

| | | |
|-----|---|----|
| 1. | Introduction..... | 1 |
| 2. | Infrastructure for Microarchitectural Power Simulation..... | 4 |
| 2.1 | Where to Get the Source Code..... | 5 |
| 2.2 | How to Compile..... | 5 |
| 2.3 | How to Run the Simulator..... | 5 |
| 3. | MOSFET Capacitance Component..... | 8 |
| 3.1 | Model..... | 8 |
| 3.2 | Implementation..... | 9 |
| 4. | Interconnect Capacitance and Resistance..... | 13 |
| 4.1 | Models..... | 13 |
| 4.2 | Implementation..... | 15 |
| 5. | General Circuits..... | 17 |
| 5.1 | Models..... | 17 |
| 5.2 | Implementation..... | 20 |
| 6. | Memory Power Model..... | 21 |
| 6.1 | Models..... | 21 |
| 6.2 | Implementation..... | 22 |
| 6.3 | Calibration..... | 26 |
| 7. | Datapath and Execution Unit..... | 31 |
| 7.1 | Implementation..... | 31 |
| 7.2 | Calibration..... | 32 |
| 8. | Clock Distribution Tree..... | 34 |
| 8.1 | Models..... | 34 |
| 8.2 | Implementation..... | 38 |
| 9. | I/O..... | 40 |
| 9.1 | Model..... | 40 |
| 9.2 | Implementation..... | 43 |
| 10. | Conclusion..... | 45 |
| | Appendix A - Sim-iPAQ..... | 46 |
| | References..... | 50 |
| | Publications..... | 53 |

1. Introduction

Power consumption has quickly become a key design constraint in microprocessor designs, from low-end embedded processors to high-end, high-performance systems. The embedded processors found in PDAs and cell phones must utilize energy efficient designs, as their energy payload is limited by form factor and weight constraints. With battery power density improving only at a rate of about 5% per year, increase in battery lifetime must come about through improvements in the energy efficiency of system components. To create power-sensitive designs, accurate power estimation combined with architectural or system level performance simulation is a key design tool that permits rapid early design studies that gauge trade-offs between performance and power.

Recently, several microarchitectural-level power estimation tools have been introduced [1, 2, 3] in academia, and they have been widely adopted for use in design studies that require power modeling. In all of these tools, microprocessor power is estimated by accruing power as estimated by the power models for each access to microarchitectural functional blocks. In *Wattch* [1], Brooks et al. extended CACTI, an access and cycle time model for on-chip caches [4], to model the power dissipation of on-chip storage blocks such as caches, register files, and branch target buffers. The model used in *Wattch* resorts to a fast approximation that is well suited for high-end designs containing large and complex memory, but the power consumption of datapath and execution blocks is estimated by a single, per-access value, which is not scalable for the technology nor different circuit styles. Although their approach is a good approximation for the high-end application domain, we believe that embedded designs require more accurate modeling based on the specific switching activity within each execution block.

In *SimplePower* [2], Vijaykrishnan et al. incorporated *register-transfer level* (RTL) power models based on *look-up tables* (LUT) into a microarchitectural simulator. Each LUT contains a

set of pre-characterized power dissipations for a datapath component, and each entry of the LUT, indexed by the Hamming distance between subsequent input vector pairs, returns the estimated power of the component [5]. The objective of this tool is to provide a framework to quickly evaluate a range of architectural and algorithmic trade-offs during the early design stages. To this end, it targets a reference processor design for the pre-computation of capacitance tables. This reference design, while accurate enough for the purpose of trade-off analysis, is not easily modifiable to describe specific alternative designs that may have different datapath widths, smaller feature sizes, or different technologies. On the other hand, an evaluation of power dissipation in later design stages would obviously benefit from referencing the specific design under development. Those microarchitectural-level power modeling tools have been invaluable in giving computer architects the insights necessary to develop first-generation microarchitectural power optimizations. However, a rapidly changing technology landscape combined with increasingly complex microarchitectural features has brought about an erosion in the fidelity of existing power models.

In this report, we outline the methodology behind the Sim-Panalyzer program. It is an augmentation to the SimpleScalar performance simulator that allows the user to estimate power consumption. It is broken out into several components that model distinct parts of a computer: cache power models; datapath and execution unit power models; clock tree power models; and I/O power models. These power models can be configured into an augmented SimpleScalar simulator that will then produce power consumption figures.

There are a number of artifacts in SimpleScalar that can cause the event counting needed to calculate dynamic power. These artifacts are discussed in [27]. Our power analyzer program, Sim-Panalyzer, accounts for these.

The rest of the report is divided as follows. Section 2 details the infrastructure and explains how to use Sim-Panalyzer. Sections 3 through 5 provide some background for our approach to modeling MOSFETs, interconnect, and circuits in general. Sections 6 through 9 apply these modeling techniques to cache power, datapath and execution unit power, clock tree power, and I/O power, respectively. Section 10 adds some concluding remarks. The appendix on Sim-iPAQ is included because it represents a typical low power platform, and it was developed as part of the same project that supported the development of Sim-Panalyzer. Finally, the Publications section lists papers that were written with the support of this project.

2. Infrastructure for Microarchitectural Power Simulation

Sim-Panalyzer is an infrastructure for microarchitectural power simulation. It is implemented on top of “sim-outorder”, a component within the SimpleScalar [6] simulator. To minimize the modification of the original “sim-outorder.c”, we implemented minimum interfaces to gather microarchitectural activities such as cache accesses. Originally, this project was targeted for the ARM instruction set architecture (ISA) in which there are no complex microarchitectural block such as an instruction queue (IQ), re-order buffer (ROB), branch predictor, floating-point unit, etc. However, now we also provide a platform for the Alpha ISA. Our main focus is on basic microarchitectural blocks and major power dissipation sources such as clock distribution trees, external I/O, on-chip memories, and execution blocks.

The user must specify an effective switching capacitance per access, which is used to compute the energy dissipation of each microarchitectural block. Sim-Panalyzer computes the energy dissipation with the switching capacitance multiplied by the number of microarchitectural accesses. A different scheme is applied for external I/O accesses; we provide a more detailed transaction model to count I/O pin switches in a cycle accurate way. We also provide various routines and parameters to guide the user to estimate the effective switching capacitance per access of each block. Those routines are technology scalable. Thus, we provide usages and examples on how to port different technologies into our infrastructure.

In addition to these features, we provide a power modeling methodology and library to support more sophisticated and accurate power models. In the library, we provide basic building blocks for the embedded logic simulator and switching capacitance extraction for CMOS gates. The logic simulator collects the number of switchings in each internal node of the target circuit or functional block and the capacitance extractor estimates the switching capacitance of each node.

This library supports hierarchical implementations of functional blocks. Thus, the users can re-use the previously implemented sub-blocks to build more complex functional blocks.

2.1 Where to Get the Source Code

Source code can be downloaded from our website. Go to it at <http://www.eecs.umich.edu/~panalyzer>, and click on the link to Sim-Panalyzer 2.0. The source code, “sim-panalyzer-2.0.tar.gz”, is a compressed tar ball file.

The tar ball version has been created in a Linux x86 environment. We have not tested our code for other operating systems and target machines.

2.2 How to Compile

Untar “sim-panalyzer-2.0.tar.gz” into your install directory. Sim-Panalyzer has currently been compiled using gcc 3.2. Other gcc versions have not been tested thoroughly, therefore we recommend that you compile with this version of gcc. We compiled the source code using GN make. ‘make sim-panalyzer’ generates a binary for the simulator. Go to the root directory for each version ‘./Implementations/targetmachine’ and execute this command. This should generate the executable file ‘sim-panalyzer’. For simple tests you can execute small programs under the “./Implementations/targetmachine/tests” directory. These are provided from the simplescalar toolset.

Sample tools used to extract effective capacitance for various functional blocks can be built by going to the ‘./pmodel/’ directory and executing *make*.

2.3 How to Run the Simulator

We tried to decouple the power related configurations from the architectural configurations. To simplify use, we have created a separate script file that parses the *cmd* file. The format for a *cmd* file is similar to a Microsoft Windows *ini* file. We divide the configuration variables into sections and parse through these sections to generate an appropriate configuration for our simulator.

```
[Component]
AIO
DIO
IL1 Cache
DL1 Cache
IL2 Cache
DL2 Cache
ITLB
DTLB
Branch_Predictor
Bimodel
Level1
Level2
BTB
RAS
IRF
FPRF
Random Logic
Clock

[Global]
supply_voltage=1.8
frequency=200

[AIO]
frequency=200
IO_voltage=3.3
numberofbufferstages=5
microstrip length=10
external load=1

[DIO]
frequency=200
IO_voltage=3.3
numberofbufferstages=5
microstrip length=10
external load=1
```

Figure 1: Example of *cmd* file

An example of a *cmd* file is shown in Figure 1. Power configurations can be given as follows below.

The [Component] section that is shown in the beginning of Figure 1 represents the components we intend to analyze for power. Currently the components we support are Caches, Branch

Target Buffers, Branch Predictors, Register files, Clock Trees & Random Logic. Based on the chosen components in the [Component] section we define the configuration variables in the following subsections. For example, the [AIO], which configures the address IO pads, has the parameters “frequency” for the bus frequency, “IO_voltage” to describe the supply voltage for the IO pad, “Buffer ratio” for buffer sizing, “microstrip length” for modeling the PCB, and finally “external load” to model the load that is connected to this IO. “test_arm.cmd” & “test_alpha.cmd”, which are located in the source code, are template command files the user can use as a reference.

It is important to note that in the *cmd* file, we assume capacitance to be in pF, time unit to be in ps, frequency to be in MHz, and voltage to be in V.

