A Hardware/Software Co-Simulation Environment for Graphics Accelerator Development in ARM-Based SOCs

Dan Crisu, Sorin Cotofana, and Stamatis Vassiliadis
Computer Engineering Laboratory
Faculty of Information, Technology, and Systems
Delft University of Technology
Mekelweg 4, 2600 GA Delft, The Netherlands
Phone: +31 15 2783644 Fax: +31 15 2784898
E-mail: {dan|sorin|stamatis}@ce.et.tudelft.nl

Abstract— **This paper focuses on the challenging aspects of developing a versatile hardware/software co-design and co-simulation environment for the development of 3D graphics hardware accelerators in ARM-based system-on-chip designs. The tool we propose integrates the ARMulator, the cycle-accurate instruction-level simulator for the ARM low-power processor family, with an augmented open source SystemC modeling framework and simulation engine, which allows the development of cycle-accurate or more abstract models of software algorithms, hardware architectures, and system-level design. The tool permits the simulation of an entire computer graphics pipeline allowing experimental software/hardware partitioning schemes, and performance monitoring in terms of throughput and power consumption. Moreover, it provides graphical output for the visualization of the potential impact tweaking the algorithms or the bit operand width precision may have on the resulted image quality.**

Keywords—**graphics architecture; hardware/software co-simulation; system-on-chip; ARM processor; ARMulator; SystemC**

I. Introduction

With the recent proliferation of embedded systems using the system-on-chip (SOC) design paradigm such as mobile PDA's (Personal Digital Assistants), cellular phones, and other portable computing appliances, the request for increasingly fast, graphics-reach user-friendly interfaces and entertainment environments opened new market opportunities for 3D real-time rendering graphics systems meant to accelerate these features. The challenge posed by the severe cost constraints on products for the mobile consumer market requires a new breed of graphics rendering hardware to be developed with a very low power consumption and low implementation costs. This implies that performance/power/cost trade-offs have to be investigated and decisions have to be made early in the design process regarding the most suitable partition between features that must be provided by software and features that will be mapped in hardware.

In this context, designing and assessing the performance of hardware architectures for accelerating graphics has proved to be a difficult endeavor. Among the reasons are the absence of well-established benchmark programs (although graphics system specifications like OpenGL [1] and Direct3D [2] exist for many years) and the lack of specific tools to support the graphics architecture development process [3]. As a consequence, heterogeneous design exploration frameworks were created by connecting custom-made tools with tools borrowed from other fields of computer architecture research, thus raising a lot of problems that have to be solved like the interoperability, flexibility, and specificity for the intended purpose [3] [4] [5]. To overcome these difficulties, and to produce an efficient and productive design environment, a first attempt toward an integrated software/hardware co-design framework for graphics hardware accelerators was presented in [6]. In this framework, a number of software tools were developed in the C++ language to work either standalone or alongside hardware models written in a high level hardware description language (VHDL) to aid in algorithm research and hardware design. The software tools are based on an extensive library of C++ classes designed to enable accurate software modeling of both number representation (multiple precision fixed- and floating-point data types) and hardware arithmetic units in a graphics pipeline. This library of classes allows also the exploration of the graphics algorithms at both algorithmic and behavioral levels, prior to the hardware design using VHDL. Mixed-model simulation (C++/VHDL) is also possible. Although this co-design framework addresses well the issue of specificity and although its generality allows a large number of problems pertaining to graphics accelerator development to be solved within the framework, the mixed-model simulation is cumbersome. The run-time integration and synchronization during simulation of the parts modeled in VHDL (behavioral or RT level) with the software mod-

els of the rest of the graphics pipeline components is performed via a file sharing mechanism, which by definition it is slow and it is delayed further by the protocol of data exchange needed at the C++/VHDL interface. Another drawback of the framework is the impossibility to refine a high-level functional model down to the implementation in a single language. Usually at the RT level the refinement of the model in C++ has to be ended abruptly and the model has to be prepared for hardware synthesis by being completely rewritten in VHDL without taking advantage of any previously developed source code, making this a very tedious and error-prone process.

In addition to the imperious need of an early performance assessment of the hardware architectures for accelerating graphics, the energy consumption is a critical system-level design factor to be accounted for the graphics architectures that are targeted to the embedded portable appliance market. Studies have demonstrated that circuit- and gate-level techniques have less than a $2\times$ impact on power, while architecture- and algorithm-level strategies offer savings of $10 - 100\times$ or more [7]. Hence, the greatest benefits are derived by trying to assess early in the design process the merits of the potential implementation. Ideally, when designing a graphics accelerator for an embedded system, a designer would like to explore a number of architectural alternatives and test functionality, energy consumption, and performance without the need to build a prototype first.

Usually, typical mobile embedded systems are built of commodity components and have a microprocessor-based architecture. Full system evaluation is often done on prototype boards resulting in long design times. Power consumption estimation can be done only late in the design process, after the prototype board was built, resulting in slow power tuning turnarounds that doesn't meet the requirement of fast time to market. On the other hand, using field programmable gate array (FPGA) hardware emulators for functional debugging, with a fast prototyping time, can neither give accurate estimates of energy consumption nor of the performance.

Among the tools preferred for early performance assessment at the algorithmic and architectural level, in the last decade, were the cycle-accurate instruction-set simulators. Unfortunately, for power consumption estimation this approach was seldom easy to follow. There were only a few academic tools for power estimation (all based on or integrated in the SimpleScalar instruction set simulator toolset framework [8] [9] [10]) and almost no commercial products.

For several target general purpose processors a number of techniques emerged in the last few years. The processor energy consumption for an instruction trace was generally estimated by instruction-level power analysis [11] [12]. This technique estimates the energy consumed by a program by summing the energy consumed by the execution of each instruction. Instruction-by-instruction energy costs, together with non-ideal effects, are precharacterized once for each target processor. A few research prototype tools that estimate the energy consumption of processor core, caches, and main memory have been proposed [13] [14]. Memory energy consumption is estimated using cost-per-access models. Processor execution traces are used to drive memory models, thereby neglecting the non-negligible impact of a non ideal memory system on program execution. The main limitation of these approaches is that the interaction between memory system (or I/O peripherals) and processor is not modeled. Cycle-accurate register-transfer level energy estimation was proposed in [9]. The tool integrates RT level processor simulator with DineroIII cache simulator and memory model. It was shown to be within 15% of HSPICE simulations.

The drawback of all the above methods to estimate the power consumption is that they are based on certain architectural templates, i.e., general purpose processors and can be hardly adapted to model graphics accelerators embedded in system-on-chip designs.

This paper focuses on the challenging aspects of developing a versatile hardware/software co-design and co-simulation environment for the development of graphics hardware accelerators in ARM-based system-on-chip designs. The tool we propose integrates the ARMulator [15], the cycle-accurate instruction-level simulator for the ARM low-power processor family, with an augmented open source SystemC modeling framework and simulation engine [16] [17], which allows the development of cycle-accurate or more abstract models of software algorithms, hardware architectures, and system-level design. The fundamental motivator for choosing the SystemC modeling language is the possibility to refine down to the implementation details an entire system specified at higher levels of abstraction, i.e., at the functional level, in a single language. Also, automatic synthesis of hardware is also possible from the RTL subset of the SystemC language [18]. The tool permits the simulation of an entire computer graphics pipeline allowing experimental software/hardware partitioning schemes, and performance monitoring in terms of throughput and power consumption estimated at the RT level. Moreover, it provides graphical output for the visualization of the potential impact tweaking the algorithms or the bit operand width precision may have on the resulted image quality.

