# GRAAL: A Framework for Low-Power 3D Graphics Accelerators

**Ben Juurlink, Iosif Antochi, Dan Crisu, Sorin Cotofana, and Stamatis Vassiliadis** ▪ *Delft University of Technology*

With the advent of the system-on-chip (SoC) design paradigm for embedded systems, 3D graphics accelerators for mobile platforms are becoming increasingly popular. To address the lack of specific tools for exploring this graphics architectural-design space and to enable early design trade-offs among image quality, performance, power, and cost, we developed GRAAL (GRAphics AcceLerator). GRAAL is a versatile framework that supports hardware/software cosimulation and codesign. The framework is open in that it's built on a basic library of SystemC register-transfer-level (RTL) models of graphics pipeline components. If required, designers can easily augment these models with new components. This flexibility, along with a well-structured design methodology, offers designers a coherent environment for design-space exploration. GRAAL lets designers verify and evaluate various architectural solutions without having to complete a low-level design. It also includes tools to assist visual debugging of the graphics algorithms implemented in hardware and to estimate an accelerator's throughput, power consumption, and area.

One significant feature of GRAAL that compares favorably with other 3D graphics frameworks such as Attila[1] is that it allows the evaluation of traditional as well as tile-based renderers. Tile-based rendering is promising from a low-power perspective because it decomposes a scene into smaller parts—tiles that can be rendered one by one. A small memory integrated into the graphics accelerator can store the color components and depth (*z*) values of one tile. Thus, most access is local, on-chip access, which consumes significantly less power than access to off-chip frame and *z*-buffers.

A power/energy-estimation framework is another important part of GRAAL. We developed two power-estimation strategies for use early in design. This approach is an alternative to field-programmable gate array (FPGA) prototypes, which can't offer accurate energy or performance estimates. Furthermore, academic power-estimation tools (such as Wattch[2]) and instruction-level power analysis (for example, see Vivek Tiwari and colleagues[3]) are based on general-purpose processors and can hardly be adapted to model SoC designs.

We begin our article with an overview of the framework. Then we describe the tiling engine, which sends the graphics primitives to the rasterizer in tile-based order; this description includes benchmark study results for our tile-based rendering approach. We conclude with the power-estimation strategies.

**The GRAphics AcceLerator (GRAAL) design-exploration framework is an open system that offers a coherent development methodology for hardware/software cosimulation and codesign of embedded 3D graphics accelerators. GRAAL incorporates tools to help visually debug graphics algorithms implemented in hardware and to estimate performance in terms of throughput, power consumption, and area.**

## The proposed simulation framework

Figure 1 (next page) depicts the GRAAL tool set. Running OpenGL applications through the Mesa library generates input to the simulator. The library is augmented so that it sends specific rasterization primitives to the simulator. The application can also be run through an OpenGL tracer that logs the graphics library calls. We used the tracer to generate the workloads used in the benchmark study.

Gathering useful design exploration data requires a graphics library, a driver, and a graphics software application. To this end, we developed an OpenGL-compatible library for the GRAAL framework, borrowing source code from the Mesa library together with the device driver for the graphics

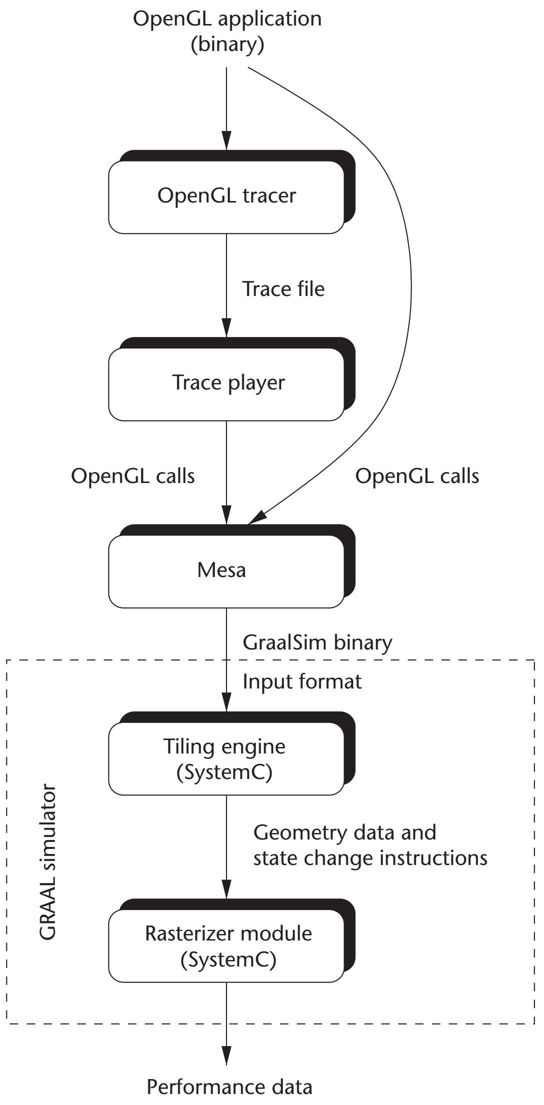

**Figure 1. GRAAL tool set. Running OpenGL applications through the OpenGL tracer, the trace player, and the Mesa implementation of the OpenGL library generates input to the simulator.**

accelerator. If the OpenGL application's source code is available, the only subsequent steps to be taken are to port the application to the SoC platform software environment and then compile and link it with the OpenGL-compatible library using native SoC platform application-development tools. However, graphics applications with advanced features that stress the graphics accelerator are often available only in binary form, usually for PCs. For these situations, we dynamically link the binary application to a tracer library, which traces and logs all calls and relevant data in files. Then we use a player to re-create the original graphics calls and data from those files. Kari Kangas and his colleagues at Nokia propose an alternative to using interactive real-world 3D graphics workloads.[4] They describe a scalable synthetic content approach for measuring OpenGL ES (embedded systems) 3D graphics performance of mobile devices.

The Mesa module executes the Transform and Lighting (TnL) stages of the OpenGL pipeline in software and sends rasterization instructions to the tiling engine. The tiling engine and the rasterizer module form the core of the simulator. Both modules are implemented in SystemC. The tiling engine sorts the primitives into bins corresponding to tiles and sends them to the rasterizer in tile-based order. We have evaluated several sorting or scene-management algorithms (described later). Because the tile size is configurable, we can simulate a conventional renderer by setting the tile size to the frame size.

Finally, the rasterizer module receives geometry data and state-change instructions from the tiling engine, renders the primitives, and generates performance data. The rasterizer is divided in the functional stages of the OpenGL pipeline, but a functional stage describes only the task to be performed and not the way it's executed in the underlying hardware pipeline. Designers can divide a functional pipeline stage into several hardware pipeline stages, implement two functional pipeline stages in one hardware stage, or parallelize a hardware pipeline stage to meet performance demands. Moreover, for every function performed in the rasterizer stage, several hardware algorithms might exist. Within the data path, the algorithm can employ various fixed-point data formats and precision levels that might affect the generated image's quality. As a consequence, designers must explore different image quality, performance, power, and cost trade-offs to choose the best solution for the graphics accelerator under development—and do so for the huge design space under tight time-to-market constraints.