The power configurations are then integrated with the architectural configurations and create a single configuration file. We provide architectural templates for a 4-wide issue Alpha microprocessor and the SA1100 StrongARM. Power configuration templates are also provided for these two microprocessors in the “./cmd_files/” directory. The typical method for executing Sim-Panalyzer would be executing the *gen_cfg_<target machine>.pl* script and then using the generated output file as the configuration file for Sim-Panalyzer.

```
> gen_cfg_<target machine>.pl <architectural config filename> <PA cmd filename>
```

```
>sim-analyzer -config <configuration filename> <executing program> <program parameters>
```

3. MOSFET Capacitance Component

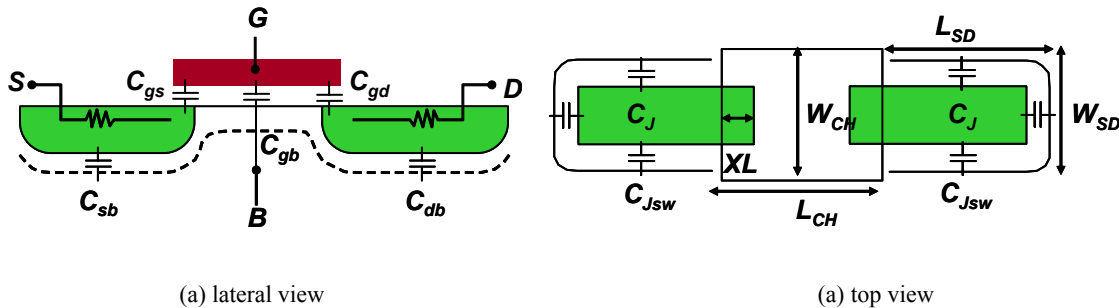
3.1 Model

For accurate *dynamic* power estimation of a circuit, it is important to understand the intrinsic capacitance components of a transistor, because the dynamic power dissipation is estimated based on those capacitance values and the activity ratio of the transition nodes. Figure 2 shows the intrinsic capacitance components in the transistor (or MOSFET). In Figure 2-(a), G, B, S, D nodes represent gate, body, source, and drain. C_J and C_{JSW} in Figure 2-(b) represent the junction bottom area and sidewall capacitance. L_{CH} , W_{CH} , L_{SD} , and W_{SD} represent the channel length and width, and junction length and width of the transistor. In deep sub-micron technology, most of the dynamic power dissipation is due to the charging and discharging of gate and source/drain capacitance during each transition. Therefore, we need an accurate, yet simple model to estimate these capacitance components accurately for our power estimation technique. The gate capacitance can be computed as follows [7]:

$$C_{gate} \cong (C_{poly} \times (L_{CH} \angle 2XL) + C_{ovlp}) \times W_{CH} \tag{1}$$

where C_{poly} , C_{ovlp} , and XL are gate poly capacitance per unit area and gate overlap (with source and drain) capacitance per unit length, and gate overlap length, respectively, see the SPICE parameters specified in [8]. Moreover, L_{CH} is usually fixed to be the minimum channel length of

Figure 2: The intrinsic capacitance components in a transistor.



the technology for digital circuits, thus the only unknown variable in the expression is the channel width W_{CH} . For the computation of source and drain capacitances we use:

$$C_D = (AD) \times C_J + PD \times C_{JSW} \quad (2)$$

where $AD = L_{SD} \times W_{SD}$ is the drain area and $PD = 2 \times (L_{SD} + W_{SD})$ is the drain perimeter. AD and PD can usually be extracted from the physical layout. Alternatively, it is possible to obtain a rough estimate based on the design rule set of the target technology and the design structure: L_{SD} and W_{SD} can be approximated as $3 \times L$ and W for small size devices.

3.2 Implementation

In Sim-Panalyzer, technology specific parameters are specified in “technology.h”. Table 1 summarizes the TSMC 0.18 μm technology parameters used in “technology.h”. All those parameters were obtained from MOSIS parametric test results for TSMC 0.18 μm CMOS runs [8]. If users need to run experiments for different technologies, they can update “technology.h” accordingly. Figure 3 shows part of a MOSIS parametric test for a TSMC 0.18 μm CMOS run. In the

Table 1: Technology parameters.

| Notation | Physical property | Definition in “technology.h” | Sample value |
|------------|---|------------------------------|-------------------------|
| L_{CH} | Minimum channel length for the defined technology | LCH | 0.18 μm |
| XL | Gate (poly) overlap length | XL | 0.02 μm |
| C_{poly} | Gate (poly) capacitance per unit area | CPOLY_NDIFF (NMOS) | 8460aF/ μm^2 |
| | | CPOLY_PDIFF (PMOS) | 8250aF/ μm^2 |
| C_{ovlp} | Gate (poly) overlap (with source and drain) capacitance per unit length | COVLP_NDIFF (NMOS) | 860aF/ μm |
| | | COVLP_PDIFF (PMOS) | 662aF/ μm |
| C_J | Source/drain junction capacitance per unit area | CJ_NDIFF | 970aF/ μm^2 |
| | | CJ_PDIFF | 1171aF/ μm^2 |
| C_{JSW} | Source/drain junction side-wall capacitance per unit length | CJSW_NDIFF | 261aF/ μm |
| | | CJSW_PDIFF | 225aF/ μm |
| L_{SD} | Source/drain minimum length | LSD | 0.54/ μm |

Figure 3: MOSIS parametric test results for a TSMC 0.18µm CMOS run.

| PROCESS PARAMETERS | N+ | P+ | POLY | PLY+BLK | MTL1 | MTL2 | N+BLK | UNITS |
|----------------------|-----|------|------|---------|------|------|-------|----------|
| Sheet Resistance | 6.8 | 7.7 | 8.0 | 326.2 | 0.08 | 0.08 | 61.8 | ohms/sq |
| Contact Resistance | 9.6 | 10.2 | 8.9 | | | 4.87 | | ohms |
| Gate Oxide Thickness | 41 | | | | | | | angstrom |

| PROCESS PARAMETERS | MTL3 | MTL4 | MTL5 | MTL6 | POLY_HRI | N_W | UNITS |
|--------------------|------|-------|-------|-------|----------|-----|---------|
| Sheet Resistance | 0.08 | 0.07 | 0.07 | 0.03 | | 932 | ohms/sq |
| Contact Resistance | 9.74 | 14.38 | 18.98 | 21.32 | | | ohms |

COMMENTS: BLK is silicide block.

| CAPACITANCE PARAMETERS | N+ | P+ | POLY | M1 | M2 | M3 | M4 | M5 | M6 | M5P | N_W | UNITS |
|------------------------|-----|------|------|----|----|----|----|----|----|-----|-----|---------|
| Area (substrate) | 981 | 1142 | 104 | 38 | 19 | 13 | 8 | -- | 3 | | 75 | aF/um^2 |
| Area (N+active) | | | 8460 | 53 | 20 | 14 | 11 | 9 | 8 | | | aF/um^2 |
| Area (P+active) | | | 8258 | | | | | | | | | aF/um^2 |
| Area (poly) | | | | 61 | 17 | 10 | 7 | 5 | 4 | | | aF/um^2 |
| Area (metall1) | | | | | 39 | 15 | 9 | 7 | 5 | | | aF/um^2 |
| Area (metal2) | | | | | | 39 | 14 | 9 | 6 | | | aF/um^2 |
| Area (metal3) | | | | | | | 39 | 15 | 9 | | | aF/um^2 |
| Area (metal4) | | | | | | | | 41 | 14 | | | aF/um^2 |
| Area (metal5) | | | | | | | | | 37 | 987 | | aF/um^2 |
| Area (no well) | 147 | | | | | | | | | | | aF/um^2 |
| Fringe (substrate) | 252 | 210 | | 18 | 59 | 53 | 41 | 24 | -- | | | aF/um |
| Fringe (poly) | | | | 69 | 38 | 28 | 23 | 20 | 17 | | | aF/um |
| Fringe (metall1) | | | | | 59 | 35 | | 22 | 19 | | | aF/um |
| Fringe (metal2) | | | | | | 53 | 35 | 28 | 23 | | | aF/um |
| Fringe (metal3) | | | | | | | 51 | 35 | 28 | | | aF/um |
| Fringe (metal4) | | | | | | | | 56 | 36 | | | aF/um |
| Fringe (metal5) | | | | | | | | | 55 | | | aF/um |
| Overlap (N+active) | | | 860 | | | | | | | | | aF/um |
| Overlap (P+active) | | | 662 | | | | | | | | | aF/um |

T3AZ SPICE BSIM3 VERSION 3.1 PARAMETERS

SPICE 3f5 Level 8, Star-HSPICE Level 49, UTMOST Level 8

* DATE: Dec 17/03

* LOT: T3AZ WAF: 3097

* Temperature_parameters=Default

```
.MODEL CMOSN NMOS (
+VERSION = 3.1          TNOM = 27          TOX = 4.1E-9
+XJ = 1E-7             NCH = 2.3549E17     VTH0 = 0.3665129
+K1 = 0.5924639       K2 = 3.654968E-3      K3 = 1E-3
+K3B = 3.6779438      W0 = 1E-7           NLX = 1.972849E-7
+DVT0W = 0            DVT1W = 0           DVT2W = 0
+DVT0 = 1.1143034     DVT1 = 0.3041866    DVT2 = 0.0441263
+U0 = 261.4948585     UA = -1.479128E-9   UB = 2.465229E-18
+UC = 6.659312E-11    VSAT = 1.033566E5   A0 = 2
+AGS = 0.4404112     B0 = 3.908023E-8    B1 = 5E-6
.
.
.
```

Figure 4: Basic data structure for transistor capacitance estimation in “technology.h”.

```
/* channel type*/
typedef enum {PCH /* PMOS */, NCH /* NMOS */} channel_t;

/* cmos gate transistor sizes */
typedef struct {
    double WPCH; /* PMOS transistor channel width */
    double WNCH; /* NMOS transistor channel width */
} cmos_t;
```

results, process parameters such as sheet and contact resistance, capacitance parameters, and SPICE parameters can be seen. In particular, the capacitance parameters are used to build Table 1. For example, to estimate poly (gate) capacitance over the N+ active area — simply poly used to build NMOS — per unit area, we should lookup the intersecting number from the “POLY” column and the “Area (N+active)” row. This is how “CPOLY_NDIFF” is obtained for 0.18 μ m technology in Table 1; each layer’s capacitance is dependent on the bottom layer connected to ground (GND).

