloc` and `ext` as an optimization. This can only happen if the target compiler thinks that the exchange is safe. Consequently, the imitation of the function calls in cross replay is also safe.

6. COMMUNICATION

Both the functional correctness and timing precision of multiprocessor applications rely on the accuracy of communication and synchronization. There is a lot of work on cycle accurate modeling of whole systems which take accurate peripheral latency and communication congestion into account. HySim offers a replacement of traditional ISSes, and can be easily integrated into such systems. However, since the motivation of this work is to increase the simulation speed by raising the abstraction level, we also use a simple, abstract model of communication which is described in the experimental results section. Here we only discuss the synchronization between PEs.

In a multiprocessor system, there is a global time, and each processor has its local time. For virtualized functions, it is neither necessary nor possible to synchronize with the global time. When a program is exiting the virtual mode (either through invocation to a non-virtualized function, or returning from a virtualized function), the performance for the virtualized execution is estimated and updated to the corresponding processor's local time. The synchronization policy is to run a processor only when its local time is earlier than the global one.

Since functions at virtual mode do not synchronize with each other or with non-virtualized functions, there has to be one restriction that any function accessing `volatile` variables cannot be mapped to VCP. Thus, it can be ensured that the virtualized parts of an application do not need synchronization. Note that virtualized parts can still invoke non-virtualized subroutines which can do the communication.

This work support communication based on shared memory (by detecting `volatile` variables) and DMA. Currently there is a limitation that interrupts can not be handled in virtual mode. We plan to address this in future.

7. EXPERIMENTAL RESULTS

This section presents some experimental results for our simulation framework. Two sets of experiments have been done for evaluating our framework. Firstly, the performance estimation for a single RISC and a VLIW DSP processor have been obtained for a set of benchmarks. Then, these components have been integrated into an MPSoC model including a communication infrastructure, and multiprocessor performance estimations have been obtained. Although there is an automatic partitioner [8] designed for fast forwarding, for performance estimation purpose, the applications are mannually partitioned into virtualized and non-virtualized parts. All the experiments have been performed on a simulation host with Athlon64 X2 5200+ processor and 4 GB of memory, running Fedora Core version 4.

7.1 Single Processor Result

For the single processor performance estimation, we have selected one representative each from the RISC architecture class (32-bit little-endian MIPS-4K RISC processor) and the DSP/VLIW architecture class (a floating-point clustered-VLIW DSP named mAgic [10]).

Table 1 shows the results for 5 embedded applications selected to evaluate MIPS-4K. Except for `JPEG Dec`, the simulation speedup is significant. Investigation shows that the root of such limited speedup for `JPEG Dec` is an inefficient partitioning. The function being mapped to VCP is `jpeg_idct_islow` which accesses data mostly from the ISS memory space. This lowers the speedup considerably.

The performance estimates have been compared to a cycle accurate ISS. The cache simulation is quite accurate (between 0.6% and 8%) except for `MD5`. By looking into the generated target binary of `MD5`, we find that about one third of memory accesses are from spills and restores introduced by the register allocator. In our 3-AC IR, all the scalar local variables are assumed to be in register and no memory accesses are generated for them. This is the source of the discrepancy.

Single processor simulation speed improvement for mAgic is shown in Table 2. The simulation speed improvement is 10 to 50 times with tracing is enabled. And the overhead of cross replay is quite marginal (0.6% to 9%).

7.2 MPSoC Evaluation

To illustrate the usage of hybrid simulation for MPSoCs, a multiprocessor software-hardware co-design case has been studied. The target application is a multi-frame edge detection algorithm which benefits from parallelization. The design space parameters are data partitioning of the application and architectural configuration. The objective is to observe the effects of these parameters on the latency and throughput of frame-processing.