Figure 2 depicts the simulation framework, which is the tool set's core, in more detail. Central elements are

- the reference implementation of a SystemC simulator;
- the GRAAL simulator, our own custom-designed tool, which acts as a graphical front end for SystemC simulation control, data visualization, and performance estimation;
- the SystemC model of the candidate graphics accelerator designed using our (extendable) library of graphics pipeline components modeled in SystemC at RTL;
- third-party SystemC models of SoC components, including a processor core, memories, and peripherals to simulate the entire system; and
- the graphics application in binary form, which runs on the processor core and uses the accelerator's capabilities.

Within this framework, we can run different applications on the virtual system as well as develop, verify, evaluate, and optimize different graphics accelerator microarchitectures without realizing a physical prototype.

### *3D graphics component library*

To facilitate design exploration, we modeled a library of OpenGL-compatible hardware modules in SystemC. The modules include all rasterizer stage functionality and can be plugged together to build a complete graphics accelerator. A designer can use these models as microarchitectural templates to support further refinement. The library amounts to approximately 28K lines of SystemC code and has the following features:

- All modules have a fully parameterizable data path using SystemC/C++ templates.
- All modules (except the system interface) are described at RTL with operators specified at the word level in the data path. (They can be further refined to the bit level.)
- All modules can be configured to support tile-based rasterization.
- Besides modules that implement OpenGL functionality, the library provides modules for tile and state management and for interfacing in SoC using the Open SystemC Initiative (OSCI) transaction-level models; it also provides finer-grain modules to implement various data-path operators and pseudomodules to implement performance-related counters or to allow graphical visualization.

Using the library, the simulation framework can gather the following performance data during simulation:

- number of frames generated per second;
- number of primitives rasterized (shaded, antialiased, and textured);
- number of fragments entering the per-fragment operations stage and number of fragments discarded in each substage (pixel flow estimation);
- total number of clock cycles to produce a frame or a number of frames;
- number of clock cycles the graphics unit is busy processing and where these cycles are spent (rasterization setup, pixel fill engines, texture units);
- number of clock cycles spent stalling the hardware units;
- number of transactions and data traffic at the graphics accelerator interface to the overall system;
- frame buffer and local buffers usage (number of reads, writes, and stalls);

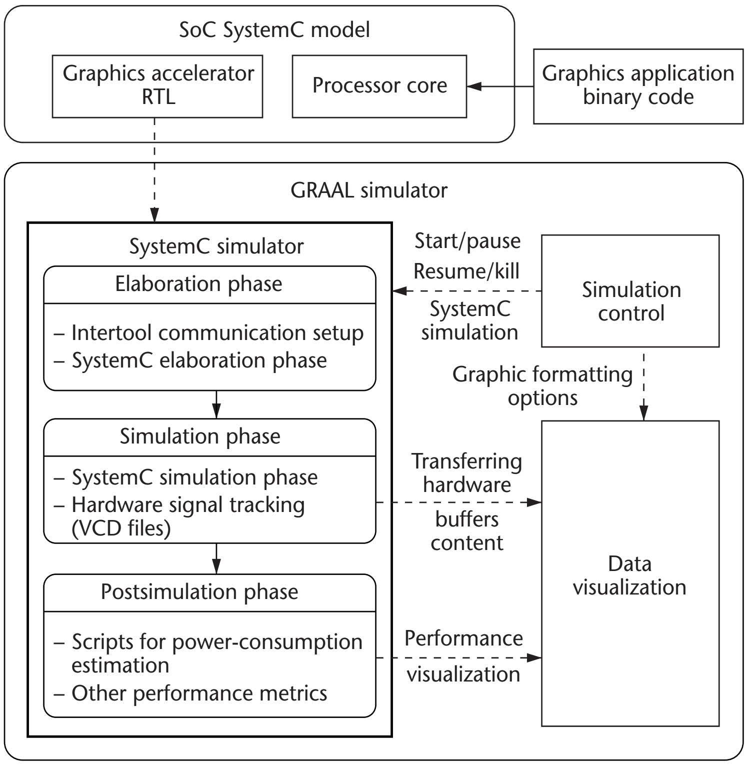
SoC SystemC model
Graphics accelerator RTL
Processor core
Graphics application binary code
GRAAL simulator
SystemC simulator
Elaboration phase
– Intertool communication setup
– SystemC elaboration phase
Simulation phase
– SystemC simulation phase
– Hardware signal tracking (VCD files)
Postsimulation phase
– Scripts for power-consumption estimation
– Other performance metrics
Start/pause
Resume/kill
SystemC simulation
Simulation control
Graphic formatting options
Transferring hardware buffers content
Performance visualization
Data visualization

**Figure 2. The GRAAL simulator framework. The framework integrates SystemC simulation together with simultaneous graphics visualization and performance estimation.**

- runtime communication of graphics-related data to graphical visualization modules; and
- hardware signal transitions in VCD (value change dump) files for on-screen debugging of hardware traces and estimations of power consumption.

A designer can employ all these statistics to check the balance of the graphics microarchitecture and to suggest changes in the next iterations of the design-exploration process.

### *Visualization and simulation control*

The GRAAL simulator provides a graphical front end for SystemC simulation control, data visualization, and performance estimation in our design-exploration framework. Figure 3 (next page) shows the interface. It features command menus that let the designer visualize—interactively and at a customizable detail level—the content of the various system buffers and overlay it with other auxiliary data (such as the pixel center and primitive geometry) that can help with visual debugging. The GRAAL simulator is implemented using the Open Software Foundation's Motif toolkit for Unix/X Window System workstations. For data visualization, we employed the pseudomodules mentioned earlier to snoop relevant buses and to communicate the captured data via first-in, first-out special files to the GRAAL simulator.