To support transistor-level modeling, two basic data structures are provided in Figure 4. “channel_t” represents transistor channel type — “PCH” for p-type and “NCH” for n-type channel. The n- and p-type channels are used to build NMOS and PMOS, respectively. The structure “cmos_t” contains the transistor width for a pair of complementary PMOS — “WPCH” — and NMOS — “WNCH” transistors.

Table 2: Basic transistor capacitance component estimation functions in “technology.c”.

| Function Name | Argument | | | Return Data-type |
|----------------------------|------------------|----------------|-------------------------|------------------|
| | Data-type | Name | Property | |
| <i>estimate_MOSFET_CG</i> | <i>channel_t</i> | <i>channel</i> | transistor channel type | <i>double</i> |
| | <i>double</i> | <i>WCH</i> | transistor width | |
| <i>estimate_MOSFET_CSD</i> | <i>channel_t</i> | <i>channel</i> | transistor channel type | <i>double</i> |
| | <i>double</i> | <i>WCH</i> | transistor width | |

Table 2 shows the corresponding functions in “technology.c”, calculating both C_{gate} and C_D . Those functions are fundamental routines to build a power model for more complex circuits or functional blocks.

4. Interconnect Capacitance and Resistance

4.1 Models

There are two ways to estimate the interconnect capacitance and resistance. One way is to use the sheet resistance for each interconnect layer and capacitance parameters specified in Figure 3. However, it is complicated to determine which layer should be used for the interconnect estimation; this requires significant knowledge of the circuit layouts and other applicable design features. The other way is to use the Berkeley Predictive Technology Model (BPTM) [9].

The BPTM estimates the interconnect capacitance and resistance for the given interconnect material and dimensions, and dielectric material between the layers. The total interconnect capacitance consists of C_g — area and fringe capacitance to the underlying plane and C_c — coupling capacitance to the adjacent interconnects. Those capacitance components are estimated by:

$$C_g = \varepsilon \left[\frac{w}{h} + 2.2 \left(\frac{s}{s + 0.7h} \right)^{2.2} + 1.2 \left(\frac{2}{s + 1.5h} \right)^{0.76} \cdot \left(\frac{t}{t + 4.5h} \right)^{0.12} \right] \quad (3)$$

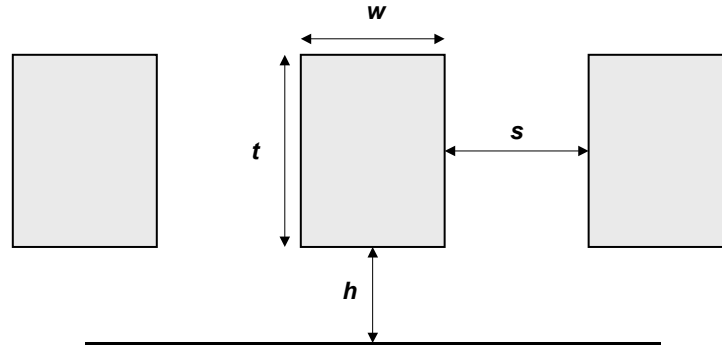
$$C_c = \varepsilon \left[1.4 \cdot \frac{t}{s} \cdot \exp\left(\frac{\angle s}{s + 8.0h}\right) + 2.4 \left(\frac{w}{w + 0.3} \right)^{0.26} \cdot \left(\frac{h}{h + 9.0} \right)^{0.75} \exp\left(\frac{\angle 2s}{s + 6h}\right) \right] \quad (4)$$

where w , s , t , and h represent *width*, *space*, *thickness*, and *height* of the interconnect and those models are accurate in $0.16 < w < 2$, $0.16 < s < 10$, $0.15 < t < 1.2$, $0.16 < h < 2.7$ ranges, see Figure 5 for the dimensions of the interconnect model.

For global interconnect, which is usually the top interconnect layer, the total interconnect capacitance C_t is modeled by:

$$C_t = 2 \times C_c + C_g \quad (5)$$

Figure 5: The dimensions of the interconnect model.



assuming that there are three adjacent interconnects in the left, right, and bottom sides of the interconnect. Hence, C_t becomes a sum of C_g and C_c multiplied by two. However, for local and intermediate interconnects, there is another layer on top of the interconnect. This doubles C_g in the total interconnect capacitance equation. Therefore, the total interconnect capacitance C_t becomes:

$$C_t = 2 \times C_c + 2 \times C_g \quad (6)$$

Caveat: both the coupling and ground interconnect capacitance is sensitive to the space between the adjacent interconnects. Therefore, users should apply the interconnect capacitance model appropriately depending on the spacing and the existence of the adjacent interconnect.

Depending on the interconnect material, the *resistivity* is different. For instance, *Cu* (copper) and *Al* (aluminum) have 2.2 and 3.3 Ω/cm for resistivity, respectively. The interconnect resistance for the given resistivity and dimensions is estimated by:

$$R = \frac{\rho \cdot l}{w \cdot t} \quad (7)$$

where ρ is the resistivity of the interconnect material.

4.2 Implementation

In Sim-Panalyzer, we provide the interconnect capacitance and resistance estimation functions in “technology.c”, see Table 3 for the provided functions. In Table 3, “*estimate_interconnect_CG*” and “*estimate_interconnect_CC*” estimate C_g and C_c capacitance in (3) and (4), respectively for the interconnect capacitance. Depending on the layer and the existence of the adjacent interconnect, users should combine those two equations appropriately. For the interconnect resistance estimation, “*estimate_interconnect_R*” which is based on (7) is provided.

However, to estimate the interconnect capacitance and resistance properly, the users are required to provide appropriate interconnect dimensions for the estimation model. Table 4 shows typical parameters for interconnect capacitance and resistance estimation for 0.18, 0.13, and 0.10 μm technologies. *space* and *width* in Table 4 represents the minimum spacing between the

Table 3: Interconnect capacitance and resistance estimation functions in “technology.c”.

| Function Name | Argument | | | Return |
|---------------------------------|---------------|------------|---------------------|---------------|
| | Data-type | Name | Property | Data-type |
| <i>estimate_interconnect_CG</i> | <i>double</i> | <i>l</i> | length | <i>double</i> |
| | | <i>s</i> | space | |
| | | <i>w</i> | width | |
| | | <i>h</i> | height | |
| <i>estimate_interconnect_CC</i> | | <i>t</i> | thickness | |
| | | <i>k</i> | dielectric constant | |
| <i>estimate_interconnect_R</i> | <i>double</i> | <i>l</i> | length | <i>double</i> |
| | | <i>s</i> | space | |
| | | <i>w</i> | width | |
| | | <i>rou</i> | resistivity | |

Table 4: Interconnect capacitance and resistance estimation parameters.

| 0.18/0.13/0.10 μm | width (μm) | space(μm) | thickness(μm) | height(μm) | dielectric K |
|------------------------------|-------------------------|------------------------|----------------------------|-------------------------|--------------|
| Local | 0.28/0.20/0.15 | 0.28/0.20/0.15 | 0.45/0.45/0.30 | 0.65/0.45/0.30 | 3.5/3.2/2.8 |
| Intermediate | 0.35/0.28/0.20 | 0.35/0.28/0.20 | 0.65/0.45/0.45 | 0.65/0.45/0.30 | |
| Global | 0.80/0.60/0.50 | 0.80/0.60/0.50 | 1.25/1.20/1.20 | 0.65/0.45/0.30 | |

interconnect. These values should be adjusted based on the specific circuit layout dimension. The *intermediate* parameters are used for interconnects in bit-lines and word-lines in the memory structure and the *global* parameters are used for interconnects in system clocks and power distribution networks.

5. General Circuits

5.1 Models

For digital circuits, once node capacitances are estimated, the next step is to gather node-switching information. We compute each switch *on the fly* during microarchitectural simulation, because total dynamic power dissipation is heavily dependent on the number of switches at the internal nodes [10][11] for some circuit blocks. To compute the number of switches in each node, it is necessary to perform logic simulation. Traditionally, event-driven logic simulation is much slower than compiled-code levelized logic simulation. Event-driven logic simulation is indispensable for accurate timing-level simulation. However, levelized logic simulation is enough to compute approximated number of switches in each node.

To support logic simulation in a microarchitectural simulator and to enhance the simulation speed, we provide a set of generic data structures and functions that enable users to combine these basic blocks to build a more complex functional block. In this modeling technique, the users should connect each individual transistor and give transistor sizes in the modeled block. This tedious procedure is inevitable because the power dissipation of functional blocks can be different by more than 100% depending on the circuit style and transistor sizes. During the initialization process, the specialized logic simulator for a specific functional block estimates the switching capacitance for every internal node. This logic simulator is embedded in the microarchitectural simulator to capture necessary inputs and access activities, and generate the accumulated power dissipation statistics, accordingly.

First, we explain the generic data structures to model a circuit node and logic gate, see Figure 6. “node_t” contains a logic value, node switching capacitance, and energy dissipation of the node. This is one of the basic building blocks to modeling a generic gate. “lgate1_t” and

Figure 6: A generic node data structure

```
/* netlist node type */
typedef struct {
    bit_t lvalue; /* logic value */
    double capacitance; /* node capacitance */
    double energy; /* transition energy */
} node_t;

/* 1-input generic logic gate type */
typedef struct _lgate1_t lgate1_t;
struct _lgate1_t {
    node_t *Y; /* current output */
    node_t *A; /* connected input node ptrs */
    double energy; /* energy dissipation of the gate */
    double (*lgate_op)(lgate1_t *lgate, double voltage);
    /* logic and energy evaluation fn of the gate */
};

/* 2-input generic logic gate type */
typedef struct _lgate2_t lgate2_t;
struct _lgate2_t {
    node_t *Y; /* current output */
    node_t *A, *B; /* connected input node ptrs */
    double energy; /* energy dissipation of the gate */
    double (*lgate_op)(lgate2_t *lgate, double voltage);
    /* logic and energy evaluation fn of the gate */
};

.
.
.
```

“lgate2_t” represent generic data structures for 1- and 2-input gates. This generic logic gate type can be easily extended to model 3- or more input gates by adding input nodes in the data structure.