The rest of the paper is organized as follows. The func-

tions that are performed by a 3D graphics accelerator in a real-time graphics rendering system are presented in Section II. The benefits of the system-level modeling that can be derived by employing the SystemC language are presented in Section III. The design exploration framework that we propose for embedded graphics accelerator development in ARM-based system-on-chip designs is discussed in Section IV. Finally, Section V presents the conclusions and describes future work in the area.

II. A Typical 3D Graphics Rendering System

A 3D graphics rendering system is organized conceptually as a number of stages chained in a pipelined fashion. The main function of this pipeline is to generate, or render, a two-dimensional image on a raster screen, given a virtual camera, a scene populated with three-dimensional objects, light sources and atmospheric conditions (fog). The objects in the scene can be made of different materials, can have different colors, or have different images imprinted on their surface (textures). Furthermore, their appearance may be affected by the light sources, the lighting models, and the atmospheric conditions. The conceptual stages of the graphics pipeline are the *application*, the *geometry*, and the *rasterizer stage* [19] [20]. They are presented in Figure 1.

The application stage is dealing only with object specification and scene management tasks. To enumerate only a few, it specifies how the objects are constructed from connected geometrical forms, what material properties are bound to the objects, it specifies the kind and the number of objects and light sources in the scene, the spatial or temporal relationships intra- or inter-objects, or between objects and light sources. The most important outcome of the computations performed in this stage is the scene decomposition in a number of rendering primitives (points, lines, triangles) to be fed to the next stage. The application stage is always implemented in software, performs floating-point computations, and the outcome of the computations performed here has a tremendous impact on the workload of the subsequent stages.

The geometry stage is responsible for the majority of the per-primitive operations or per-vertex operations. Basically, in this stage, matrix transformations are applied to the rendering primitives received from the application stage, resulting in a perspective mapping of the triangles to the 2D display. This conceptual stage has several functional stages: the *model and view transform*, the *lighting*, the *projection*, the *clipping*, and the *screen mapping stage*. In the model and view transform stage, the rendering primitives are remapped from their own system of coordinates to a common system of coordinates having the camera at its origin. In the lighting stage, for the incoming primitives that are to be affected by the light sources, a lighting equation is used to compute a color at each vertex of the primitive. This equation takes into account the location of the light sources and their properties, the position and normal of the vertex, and the properties of the material belonging to the vertex. In the projection stage, usually a perspective transform is applied to the incoming primitives. The perspective transform mimics the way we perceive the objects' size with the distance towards horizon. In the clipping stage, only the primitives wholly or partially inside the viewing volume (what can be seen through the camera) are passed to the next stage. The primitives partially inside the viewing volume have to be clipped, and the part that is outside the viewing volume is not propagated. In the screen mapping stage the (clipped) primitives have still three-dimensional coordinates. However, the x and y coordinates of each primitive's vertex are specified in a different range than the coordinate range of the screen and, necessarily, they have to be "stretched" (translated + scaled) to form coordinates in the screen range. The z coordinate is not affected by this mapping and it is propagated unchanged. Thus, the primitives that have survived at the end of the geometry stage are passed on to the rasterizer stage with the vertices specified in the new *screen coordinates* x and y, and the old coordinate z. The geometry stage is usually implemented in software, although high-performance graphic systems exist that implement this conceptual stage in hardware. The operations performed in this stage are floating-point computations.

Given the primitives received from the geometry stage with transformed and projected vertices, colors, and texture coordinates computed for this vertices, the goal of the rasterizer stage is to assign correct colors to the pixels on the screen to render an image correctly. This process is called *rasterization* or *scan conversion*. Unlike the geometry stage, which handles per-primitive operations, the rasterizer stage handles per-pixel operations. During rasterization, the information needed for the screen pixels covered by the primitive is interpolated from the data (colors and texture coordinates) associated with its projected vertices on the screen. The two-dimensional images of the projected primitives are stored in a memory called the *frame buffer*, which is read periodically by the display controller to form the image on the screen. The rasterization is concerned only with the producing of a series of frame buffer addresses and values (called *fragments*) using a two-dimensional description (screen coordinates x and y, colors, and texture coordinates) of the vertices of a point, line segment, or polygon. Each fragment so produced is fed to the next functional stage (inside the con-

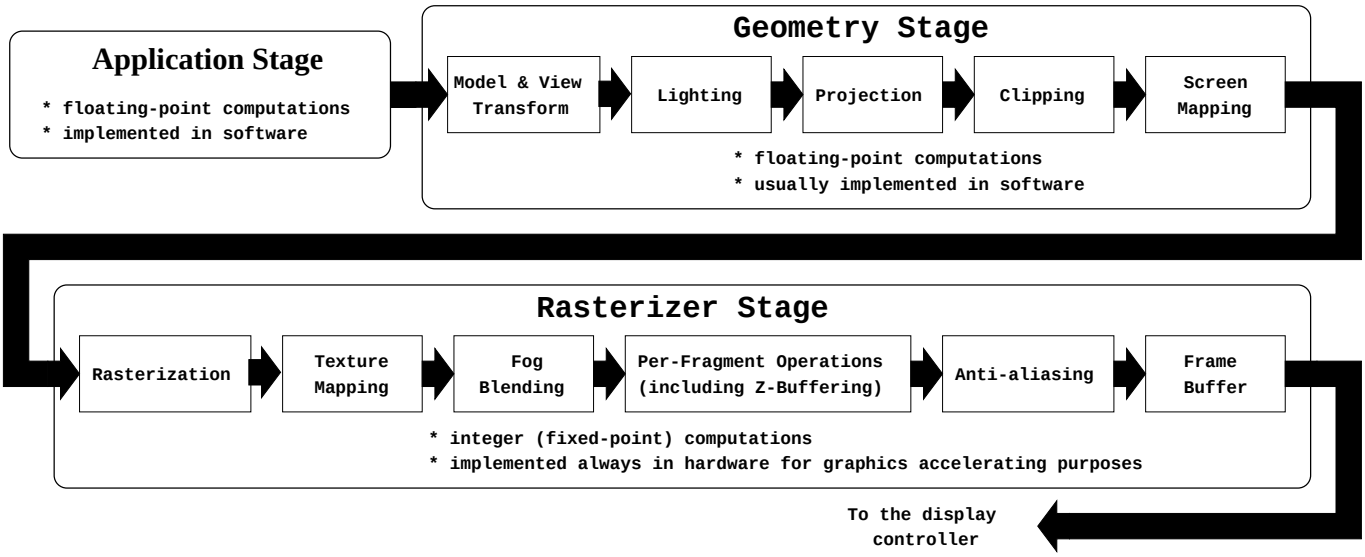

Fig. 1. A typical 3D graphics pipeline.

ceptual rasterizer stage) that performs operations on individual fragments before they finally alter the frame buffer. These operations include color alteration based on the textures assigned per primitive and texture coordinates, fog blending, conditional updates into the frame buffer based on incoming and previously stored depth values z in the *depth buffer* or *z-buffer*, blending of incoming fragment colors with stored colors, as well as masking and other logical operations on fragment values. Due to the sampling process involved by rasterization, a chain of filtering operations followed by resampling may be necessary on fragment values to alleviate the inherent aliasing phenomenon (e.g., the staircasing effect of lines drawn on a raster screen). Finally, the fragments that will survive in the frame buffer, after all of the primitives have been processed, will produce the final image. The rasterizer stage is implemented in hardware wherever exists the need for graphics acceleration and it usually involves only integer (fixed-point) arithmetic.