**Table 1. Number of triangles transferred for various tile sizes.**

| | Benchmarks | | | | | |
|---|---|---|---|---|---|---|
| **Tile size** | **Q3H** | **Tux** | **AW** | **ANL** | **GRA** | **DIN** |
| 16 × 16 | 21,300 | 8,204 | 15,627 | 18,731 | 9,416 | 9,416 |
| 16 × 32 | 15,600 | 6,101 | 14,464 | 13,850 | 8,215 | 7,905 |
| 16 × 64 | 13,009 | 5,143 | 13,911 | 11,555 | 7,624 | 7,142 |
| 32 × 16 | 14,662 | 5,539 | 14,187 | 14,823 | 7,183 | 7,954 |
| 32 × 32 | 10,526 | 4,148 | 13,090 | 10,689 | 6,217 | 6,591 |
| 32 × 64 | 8,671 | 3,576 | 12,567 | 8,745 | 5,742 | 5,904 |
| 64 × 16 | 11,360 | 4,225 | 13,480 | 12,910 | 6,071 | 7,216 |
| 64 × 32 | 8,006 | 3,245 | 12,416 | 9,150 | 5,223 | 5,928 |
| 64 × 64 | 6,518 | 2,813 | 11,908 | 7,308 | 4,807 | 5,278 |
| 640 × 480 | 3,404 | 1,822 | 10,768 | 4,321 | 3,603 | 4,083 |

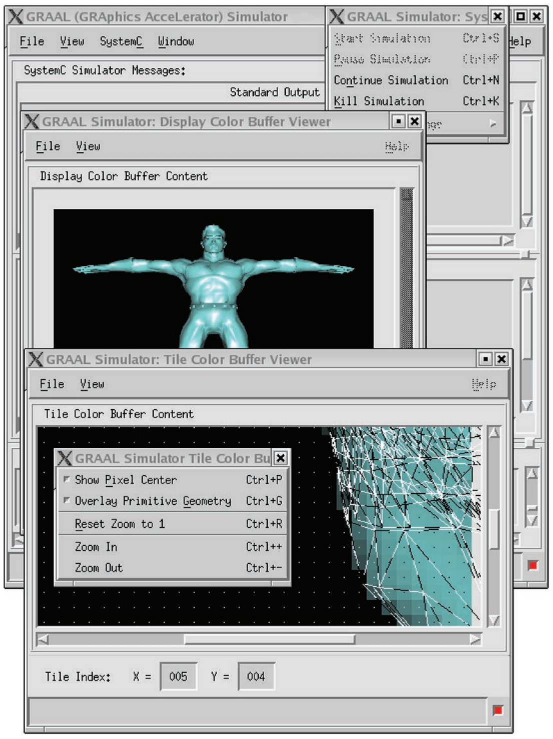
GRAAL (GRAphics AcceLerator) Simulator
File View SystemC Window
SystemC Simulator Messages:
Standard Output
GRAAL Simulator: Sys
Start Simulation Ctrl+S
Pause Simulation Ctrl+P
Continue Simulation Ctrl+N
Kill Simulation Ctrl+K
Help
GRAAL Simulator: Display Color Buffer Viewer
File View Help
Display Color Buffer Content
GRAAL Simulator: Tile Color Buffer Viewer
File View Help
Tile Color Buffer Content
GRAAL Simulator Tile Color Bu
Show Pixel Center Ctrl+P
Overlay Primitive Geometry Ctrl+G
Reset Zoom to 1 Ctrl+R
Zoom In Ctrl++
Zoom Out Ctrl+-
Tile Index: X = 005 Y = 004

**Figure 3.** SystemC simulation control and graphical visualization in the GRAAL simulator. The simulation can be controlled accurately using the menus, and various markers can be overlaid to aid in the visual debugging of algorithms.

## Tiling engine

The tiling engine sorts the primitives into bins and sends them to the rasterizer in tile-based order. As mentioned earlier, tiling appears to be promising for low-power implementations because it reduces the memory bandwidth required between the rasterizer and the (external) frame and *z*-buffers. Accesses to external memory often dissipate more energy than data paths and control units do, so reducing accesses can provide significant energy savings.

In this section, we investigate how much external data traffic is saved by a tile-based renderer compared to a traditional renderer. Because the sorting step requires memory bandwidth, we also evaluate several algorithms for performing this step. Finally, we investigate how to reduce the state-change information.

### *Memory bandwidth requirements of tile-based rendering*

We begin by examining how the amount of external data traffic varies with tile size to identify the tile size that yields the best trade-off between data traffic volume and area needed for on-chip buffers. Furthermore, we measure how much external data traffic is saved by a tile-based renderer. Previous studies haven't presented such measurements[5] or focused on the *overlap*—that is, the average number of tiles covered by a primitive.[6]

We used the GraalBench benchmark suite.[7] Q3L and Q3H are traces of the popular Quake III game. Tux is a freely available game that runs on Linux. AW is the AWadvs-04 test that's part of the Viewperf 6.1.2 package. ANL, GRA, and DIN are Virtual Reality Markup Language scenes chosen on the basis of their diversity and complexity. All workloads use VGA resolution (640 × 480) except Q3L, which uses QVGA resolution (320 × 240). Furthermore, the first three traces consist of about 1,400 frames while the latter four consist of about 600 frames.

Table 1 depicts the number of triangles transferred from the tiling engine to the rasterizer for various tile sizes. The last row shows the number of triangles transferred if the tile size is equal to the window size. The overlap can therefore be obtained by dividing the number of triangles transferred for a specific tile size by the number given in the last row.

Obviously, if the tile size increases, the number of transferred triangles decreases, because there is less overlap. In our design, however, it's important to use as little internal memory as possible. Therefore, we must make a trade-off. The results show that using tiles smaller than 32 × 32 pixels significantly increases the number of triangles transferred. For example, if a tile size of 16 × 16 is employed, the amount of geometrical data sent to the rasterizer increases by 2.02 times for the Q3H benchmark and by 1.97 times for the Tux benchmark. On average, using the geometric mean, a tile size of 16 × 16 increases the number of triangles sent by 1.62 times when compared to 32 × 32 tiles.

On the other hand, employing tiles larger than 32 × 32 reduces the amount of geometrical data only marginally. For example, using 64 × 64 instead of 32 × 32 tiles reduces the data by 1.35 times (geometric mean). This indicates that for the considered workloads, a tile size of 32 × 32 is the best trade-off between the number of triangles sent to the rasterizer and the internal buffer size. The resolution might affect the optimal tile size for nonscalable applications. In practice, however, most applications scale their content complexity according to the resolution. Therefore, a resolution change can't substantially affect the optimal tile size. If each element is represented by 32 bits for RGBA (red, green, blue, alpha) color, 24 bits for depth, and 8 bits for stencil, 64 kilobits are required to implement 32 × 32 tile buffers. In a static RAM implementation, this corresponds to about 96K equivalent gates. We can compare this to the current gate budgets available for mobile graphics accelerators, which are around 200K to 500K gates.

Figure 4 depicts the total amount of external data traffic produced by the traditional rendering (left bar) and tile-based rendering (right bar) for each benchmark on a tile size of 32 × 32. For each total, the traffic is divided into data sent from the tiling engine to the rasterizer (light purple) and data transferred between the rasterizer and the external frame, z-buffers, and texture memory (dark purple). Since the latter component is much larger than the former, the tile-based renderer reduces the total external traffic volume significantly, by 1.96 times on average (geometric mean). For some workloads, however, the advantage of tile-based rendering is marginal. For example, for the AW benchmark it is only 23.4 percent. This is because tile-based rendering decreases the data transferred between the rasterizer and the external frame, z buffers, and texture memory (depending on the overdraw—that is, the number of pixels written to a buffer divided by the buffer size, in pixels) but increases the amount of data sent from the tiling engine to the rasterizer (depending on the overlap). Hence, tile-based renderers are more suitable than traditional renderers for workloads with low overlap and high overdraw, a trend foreseen for the future. For workloads with high overlap and low overdraw, on the other hand, tile-based renderers do not reduce the total amount of external data traffic significantly.