The target hardware has 1 MIPS-4K (running at 200 MIPS, with 512 byte cache) processor and configurable number (8 or 4) of mAgics (running at 100 MIPS with 320K byte scratch-pad memory). A shared bus (*100 Million Cycles per Second* (MCPS)) and a global DMA (4 bytes per cycle) are used for communication. The bus and DMA are modeled in an abstract way. The DMA has 8 programmable channels and are activated in a round robin way. No bus congestion is modeled.

The hot-spot of the software is a spatial edge detection procedure (`susan_edges`), involving a lot of floating point computations. This procedure is mapped to the DSPs and the RISC is used for input and output. Two different schemes are evaluated for this application. The first scheme (called *coarse grained scheme*) distributes whole frames to the DSPs minimizing the synchronization overhead. The second (called *fine grained approach*) processes the frames one by one, and partitions them into blocks which can be processed parallely. Since these blocks have to be overlapped because of margin effect, the second scheme involves more communication overhead.

The design space has two dimensions (software partitioning, and the number of DSPs) and 4 schemes in total. To figure out which proposal is better, fast simulation is desired.

Table 3 shows the results of enumerating various designs. Compared to the detailed simulation involving cycle accurate ISS, an error rate of less than 3% is achieved at almost 3-5× simulation speed. Still, we can see the speedup is much smaller than single processor hybrid simulation. This is because of the global synchronization. Note that the speed of MIPS-4K cycle accurate ISS is about 5 MIPS, and it has to simulate two instructions for each bus cycle. Moreover, the dispatching and data collection loop in MIPS-4K uses volatile variables and can not be virtualized. This event loop executes on MIPS-4K even when the DSPs are processing data, and since it can not be virtualized, forms the major performance bottleneck.

Results are also presented for AES and DES cryptographic applications. They show a similar trend in terms of simulation speed and performance estimates.

Application	Performance (M cycles)	Estimated Performance (M cycles)	Error Rate (%)	Cache Misses (M)	Estimated Cache Misses (M)	Error Rate (%)	Detailed Simulation Speed (MCPS)	Hybrid Simulation Speed (MCPS)	Speedup (times)
DES	281.7	282.7	+0.4	11.84	11.53	-2.6	7.8	70.7	9.0
MD5	67.9	70.1	+3.1	0.37	0.28	-22.7	3.8	23.4	6.2
G721 Enc	371.7	404.9	+8.9	3.42	3.40	-0.6	4.0	18.4	4.6
G721 Dec	329.7	331.2	+0.5	1.20	1.18	-1.6	3.7	14.4	3.9
JPEG Dec	24.2	21.9	-9.4	0.58	0.53	-8.1	4.0	5.5	1.4

Table 1: Performance Estimation for MIPS-4K

Application	Simulated Instructions (M insn.)	① Detailed Simulation Speed (MIPS)	② Hybrid Simulation (without Cross Replay) Speed (MIPS)	③ Hybrid Simulation (with Cross Replay) Speed (MIPS)	Cross Replay Overhead (②/③ - 1) (%)	Speedup (③/①) (times)
Edge Detection	3061.5	8.5	248.0	246.6	0.6	29.2
DES	17.6	4.0	44.7	41.7	6.7	10.4
FIR	215.7	5.0	260.9	237.3	9.0	47.3
FFT	146.7	4.9	108.5	103.7	4.4	21.2

Table 2: Virtual Mode Speedup for mAgic DSP.

Application	Architectural Configuration	Simulated Bus Cycle (M)	Estimated Bus Cycle (M)	Error Rate (%)	Latency per Frame	Estimated Latency per Frame	Error Rate (%)	Simulation Speed (KCPS)	HySim Speed (KCPS)	Speedup (times)
Coase-grained Susan	8 DSPs	189.4	194.7	+2.8	59.03 ms	60.67 ms	+2.8	113	411	3.6
Fine-grained Susan	8 DSPs	431.3	428.6	-0.6	16.83 ms	16.72 ms	-0.7	114	384	3.4
Coase-grained Susan	4 DSPs	373.9	385.0	+3.0	58.33 ms	60.08 ms	+3.0	209	515	2.5
Fine-grained Susan	4 DSPs	485.1	482.9	-0.5	18.93 ms	18.84 ms	-0.5	218	526	2.4
Parallel Triple DES	8 DSPs	3.47	3.47	+0.0	N.A.	N.A.	N.A.	83	434	5.3
Parallel AES	8 DSPs	24.6	23.7	-3.7	N.A.	N.A.	N.A.	120	312	2.6

