Creation and Utilization of a Virtual Platform for Embedded Software Optimization: An Industrial Case Study

Sungpack Hong[c1], Sungjoo Yoo[1], Sheayun Lee[2], Sangwoo Lee[3], Hye Jeong Nam[4], Bum-Seok Yoo[3], Jaehyung Hwang[2], Donghyun Song[2], Janghwan Kim[5], Jeongeun Kim[5], HoonSang Jin[1], Kyu-Myung Choi[1], Jeong-Taek Kong[1], SooKwan Eo[1]

{[1]CAE Team, System LSI Division, [2]Flash Software Group, Memory Division, [3]Storage Team, System LSI Division, [4]HDD Development Group, Storage Division}, Semiconductor Business, Samsung Electronics CO., LTD
[5]Mobile SW Platform, Software Laboratories, Corporate Technology Operations, Samsung Electronics CO., LTD
[c]sp7.hong@samsung.com

Abstract

Virtual platform (ViP), or ESL (Electronic System Level) simulation model, is one of the most widely renowned system level design techniques. In this paper, we present a case study of creating and applying the ViP in the development of a new hard disk system called Hybrid-HDD that is one of the main features in the Windows VISTA (R). First, we summarize how we developed the ViP including the levels of timing accuracy of models, automatic generation of models from RTL code, external subsystem models, etc. Then, we explain how we exploited the ViP in software optimization. Compared with the conventional flow of software development, e.g. based on the real board, the ViP gives a better profiling capability thereby allowing designers to find more chances of code optimization. Based on the simulation and analysis with the ViP, the software optimization could improve system performance by more than 50%. However, in our case study, we found that the current ViP technique needs further improvements to become a true ESL design technique.

Categories and Subject Descriptors

B.8.2 [**Performance Analysis and Design Aids**]

General Terms: Performance, Design

Keywords: Virtual Platform (V*i*P), Embedded Software Optimization, Hybrid HDD (Hard Disk Drive)

1. Introduction

Virtual platform (V*i*P), or ESL (Electronic System Level) simulation environment, is one of the most widely renowned system level design techniques [1]. The conventional application of V*i*P is fast prototyping and architecture exploration. There have been proposed several EDA solutions of ViP development [4-7], modeling standards [2,3,8], and modeling methods [12-15,18,19].

The contribution of this paper is to report an industry case study of applying the V*i*P technique to real product development. We focus on two aspects: practical issues for V*i*P development and SW optimization based on the V*i*P.

First, V*i*P development is not just assembling existing library components (e.g. processor model, bus model, etc.) available in commercial tool libraries (e.g. MaxLib of ARM ESL). Instead, it demands significant amount of manual works for modeling custom components (e.g. data path). To make matters worse, the development is always conducted in a tight time-to-market constraint but with a limited manpower. Thus, there should be a practical solution to create an effective V*i*P efficiently within the given design period. We report our experience of V*i*P development in such a circumstance.

Second, to the best of the authors' knowledge, there have been few reports of applying the V*i*P technique to SW optimization in industrial designs. Our case study shows that the V*i*P is practically beneficial as in the cases of conventional architecture exploration [23-26] or co-verification [16, 17], since it provides fast and accurate simulation with abundant visibility, i.e. profiling capability.

The rest of our paper is organized as follows. Section 2 gives an overview of applying the V*i*P technique to Hybrid HDD system design. We present the details of V*i*P development in Section 3. In Section 4, the software optimization based on the V*i*P is explained. The lessons learned in the case study are summarized in Section 5. We conclude the paper in Section 6.

2. Overview

Target System: Hybrid HDD

Recently, a new concept of hard disk drive (HDD), namely a hybrid HDD, has been proposed where a large-sized flash memory is attached to a legacy disk drive as a non-volatile (NV) cache. It has two main advantages. First, the NV cache reduces mechanical movement of the disk, thus enabling significant reduction in the power consumption of HDD. Second, by putting frequently accessed data into the NV cache, drastic loading time reduction can be achieved, since seek and rotation delay of the legacy HDD is no longer required; thus features like instant boot-on is now available. Microsoft® announced their support of this novel type of HDD in the next version of Windows OS [20].