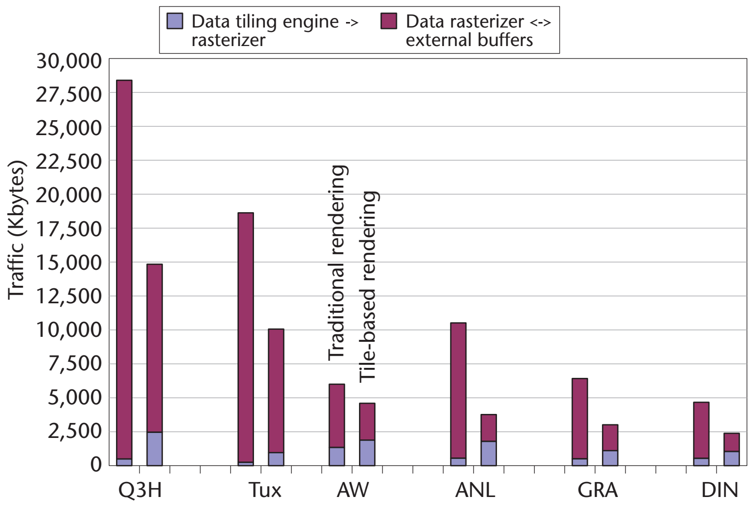

**Figure 4. Total external data transferred per frame for traditional and tile-based rendering on the GraalBench benchmark suite.**

## Scene-management algorithms

The primitives must be sent to the rasterizer in tile-based order. We've developed several scene-management algorithms to do this and evaluated their computational cost and memory requirements.

An important part of the scene-management algorithm is the test that determines if a triangle overlaps a tile. Commonly employed is the so-called bounding-box (BBOX) test, which checks if the axis-aligned bounding box of a triangle intersects the tile. The BBOX test is relatively low cost; it requires only four comparisons. However, it's imprecise because the BBOX might intersect with a tile even though the triangle doesn't, as illustrated in Figure 5.

We have developed an exact test called the linear edge test (LET). Other researchers have proposed this test before but in a different context,[8] and we have adapted it so that no coverage mask is needed. However, LET is computationally more expensive than the BBOX test.

We have proposed and evaluated the following scene-management algorithms:

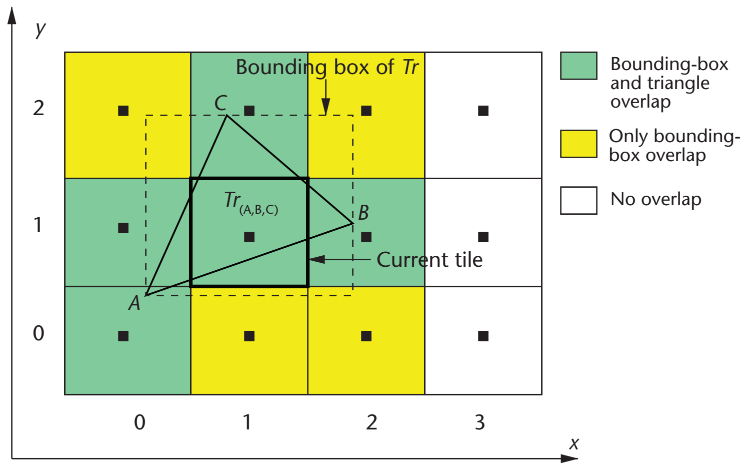

**Figure 5. Imprecision in the triangle-to-tile bounding-box (BBOX) test. In this case, the BBOX overlaps the yellow tiles but the triangle doesn't.**

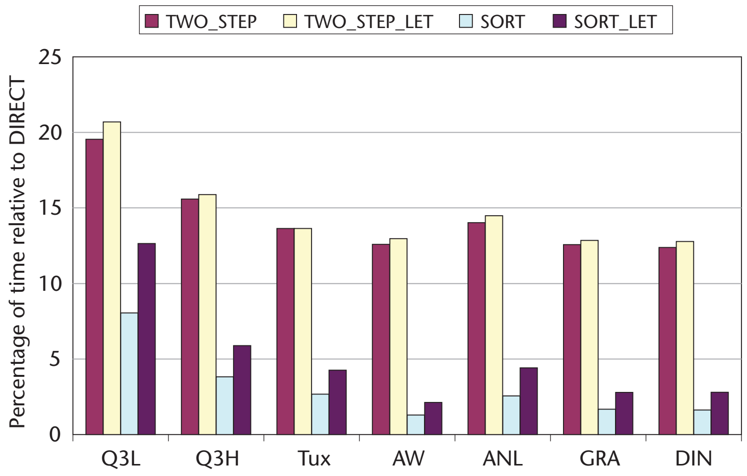

**Figure 6. Estimated time taken by each scene-management algorithm relative to the time taken by the DIRECT algorithm.**

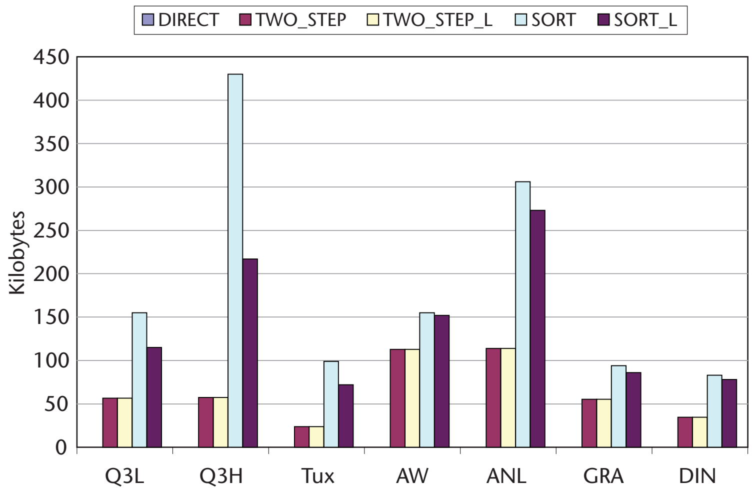

**Figure 7. Memory requirements of the scene-management algorithms. The additional memory required for benchmarks with a small overlap factor (such as AW) is insignificant, but it's considerable for benchmarks with a large overlap factor (such as Q3H).**

- DIRECT: This algorithm scans all primitives for each tile and sends the primitives whose BBOX overlaps the current tile to the rasterizer. Its main advantage is that it requires no memory in addition to the scene buffer, which holds the initial scene geometry and state changes and, optionally, the geometry and state changes sorted per tile.
- TWO_STEP: This algorithm consists of two phases. First, each triangle's BBOX is computed and stored in a buffer. This avoids recomputing the BBOX for each triangle/tile tuple. In the second phase, all triangles are scanned for each tile, and the triangles whose BBOX overlaps the current tile are sent to the rasterizer.
- TWO_STEP_LET: This algorithm is identical to TWO_STEP except that the second phase employs the LET. Because LET contains the BBOX test, the main LET is applied only to the triangles that passed the BBOX test.
- SORT: This algorithm has a buffer for each tile with pointers to the primitives that overlap the tile according to the BBOX test. For each tile, only the primitives that have a pointer in the corresponding buffer are sent to the rasterizer.
- SORT_LET: This algorithm is identical to SORT except that the second phase uses the LET.