In fact, due to the computational explosion in the rasterizer stage (operations performed per-pixel, not per-primitive), this stage constitutes the only stage that it is truly implemented in every *graphics hardware accelerator*. The rasterizer stage is divided in the functional stages that were presented, but a functional stage describes only the task to be performed in the pipeline, it does not specify the way the task is executed in the underlying hardware pipeline. A functional pipeline stage may be divided in several hardware pipeline stages, or two functional pipeline stages may be implemented in one hardware pipeline stage. Also, a hardware pipeline stage may be parallelized in order to meet high performance demands. On the other hand, for every function performed in the rasterizer stage a considerable number of hardware algorithms exists. A description of these algorithms is beyond the scope of the paper (to start with, the reader is referred to [19] [20]). Within a hardware algorithm datapath, various fixed-point data formats and precision can be employed that might have an impact on the quality of the generated image. As a consequence, different performance-power-cost trade-offs would have to be explored by a designer to choose the best solution for the to be developed graphics hardware accelerator.

To summarize, the conceptual stages of the 3D graphics pipeline are mapped on a typical computer system equipped for 3D graphics as described in the sequel. The graphics software application is running on the host processor of the system. The software application corresponds to the conceptual application stage of the graphics pipeline. The software application is relying on a 3D graphics library (perceived in the sense of a software interface to the graphics hardware [21]) like OpenGL or Direct3D to have its graphic calls taken care further. This 3D graphics library executes usually the conceptual geometry stage on the host processor. The code that implements the geometry stage in the library can further make calls to the graphics hardware accelerator by means of a standardized, virtual interface. However, between this virtual interface and the graphics hardware accelerator (on which the conceptual rasterizer stage is mapped) there is another piece of code executed on the host processor called a device driver. This device driver performs the function of a hardware abstraction layer and it translates the calls through the virtual interface in actual memory-mapped or programmable I/O

instructions (seen from the host processor point of view) particular to the graphics hardware accelerator's input and output register port mapping in the system address space. Finally, the rasterizer stage is executed in hardware on the graphics hardware accelerator. The benefit of this scheme is the resultant portability of the graphics software application between computer systems equipped with different 3D graphics accelerators, at the cost of changing only the device driver. This is usually a non-issue, the device driver being developed jointly with the graphics hardware accelerator.

III. SystemC Modeling Benefits

SystemC represents the newest system-level specification and design language [16] [17] [22], and it was a response to the increasing system complexity facing the designers nowadays. For example, a modern system-on-chip may well contain one or more processors (for control, digital signal processing, multimedia processing), on-chip memories, accelerated hardware units for dedicated functions, peripheral control devices, linked together by a complex on-chip communication network. Along, complex layered software architectures are necessary to coordinate the inner working of such integrated device. As a consequence, the SystemC language was developed to allow in a single language to be specified, simulated, designed, and implemented complex systems with functionality implemented both in hardware and software forms.

More in particular, the SystemC language is developed on top of the C++ language, in order to capitalize on the extensive infrastructure of capture, compilation, and debugging tools already available. In addition, with an object-oriented language base, it allows for modeling flexibility, parametrization, and facilitates reuse, through capabilities such as templates and inheritance. Intellectual property (IP), available from third-parties, can be delivered in a secured form (in a compiled form as object code) to be linked and embedded in the system description. Also, the SystemC language does not impose any constraints on the design flow, the designer is free to use whatever approach she/he might consider fit for the system. As said, it is possible to use top-down, bottom-up, or even middle-out design flow methodologies without any restriction.

Regarding the way a particular SystemC model reflects its real-world implementation, there are many aspects of accuracy that SystemC can capture:

- *structural accuracy* — whether the partitioning between hardware and software is modeled and whether major hardware and software modules within the implementation are evident, whether the signals and pins of the actual implementation are reflected in the hardware model, or for software models, whether the communication between tasks is refined down to the level of communication mechanisms provided by the target real-time operating system;
- *timing accuracy* — whether the model reflects the timing of the actual implementation expressed in absolute time units, or at the clock-cycle level, or whether the timing is not expressed at all;
- *functional accuracy* — whether certain complex functionality is simplified in a high-level model to speed up the simulation, or whether the functionality is reflected down to minute detail in the model;
- *data organization accuracy* — whether the software data structures and the data layout within the model match those used within the implementation;
- *communication protocol accuracy* — whether the communication protocols used in the target implementation are reflected in the model and at what level of abstraction;
- *hierarchical accuracy* — whether the accuracy issues discussed above are extended also to the child modules of the particular module model.

As a consequence, multiple levels of abstraction can be selected to model a complex system: the straightforward executable specification, the untimed or timed functional model, the transaction-level model, the behavioral hardware model, the pin-accurate, cycle-accurate hardware model, or the register-transfer (RT) level model. Even it may be possible to mix these levels of abstraction when modeling various parts of a system [22]. Also, automatic synthesis of hardware is possible from the RTL subset of the SystemC language [18] which allows a single language to be used for modeling, design exploration via simulation and refinement, and hardware synthesis.

We will present in the sequel a short overview of the SystemC language philosophy. The SystemC language and its support libraries are built entirely on standard C++. This means that any program written in SystemC may be compiled with a C++ compiler to produce an executable program. To introduce its own semantics based on own higher-level constructs, the SystemC uses a layered approach in its specification. The base layer of SystemC provides an event-driven simulation kernel. This kernel works with events and processes in an abstract manner, without knowing their semantics. An event is represented by an object that determines whether and when a process's execution should be triggered or resumed. The processes support (static or dynamic) sensitivity lists to specify on what kind of events they have to react. The notification of events that cause sensitive processes to be triggered can be immediate at the current simulation time instant, can be delta-delay notifications after an infinitesimal amount

of time, or nonzero-delay notifications after a specified amount of time has elapsed. The three flavors of event notification have different consequences and allow hardware and software behavior to be mimicked. Other elements of SystemC include modules and ports for representing structural information, and interfaces and channels as an abstraction for communication. Channels are used for holding and transmitting data, and an interface describes the (sub)set of operations that the channel provides for manipulating data. Ports are well-defined boundaries of a module that act like proxy objects to facilitate access to channels through interfaces. The kernel and these abstract elements together form the core language. Alongside the core language is a set of data-types (and their associated overloaded operators) that are useful for hardware modeling and certain kinds of software programming (in digital signal processors): fixed-precision and arbitrary-precision integral types, bits and bit-vectors types, four-valued logic types, resolved types, fixed-point types etc. Equally, the native C++ data types can be employed. The elementary channels library layer is immediately above the core language, and include models that are widely applicable, such as signals (that can be used to model hardware signals), timers, and FIFO buffers. Although are specified apart from the core language, the data types and the elementary channels library have a broad usefulness and are regarded as part of the SystemC standard. On top of these libraries belonging to the standard, using native C++ mechanisms, other high-level libraries can be implemented to accommodate different models of computation and design methodologies. For more detailed information, the reader is referred to [22].