In Figure 7, we show a modeling example of the two-input CMOS NAND gate — the most basic logic component along with the inverter — consisting of four transistors using our proposed methodology. First, we declare input/output nodes — A, B, and Y and a generic gate — NAND2. Second, we create a 2-input logic gate using “create_lgate2” connecting the necessary inputs, assigning transistor widths, relating logic and assigning a energy evaluation function — NAND2_op to lgate_op in the gate data structure. By calling “lgate_op”, the logic value and the energy is evaluated.

```

/* declare a generic gate for NAND */
lgate2_t NAND2;
/* declare gate node */
node_t A, B, Y;

/* declare variables storing the transistor widths of the
gate*/
cmost_t AW, BW;

/* assign transistor sizes */
AW.WPCH = WP1; AW.WNCH = WN1; BW.WPCH = WP2; BW.WNCH = WN2;

/* create NAND2 gate create */
NAND2 = create_lgate2(&A, &B, &AW, &BW, Static, NAND2_op);

/* estimate node switching capacitance */
NAND2->Y->capacitance
    = estimate_capacitance_CSD(PCH, ...) + ...

/* evaluate logic and energy dissipation */
A.lvalue = 1; B.lvalue = 0; /* assign inputs */
energy = lgate_op->(NAND2, 1.8); /* evaluate logic and energy dissipation*/

```

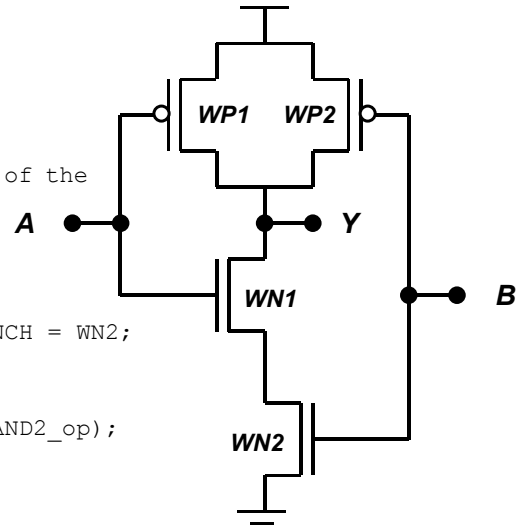


Figure 7: Power modeling of a 2-input NAND gate

At the netlist level, multiple gates are created and connected together to simulate the entire logic block. We levelize each gate or netlist primitive and simulate each gate one after the other, in a sequence compatible with the partial ordering imposed by levelization. This approach corresponds to the levelized cycle-based simulation technique in logic simulation [12]. As a small example, the following illustrates how to create a netlist for a combinational circuit and simulate the internal node activity. The example shows a combinational circuit consisting of 2-input NOR and 2-input NAND gates. We are able to evaluate the correct output logic value by evaluating the gates in order of increasing distance from the primary inputs. The levelized approach we use here most often performs better than an event-driven simulation since we trade having to maintain an event queue at the expense of simulating every gate in the netlist for each time interval [12][13]. A downside of the levelized approach is that we lose information on arrival times of signals, thus we cannot evaluate power dissipation due to glitches and temporary transitions. However, a well-

designed combinational circuit should not generate many glitches, in which case our model is fairly accurate.

This proposed modeling methodology can be extended to model a more complex datapath or memory circuit. In both cases, since they have regular structures, they can be modeled easily in a iterative manner after modeling one component.

5.2 Implementation

We provide a sub-set of data structures and generic functions for 1-, 2-, and 3-input gate types in “./pmodel/logic.h” and “./pmodel/logic.c”. The detailed usage is described in the source code. With the implemented data structures and generic functions, the users can easily extend those according to the instructions given in “logic.h” and “logic.c”. Table 5 lists the implemented generic gate modeling functions.

Table 5: Generic gate modeling functions in “logic.c”.

| Function Name | Argument | | | Return |
|---|--|---------------------|--|-------------------|
| | Data-type | Name | Property | Data-type |
| <i>create_lgate</i> “ <i>n</i> ” (<i>n</i> =1, 2, ...) | <i>node_t</i> * | <i>A, B, C, ...</i> | input — A, B, C, ... | <i>lgate1_t</i> * |
| | <i>cmos_t</i> * | <i>a, b, c, ...</i> | transistor widths connected to A, B, C, ... | |
| | <i>lgstyle_t</i> | <i>lgstyle</i> | logic style | |
| | <i>double</i> (*) | <i>lgate_op</i> | function pointer evaluating logic and energy dissipation of the gate | |
| <i>lgate_op</i> | <i>lgate</i> “ <i>n</i> ”_ <i>t</i> (<i>n</i> =1, 2, ...) | <i>lgate</i> | gate to be evaluated | <i>double</i> |
| | <i>double</i> | <i>voltage</i> | supply voltage of the gate | |

6. Memory Power Model

In modern microprocessors, *static random access memory* (SRAM) is extensively used for caches, TLBs, BTBs, branch predictors, register files, instruction queues, etc. For instance, 40% of the total power in the Alpha 21264 and 60% of the total power of the StrongARM processor is devoted to cache and memory structures [14][15]. As feature sizes shrink and supply voltages decrease along with the word-line pulse technique [16], bit-line voltage swings during read operations decrease to 100mV. This has dramatically reduced the power consumption from the bit-line.

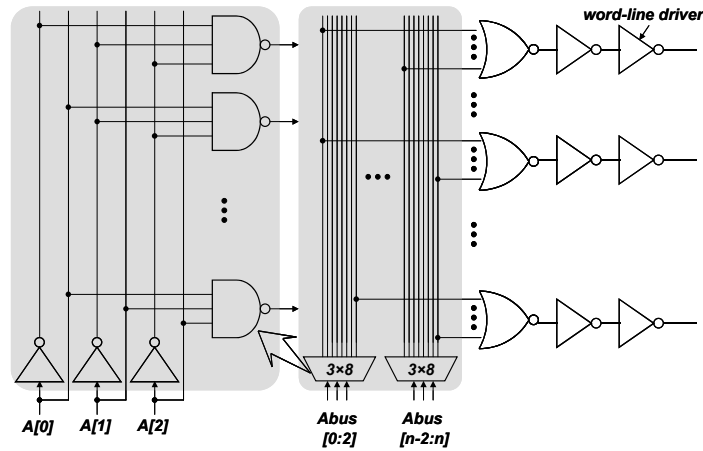
In a first order approximation, users can obtain an effective switching capacitance for running Sim-Panalyzer from the power model — energy dissipation per access — provided by the modified CACTI in our tool set. However, the following modeling technique can be applied to Sim-Panalyzer to estimate more accurate memory power dissipation.

6.1 Models

While the bit-line power dissipation is independent from the switching activity of the data due to the complementary structure of bit-lines, the power dissipation of the decoder is heavily dependent on the switching events of the decoder address inputs. Hence, we need to build a switching event-sensitive power model for the decoder. We present an example on how to use the technique just presented in Section 5 to model a 7×128 decoder designed with the TSMC $0.18\mu\text{m}$ technology Artisan standard cell library and Synopsys® Design Compiler®. Figure 8 shows the 7×128 decoder logic. The decoder logic has a regular structure consisting of a set of NANDs, NORs, and INVs. To measure bit-line energy consumption, we used the following equation:

$$E = C_{bit-line} \times V_{DD} \times \Delta V_{swing}, \quad (8)$$

Figure 8: 7×128 decoder logic.



where $C_{bit-line}$ is the bit-line capacitance per memory column and ΔV_{swing} is the bit-line voltage swing. $C_{bit-line}$ includes the bit-line interconnect, the access transistor drain capacitance, and the pre-charge circuit drain capacitance. The bit-line interconnect capacitance was estimated based on the actual SRAM dimensions and using available MOSIS parametric test results from the TSMC 0.18 μm technology fabrication run [8]. The access transistor drain capacitance connected to the bit-line was estimated using (2).

6.2 Implementation

Figure 9 and Figure 10 show the corresponding descriptions of functions for creating the 3-to-8 decoder module “create_module_dec3x8”, and evaluating logic and energy “dec3x8_op”. The cycle-based logic simulator for the decoder was derived by instantiating and connecting those gates in an iterative way in Figure 9. The switching capacitance of each node was automatically estimated depending on the circuit topology. Then, the generated logic simulator annotated with the extracted capacitance is embedded in the microarchitectural simulator with

Figure 9: An example of modeling an 3×8 decoder.

```
/** dec3x8 consists of a set of inverters and nand gates. These routines describe modeling examples for the decoder */
module_dec3x8_t * /* return a 3x8 decoder module instance pointer */
create_module_dec3x8(
    node_t *A[], /* input node pointers */
    node_t *Y[], /* output node pointers */
    ...
    double (*module_op)(module_dec3x8_t *module, double voltage) /* module logic and energy evaluation fn */)
{
    module_dec3x8_t *module; /* top module data structure */
    ...
    /* create module*/
    module = (module_dec3x8_t *)malloc(sizeof(module_dec3x8_t));

    /* level-0 : create netlists for inverted address */
    for(i = 0; i < 3; i++) {
        INVA[i] = create_lgate1(A[i], &x1[0], Static, INV_op);
        A_[i] = INVA[i]->Y; }
    /* estimate drain capacitance of the connected INV gates */
    for(i = 0; i < 8; i++)
        INVA[i]->Y->capacitance += (estimate_MOSFET_CSD(PCH, ...) + ...);

    /* level-1 : create netlists for NAND gates */
    NAND3[0] = create_lgate3(A_[0], A_[1], A_[2], ..., NAND3_op);
    NAND3[1] = create_lgate3(A[0] , A_[1], A_[2], ..., NAND3_op);
    NAND3[2] = create_lgate3(A_[0], A[1] , A_[2], ..., NAND3_op);
    NAND3[3] = create_lgate3(A[0] , A[1] , A_[2], ..., NAND3_op);
    NAND3[4] = create_lgate3(A_[0], A_[1], A[2] , ..., NAND3_op);
    NAND3[5] = create_lgate3(A[0] , A_[1], A[2] , ..., NAND3_op);
    NAND3[6] = create_lgate3(A_[0], A[1] , A[2] , ..., NAND3_op);
    NAND3[7] = create_lgate3(A[0] , A[1] , A[2] , ..., NAND3_op);

    /* estimate drain capacitance of the connected NAND gates */
    for(i = 0; i < 8; i++)
        NAND3[i]->Y->capacitance += (3. * (estimate_MOSFET_CSD(PCH, ...) + ...));

    /* level-2 : connect outputs of the netlist */
    for(i = 0; i < 8; i++)
        Y[i] = NAND3[i]->Y;

    ...
    return module;
}
```

Figure 10: An example of evaluating an 3×8 decoder.

```
/* dec3x8 logic and energy dissipation evaluation function */
double /* return energy */
dec3x8_op(
    module_dec3x8_t *module /* module to be evaluated */,
    double voltage /* supply voltage */)
{
    /* temporary pointers */
    lgate3_t **NAND3;
    lgate1_t **INVA;
    node_t **Y;

    double energy;
    int i;

    /* retrieve node and gate instance pointers */
    Y = module->Y;
    NAND3 = module->NAND3;
    INVA = module->INVA;

    /* logic and energy dissipation evaluation */
    energy = 0.;
    /* level-0: inverter gate logic and energy dissipation evaluation */
    for(i = 0; i < 3; i++)
        energy += INVA[i]->lgate_op(INVA[i], voltage);

    /* level-1: NAND gate logic and energy dissipation evaluation */
    for(i = 0; i < 8; i++)
        energy += NAND3[i]->lgate_op(NAND3[i], voltage);

    /* return energy */
    return energy;
}
```

an interface routine passing the current address bus value to the logic simulator and returning the estimated energy consumption to the microarchitectural simulator.