As we've described elsewhere in detail,[9] we evaluate the computational and memory requirements of the algorithms by determining them analytically and simulating the GraalBench traces to measure statistics such as BBOX and LET overlap factors. We estimate other parameters by counting the number of elementary operations (assignments, comparisons, and so on) required to implement the operation. We did this because cycle-accurate simulations would be too time-consuming, taking several weeks or more.

Figure 6 depicts the time each algorithm takes relative to the time taken by the DIRECT algorithm to send the primitives to the rasterizer in tile-based order. As expected, DIRECT requires the largest number of operations by far, while SORT takes the least amount of time. On average, across all benchmarks, SORT is 44 times faster than DIRECT. The TWO_STEP algorithm, even though it also scans the entire scene buffer for each tile, has reasonable performance. On average, it is 6 times slower than SORT. Furthermore, TWO_STEP_LET is only slightly slower than TWO_STEP and is, therefore, preferable because it sends fewer triangles to the rasterizer, which means the computational load on the rasterizer is reduced. SORT_LET, on the other hand, is on average 1.6 times slower than SORT.

Figure 7 depicts additional memory required by each algorithm. As explained before, DIRECT does not require any additional memory. Furthermore, as expected, SORT needs the most additional memory because it's proportional to the number of triangles and the BBOX overlap factor. Because the LET test is exact and the BBOX test is not, SORT_LET requires less memory than SORT. However, the difference is significant only for one benchmark (Q3H), for which SORT needs almost twice as much additional memory as SORT_LET. The difference is much smaller for the other benchmarks (1.17 times as much, on average). The reason is that the BBOX test is fairly exact for all benchmarks except Q3H. The TWO_STEP and TWO_STEP_LET algorithms require the same amount of memory. On average, TWO_STEP requires 3.2 times less additional memory than SORT.

However, this difference depends strongly on the benchmark. For benchmarks with a small overlap factor (such as AW), the difference isn't significant. For benchmarks with a large overlap factor (in particular Q3H), the difference is considerable.

### State-management algorithms

Tile-based rendering reduces the memory traffic between the rasterizer and the off-chip frame and *z*-buffers. But it increases the amount of state-change information such as enable/disable *z*-testing and create/delete texture commands that the tiling engine must send to the rasterizer. This is because the tiling engine might have to send the same state-change operation to the rasterizer several times. Consider, for example, the following instruction stream:

```
      EnableDepth
      Triangle(1)
      DisableDepth
      Triangle(2)
      EnableDepth
      Triangle(3)
```

Assume that tile 1 intersects with triangles 2 and 3 and that tile 2 intersects with triangles 1 and 3. If the tile-based driver duplicates the state-change operations for each tile, it generates the following instruction stream:

```
Tile 1
      EnableDepth
      DisableDepth
      Triangle(2)
      EnableDepth
      Triangle(3)

Tile 2
      EnableDepth
      Triangle(1)
      DisableDepth
      EnableDepth
      Triangle(3)
```

However, the italics indicate state-change operations that we can remove. For example, we can remove the first EnableDepth command because it is immediately followed by DisableDepth.

Determining which state-change operations can be removed and when isn't always trivial. For instance, if the driver encounters a DeleteTexture command while rendering the current tile, the texture can be safely deleted only when all primitives (from all tiles) using this texture are rendered or when multiple copies of the texture are kept in memory. Including all state-change operations to each tile isn't practical because it requires duplicating large numbers of state variables (for example, texture objects). In some cases, the state-change operations account for 63 percent of the data sent to the rasterizer.

When using tile-based rendering, the following algorithms can handle state information correctly:

- *Partial-rendering algorithm.* Whenever this algorithm encounters an instruction with side effects (for example, DeleteTexture), the driver renders all previously buffered instructions and then executes the instruction. This solution can introduce significant rendering overhead. For each partial rendering, the introduced overhead consists of saving and reloading the contents of the enabled tile buffers (for example, color, depth, and stencil) from the global buffers and also the state information save and reload operations.
- *Delayed-execution algorithm.* When this algorithm encounters a command with side effects, the driver postpones executing the command until all primitives depending on it have been rendered or until the end of the current frame is reached. This approach reduces the overhead encountered in the partial-rendering algorithm.

Figure 8 depicts the amount of state-change information sent to the rasterizer using these two algorithms on a tile size of 32 × 32. The delayed method reduces the number of writes to the accelerator by filtering the state information and eliminating unnecessary writes. State information for the Q3L, Q3H, GRAZ, and ANL components is reduced by 23 to 58 percent. The state traffic for the AW, Tux, and DIN workloads, however, is not decreased substantially because tiling doesn't introduce any unnecessary state changes for these workloads.