Required Design Changes

Though many of our designs of the new hybrid HDD are inherited from the legacy normal HDD, we still required significant renovations in both hardware side and software side. On the

CODES+ISSS'06, October 22–25, 2006, Seoul, Korea.

hardware side, the NV cache related data-path had to be added to the existing HDD controller LSI, including a controller for OneNAND® flash memory [21]. On the software side, firmware (FW) [1] had to manipulate additional tasks having more complexity, e.g. managing consistency of data spread over disk, SDRAM, and flash.

In addition, there were two other market-driven constraints. First, the total performance should never be inferior to the legacy HDD system, even when NV cache is not utilized. Second, time-to-market deadline was tight; it was fixed by the launch schedule of Windows VISTA.

Introduction of Virtual Platform

There were three major challenges in the development of hybrid HDD. 1) How to increase system performance? The hybrid disk should never be slower than legacy HDD. 2) How to verify architectural decisions? There was not much time to change the structure of hardware or software afterwards. 3) How to accelerate FW development? Time to develop FW would dominate the total development time.

The virtual platform (V*i*P) [2] was introduced as a possible solution for all three challenges. Its objectives were as follows:

- To measure system performance and analyze its bottlenecks; especially the bottlenecks of the FW side.
- To find out possible optimization points from the result of bottleneck analysis.
- To evaluate various architectural decisions of both hardware (HW) side and FW side at the early stage of development
- To provide an early-access platform for FW development ahead of chip fabrication
- To exploit other advantages of V*i*P like co-verification as much as possible

In result, we have achieved the first three objectives quite successfully, while last two objectives were not so satisfactory. We will cover these issues in later sections in detail.

Early Software Development and Optimization

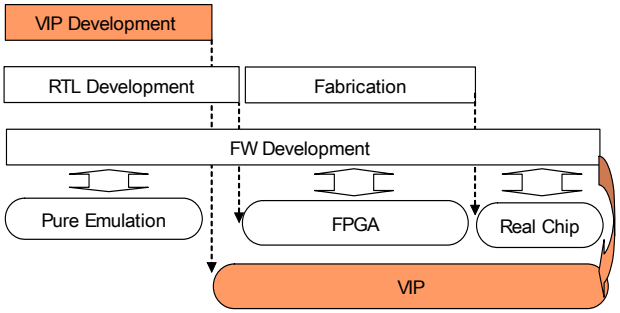


Fig. 1 Software development process enhanced with a V*i*P

Fig. 1 depicts how the conventional embedded system development process is enhanced with a V*i*P, where the newly introduced process (V*i*P Development) and its outcome (V*i*P) are denoted by color-filling. As you can see from the figure, the amount of early availability as a development platform is heavily dependent on the actual development period of both the V*i*P and the RTL. However, even when the ViP is not significantly earlier than FPGA, it can still accelerate SW development by assisting in optimization; owing to the V*i*P, optimization can be initiated at the early stage of development, which prevents huge code modification after silicon fabrication in the conventional flow.

3. Virtual Platform Development

3.1 General Issues on V*i*P Development

There are three general criteria in ESL model: timing accuracy, simulation speed, and development time. Unfortunately, these criteria are not orthogonal; accurate models are slow in simulation and require substantial time to develop. What is worse, to be practically useful, a V*i*P is required to achieve all of these criteria to some degree. In our case the V*i*P should 1) be accurate enough to analyze system performance including HW-SW interaction, 2) be fast enough to execute whole FW for meaningful time[3], and 3) be available earlier than real-chip environment. The following subsections explain our work to meet this mission.