The opportunities offered by the SystemC language in setting the infrastructure needed by hardware/software cosimulation, system design exploration, and hardware design made it the vehicle of choice in our endeavor to investigate and develop a 3D graphics hardware accelerator to augment ARM-based system-on-chip designs.

IV. The Proposed Design Exploration Framework

In this section we present our design exploration scenario for a low-power, low-cost 3D graphics hardware accelerator to be employed as a coprocessor or a peripheral unit coupled with an ARM CPU core on a system-on-chip design.

A. System Model

Figure 2 gives an overview of the system modeling strategy that we propose. The central elements of our architectural design exploration framework are:

- ARMulator [15], a cycle-accurate instruction-level simulator for the ARM low-power processor family, without power estimation capabilities;
- a reference implementation of a SystemC simulator, that can be downloaded as source-code from the OSCI organization's WWW site [23];
- GRAAL (GRAphics AcceLerator) Simulator, our own custom-designed tool that acts as a graphical front-end for SystemC simulation control and data vizualization;
- our custom-designed library for RTL power consumption estimation;
- a model in the SystemC language of the to be developed 3D graphics hardware accelerator that is the subject of our investigation;
- the graphics software application in a binary form, which will run on the ARMulator model of the ARM processor, and will make use of the capabilities of the graphics hardware accelerator.

The simulation scenario can be explained briefly as follows. The instruction set architecture of the ARM family of processors offers room for extensions to be added by providing the so called coprocessor instructions. Referring to the Figure 2, the inputs from the user are a description in SystemC of a candidate architecture for the 3D graphics hardware accelerator coupled as a coprocessor and the graphics software application program to be run on the ARM processor. The graphics software application will link statically the graphics libraries (e.g., OpenGL) with the afferent device drivers for the graphics hardware accelerator (as discussed in Section II). The provided program is then compiled using the ARM native compiler. Usually, the device driver code will embed, beside ARM native instructions, specific coprocessor instruction. These specific instructions, when executed on the ARMulator, will be recognized as non-native or coprocessor instructions and they will trigger callback functions, installed using the ARMulator API (application programming interface), so specific actions (e.g., new data or commands are fed to the hardware description simulated in SystemC) can be taken[1]. Moreover, every clock cycle (this might be the case only when the graphics hardware accelerator is modeled at cycle-accurate level), the ARMulator will sent signals to the SystemC simulator to advance the state of the simulated hardware description one more clock cycle. In this simple way the simulated hardware description of the graphics accelerator will process its own data in lockstep with the ARM processor pipeline. Also, irrespective of the level of abstraction chosen to model the graphics hardware

[1]For the synchronization between the ARMulator and the SystemC simulator a two-phase reliable communication protocol (request-acknowledge) is employed.

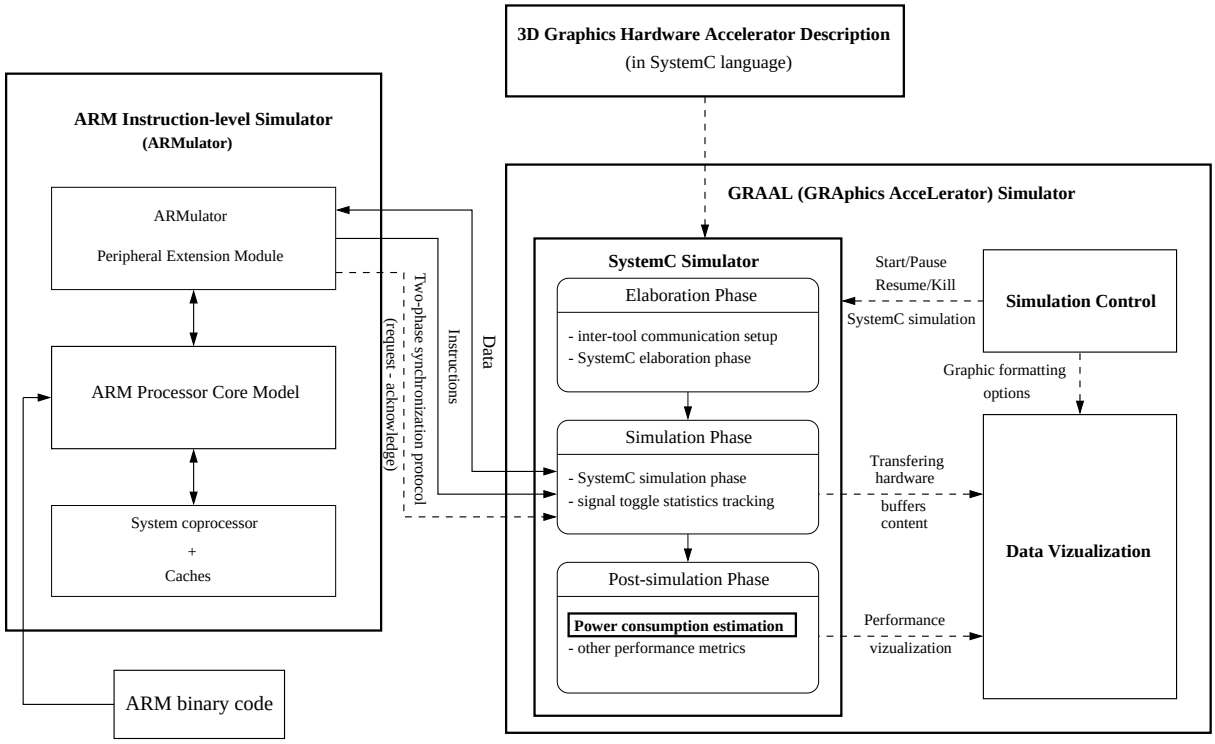

Fig. 2. The graphics hardware accelerator design exploration framework.

accelerator, the content of the relevant hardware buffers inside the accelerator can be visualized in real-time during the simulation. Moreover, if the graphics hardware accelerator is modeled at the register-transfer level (RTL), every clock cycle the activity on internal relevant signals is also collected and, after the simulation is finished, it is modeled statistically. Thus, at the end of the simulation (in the post-simulation phase referring to Figure 2), the total power consumption of the graphics hardware accelerator per graphics software program executed on the ARM processor is estimated. In the virtue of the SystemC's rich expressiveness, other performance metrics might be estimated with ease.

In the following sections, particular aspects of interest to our design exploration framework implementation are presented.

B. Simulation control

The GRAAL (GRAphics AcceLerator) Simulator program, presented in Figure 2, was developed to provide a graphical front-end for SystemC simulation control and data visualization in our graphics hardware accelerator design exploration framework. For the time being, it is a stand-alone program, implemented using the OSF Motif toolkit [24] [25] for UNIX / X Window System workstations. As soon as the undergoing refinement of the reference SystemC simulator implementation will settle down to a more stable form, the rationale behind keeping the GRAAL Simulator program completely distinct from the SystemC simulation engine will disappear. At that point in time, for the sake of efficiency, the code to provide a graphical front-end for the SystemC simulation engine will be transferred from the GRAAL Simulator program and adapted to the SystemC simulator implementation internals.