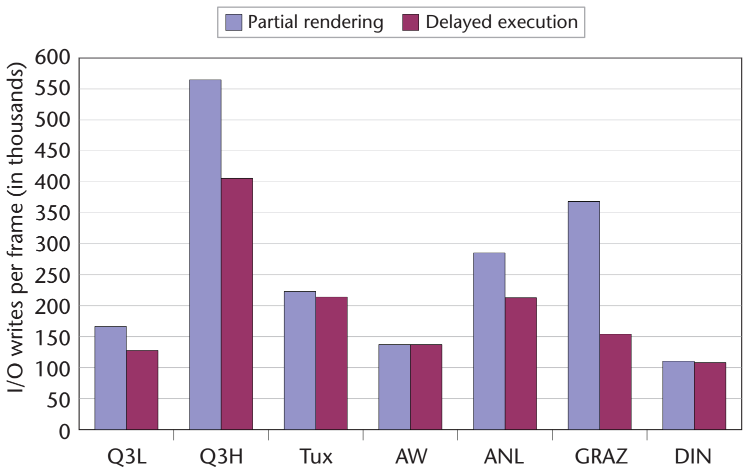

**Figure 8. Average number of state information writes to the accelerator per frame on the GraalBench benchmark suite.**

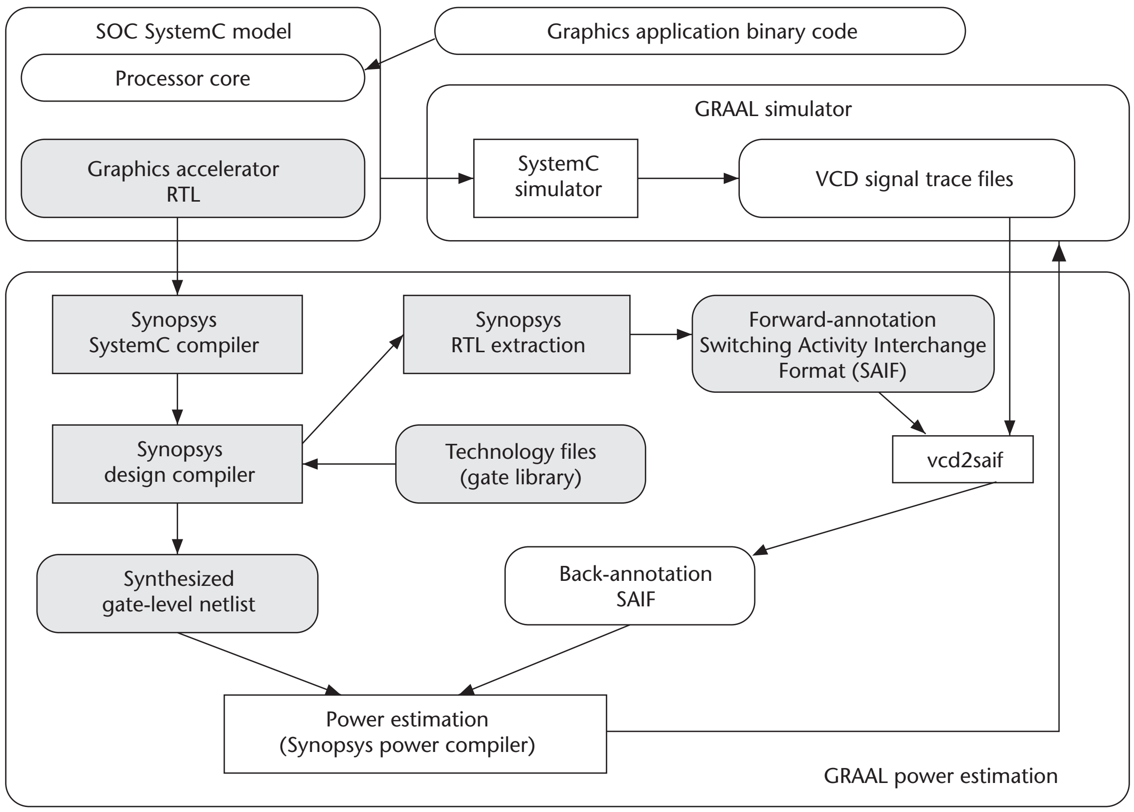
SOC SystemC model
Processor core
Graphics accelerator RTL
Graphics application binary code
GRAAL simulator
SystemC simulator
VCD signal trace files
Synopsys SystemC compiler
Synopsys RTL extraction
Forward-annotation Switching Activity Interchange Format (SAIF)
Synopsys design compiler
Technology files (gate library)
vcd2saif
Synthesized gate-level netlist
Back-annotation SAIF
Power estimation (Synopsys power compiler)
GRAAL power estimation

**Figure 9. GRAAL netlist-level power estimation strategy. Netlists generated via synthesis together with signal changes captured during simulation are used to generate signal switching activities employed by the power estimator.**

## Power estimation

Studies have demonstrated that circuit- and gate-level techniques can reduce power by up to 2 times, but architecture- and algorithm-level strategies offer savings of 10 to 100 times or more.[10] Assessing the merits of a potential implementation early in the design process brings the largest benefits. The GRAAL framework offers two power-estimation strategies based on SystemC simulation. Both strategies estimate the average power consumption over the entire simulation period and produce the energy drawn from the battery as a by-product.

### *Netlist-level power estimation*

Figure 9 illustrates the first strategy.[11] It employs several Synopsys tools (CoCentric SystemC compiler, design compiler, and power compiler). All steps are automated with custom-developed scripts for driving the tools. The prerequisites are a power precharacterized library of standard cells and an initial hardware synthesis step of the SystemC model to produce the gate-level netlist used by the tools.

The darker-colored steps in Figure 9 must be performed once for every candidate implementation. The RTL power-consumption estimation acquires information about switching activity from SystemC RTL simulation. The switching activity is obtained by translating the VCD files—where the hardware-signal traces are logged—to the Switching Activity Interchange Format (SAIF) files, the format recognized by the tools.

We can categorize switching activity as synthesis-variant switching activity (SVSA) and synthesis-invariant switching activity (SISA). SVSA, obtained from presynthesis logic signals in the simulator, comes from the design's combinational the logic, which will be heavily optimized during the synthesis process. SISA comes from the synthesis invariant elements, which generally include inferred registers, inferred tristate devices, hierarchical boundaries, and so on. Because synthesis doesn't modify these elements, SISA information is still valid after the RTL design is mapped to the gate-level netlist.

SVSA information is less accurate and is estimated statistically by propagating synthesis-invariant element-switching activity probabilities to the synthesis-optimized combinational logic trees. This strategy provides estimates accurate enough to be used in the microarchitecture exploration phase, where power estimates corresponding to two candidate microarchitectures are expected to be significantly different (greater than 100 percent).

### *Architecture-level power estimation*

The second strategy uses an approach described by Dan Crisu and his colleagues.[12] The strategy requires more technology-dependent data from the user than the first strategy, and it can deliver estimates within 25 percent of circuit-level simulation accuracy. It's based on Paul Landman's work,[10] which separately analyzed the four main classes of chip components: data path, memory, control, and interconnect.