3.2 Creating Virtual Models

Limiting Modeling Scope for Early Development

The rule for rapid development was simple; do only the least of what you have to do. Components out of our interest, like error-correction modules, are simply omitted. Functionalities that are transparent to FW are also ignored. For example, the physical and link layer activities of SATA communication were just replaced with simple timed data-copy routines in the model.

Similarly, only selected set of registers are implemented in the V*i*P. Of course, it should be avoided that FW, running on the V*i*P, accesses an unimplemented register and malfunctions. We used following two approaches to prevent such case: 1) Adding #ifdef directives in the FW code to prevent accessing unimplemented registers, 2) Providing dummy registers including fixed status bits (e.g. CRC_OK flags that are always read as true) in the model. As a result, only about 30% of registers were sufficient to be modeled for FW execution.

Timing Accuracy

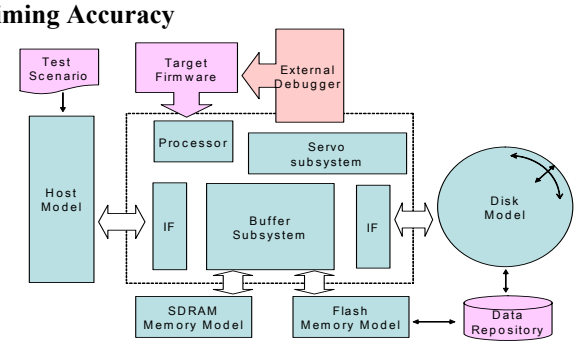


Fig. 2 The virtual platform: top-level block diagram

Fig.2 shows the top-level block diagram of our V*i*P. It contains the models for five major subsystems (processor, servo, buffer, host interface, disk interface) inside of the chip as well as those for four external subsystems (host, disk, flash and SDRAM). Though our objectives require high level of timing accuracy of

[1] We use terms *software (SW)* and *firmware (FW)* interchangeably in this paper.

[2] We adopted ARM ESL's SoCDesigner™ (also known as MaxSim) as for an ESL simulation engine.

[3] At least, 1 second of data transfer should be simulated.

the V*i*P as a whole, we did not model all the components with full details. First, we classified all the components to be modeled by accuracy requirements. Some component should be fully cycle-accurate, since a few cycles' mismatch between the model and the RTL code will cause a significant mismatch in performance measurement. A typical example is the memory controller. On the contrary, for data-dominant components (e.g. DMA engine), only the throughput and latency are important. Table 1 summarizes our classification of components in terms of accuracy requirement.

Table 1. Model classification by timing accuracy requirement

Timing Accuracy Requirement	Components
Full Accuracy	Buffer data path components[4], CPU Core
Latency-Throughput Compatible	Disk Interface, Host Interface, Memory devices
Untimed	CPU Accessories, Buffer control path components[5]

Second, depending on the different requirements of timing accuracy, we used different strategy to implement each model; we chose appropriate HW affinity[6] from four levels listed in Table 2 and applied corresponding implementation method.

Table 2. Levels of HW affinity of V*i*P model

HW Affinity	Model Implementation Method
Equivalent	Directly generate from RTL
Strong	Mimic RTL structure; state machine, major control signals, pipeline…
Weak (Behavioral)	Describe functional behavior annotated with timing parameters
Dummy	Dummy register-map to prevent FW access fault

Automatic Model Creation

Recent technology enables automatic generation of a C/C++ model from HDL description [11, 22, 28]. Its advantages are 1) highest affinity to RTL 2) fast model creation free from human error, and 3) no demand for knowledge on the RTL code. However, it also has disadvantages; 1) not applicable to some blocks (e.g. analog interface blocks), 2) slower than pure ESL model[7], and 3) the generated model has signal interfaces only[8].

We applied this technology to create a cycle-accurate model of a 3rd-party IP, mainly because we lacked of knowledge on its internals. It is notable that since this IP exploits only standard interfaces (e.g. AHB), it was relatively easy to create signal-transaction interface converters for them. On the other hand, simulation speed was degraded around 45 % due to this single component.