In Figure 9, “`create_module_dec3x8`” creates a module instance for 3-to-8 decoder. This module consists of instantiating and connecting the basic gates to implement the 3-to-8 decoder function. First, a memory space for the module data structure is allocated with “`module = (module_dec3x8 *)malloc(sizeof(module_dec3x8))`”. Second, the decoder components are created and instantiated in the levelized order — level 0 to 2. In level 0, inverter gates are created and instantiated to generate inverted address bus signals. In level 1, NAND gates are created and instantiated to form the 3-to-8 decoder function. In level 2, the outputs of the NAND gates are connected to the output node “`Y`”.

After the creation of gate instances, we estimate the source/drain capacitance of each gate; the estimation of the input gate capacitance is automatically done by the logic gate creation functions. The reason the source/drain capacitance should be estimated separately is that the output drain capacitance estimations are different for each gate type depending on the circuit topology while the gate input capacitance is independent from the gate type.

In Figure 10, to evaluate the 3-to-8 decoder, the module function “`dec3x8_op`” is derived by evaluating each gate instance in the levelized order. First, the inverter gates are evaluated with the applied address bus. Second, the NAND gates are evaluated with the updated node logic values from level 1. The implemented source codes ‘`./pmodel/dec3x8.h`’ and ‘`./pmodel/dec3x8.c`’ explain in detail on how it works. See Table 6 for information on the implemented function prototypes.

Figure 11 and Figure 12 show the corresponding descriptions of functions for creating the 7-to-128 decoder module “`create_module_dec7x128`”, and evaluating logic and energy

Table 6: 3-to-8 decoder modeling functions in “dec3x8.c”.

| Function Name | Argument | | | Return |
|-----------------------------|------------------------|------------------|--|-------------------------|
| | Data-type | Name | Property | Data-type |
| <i>create_module_dec3x8</i> | <i>node_t</i> ** | <i>A, Y</i> | input — A, and output — Y | <i>module_dec3x_t</i> * |
| | <i>cmos_t</i> ** | <i>x3WCH</i> | transistor widths for 3-input NAND gates | |
| | <i>cmos_t</i> ** | <i>x1WCH</i> | transistor widths for 3-input inverter gates | |
| | <i>double</i> (*) | <i>module_op</i> | function pointer evaluating logic and energy dissipation of the module | |
| <i>dec3x8_op</i> | <i>module_dec3x8_t</i> | <i>module_op</i> | module to be evaluated | <i>double</i> |
| | <i>double</i> | <i>voltage</i> | supply voltage of the module | |

“dec7x128_op”. “create_module_dec7x128” instantiates 3 “dec3x8” modules, NOR gates, and inverters to implement the “dec7x128” module. In addition, “dec7x128_op” reuses “dec3x8_op” to evaluate the logic and energy dissipation. The implemented source codes ‘./pmodel/dec7x128.h’ and ‘./pmodel/dec7x128.c’ explain in detail on how it works. See Table 6 for information on the implemented function prototypes.

The hierarchical structure along with the re-use property allows users enormous flexibility and reduces the tremendous modeling efforts. We can create a more complex function module by instantiating simple modules and connecting the nodes. The above examples are for a specific circuit type of memory decoder, but if the users want to evaluate different circuit styles with the same decoder function, they only have to recombine the gates and reconnect the internal nodes.

6.3 Calibration

Figure 13-(a) shows the calibrated energy consumption of a 4KB SRAM power model against HSPICE measurement. In the figure, each point represents the energy consumption for each applied vector. For the HSPICE experiment, we modeled and simulated the whole 7×128

Figure 11: An example of modeling an 7×128 decoder.

```
/** dec7x128 consists of a set of inverters and nand gates. These routines describe
modeling examples for the decoder */
module_dec7x128_t * /* return a 3x8 decoder module instance pointer */
create_module_dec7x128(
    node_t *A[], /* input node pointers */
    node_t *Y[], /* output node pointers */ ...,s
    double (*module_op)(module_dec7x128_t *module, double voltage) /* module logic and
energy evaluation fn */)
{
    module_dec7x128_t *module; /* top module data structure (1) */
    ...
    /* allocate space for the module instance (2) */
    module = (module_dec7x128_t *)malloc(sizeof(module_dec7x128_t));

    /* level-0 : create netlists for 3x8 decoders */
    /* create module instances and connect the nodes */
    dec3x8[0] = create_module_dec3x8(A, dec3x8Y0, &x3WCH[0], &x1WCH[0], dec3x8_op);
    dec3x8[1] = create_module_dec3x8(A+3, dec3x8Y1, &x3WCH[0], &x1WCH[0], dec3x8_op);
    dec3x8[2] = create_module_dec3x8(A+6, dec3x8Y2, &x3WCH[0], &x1WCH[0], dec3x8_op);

    /* level-1 : create netlists for nor gates */
    /* allocate space for the NOR instances */
    for(i = 0; i < 8; i++) {
        for(j = 0; j < 8; j++) {
            /* create gate instance and estimate the output node drain capacitance */
            NOR3[8*i+j] = create_lgate3(dec3x8Y0[j], dec3x8Y1[i], dec3x8Y2[0], ...);
            NOR3[8*i+j]->Y->capacitance += (1.*(estimate_MOSFET_CSD(PCH,...) + ...));
        }
    }
    for(i = 0; i < 8; i++) {
        for(j = 0; j < 8; j++) {
            /* create gate instance and estimate the output node drain capacitance */
            NOR3[64+8*i+j] = create_lgate3(dec3x8Y0[j], dec3x8Y1[i], dec3x8Y2[1], ...);
            NOR3[64+8*i+j]->Y->capacitance += (1.*(estimate_MOSFET_CSD(PCH,...) + ...));
        }
    }
    /* level-2 : create netlists for inverters after nor gates */
    INVNOR = (lgate1_t **)calloc(128, sizeof(lgate1_t *));
    for(i = 0; i < 128; i++) {
        INVNOR[i] = create_lgate1(NOR3[i]->Y, &x1WCH[2], Static, INV_op);
        INVNOR[i]->Y->capacitance += (1.*(estimate_MOSFET_CSD(PCH, ...) + ...));
    }

    /* level-3 : create netlists for inverters after INVNOR gates */
    for(i = 0; i < 128; i++) {
        /* create gate instance and estimate the output node drain capacitance */
        INVWL[i] = create_lgate1(INVNOR[i]->Y, &x1WCH[3], Static, INV_op);
        INVWL[i]->Y->capacitance += (1.*(estimate_MOSFET_CSD(PCH, ...) + ...));
    }

    /* level-4 : connect outputs of the netlist */
    for(i = 0; i < 128; i++)
        Y[i] = INVWL[i]->Y;
    ...
    return module;
}
```

Figure 12: An example of evaluating an 7×128 decoder.

```
/* dec7x127 logic and energy dissipation evaluation function */
double
dec7x128_op(
    module_dec7x128_t *module,
    double voltage)
{
    module_dec3x8_t **dec3x8;
    lgate3_t **NOR3;
    lgate1_t **INVNOR;
    lgate1_t **INVWL;
    node_t **Y;

    double energy;
    int i, j;

    /* retrieve node and gate instance pointers */
    Y = module->Y;
    dec3x8 = module->dec3x8;
    NOR3 = module->NOR3;
    INVNOR = module->INVNOR;
    INVWL = module->INVWL;

    /* logic and energy dissipation evaluation */
    energy = 0.;
    /* level-0: 3x8 decoder logic and energy dissipation evaluation */
    for(i = 0; i < 3; i++)
        energy += dec3x8[i]->module_op(dec3x8[i], voltage);

    /* level-1: NOR logic and energy dissipation evaluation */
    for(i = 0; i < 128; i++)
        energy += NOR3[i]->lgate_op(NOR3[i], voltage);

    /* level-2: INVNOR logic and energy dissipation evaluation */
    for(i = 0; i < 128; i++)
        energy += INVNOR[i]->lgate_op(INVNOR[i], voltage);

    /* level-3: INVWL logic and energy dissipation evaluation */
    for(i = 0; i < 128; i++)
        energy += INVWL[i]->lgate_op(INVWL[i], voltage);

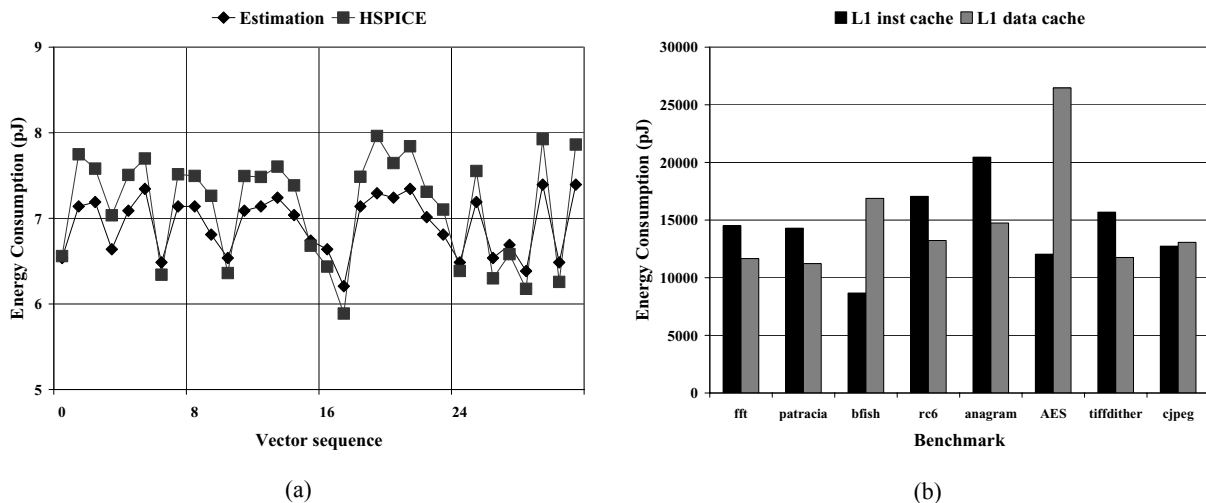
    /* return energy */
    return energy;
}
```