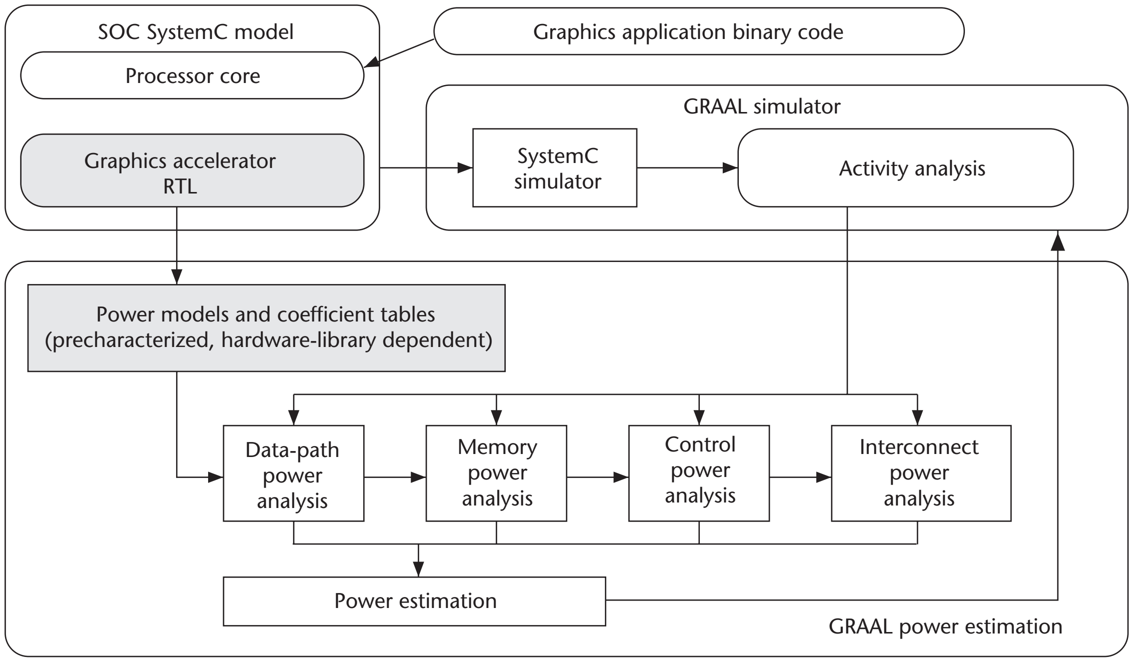

**Figure 10. GRAAL architecture-level power estimation strategy. Template circuit classes derived from power-precharacterized hardware modules are used together with signal activities to estimate power consumption.**

The strategy requires a library of hardware cells consisting of various operators for the data-path part; gates for control logic; and bit cells, decoders, and sense amplifiers for memory cores. Once such a library exists, it can be precharacterized via gate-level and circuit-level simulations, resulting in a table of effective capacitive coefficients for every library element. Using this table and the activity statistics derived during the architectural-level simulation, we can estimate power consumption. The precharacterization must be done only once, and the power estimation requires only the effective capacitive coefficients table. We note here that precharacterization results are valid only for a specific library of hardware cells and a given IC technology.

Figure 10 illustrates the architecture-level power-estimation strategy we propose. User inputs describe a candidate architecture in structural SystemC and the application program. Every clock cycle, the simulation collects the activity on relevant internal signals and sends it to the power-analysis units. The power calculator estimates total power consumption of the graphics accelerator per program executed on the host processor.

Components relevant to the architectural power/energy-estimation framework are

- precharacterized power models and effective-capacitance coefficient tables that contain, for a library of hardware cells, all technology-dependent information required by the power-analysis modules;
- an activity-analysis module that feeds the power-analysis modules with statistics about signal activity inside the simulated hardware description;
- power-analysis modules that estimate the power consumption in the data path, control, memory, and interconnect according to statistics from the activity-analysis module and lookups in the effective-capacitance coefficient tables; and
- a power-estimator module that adds power consumption estimates of the data path, control, memory, and interconnect and produces total power consumption.

The architecture-level power-estimation strategy, although faster and more accurate than the netlist-level strategy, requires a difficult precharacterization on existing libraries of hard cells. Users can employ either one or both estimation strategies, depending on the situation.

## *Case studies*

To illustrate the two power/energy estimation strategies, we designed two hardware circuits. We implemented the first one via synthesis from SystemC RTL. We obtained the other by employing semicustom techniques to generate the hard cells required for the architecture-level power-estimation strategy. The IC technology employed in both cases was a Taiwan Semiconductor Manufacturing Company 0.13 μm 1.2 V CMOS process.

**Synthesizable design.** The first design implements an OpenGL 1.2-compliant 3D graphics accelerator for an ARM-based SoC platform using the SystemC module library described earlier. We incorporated the following OpenGL functionality in hardware:

- triangle rasterization: flat- and Gouraud-shaded with/without antialiasing with all the options controlling rasterization;

**Table 2. Frame workload.**

| Processed triangles | Fragmments | | Frame duration (clock cycles) |
|---|---|---|---|
| | Processed | Passed to color tile frame buffer | |
| 12,362 | 9,511,077 | 8,759,891 | 6,005,722 |

**Table 3. Graphics hardware estimation results.**

| IC technology | | Standard cell library | |
|---|---|---|---|
| TSMC 0.13 $\mu$m 1.2 V | | Artisan SAGE-X | |
| **Clock frecuency** | **Standard number of cells** | **Standard cells** | **Total area** |
| 200 MHz | 78 K | 1.32 mm$^2$ | 2.15 mm$^2$ |
| **Per-frame estimated** | | | |
| **Average power** | | **Energy** | |
| 267 mW | | 8.02 mJ | |

**Table 4. Power consumption results for the ripple-carry subtracter.**

| Instruction trace | Power (estimated) | Power (simulated) | Relative error (%) |
|---|---|---|---|
| A | 211 $\mu$W | 245 $\mu$W | –14 |
| B | 262 $\mu$W | 221 $\mu$W | 19 |
| C | 173 $\mu$W | 163 $\mu$W | 6 |

- texturing with only RGBA8 internal texture format and texture fetching on demand;
- per-fragment operations: scissor test, alpha test, stencil and depth buffer test, blending, logical operation;
- entire frame buffer operations: fine control of buffer updates, clearing the buffers;
- state management: all state management for previously mentioned functionality respecting all the invariance rules imposed by the OpenGL specification.

The software driver processes the other OpenGL primitives (points, lines, and polygons with more than three vertices) and presents them to the graphics accelerator as a combination of triangles.

For the internal organization, the graphics accelerator adopts tile-based rasterization. We set the tile size at 32 × 32 pixels, which implies that all internal buffers (color, depth, and stencil buffers) composing the tile frame buffer have this size. We set the display-size resolution at 320 × 240 pixels (a quarter VGA), meaning that the display consists of 10 × 8 tiles. The fixed-point formats used at the interface with the internal data path are all unsigned. The accelerator has two pixel-processing pipelines. The screen coordinates ($X$, $Y$) are represented on 9.8 bits, the color components (R, G, B, A) on 0.8 bits, the depth component ($Z$) on 0.24 bits, and the stencil component on 8.0 bits.

We generated one frame of the AW benchmark on our virtual SoC platform using the GRAAL framework. Figure 3 shows the resulting image. Table 2 depicts a few characteristics of the frame workload.

Table 3 shows the results of the hardware synthesis of the graphics accelerator and estimated (netlist-level) average power/energy drawn from the battery per frame duration.

**Semicustom design.** To verify the power-consumption prediction accuracy of the architecture-level strategy, we precharacterized parts of a data-path library of cells (including a ripple-carry adder/subtracter) designed using the Alliance VLSI CAD System. From the layout, we extracted the subtracter's circuit for three data-path widths: 4, 8, and 16 bits.