Modeling External Components

To simulate the entire system, external subsystems outside of the target SoC also had to be modeled as in Fig. 2. As for a normal memory device (off-chip SDRAM and on-chip ROM also), a functional (un-timed) model was quite enough; its timing behavior is solely controlled by the memory controller which was modeled with full cycle-accuracy. One the other hand, a more sophisticated model was required for the OneNAND® which is an active device, but we could easily find an already-existing in-house model for such a widely used component.[9]

We took a practical approach to create a model of host device. That is, the host model plays the role of input-pattern generator; it reads a script file and, in timely manner, generates a stream of ATA commands to the virtual host-interface model. Please remind that details of SATA communication is replaced with simple function calls in our V*i*P.

A model for disk was also simply created. Instead of accurately modeling all the mechanical properties of disk-heads and platters, we used a simple linear equations to calculate seek and rotational delay as in (1), where α, β, and γ are the parameters from empirical knowledge.

$$\begin{aligned} seek\_time &= (\#Tracks)\times\alpha+\beta \\ rotational\_latency &= (\#Sectors)\times\gamma. \end{aligned} \quad (1)$$

Admittedly, above equations are too simple to reflect realistic disk behavior at a specific moment. However, the overall system performance measured by the V*i*P simulation fairly matched with the real system on the average sense.

Summary of Modeling Practice: rule of thumbs

Here, we summarize the rule of thumbs from our practice.

- Limit the number of components and functionalities to be modeled as minimal as possible.
- All the component models need not be cycle-accurate to create a cycle-accurate V*i*P.
- Automatic model creation is a viable alternative at the expense of simulation speed and model connectivity.
- You also have to model outside the chip; Keep them abstracted and exploit them as controlling knobs.

3.3 Evaluation: Cost and Quality

3. 3. 1 Overall evaluation

We evaluated our virtual platform by the three general criteria: accuracy, speed, and cost.

- Timing accuracy was initially measured by comparison of measured performance of V*i*P simulated to that of RTL simulation using short stimulus. Its result was near 99%. Afterwards, we compared V*i*P to real system with long stimulus, which matched 95 % on average. In other word,

[4] Arbitrator, FIFOs, Memory Controller, …

[5] HW Data queue, Automatic memory-pointer modifiers…; zero-delay models for these components do not affect on overall performance.

[6] We define HW affinity to be how much the model corresponds to the original RTL code. High timing accuracy comes from strong affinity; while it also requires vast modeling efforts.

[7] Note that this technology never generates a true (abstracted) ESL model. It merely provides an equivalent C/C++ description to the RTL design.

[8] Since most ESL models use transaction level interface (i.e. function calls), there must be a converter to connect models with RTL-like signal-interface to them. While converters for well-defined signal interfaces like AHB are easy to implement, or already provided by EDA vendors, there's no easy way for custom interfaces but to create them speculatively.

[9] Another alternative was to use commercial Soma behavioral model [27] as in RTL simulation. The drawback of this approach was simulation speed decrement due to the linkage to an additional simulation engine.

our V*i*P showed enough accuracy to be used for performance optimization.

- Simulation speed was 15K cycles per second.[10] With this speed, to simulate 1 second takes about 150 minutes, and 1 second is quite long time enough to characterize the FW and the HW with various test patterns.
- The cost, or time to develop the V*i*P, was 2 x 6 man-months. The V*i*P was available 5 months ahead of real chip environment, and 3 weeks ahead of FPGA.

3.3.2 Unsuccessful Objectives

This section analyzes why our V*i*P failed to satisfy some of the original objectives.

As a Verification Environment

A V*i*P cannot but have only limited merits as an verification environment. Most of all, it can not validate hardware design directly, since models are not equivalent to the RTL; when detecting a malfunction in V*i*P simulation, one can not sure there exists the same malfunction in the real system.[11]

A V*i*P may be useful for verification if it is co-simulated with RTL and works as a real pattern generator including SW execution. However, in this case simulation speed will be shackled by RTL simulator. Also extensive use of signal-transaction converters is required..