Table 7: 3-to-8 decoder modeling functions in “dec3x8.c”.

| Function Name | Argument | | | Return |
|-----------------------------|------------------------|------------------|--|-------------------------|
| | Data-type | Name | Property | Data-type |
| <i>create_module_dec3x8</i> | <i>node_t</i> ** | <i>A, Y</i> | input — <i>A</i> , and output — <i>Y</i> | <i>module_dec3x_t</i> * |
| | <i>cmos_t</i> ** | <i>x3WCH</i> | transistor widths for 3-input NAND gates | |
| | <i>cmos_t</i> ** | <i>x1WCH</i> | transistor widths for 3-input inverter gates | |
| | <i>double</i> (*) | <i>module_op</i> | function pointer evaluating logic and energy dissipation of the module | |
| <i>dec3x8_op</i> | <i>module_dec3x8_t</i> | <i>module_op</i> | module to be evaluated | <i>double</i> |
| | <i>double</i> | <i>voltage</i> | supply voltage of the module | |

decoder and a dummy 128×256 bit memory array; we modeled just one column of 128 cells multiplied by 256 to speed up the simulation. As seen in Figure 13-(a), the estimated energy consumption follows the actual measurement result closely for each applied vector. The proposed technique has an average 7% estimation error for 1K vectors compared to the HSPICE measurement. However, when comparing the execution time, the proposed technique completed within a

Figure 13: Calibration of SRAM energy consumption model in (a) and L1 instruction and data cache energy consumption in (b).



few seconds while the HSPICE took 3.4 hours on UltraSparc80® 450MHz dual processors with a 4MB L2 cache.

Figure 13-(b) shows the total accumulated energy consumption of the 4KB L1 instruction and data caches obtained by running 10 million instructions for a subset of embedded benchmark programs from the MiBench Benchmark Suite [17]. The proposed power models were embedded in Sim-Panalyzer with the StrongARM configuration for this experiment. In the process of estimating energy the actual address stream was applied to the SRAM power model on the fly.

The estimated energy consumption results show that total energy consumption can be significantly different depending on the benchmark programs even if the same number of instructions are executed. Usually, the instruction cache consumes more energy than the data cache. However, the *average energy dissipation per access* of a data cache is usually higher than that of an instruction cache. The primary reason for this energy consumption behavior is that the activity ratio of an instruction cache is higher than the data cache, while the address stream supplied to the data cache is more non-sequential than the instruction cache, which means more switching events in the address bus. These characteristics imply that both input switches and the access activities for the application-specific functional block must be considered for accurate power estimation of embedded microprocessors. In terms of overhead for the microarchitectural simulator, the proposed technique increases the execution time by 3% for both the instruction and data cache.

7. Datapath and Execution Unit

7.1 Implementation

We now present an example on how to use the proposed technique to implement a datapath component and generate power estimations that interfaces at run-time with the micro-architectural simulator. For this example, we consider a 32-bit carry-select adder consisting of eight 8-bit ripple-carry adders as reported in Figure 14. For each 8-bit add, two 8-bit ripple-carry adders are used to compute the results in parallel for zero and one carry-ins, respectively. The first step is to construct the basic block for a full adder by instantiating the necessary logic gates: the construction of the class *FullAdder* creates all the internal gates and it properly connects them, so that two output nodes, *S* and *CO*, produce the correct functionality. By this point, the main program can create and connect the full adder blocks employing a loop shown in Figure 14. Note how the program structure lends itself naturally to the parameterization of the bus width. By instantiating eight 8-bit ripple-carry adders, we are able to build a 32-bit carry adder. By this point, the model includes a complete description of the logic block under observation. The last two steps provide an interface to the microarchitectural simulator by retrieving *on-the-fly* at each cycle the input vectors corresponding to the two operands of the add operation, and proceeding with the power/

Figure 14: An example of modeling an 8-bit RCA.

```
/* step 1: create netlist */
for(i = 1; i < WIDTH; i++) {
    /* create and connect FullAdder instances */
    FA[i] = FullAdder(A[i], B[i], FA[i-1].CI, CO[i], SO[i]...);}

/* step 2: load input vectors */
A.apply(LOp); B.Apply(ROp);

/* step 3: logic and energy evaluation*/
for(i = 0; i < WIDTH; i++) {
    energy += FA[i].GateOp(voltage);}
```

Table 8: 3-to-8 decoder modeling functions in “dec3x8.c”.

| Function Name | Argument | | | Return |
|-----------------------------|------------------------|------------------|--|-------------------------|
| | Data-type | Name | Property | Data-type |
| <i>create_module_dec3x8</i> | <i>node_t</i> ** | <i>A, Y</i> | input — A, and output — Y | <i>module_dec3x_t</i> * |
| | <i>cmos_t</i> ** | <i>x3WCH</i> | transistor widths for 3-input NAND gates | |
| | <i>cmos_t</i> ** | <i>x1WCH</i> | transistor widths for 3-input inveter gates | |
| | <i>double</i> (*) | <i>module_op</i> | function pointer evaluating logic and energy dissipation of the module | |
| <i>dec3x8_op</i> | <i>module_dec3x8_t</i> | <i>module_op</i> | module to be evaluated | <i>double</i> |
| | <i>double</i> | <i>voltage</i> | supply voltage of the module | |

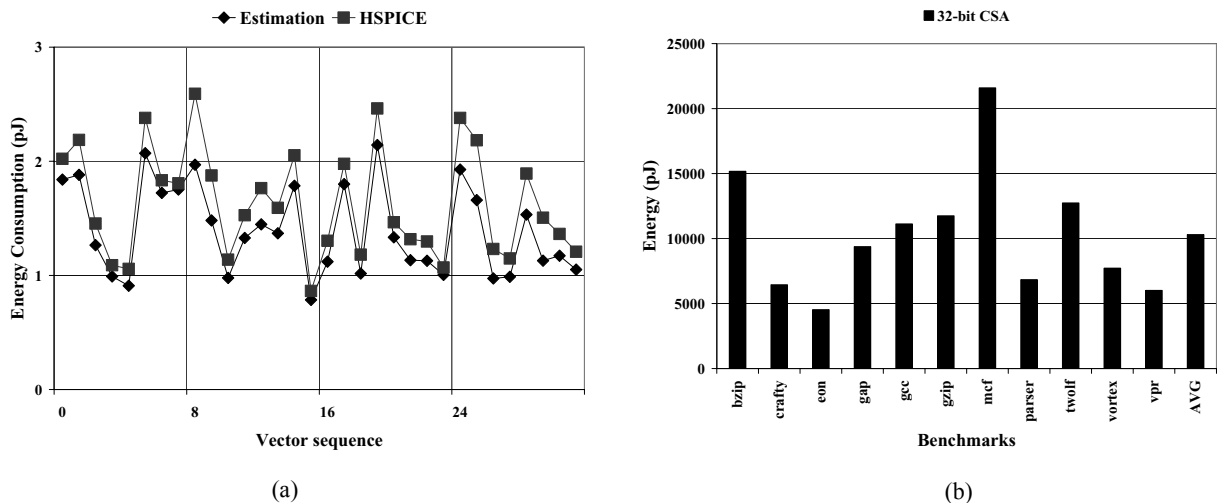
logic simulation. The implemented source codes ‘./pmodel/rca.h’ and ‘./pmodel/rca.c’ explain in detail on how it works. Those power models are parameterizable; by changing “WIDTH” in the function, the bit-width of the implemented adder can be easily changed.

7.2 Calibration

We calibrated our model by comparing the results with the corresponding HSPICE circuit simulation. Figure 15-(a) shows a calibration using the carry-select adder in the previous case study. Each point in the graph represents dissipated energy estimated or measured by applying each vector to the circuit. The diagram indicated that the technique proposed tracks the actual power dissipation of adders very well; we found that the average estimation error by applying 1K vectors is around 9%. The steady under-approximation error of the power estimator can be explained by two sources of power dissipation that our model does not take into account: glitches occurring because of the relative delays among signal propagation times and temporary short circuits due to both PMOS and NMOS transistors being turned on during the transition.

To produce the graph in Figure 15-(b), we simulated the SPEC2000 INT benchmark programs [18] while running the power simulator on our 32-bit adder component. For each benchmark program, we applied 32K vectors to the power model. The results show the total energy that was dissipated in the adder. Note how the total energy dissipation profiles present high variations over different benchmark programs. For instance, *mcf* consumes 480% more energy than *eon* which seems to indicate that the amount of data activity plays an important role in the accurate estimation of the power dissipation of a datapath component. Because of its accuracy and flexibility, this technique could easily be applied in trade-off studies of various solutions for datapath circuits, or for optimization of power dissipation in embedded processors where the datapath constitutes a significant portion of the total power dissipation.