The data visualization aspects of the GRAAL Simulator program will be covered in Subsection IV-C. In this subsection only SystemC simulation engine control aspects of the GRAAL Simulator program will be discussed. For the sake of brevity, the executable program that will result from linking the compiled SystemC model alongside the SystemC simulation engine libraries will be called throughout this subsection the SystemC simulator program.

The basic idea is to enable the GRAAL Simulator program to launch the SystemC reference simulator as an independent software entity. The challenges that have to be solved by the GRAAL Simulator program is to be able to control a number of aspects of the SystemC simulation without making any slightest modification to the source code of the SystemC reference simulator. These aspects involve the capability to start, pause, resume, and abort the SystemC simulation process at any time, and to have all the messages normally displayed by the SystemC reference

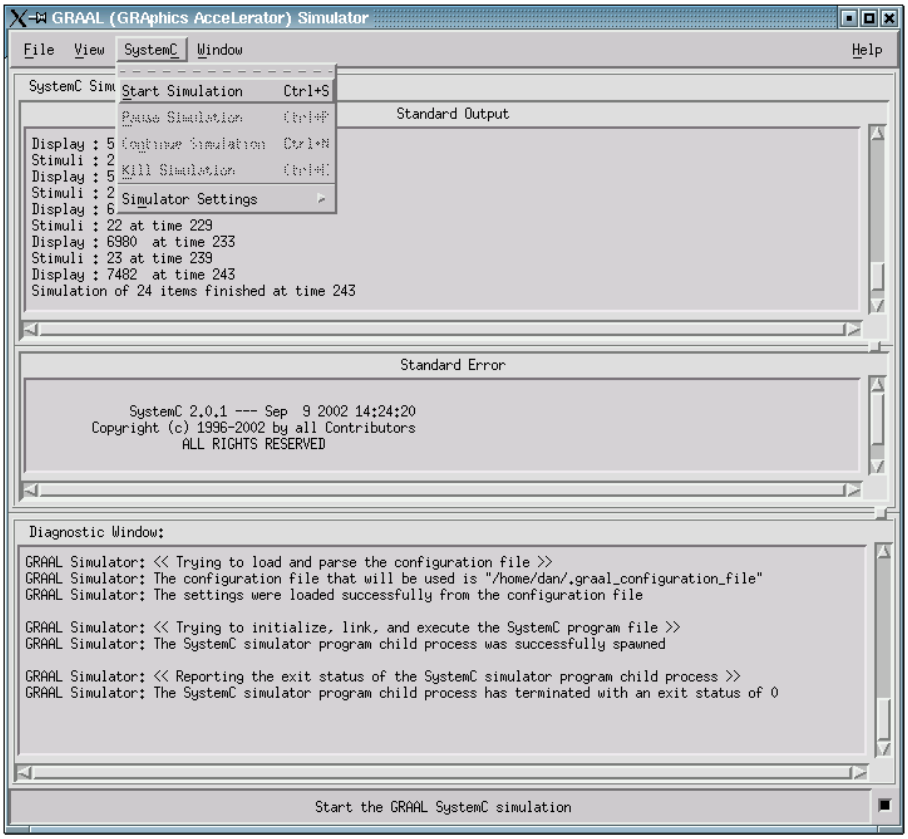

Fig. 3. SystemC simulation control in GRAAL Simulator program.

simulator redirected and captured in the graphical front-end of the GRAAL simulator program. Fortunately, the mechanisms provided by the system libraries of the UNIX operating system [26] and the OSF Motif toolkit API made this task possible. Before presenting details about how the control was implemented, a few words about these software mechanisms are necessary in order to facilitate the understanding of the algorithmic steps taken.

Processes are the primitive units for allocation of system resources. Each process has its own address space and one or more threads of control. A process executes a program; one can have multiple processes executing the same program, but each process has its own copy of the program within its own address space and executes it independently of the other copies. Each process is identified by a unique *process ID number*. Processes are organized hierarchically. Each process has a *parent process* that explicitly created it. The processes created by a given parent are called its *child processes*. A child inherits many of its attributes from the parent process. Child processes are created with the `fork()` system call. The child process created by *forking* a process is a copy of that (parent) process, except that it has its own process ID. A newly forked child process continues to execute the same program as its parent process, at the point where the `fork()` system call returns. The return value of the `fork()` system call is used to find out if the program is running in the parent process or in the child process. The child process can execute another program from that of its parent by using the `execv()` system call. The program residing on the filesystem that the child process is executing is called its *process image*. Starting execution of a new program causes the child process to forget all about its previous process image; when the new program exits, the child process exits too, instead of returning to the previous process image (parent's process image). To control the evolution of a child process from the parent process, *signals* (software interrupts delivered to a process) can be delivered from the parent process to the child process. Among them it can be mentioned `SIGSTOP` to pause a process, `SIGCONT` to resume a paused process, and `SIGKILL` to abort a process. To make a child process to announce its parent process of its completion, a `SIGCHLD` signal handler for the child process has to be registered within the parent process. Then the `waitpid()` system call has to be used in the parent process to request exit status information from the child process.

To redirect the output messages from a child process to

the parent process, *pipes* are employed. A pipe is a mechanism for interprocess communication; data written to the pipe by one process can be read by another process. The data is handled in a first-in, first-out (FIFO) order. The pipe has no name; it is created by the `pipe()` system call in the parent process before forking and it is inherited from the parent process by the child process after the forking takes place. Then a call to the `dup2()` system call is performed in the child process (before the `execv()` system call) to force the child process, after a new process image has been loaded, to use one end of the pipe as its standard output channel (a pipe have to be employed for each of the descriptors of the standard output streams `stdout` and `stderr`). The other end of the pipe will be used by the parent process to capture the messages sent by the child process on one of its standard output streams. Also, to report in the parent process errors encountered early in the child process (e.g., to report that the `execv()` system call to load a new process image has not succeeded), another pipe has to be set in the parent and inherited by the child process for this purpose.

An issue that will further increase the complexity of the scheme explained above has to do with the communication asynchronism between the child process and a parent process that employs an X-based graphical user interface (GUI). A signal such as `SIGCHLD` or data sent through the pipes can be received from the child process asynchronously in the context of the current running parent process. If the parent process would like to display the data received or to display messages about the child process status by making X routine calls right away from inside the handlers that the parent has installed for this eventuality, this will lead to problems. This scheme of operations can severely interfere with the transmission and processing of X protocol messages between the X server and the application client (the parent process), because this asynchronous events could potentially be delivered right in the middle of an X call which is manipulating the event queue. Any attempt by the handler to call further X routines in these circumstances might garbage any messages which are in progress. The solution to this problem is to register these handlers with the Xt Toolkit Intrinsics library (a library on which OSF Motif toolkit is built) that will delay the processing of the received events until a time when the parent process knows it is safe to make X routine calls.