As an Early-Access Platform for FW Development

To our dismay, our V*i*P was available only 3 weeks ahead of FPGA. That is not surprising, though; while the entire V*i*P had to be modeled from the scratch, most of the RTL designs were already there in the legacy design.

More importantly, our V*i*P had a significant disadvantage as a FW development environment; it was too slow. For example, if an execution takes 1 second in a real environment, FW designers expect it takes within 10 ~ 100 seconds in the virtual environment, but can not endure 10 ~ 100 minutes since they have execute codes over and over at the development stage. Simulation speed should be the major concern for this usage.

4. Software Optimization based on the V*i*P

In our work, the main benefit of the V*i*P was its assists in software optimization. This section gives the details of our software optimization using the V*i*P.

4.1 SW Optimization and Performance Improvements

From the beginning of the project, we expected the FW would be the main bottleneck of system performance, since the new FW had to handle lots of complex tasks in addition to the legacy ones. If we had followed the conventional design flow without a V*i*P, system performance could be measured only after the fabrication of silicon and functional implementation of FW are completed. And even then, the detailed bottleneck analysis would never be an easy task. However, thanks to the V*i*P, we could start software optimization much earlier in the design cycle; the FW designers could optimize their code while they were still implementing functionalities.

Fig. 3 shows the history of system performance improvement with FW optimizations (versions). In the figure, performance is normalized with respect to that of the first version. At each step of version-up, we optimized the FW codes using the analysis result of V*i*P simulation. In result, as you see in the Fig 3, the fourth version showed more than 50% performance improvement in read operations solely by FW optimization.[12]

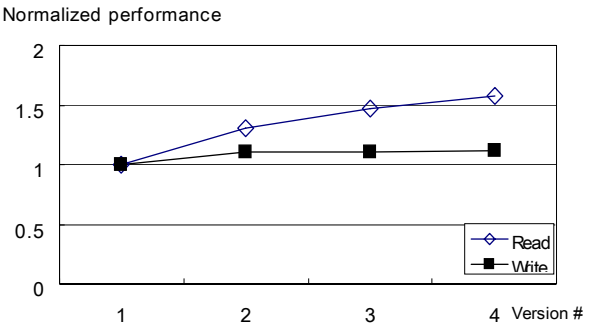


Fig. 3 The improvement of system performance over firmware versions

1.1 V*i*P as an Optimization Environment

Table 3. Comparisons of FW development environments

	Pure Emulation	FPGA	ViP
Speed	1s GHz	10s MHz	10s KHz
Time accuracy	Not at all	Unreliable*	More than 90 %
Visibility	Function call counts	Limited (GPIO, UART, ETM)	Full trace with timing information
FW execution	HAL implementation required	Almost no modification	Almost no modification
Usefulness	High-level algorithm development	Low-level development and verification	Performance analysis

Table 3 compares various FW development environments to show the merits of the V*i*P as an optimization environment. Pure emulation is fast but gives little information on timing. Also all the register access routines have to be re-written for full FW execution. FPGA is a better alternative, but its timing accuracy may be still not satisfactory due to following reasons: 1) FPGA supports only limited number and frequencies of clocks[13], and 2) memory hierarchy including on-chip memories is discrepant to the real design.[14] On the other hand, a V*i*P can provide enough time accuracy for bottleneck analysis (either in HW or SW side). Furthermore, low simulation speed is not a big concern in this

[10] We used cycle-accurate (not ISS) ARM core model supplied by Maxlib, which runs at 300Kcps by itself. However, other cycle-accurate models like detailed buffer subsystem degrades simulation speed, not to mention automatic models.

[11] Still, one can suspect HW/FW bugs in this case. As for our experience, we could spot out DMA/CPU cache coherence problem by inspecting V*i*P simulation traces.