Figure 15: Calibration of 32-bit CSA energy consumption model in (a) and total energy consumption in (b)



8. Clock Distribution Tree

Ever increasing clock frequencies and die area of microprocessor designs require more aggressive clock distribution networks (or trees). As a result, the fraction of total clock power dissipation has become more significant, depending on the target clock frequency and the maximum allowable clock skew. In the case of a small embedded processor design, such as the StrongARM, the clock distribution network consumes only 10% of the total power [15]. However, for the Alpha 21264 microprocessor, the clock consumes up to 32% (23W) of the total average chip power (72W) [14], and the percentage of total clock power is expected to keep increasing for high-end microprocessors that employ aggressive clock frequencies and pipeline-depths [25]. Hence, accurate estimation of clock power is an important key for accurate total microprocessor power estimation, and the fraction of clock power is substantial whether we consider embedded or high-end microprocessors.

8.1 Models

In a tree-style clock distribution system, the power consumption of a clock distribution tree consists of three components:

- Clock distribution tree interconnects.
- Clock buffer gates and parasitics.
- Clocked nodes.

In the Alpha 21264, the power dissipation of the clock distribution tree interconnect and buffers is 65% of the total clock distribution system power. Assuming that an H-tree style clock distribution system is employed, the total interconnect capacitance of a clock distribution tree becomes:

$$C_{H-tree} = c_{int} \times \sqrt{A_{die}} \times 2^{N_{tree} < 1} \times \sum_{i=1}^{N_{tree}} \frac{1}{2^{\lfloor i/2 \rfloor + 1}} \quad (9)$$

where c_{int} , A_{die} , and N_{tree} represent the interconnect capacitance per unit length, chip die area, and the number of levels of depth of the tree, respectively [7]. Also, N_{tree} , the depth of the tree, is given by:

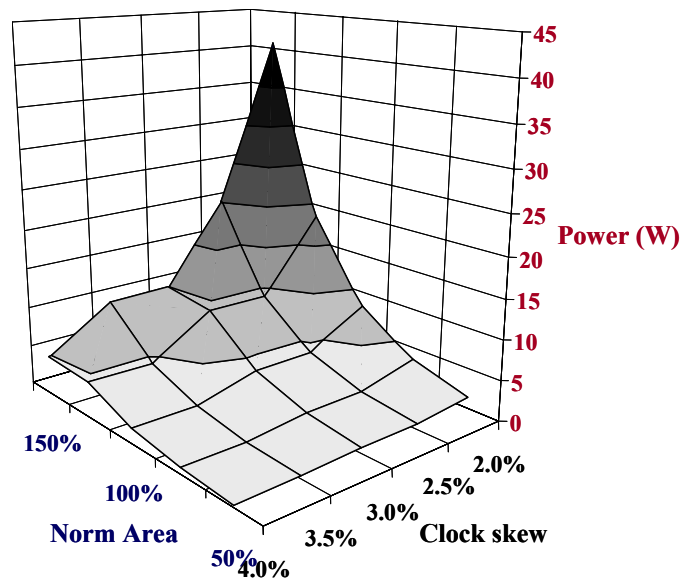
$$N_{tree} = \sqrt{A_{die} \times \frac{r_{int} \times c_{int}}{c_{skew}} + 1}, \quad (10)$$

where r_{int} and c_{skew} represent the interconnect resistance per unit length and maximum allowable clock skew.

As seen from (9) and (10), there are several variables that impact the capacitance of a clock distribution interconnect; the estimation of clock distribution interconnect capacitance is more complicated in the case of other clock distribution styles such as balanced H-tree or tree driven grids. Hence, it is extremely difficult to estimate all needed parameters accurately at the microarchitectural level. We estimate c_{int} and r_{int} using the interconnect capacitance and resistance estimation model we provide in (3), (4), and (7) (see Table 3 for the functions provided for the interconnect parameter estimations in Sim-Panalyzer).

In Wattch, for example, the chip die area and the depth of the clock distribution tree are fixed. However, both parameters can change significantly as the microarchitectural and circuit parameters are changed. For instance, the addition of on-chip L2 caches cause major increase in die area. In addition, the physical implementation phase, such as placement and route, can affect the global chip area. Even worse, estimating parameters such as maximum allowable clock skew requires an in-depth understanding for both circuit design and semiconductor process knowledge incorporated with the target chip specification.

Figure 16: Power consumption of clock distribution tree for maximum allowable clock skew and microprocessor die area.



H-tree is assumed for the clock distribution tree topology. The 100% normalized die area corresponds to that of Alpha 21264 implemented with 0.35 μ m technology. The clock skew is relative fraction of 600MHz clock frequency.

Figure 16 shows the sensitivity of a clock distribution tree power to the maximum allowable clock skew and chip die area of a microprocessor. In this experiment, an “H-tree” is assumed for the clock distribution tree topology. The 100% die area in Figure 16 corresponds to that of the Alpha 21264 implemented with 0.35 μ m technology. The clock skew in Figure 16 is the relative fraction of the 600MHz clock frequency of the Alpha 21264 microprocessor. According to the estimation used in (9) and (10), the estimated global clock distribution tree power with 2.9% maximum allowable clock skew is 4W, which agrees with the published value in [26]. However, as seen in Figure 16, the clock distribution tree power has exponential dependency on the microprocessor die area and maximum allowable clock skew. For instance, the estimated clock distribution tree power dissipation increases by 200% when the target clock skew is changed from 2.5% to 2% at the 100% die area point. If the microprocessor die area increases from 100% to 125%, a 2.5%

clock skew point also results in 200% increase of the clock distribution tree power. Hence, a slight misprediction of either clock skew or chip area incurs a significant error in power estimation.

The power consumption in Figure 16 is estimated only with the clock distribution tree wire capacitance. The clock distribution buffers also dissipate a substantial amount of power, which can be estimated by:

$$C_{sw, clk} = (C_{H-tree} + C_{clk load}) \times \left(\frac{1}{1 \angle \frac{1}{a_{clk buffer}}} + 1 \right) \quad (11)$$

where $a_{clk buffer}$ represents the tapering factor or the optimal stage ratio for the clock buffer [7] and $a_{clk load}$ for Alpha 21264 is around 2.7nF. (9), (10), and (11) give a total power of 26W for the clock distribution including switching of clocked nodes, which is similar to 23W reported in [26]. Out of the 26W in total clock distribution power, 14.7W is consumed by the clock buffers, which is quite significant. However, to estimate the clock buffer power accurately, the exact amount of clock node capacitance must be known, which requires detailed information on the number and sizes of flip-flops in the microprocessor. Depending on the individual device sizes and numbers, the power consumption of a clock distribution system can be quite different.

In summary, it is extremely difficult to estimate the clock power accurately due to too many uncertainties at the microarchitectural level, In particular, one must have accurate die area and clock node capacitance estimates of the target microprocessor which are strongly dependent on changes in the microarchitectural and circuit implementation. However, accurate power estimation of the clock distribution system is very important since it comprises a large fraction of power dissipation and it can vary in a wide range with small changes to parameters. Therefore, a proper

clock skew, die area, and clocked node capacitance must be specified for accurate estimation of total clock power dissipation.

8.2 Implementation

In Sim-Panalyzer, we provide a data structure to specify the clock distribution tree styles; we support two clock tree styles — H tree and balanced H tree, see Figure 17 for the clock tree specifying data structure. This data structure is used to specify the clock distribution tree style for the clock distribution tree switching capacitance estimation function — “estimate_switching_CT” — based on (9), (10), and (11). The “estimate_switching_CT” returning total switching capacitance of the clock tree is provided with other related sub-routines in “clock.c”. The users should specify the clock tree style, target clock skew, clocked die area, clocked node capacitance, and the number of optimal buffer stages for the clock distribution buffer.

Finally, we provide a command-line executable “clock-panalyzer” with the following options:

Figure 17: Basic data structure for clock distribution tree style in “technology.h”.

```
/* clock tree style type*/
typedef enum {Htree /* H-tree */, balHtree /* balanced H-tree */} clocktree_style_t;
```

Table 9: Clock tree switching capacitance estimation functions in “clock.c”.

| Function Name | Argument | | | Return |
|------------------------------|--------------------------|---------------------|--|---------------|
| | Data-type | Name | Property | Data-type |
| <i>estimate_switching_CT</i> | <i>clocktree_style_t</i> | <i>style</i> | clock tree style — H-tree or balH-tree | <i>double</i> |
| | <i>double</i> | <i>cskew</i> | clock skew | |
| | <i>double</i> | <i>area</i> | clocked die area | |
| | <i>double</i> | <i>switching_CN</i> | clocked node capacitance | |
| | <i>int</i> | <i>nbstage_opt</i> | number of optimal buffer stage | |

- *-t clock tree style (e.g., -t Htree or -t balHtree)*
- *-s clock skew in pico-second (ps) (e.g., -s 20)*
- *-a die area in mm² (e.g., -a 100)*
- *-n number of clock buffer stages (e.g., -n 4)*
- *-l clocked node capacitance in pico-Farad (pF) (e.g. -l 20)*

This is useful to perform a clock power trade-off study among those option parameters such as clock tree style, clock skew, die area, clocked node capacitance, etc.

9. I/O

Generally, the I/O circuits (dis)charge a large amount of loading capacitance as well as require higher supply voltage than the microprocessor core (e.g., 3.3V). This makes the I/O circuits a major contributor to the peak power dissipation of a microprocessor. Although the microprocessor may not frequently access the external memory through the I/O in the presence of L1 and L2 on-chip caches, a significant amount of power will still be consumed by the I/O circuits; the Alpha 21264 I/O circuit consumes 5% (~3.5W) of total power on average [14]. Furthermore, the fraction of power dissipated by the I/O circuits will be significantly increased because there is no on-chip L2 or L1 cache in the embedded microprocessors. However, the power consumed by I/O circuits has been ignored or not modeled properly in most frameworks for microarchitectural power estimation. There are two sources of error: 1) the lack of detailed information about the external loading capacitance connected to the I/O circuit, and 2) the I/O bus transaction model used in microarchitectural simulator.