The algorithmic steps (simplified) performed to set up the SystemC simulation engine as a child process of the GRAAL Simulator process are presented next. These steps are performed in the GRAAL Simulator program up to the forking point, then the steps will be split in steps taken in the parent process and in the child process. The steps are:

1. create a pipe (the error pipe) that will be used to catch and report errors from the child process before the SystemC simulator program image is loaded by the child process;
2. create a pipe (the `stdout` pipe) for redirecting the messages on the `stdout` stream of the child process;
3. open or create a file with a user-specified name for logging the messages on the `stdout` stream of the child process;
4. create a pipe (the `stderr` pipe) for redirecting the messages on the `stderr` stream of the child process;
5. open or create a file with a user-specified name for logging the messages on the `stderr` stream of the child process;
6. perform forking and by examining the return value (the process ID) of the `fork()` system call, branch in:

- steps for the child process

(a) using `dup2()` system call, connect the `stdout` stream of the child process to one end of the `stdout` pipe;
(b) using `dup2()` system call, connect the `stderr` stream of the child process to one end of the `stderr` pipe;
(c) load using the `execv()` system call the reference SystemC simulator as a new process image for the child process;
(d) report any errors from the previous steps to the parent process via the error pipe.

- steps for the parent process

(a) install a handler for the `SIGCHLD` signal sent by the child process and register the handler with the Xt Toolkit Intrinsics library to process X routine calls safely;
(b) install a handler for the error messages received through the error pipe from the child process and register the handler with the Xt Toolkit Intrinsics library;
(c) install a handler for the messages received through the `stdout` pipe from the child process and register the handler with the Xt Toolkit Intrinsics library;
(d) install a handler for the messages received through the `stderr` pipe from the child process and register the handler with the Xt Toolkit Intrinsics library;

From now on, all the communication aspects between the GRAAL Simulator program and the reference SystemC simulator are in place. Moreover, as an outcome of the algorithmic steps mentioned, the SystemC simulation is running. By using the child process ID of the SystemC simulator program and the signals `SIGSTOP`, `SIGCONT`, and `SIGKILL`, the GRAAL simulator program can pause, resume, or abort the SystemC simulation. The messages displayed by the SystemC simulator program are also logged in user-specified files. The simulation control panels are presented in Figure 3. The first two panels are allocated for the messages displayed by the reference Sys-

temC simulator. The third panel provides feedback about the tasks that are currently performed by the GRAAL Simulator program in response to the actions selected by the user from the graphical menus. It is also used to signal different error conditions that might appear.

C. Data visualization

The design exploration framework being developed in particular to investigate graphics hardware accelerator architectures, means to visualize in a graphical form various buffers inside the graphics hardware accelerator has been provided. In this way, it is possible, while the SystemC simulation is running, to analyze the impact tweaking the algorithms or the bit operand width precision may have on the resulted image quality. It is also possible with the data visualization tool to spot artifacts in an image, generated by the graphical algorithms employed in the graphical pipeline and hence, may be a valuable debugging tool.

The data visualization tool is flexible enough to display in real-time with regard to SystemC simulation, in different windows, all kinds of buffers that the designer of a graphics hardware accelerator would like to monitor: the display frame buffer, the depth or z-buffer, and other temporary (hardware or software) memory buffers used for image compositing tasks. It is in particular designed to visualize the buffers of a particular class of low-cost, low-power rasterizers that employs a tiling architecture. The tiling architectures were adopted as a way to counteract the huge increase in storage and bandwidth requirements of full-scene antialiasing [27]. In a tiling architecture, the screen is divided in a number of non-overlapping regions, or tiles, which are processed serially. After a complete tile is rasterized, the image resulted for that tile is copied in the global display frame buffer, then the rasterization is repeated for the next tile until all the tiles of a screen are processed. The costs of a tile-based rasterizer are much lower due to the fact that the intermediate fragment values only need to be maintained for the tile, not for the whole screen. Thus, for these types of architectures, all the buffers pertaining to a tile-based graphics hardware accelerator can be visualized and the current-processed tile location can be marked in the global display frame buffer.

To facilitate the image analysis of the buffer content, functions for zoom in, zoom out are provided, as well the possibility to capture, at the current zoom level, the buffer's data visualization window content (or only what can be seen through the window viewport) as a graphical image on disk. In addition, to aid in the algorithm development, or debugging, the center of the pixels, or the rendering primitives whose image is rasterized, can be superimposed on top of the buffer content. To help the designer to catch algorithm errors like locations left uninitialized in the buffers, those portions are drawn in the buffer's visualization window with a special stipple marking. A few of these features can be seen in Figure 4, where the content of a display frame buffer that undergoes a color test is presented.

Regarding the data visualization implementation in our design framework, a one-way communication mechanism from the SystemC simulator to the GRAAL Simulator was devised. This scheme is based on software, inter-process communication FIFOs in UNIX. A FIFO special file is similar to a pipe, but instead of being an anonymous, temporary connection, a FIFO has a name like any other file in the system. Processes open the FIFO by name in order to communicate through it. A process that writes in a FIFO special file has to have another process listening to the FIFO, otherwise it will deadlock.

For these scheme to work, multiple FIFOs, one for every buffer whose content is intended to be visualized and one for the rendering primitives that will be overlaid on the buffer content, will be employed to carry data between the GRAAL Simulator process and the SystemC simulator. Because the GRAAL Simulator program launches the SystemC simulator as a child process, it has to set first these FIFOs for listening. For this purpose, it makes use of a configuration file in which the names of the FIFO special files employed by the SystemC simulator are communicated. The two processes have also to agree about the format of the data packets that will be used in the communication via FIFOs. Hence, a common header file, in which the templates for data communication are declared and defined, will be employed by the GRAAL Simulator program and the special modules for data communication that will augment the graphics hardware accelerator SystemC model.

After the FIFOs for communication will be opened for reading in the GRAAL Simulator program, handlers will be installed to process any events (e.g., a new data packet was received) that might appear on these FIFOs. Because we want that every change in the simulated graphics hardware accelerator's buffers to be mirrored in the data visualization window, the handlers will be also registered with the Xt Toolkit Intrinsics library to process X routine calls safely.

On the SystemC simulator side, the FIFOs will be opened for writing as soon as possible, for instance at the elaboration time (Figure 2), otherwise the GRAAL Simulator will deadlock. The elaboration is that phase of execution in which the SystemC library routines undertake the preparatory work to construct and connect the objects for simulation, as prescribed by the designer. In the SystemC