Table 3: Multiprocessor Hybrid Simulation Result

8. CONCLUSION

This paper presents a novel hybrid simulation framework which can be used for multiprocessor performance estimation. The simulation framework can estimate the performance for a variety of programmable processing elements and easily cope with the software tuning of an MPSoC design environment. Experimental results show that the simulation technique can reduce simulation time significantly while delivering high accuracy in performance estimates. For multiprocessor simulation, the described techniques provide 3× to 5× simulation speed improvement with very low (3%) errors in performance estimates.

The major bottleneck for further increase in multiprocessor simulation is non-virtualizable synchronization segments running on control processors. In future we would like to investigate the acceleration of these segments. Moreover, our performance estimation is not accurate in presence of high-degree of register spills. We would also like to address this issue.

9. ACKNOWLEDGMENTS

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

10. REFERENCES

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

[2] 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.

[3] 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.

[4] T. D. Givargis, F. Vahid, and J. Henkel. Trace-driven system-level power evaluation of system-on-a-chip peripheral cores. In *ASP-DAC '01: Asia South Pacific design automation*, 2001.

[5] 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.

[6] K. Karuri, M.A. Al Faruque, S. Kraemer, R. Leupers, G. Ascheid, H. and Meyr. Fine-grained Application Source Code Profiling for ASIP Design. In *DAC '05: Design Automation Conference*, 2005.

[7] T. Kogel, M. Doerper, A. Wieferink, R. Leupers, G. Ascheid, H. Meyr, and S. Goossens. A modular simulation framework for architectural exploration of on-chip interconnection networks. In *CODES+ISSS '03: IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*, 2003.

[8] S. Kraemer, L. Gao, J. Weinstock, R. Leupers, G. Ascheid, and H. Meyr. HySim: a fast simulation framework for embedded software development. In *CODES+ISSS '07*, 2007.

[9] M. Laurenzano, B. Simon, A. Snavely, and M. Gunn. Low cost trace-driven memory simulation using simpoint. *SIGARCH Comput. Archit. News*, 33(5):81–86, 2005.

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

[11] 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.

[12] A. Muttreja, A. Raghunathan, S. Ravi, and N. K. Jha. Hybrid simulation for embedded software energy estimation. In *DAC '05*, 2005.

[13] 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.

[14] 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*, 2006.

[15] 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*, 2002.

[16] E. Perelman, M. Polito, J.-Y. Bouguet, J. Sampson, B. Calder, and C. Dulong. Detecting Phases in Parallel Applications on Shared Memory Architectures. In *IPDPS '06: IEEE International Parallel and Distributed Processing Symposium*, 2006.

[17] T. Sherwood, E. Perelman, G. Hamerly, S. Sair, and B. Calder. Discovering and exploiting program phases. *IEEE Micro*, pages 84–93, December 2003.

[18] S. Sonntag, M. Gries, and C. Sauer. Performance evaluation of packet processing architectures using multiclass queuing networks. In *ANSS '06: Annual Symposium on Simulation*, 2006.

[19] T. Wild, A. Herkersdorf, and R. Ohlendorf. Performance evaluation for System-on-Chip architectures using trace-based transaction level simulation. In *DATE '06*, 2006.

[20] R. Wunderlich, T. Wenisch, B. Falsafi, and J. Hoe. SMARTS: Accelerating microarchitecture simulation via rigorous statistical sampling. In *ISCA '03: International Symposium on Computer Architecture*, 2003.

[21] J. Zhu and D. D. Gajski. A retargetable, ultra-fast instruction set simulator. In *DATE '99*, 1999.