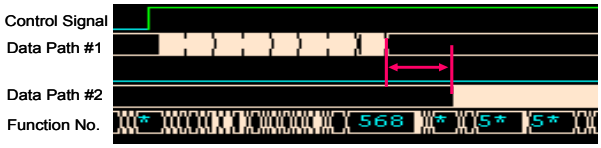
[12] Here the amount of write performance improvement is quite small. This is because the relatively low write bandwidth of flash memory plays the bottleneck in this case.

[13] In our case, FPGA could not emulate all the different clocks in the target system: SATA, SDRAM, Flash, Disk, CPU, and their dividends.

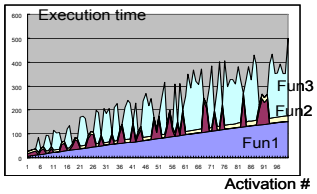
[14] FPGA usually do not support large sized on chip memories and timings are not equivalent to the real design.

case since many an analyses can be performed from one long simulation.

More importantly, a V*i*P provides sufficient visibility (i.e. traces and profile data) for optimization. Fig. 4 depicts typical analysis data obtained from our V*i*P simulation. Fig. 4 (a) is the waveform view of simulation trace, which shows both HW events and SW function calls at the same time. This view enables to track down real critical paths; whether the bottleneck lies in the SW part or in HW.



(a) Simulation Trace



(b) Function execution history

```
| Execution    Function
  Cycle        Call
(         338) < [----------]
(         160)     < [Fun1](96)
(         445)         + [Func2](96)
(         293)     > [Fun1](96)
(         116) . [----------]
(         142)     < [Func3](128)
(       29226)         + [DoThings](128)
(         258)     > [Func3](128)
(         516) . [----------]
```

(c) Function Call Trace

Fig. 4 Sample Analysis Data from V*i*P simulation

Fig. 4 (b) exemplifies the history view of execution time of selected SW functions. As shown in the figure, since the runtime of a function is not stationary but variable over time, short-term simulation (as in RTL simulation) or a taking snapshot of a function-call instance (as in logic analyzer) is not sufficient to capture the true nature of the function.

Finally, Fig. 4 (c) shows a fraction of function-call trace information. In this view, we can drill down into each sub-function and find out the most time-consuming parts of the code. Additionally, overall analysis facilities are provided: average execution cycles of functions, amount of runtimes of sub-functions, average distance between functions, and so on.

In summary, with its fair time-accuracy and abundant visibility[15], the V*i*P enabled us to effectively track down the real bottlenecks of the system.

4.3 Firmware Optimization Techniques

In this section, we explain the optimization techniques we applied and how the V*i*P was helpful for them.

Algorithmic Optimization: The most effective and fundamental optimization was, of course, the modification of top-level algorithms. Generally speaking, while no profiling method can directly propose a new optimized algorithm, the developers could find weak points of their algorithm from the information obtained from the V*i*P simulation. Also they can evaluate alternative algorithms effectively with it.

As for our case, nobody could explain the disappointingly low performance obtained by first measurement, until the V*i*P simulation revealed the true causes[16]; it was FW overhead of data management routines, which was ignorable in legacy HDD system, but now matters for flash data access. Consequently, we had to design a new tightly-coupled data management routine which includes special short-cut routines for flash data access.

Latency Hiding: This is to overlap SW and HW execution instead of their sequential executions. For example, instead of SW's busy waiting until the end of a HW task, CPU executes another task and checks the result of HW later. The V*i*P simulation can find out where those latency hidings are applicable.

In our case, flash programming and erase operations were selected for this optimization. For example SW can update all the relevant control data, except the final confirm flag, assuming successful termination of those operations while the flash device is still in operation.