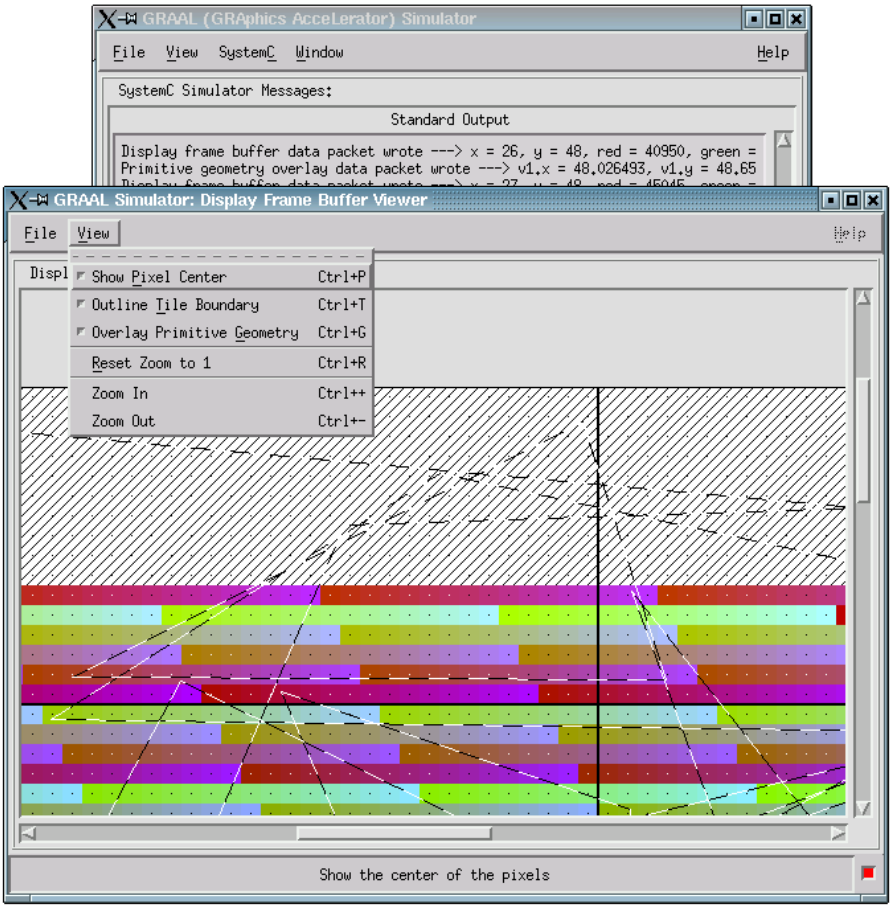

Fig. 4. Display frame buffer content visualization in GRAAL Simulator program during SystemC simulation.

simulator model of the graphics hardware accelerator, special modules for data communication will be employed alongside the modules that constitutes the graphics hardware accelerator. These special modules will be connected in parallel with the buffers receiving the same incoming data stream. This stream will be formatted into the data packets that will be pushed through the FIFO for buffer visualization in the GRAAL Simulator program. Thus, the right place to open the FIFOs for writing would be the bodies of these special modules' constructors (a module in SystemC is implemented as a C++ object).

D. Power consumption estimation

The design exploration framework can be employed for power consumption estimation once the graphics hardware accelerator model has been refined down to the register-transfer (RT) level.

To estimate the power consumption at the RT level, we adopted the estimation techniques presented in [7]. The premise for the success of such methodology consists in the existence of a library of hardware cells (various operators for datapath part, gates for control logic, and bit-cells, decoders, sense amplifiers for memory cores) specified at the layout-level. Once such a library exists, it can be precharacterized resulting in a table of effective capacitive coefficients for every element in the library. Then using only this tables and the activity statistics derived during the register-transfer level simulation the power consumption can be estimated easily. This precharacterization has to be done only once and only the effective capacitive coefficients table are needed for power estimation. The precharacterization results are valid only for a specific library of hardware cells and a given IC technology.

The power is analyzed separately for the four main classes of chip components: datapath, memory, control, and interconnect. For the first two classes, a model called the Dual Bit Type (or DBT) model is developed and it demonstrated good results, with power estimates typically within 10-15% of results from switch-level simulations. The DBT model achieves its high accuracy by carefully modeling both physical capacitance and circuit activity. The key concept behind the technique is to model the activity of the most significant (sign) bits and least significant bits separately. The DBT model applies only to parts

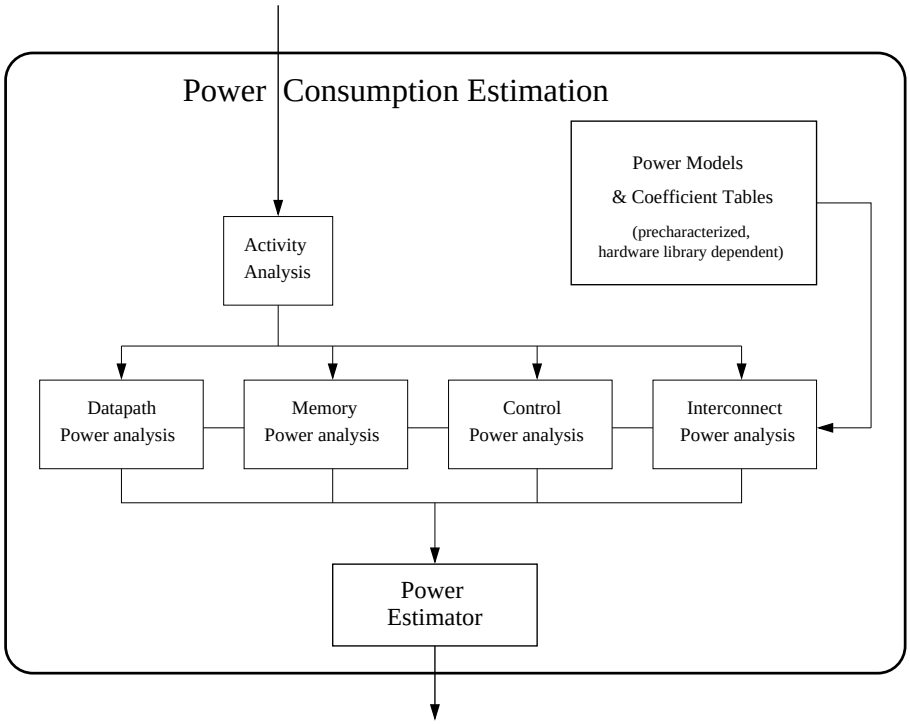

Fig. 5. The power consumption estimation strategy modeled in SystemC.

of the chip that manipulate data. A separate model is introduced to handle power estimation for control logic and signals. This model is called the Activity-Based Control (ABC) model. The method relies on the observation that although the implementation style of the controller (e.g., ROM, PLA, random logic, etc.) can heavily impact the power consumption, it is still possible to identify a number of fundamental parameters that influence the power consumption regardless of the implementation method. In a chip, datapath, memory, and control blocks are joined together by an interconnect network. The wires comprising the network have capacitance associated with them and, therefore, driving data and control signals across this network consumes power. The precise amount of power consumed depends on the activity of the signals being transferred, as well as the physical capacitance of the wires. The DBT and ABC models provide the activity information for control and data buses, but the physical capacitance depends on the average length of the wires in each part of the design hierarchy.

The details of the software implementation of this methodology for high-level power estimation and the results obtained are described in a previous paper [28]. The porting of that software implementation in the new SystemC paradigm is ongoing. Referring to the Figure 2, the power consumption estimation component modeled in SystemC can be refined as presented in Figure 5. The main components that can be identified in the figure are:

- Precharacterized Power models and Effective Capacitance Coefficient Tables Module, that contain for a library of hardware cells all the technology dependent information required by the power analysis modules to compute the power consumption; the tables are derived only once for a given library of hardware cells;
- Activity Analysis Module, that feeds the Power Analysis modules (power calculators) with statistics about signal activity inside the simulated hardware description;
- Power Analysis Modules, that estimate the power consumption in the datapath, control, memory, and interconnect based on statistics received from the Activity Analysis Module and lookups in the effective capacitance coefficient tables;
- Power Estimator Module, that adds the estimates of power consumption of datapath, control, memory, and interconnect and offers the total figure of power consumption inside the graphics hardware accelerator unit per graphics software application executed on the ARM processor.