The simulation results on the extracted netlist of the ripple-carry subtracter appear in the third column of Table 4. The clock frequency of the sample adder/subtracter is 200 MHz. The circuit-level simulation of the subtracter took several hours for the three instruction traces executed on the ARM processor, so it's clearly infeasible to obtain the power consumption for this processor directly through a circuit-level simulation of the entire graphics accelerator. The relative error between the estimated power and the power consumption obtained by circuit simulation appears in the last column of Table 4. The accuracy is quite good, well within 25 percent of a direct circuit simulation with Hspice software.

GRAAL is a versatile hardware/software cosimulation and codesign framework for 3D graphics accelerators embedded in mobile terminals. It incorporates tools to assist visual debugging of graphics algorithms implemented in hardware and to estimate the throughput, power consumption, and area. Its tiling engine sends the primitives to the rasterizer in tile-based order. Our experiments indicate that tile-based renderers are more suitable than traditional renderers for workloads with low overlap and high overdraw. On the other hand, they don't reduce external data traffic volume significantly for workloads with high overlap and low overdraw.

GRAAL includes a power/energy-estimation framework, which can be used to explore the design space and perform early evaluation of various graphics accelerator architecture candidates. The framework embeds a netlist-level and architecture-level power-estimation strategy. The case study we presented demonstrates the framework's capabilities.

Although the GRAAL framework can certainly assist designers in the microarchitectural space exploration for (tile-based) 3D graphics accelerators, it can't capture system-integration design aspects, such as workload balancing and overall data traffic optimization. Its extension toward the system level constitutes future work and will make GRAAL a more powerful design framework.

## Acknowledgments

*We gratefully acknowledge Nokia Research Center, Tampere, for kindly providing support for the GRAAL project (grant number IS00012).*

## References

1. V.M. del Barrio et al., "ATTILA: A Cycle-Level Execution-Driven Simulator for Modern GPU Architectures," *Proc. IEEE Int'l Symp. Performance Analysis of Systems and Software* (ISPASS 06), IEEE CS Press, 2006, pp. 231–241.
2. D. Brooks, V. Tiwari, and M. Martonosi, "Wattch: A Framework for Architectural-Level Power Analysis and Optimizations," *Proc. 27th Int'l Symp. Computer Architecture*, IEEE CS Press, 2000, pp. 83–94.
3. V. Tiwari et al., "Instruction-Level Power Analysis and Optimization of Software," *J. VLSI Signal Processing Systems*, vol. 13, nos. 2–3, 1996, pp. 223–238.
4. K.J. Kangas, M. Qvist, and K. Pulli, "Synthetic Content Approach for Benchmarking Mobile 3D Graphics," *Proc. Ann. Svenska Föreningen för Grafisk Databehandling Svenska Föreningen för Grafisk Databehandling Conf.* (SIGRAD 05), 2005, pp. 41–43.
5. E. Hsieh, V. Pentkovski, and T. Piazza, "ZR: A 3D API Transparent Technology for Chunk Rendering," *Proc. 34th ACM/IEEE Int'l Symp. Microarchitecture*, 2001, IEEE CS Press, 2001, pp. 284–291.
6. M. Cox and N. Bhandari, "Architectural Implications of Hardware-Accelerated Bucket Rendering on the PC," *Proc. 1997 Siggraph/Eurographics Workshop on Graphics Hardware*, ACM Press, 1997, pp. 25–34.
7. I. Antochi et al., "GraalBench: A 3D Graphics Benchmark Suite for Mobile Phones," *Proc. ACM SIGPLAN/SIGBED Conf. Languages, Compilers, and Tools for Embedded Systems* (LCTES 04), ACM Press, 2004, pp. 1–9.
8. A. Schilling, "A New Simple and Efficient Antialiasing with Subpixel Masks," *Computer Graphics*, vol. 25, no. 4, 1991, pp. 133–141.
9. I. Antochi et al., "Scene Management Models and Overlap Tests for Tile-Based Rendering," *Proc. EUROMICRO Symp. Digital System Design*, IEEE CS Press, 2004, pp. 424–431.
10. P. Landman, "High-Level Power Estimation," *Proc. Int'l Symp. Low Power Electronics and Design*, IEEE Press, 1996, pp. 29–35.
11. D. Crisu et al., "GRAAL—A Development Framework for Embedded Graphics Accelerators," *Proc. Design, Automation and Test in Europe* (DATE 04), vol. II, IEEE CS Press, 2004, pp. 1366–1367.
12. D. Crisu et al., "High-Level Energy Estimation for ARM-Based SOCs," *Proc. 3rd Int'l Workshop Computer Systems: Architectures, Modeling and Simulation* (SAMOS III), LNCS 3133, Springer, 2004, pp. 168–177.

**Ben Juurlink** *is an associate professor at Delft University of Technology, the Netherlands. His research interests include multicore and many-core processors and application-specific ISA extensions. He has a PhD in computer science from Leiden University, the Netherlands. He is a senior IEEE member and a member of the ACM and the European Network of Excellence on High Performance and Embedded Architecture and Compilation (HiPEAC). Contact him at benj@ce.et.tudelft.nl.*

**Iosif Antochi** *is a leading design engineer at Imagination Technologies Ltd., UK. His research interests include low-power 3D graphics algorithms and architectures, tile-based rendering systems, 3D graphics workload generation and characterization, computer architectures, multimedia processors, and embedded systems. He has a PhD in computer engineering from Delft University of Technology, the Netherlands. Contact him at tkg@ce.et.tudelft.nl.*

**Dan Crisu** *is a senior design engineer with Imagination Technologies Ltd., UK. He is completing his PhD in computer engineering at Delft University of Technology, the Netherlands. His research interests include low-power embedded graphics architectures, their hardware organization and modeling. He is a member of the IEEE. Contact him at dan@ce.et.tudelft.nl.*

**Sorin Cotofana** *is an associate professor with the Computer Engineering Laboratory at Delft University of Technology, the Netherlands. His research interests include computer arithmetic, parallel architectures, embedded systems, nanotechnology, design for variability and reliability, reconfigurable computing, and computer-aided design. He received his PhD in computer engineering from Delft University of Technology, the Netherlands. He is a senior IEEE member and a member of the European Network of Excellence on High Performance and Embedded Architecture and Compilation (HiPEAC). Contact him at sorin@ce.et.tudelft.nl.*

**Stamatis Vassiliadis** *was a chair professor in the faculty of Electrical Engineering, Mathematics, and Computer Science at Delft University of Technology, the Netherlands. His research interests included embedded systems, hardware/software co-design, nanocomputing, and network processing. He received his Dr.Eng. in electronic engineering from the Politecnico di Milano, Italy. He was an IEEE Fellow whose career included many awards as well as 72 US patents from a 10-year period of employment with IBM. He passed away in April 2007.*