Pipelining: This is to initiate a data-consuming HW task prior to the completion of corresponding data-producing HW task. The V*i*P simulation help to balance the trade-off of pipelining: enhancement from parallelization and degradation from resource conflict and complicated FW control. One exemplar application of our case is to initiate data transfer to the host device, while data transfer is still undergoing at the flash memory side.

DMA Access: Designing our FW, there was an argue whether to use the DMA for a short amount of data transfer or to let CPU do it; both are known to have a certain amount of overhead in our system, while their difference was unknown in a quantitative way. The V*i*P simulation gave a direct answer to this question.

Memory Relocation: This is to move critical parts of code and data from slow memory to fast one. In the conventional development flow, the decision of which parts goes to the fast memory are made from empirical knowledge. However, the V*i*P simulation helped to get a systematical solution by listing up the most time-consuming functions and frequently accessed data. In addition, the numerical information that tells exactly how much faster does the code run on the fast memory (e.g. on-chip SRAM) than on the slow memory (e.g. off-chip SDRAM). FW developers decided code/data location on the base of this information.

Data Structure Modification: Data structure also affects performance. For example, keeping a data structure to align certain memory boundary can boost access speed significantly at the cost of memory loss. In our example, we listed up candidate objects for this optimization and made up decisions using the V*i*P simulation analysis.

Overall, the V*i*P played a pivotal role in the optimization process. If there had not been the V*i*P, FW developers would have spent indefinite time to find the reasons of performance degradation. Thus they could not decide which optimization techniques should be applied to which part of the codes. In consequence, the V*i*P have saved enormous development time in the FW optimization step.

[15] To obtain this information, we implemented various profiling accessories on top of the simulator's open interface by ourselves including function call tracer, and its post-processing tool.

[16] Actually, there were still HW malfunctions blamed for performance degradation at that time. However, ViP simulation projected that the result would not be satisfactory even after those HW bugs are fixed.

4.4 Hardware Modification

The result of V*i*P simulation analysis was not only beneficial for SW optimization, but also valuable for finding HW revision points. As an example a FW task (a data-pattern generator), that was executed routinely but took significant amount of time, was identified and decided to be replaced by HW in the next version of controller LSI.

Also, architecture exploration in classical sense was also practiced to design the next version of HW. That is, using the V*i*P, we could easily test how the performance is influenced with the change of CPU types, clock speeds, memory sizes, and so on.

5. Lessons Learned: Requirements for future ESL Technology

The summary of our experience is that (1) the V*i*P needs to be available much earlier than in the current practice if it is for an early-access platform, (2) the simulation speed needs to be significantly improved for FW development, but (3), for system optimization cycle accuracy should not be sacrificed.

In our case, the availability of the V*i*P was not significantly earlier than the FPGA-based environment: only 3 weeks earlier. Generally speaking, creating archives of component models is not a complete solution, since every company has a number of custom HW-IPs which are continuously modified. Therefore, modeling methodology itself should be innovated.

Fast simulation speed is always beneficial for any V*i*P usage model. Recent trends in ESL concentrate on creating fast simulators enough to execute complete SW including big OS's like linux, at the cost of cycle-accuracy [9, 10, 29]. However, the drawback of this approach is to limit the range of applications; system performance optimization as in our case study becomes impossible.

One possible solution for above requirements is the classical two-step approach; first create a fast-running model then replace it with slow-but-accurate one later. For this approach, clean and robust macro building blocks for modeling (e.g. register file, data transfer, memory element, timing unit …) should be provided additional to current primitives. And there should be a simple way to replace the fast model with an accurate one, especially when the latter exploits different interfaces from the former. Also the burden of managing two versions must be alleviated.

Another solution can be an FPGA emulator combined with an ESL simulator. That is, a part of custom RTL design is synthesized into an FPGA while its execution is synchronized with top-level ESL-simulation. The disadvantages of this approach are 1) it is only enabled after RTL is ready, 2) A bridge to convert signal level interface to transaction level is required, and 3) events inside the RTL are hard to capture. Also there are issues like size limitation of FPGA and synchronization overhead.