To gather statistics about the signal activity, in SystemC were designed channels that inherit from the `sc_signal` primitive channel class (the `sc_signal` objects are used to model hardware signals in SystemC) and have additional code to keep track of the toggling transitions. SystemC module classes for every particular hardware class, e.g., datapath, memory, control, or interconnect have been

started to be developed with functionality (member functions) targeted to power analysis and power estimation. Then RT level modules for hardware units that have additionally embedded power estimation capability can be derived by double inheritance from the bare RT level hardware module class (the RTL modules used normally for simulation and synthesis) and from the power-aware module class for the hardware class that hardware unit belongs to. The interconnection between two such power-aware modules will be realized with the new channels that account for the toggling transitions.

V. Conclusions

In this paper were presented challenging aspects solved in the development of a versatile hardware/software codesign and co-simulation environment for the development of graphics hardware accelerators in ARM-based system-on-chip designs. The tool integrates the ARMulator, the cycle-accurate instruction-level simulator for the ARM low-power processor family, with an augmented open source SystemC modeling framework and simulation engine, which allows the development of cycle-accurate or more abstract models of software algorithms, hardware architectures, and system-level design. The fundamental motivator for choosing the SystemC modeling language was the possibility to refine down to the implementation details an entire system specified at higher levels of abstraction, i.e., at the functional level, in a single language. The tool permits the simulation of an entire computer graphics pipeline allowing experimental software/hardware partitioning schemes, and performance monitoring in terms of throughput and power consumption estimated at the RT level. Moreover, it provides graphical output for the visualization of the potential impact tweaking the algorithms or the bit operand width precision may have on the resulted image quality. Automatic synthesis of hardware can be also achieved form the SystemC RT level model of the graphics hardware accelerator.

The porting of a previous software implementation of power consumption prediction in the new SystemC paradigm is ongoing. The design exploration framework probably will suffer further modifications by replacing the ARMulator for the sake of efficiency with an ARM CPU core SystemC model. These issues will be addressed in future papers.

References

[1] M. Segal and K. Akeley. *The OpenGL Graphics System: A Specification (Version 1.2.1)*. Silicon Graphics, Inc., 1999.

[2] http://www.microsoft.com/windows/directx/default.asp.

[3] Ziyad S. Hakura and Anoop Gupta. The Design and Analysis of a Cache Architecture for Texture Mapping. In *Proceedings of the 24th International Symposium on Computer Architecture*, pages 108–120. ACM Press, 1997.

[4] Homan Igehy, Matthew Eldridge, and Kekoa Proudfoot. Prefetching in a Texture Cache Architecture. In *Proceedings of the 1998 EUROGRAPHICS/SIGGRAPH Workshop on Graphics Hardware*, pages 133–142. ACM Press, 1998.

[5] Michael Cox, Narendra Bhandari, and Michael Shantz. Multi-Level Texture Caching for 3D Graphics Hardware. In *Proceedings of the 25th Annual International Symposium on Computer Architecture*, pages 86–97. IEEE Press, 1998.

[6] Jon P. Ewins, Phil L. Watten, Martin White, Michael D. J. McNeill, and Paul F. Lister. Codesign of graphics hardware accelerators. In *Proceedings of the 1997 SIGGRAPH/Eurographics Workshop on Graphics Hardware*, pages 103–110. ACM Press, 1997.

[7] Paul Landman. High-Level Power Estimation. In *International Symposium on Low Power Electronics and Design*, pages 29–35, Monterey CA, 1996.

[8] D. Burger and T. M. Austin. The SimpleScalar Tool Set, Version 2.0. Technical Report Nr. 1342, University of Wisconsin-Madison Computer Sciences Department, June 1997.

[9] N. Vijaykrishnan, M. Kandemir, M. J. Irwin, H. S. Kim, and W. Ye. Energy-Driven Integrated Hardware-Software Optimizations Using SimplePower. *ISCA 2000*, 2000.

[10] D. Brooks, V. Tiwari, and M. Martonosi. Wattch: A Framework for Architectural-Level Power Analysis and Optimizations. In *Proceedings of the 27th International Symposium on Computer Architecture*, pages 83–94, Vancouver, BC, June 2000.

[11] V. Tiwari, S. Malik, A. Wolfe, and M. Lee. Instruction Level Power Analysis and Optimization of Software. *Journal of VLSI Signal Processing Systems*, 13(2–3):223–238, 1996.

[12] V. Tiwari, S. Malik, and A. Wolfe. Power analysis of embedded software: A first step toward software power minimization. *IEEE Transactions on VLSI Systems*, 2:437–445, December 1994.

[13] Y. Li and J. Henkel. A Framework for Estimating and Minimizing Energy Dissipation of Embedded HW/SW Systems. In *Proceedings of Design Automation Conference*, pages 188–193, 1998.

[14] B. Kapoor. Low Power Memory Architectures for Video Applications. In *Proceedings of 8th Great Lakes Symposium on VLSI*, pages 2–7, 1998.

[15] ARM Limited. *ARM Developer Suite version 1.1*, 1999.

[16] The Open SystemC Initiative, URL: http://www.systemc.org. *SystemC version 2.0 — User's Guide (Update for SystemC 2.0.1)*, 2002.

[17] The Open SystemC Initiative, URL: http://www.systemc.org. *Functional Specification for SystemC version 2.0 (Update for SystemC 2.0.1)*, 2002.

[18] Synopsys Inc. *Describing Synthesizable RTL in SystemC (Version 1.1)*, 2002.

[19] J.D. Foley, A. van Dam, S.K. Feiner, and J.F. Hughes. *Computer Graphics: Principles and Practice, Second Edition in C*. Addison-Wesley, 1996.

[20] T. Möller and E. Haines. *Real-Time Rendering*. A K Peters, Ltd., 1999.

[21] M. Woo, J. Neider, T. Davis, and D. Shreiner. *OpenGL Programming Guide, Third Edition, The Official Guide to Learning OpenGL, Version 1.2*. Addison-Wesley, 1999.

[22] T. Grötker, S. Liao, G. Martin, and S. Swan. *System Design with SystemC*. Kluwer Academic Publishers, 2002.

[23] The Open SystemC Initiative (OSCI), URL: http://www.systemc.org.

[24] A. Fountain, J. Huxtable, P. Ferguson, and D. Heller. *The Defini-*

tive Guides to the X Window System, Vol. 6A — Motif Programming Manual for Motif 2.1. O'Reilly & Associates, Inc., 2001.

[25] A. Fountain and P. Ferguson. *The Definitive Guides to the X Window System, Vol. 6B — Motif Reference Manual for Motif 2.1*. O'Reilly & Associates, Inc., 2001.

[26] W.R. Stevens. *Advanced Programming in the UNIX Environment*. Addison-Wesley, 1993.

[27] A. Herrera. Technology and Solutions for Antialiasing of Computer Graphics. Technical report, Jon Peddie Associates, 2000.

[28] D. Crisu, S.D. Cotofana, and S. Vassiliadis. An Energy-Aware Architectural Exploration Tool for ARM-Based SOCs. In *Proceedings of 12th Annual Workshop on Circuits, Systems and Signal Processing, ProRISC 2001*, November 2001.