6. Conclusion

In this paper, we presented an industrial case study of creating the virtual platform of a SoC and exploiting it for a real product development, especially for FW optimization. For the creation of the V*i*P, we applied several modeling techniques to accelerate modeling while maintaining a certain degree of accuracy and simulation speed. We took advantages of the V*i*P to the end of development steps, especially in FW optimization thereby improving system performance by more than 50%. Overall, V*i*P played a crucial role in the development of Hybrid HDD system.

7. REFERENCES

[1] "System-Level IC Design Accelerates SoC Delivery", Nikkei Electronics ASIA, Feb. 2005.

[2] SystemC White Paper, http://www.systemc.org

[3] A. Haverinen, *et al.*, "SystemC based SoC Communication Modeling for the OCP protocol", http://www.ocpip.org/, Oct. 2002.

[4] ARM-ESL SocDesigner, http://www.arm.com

[5] ConvergenSC, http://www.coware.com

[6] Magillem, http://www.prosilo.com/products/magillem

[7] Cocentric System Studio, http://www.synopsys.com/products/cocentric_studio

[8] J. A. Colgan, *et al*., Advancing Transaction Level Modeling (TLM): Linking the OSCI and OCP-IP Worlds at Transaction Level, http://www.opensystems-publishing.com/whitepapers

[9] Real Time System Model, http://www.arm.com/products/DevTools/RealTimeSystem Model1176.html

[10] Platform development kits, http://www.virtio.com

[11] CoFluent Studio, http://www.cofluentdesign.com

[12] J. A. Rawson, "Hardware/Software Co-Simulation", Proc. Design Automation Conference, pp. 439–440, 1994.

[13] R. Klein, "Miami: a hardware software co-simulation environment", Proc. International Workshop on Rapid System Prototyping, 1996.

[14] A. Nohl, *et al.*, "A universal technique for fast and flexible instruction-set architecture simulation", Proc. Design Automation Conference, 2002.

[15] S. Yoo, *et al.,* "Building Fast and Accurate SW Simulation Models based on SoC Hardware Abstraction Layer and Simulation Environment Abstraction Layer"**,** Proc. Design Automation and Test in Europe, 2003.

[16] L. Séméria, A. Ghosh, "Methodology for Hardware/Software Co-verification in C/C++," Proc. Asia and South Pacific Design Automation Conference, pp. 405-408, Jan. 2000.

[17] M. Bradley, K. Xie, "Hardware/Software Co-Verification with RTOS Application Code", http://www.techonline.com/community/tech_tipic/21082

[18] M. Hassan, *et al*., "RTK-Spec TRON: A Simulation Model of an ITRON Based RTOS Kernel in SystemC", Proc. DATE, March 2005.

[19] R. Siegmund, D. M¨uller, "SystemC*SV* : An Extension of SystemC for Mixed Multi-Level Communication Modeling and Interface-Based System Design", Proc. DATE, 2001.

[20] Microsoft, WinHEC 2005 Conference Tracks, http://www.microsoft.com

[21] Samsung, OneNAND product information, http://www. samsung.com

[22] Tenison, vtoc product information, http://www.tenison.com

[23] H.Jang, *et al*., "High-level system modeling and architecture exploration with SystemC on a network SoC: S3C2510 case study", Proc. DATE, March 2004.

[24] S.Brini, *et al*., "A flexible virtual platform for computational and communication architecture exploration of DMT VDSL modems", Proc. DATE, March 2003.

[25] G. Hadjiyiannis, *et al*., "A methodology for accurate performance evaluation in architecture exploration", Proc. DAC, June 1999

[26] S. Pasricha, *et al*., "Using TLM for exploring bus-based SoC communication architectures", Proc. ASAP, July 2005

[27] MMAV, http://www.denali.com/products_mmav.html

[28] Carbon, http://www.carbondesigns.com

[29] Virtual Platform Designer , http://www.coware.com