9.1 Model

Figure 18 shows both the memory I/O access modeling in the microarchitectural simulator and the cycle-accurate I/O bus transaction modeling. For example, SimpleScalar — baseline simulator for most microarchitectural power estimation simulators — transfers all the request data blocks at the call time of the external memory access function (e.g., `mem_access` in Figure 18) and returns the access latency. The typical microprocessor transfers the blocks one by one over several I/O bus cycles with a more complex data transfer protocol. As we have noted, the cycle-based microarchitectural simulators derive their speed from abstracting out many of the physical details. Hence, we have no idea about the details of the memory transfer protocol including exact timing and bus switching activity of address and data I/O buses. To correct this, we need a mech-

anism or modification for tracing actual I/O address and data during the I/O transactions in a cycle accurate way. To provide this mechanism, it is necessary to augment the simulator to trace I/O bus streams and feed them to the power model at the pertinent I/O transaction cycle as illustrated in Figure 18.

Figure 19-(a) shows an I/O bus power model accounting for the actual I/O bus switching activity during memory I/O bus cycles. In this model the number of “0” to “1” transitions of the I/O pin is counted by comparing the blocks transferred in the previous and current I/O bus cycles. At the initiation of the I/O bus transaction cycle, the high-impedance bus state is assumed. To estimate the power dissipation from the I/O bus at a pertinent I/O cycle, the total number of I/O pin transitions of each block is transferred to the I/O circuit power model. In general, the switching capacitance of the I/O circuit consists of the intrinsic (or internal) capacitance by the I/O circuit itself and the extrinsic (or external) capacitance by the connected chipset and by the PCB interconnect between the microprocessor and the chipset I/O pins. The amount of the extrinsic capaci-

Figure 18: The memory access I/O modeling in the microarchitectural simulator and cycle-accurate I/O transaction modeling.

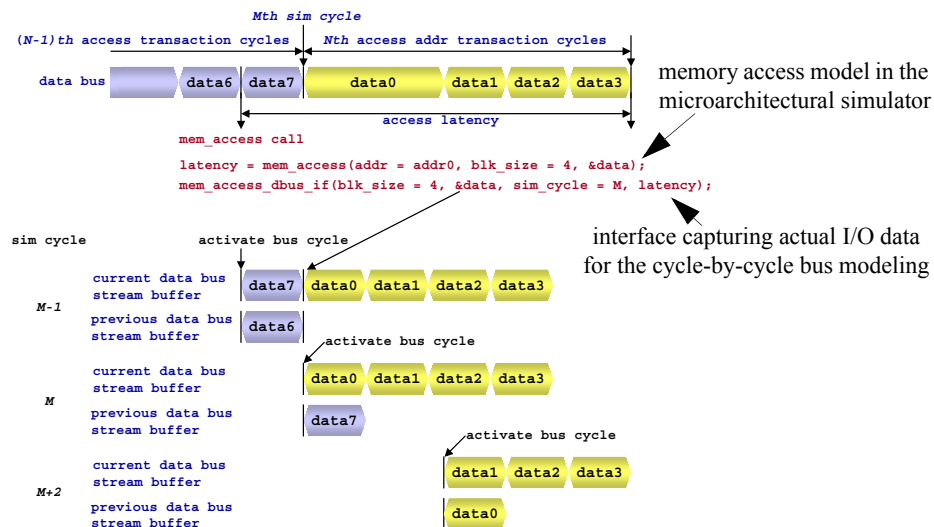
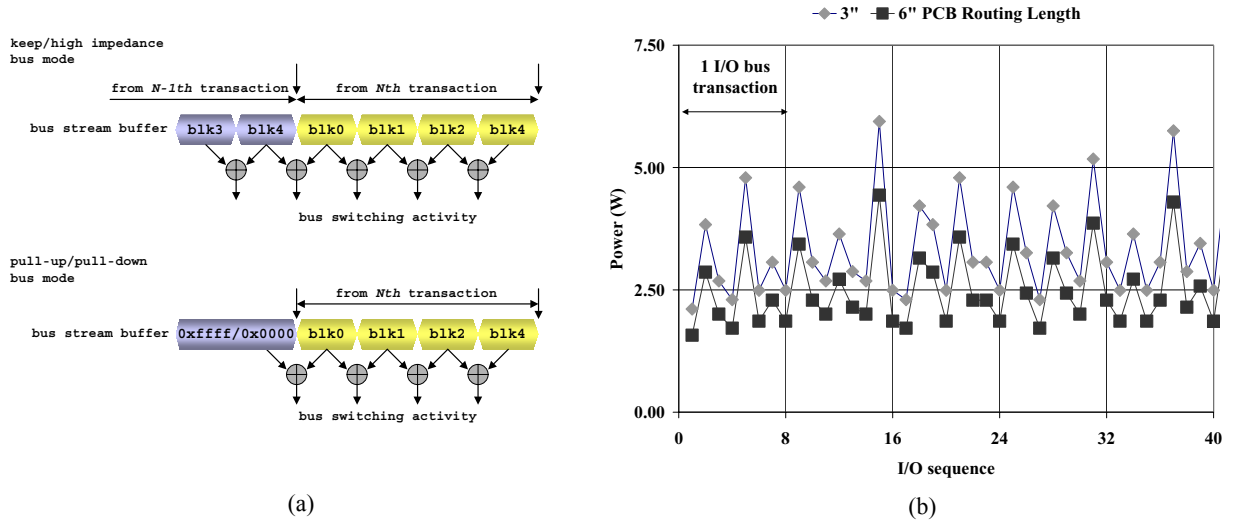


Figure 19: The I/O bus switching activity model in (a) and a snapshot of power dissipation by 64-bit processor I/O bus in (b).



In (b), it is assumed that the front system bus (or I/O bus) operating frequency is 800MHz and voltage is 2.6V [21]. The 4-wide issue machine is used for the experiment.

tance driven by the I/O circuit is more significant than that of the intrinsic capacitance by the I/O circuit itself. Therefore, it is important to estimate the extrinsic capacitance accurately.

In most high-end computer systems, the microprocessor is not directly connected to the memory module in the PC motherboard. It is connected to the memory controller through the *front system bus* (or simple I/O bus). Hence, the I/O pin capacitance of the microprocessor and chipset should be known as well as the PCB interconnect capacitance of the front system bus. According to [21], the typical package pin capacitance of both the microprocessor and chipset is 5pF per I/O pin. To estimate the PCB interconnect capacitance, some details about interconnect dimensions should be known. The interconnect dimensions and layer information is usually found in the chipset or microprocessor specification; in case the microprocessor or the chipset has not been developed, the most recent available information can be used. The estimated PCB interconnect capacitance per inch is around 2.15pF for the given specification in which the minimum and

maximum allowed front system bus interconnect lengths are 3” and 6”, respectively. Therefore, the PCB interconnect capacitance of the front system bus is between 6.5pF and 13pF depending on the interconnect length; in case of the 6” front system bus, the PCB interconnect capacitance is around 13pF, which results in a total of 23pF per pin including the package pin capacitance of both the microprocessor and chipset.

With the I/O bus capacitance and the detailed bus protocol modeling, we were able to estimate the power dissipation of the 64-bit microprocessor I/O bus with realistic parameters (see Figure 19-(b) for a snapshot of I/O bus power dissipation when running *eon*). The experiment shows that the power dissipation by the I/O bus is substantial whether the front system bus interconnect length is 3” or 6”, and it has great potential to contribute to the peak as well as the average power dissipation of the microprocessor during the I/O bus cycles. Furthermore, this experiment shows that counting switching activity in a cycle accurate way is important, because the power dissipation by I/O at a specific I/O cycle is significantly different depending on the number of I/O pin switches.

9.2 Implementation

I/O is divided into 4 subcomponents. The buffer chain, I/O pad, microstrip, and external load. We made each sub-component configurable from the *cmd* file. The I/O pad information was extracted from technology libraries. For our example, the TSMC 0.18um Artisan Cell Libraries were used. Microstrip capacitance was extracted from an impedance calculator. We used (12) to acquire PCB capacitance and made the wire length configurable. The external load is also configurable. We described I/O pads into 2 types; Bidirectional, Unidirectional. Bidirectional implies that in idle state the I/O goes to high impedance, ‘Z’, state. Unidirectional implies that it maintains the last

active value in idle state. When an external memory access occurs, we create a queue that generates I/O power estimates that occur during memory transactions. Sim-Panalyzer evaluates these estimates at the appropriate cycle. This enables us to estimate peak power for I/O in a cycle accurate manner. I/O related code is located in `./pmodel/io_panalyzer.c` and `./pmodel/io_panalyzer.h`.

$$C_0 = l \frac{0.67(\epsilon_r + 1.41)}{\log \frac{5.98h}{0.8w + t}}, \quad (12)$$

Figure 20: Cross section of Microstrip

w:width t:thickness h:height l:length



10. Conclusion

In this study, we provided power modeling methodologies for deep sub-micron microprocessors. We introduced a simple switching capacitance extraction methodology and a cycle-based logic simulation technique which can be easily embedded into a high-level microarchitectural simulator, for instance SimpleScalar. The high-level microarchitectural simulator enables the user to explore a much larger design space quickly. Combining this high-level simulator with the embedded low-level logic simulator gives us more accurate power estimation results quickly for application specific functional blocks. In addition, we illustrated and calibrated our power modeling for caches, execution units, and I/O. Our experiments show the power models track HSPICE closely for each applied vector as well as producing accurate average energy dissipation. This is achieved with a very small execution time overhead and is therefore a highly desirable method for estimating the power usage of many different microprocessor designs.

Appendix A - Sim-iPAQ

The following is the first release of the SimpleScalar iPAQ platform simulator. It is capable of booting the Linux operating system and provides a root filesystem with a variety of useful ARM Linux utilities. Comments, questions, or bug fixes may be directed to the authors via email at ss-sa@cs.colorado.edu.

In this release, the critical platform components have been implemented including ARM instruction emulation, ARM MMU support, and I/O models for the ARM iPAQ real-time clock, interrupt controller, serial devices, FLASH and DRAM memory, and the OS timer. The simulator model is able to boot the Linux kernel; however, some bootloader and Linux commands are still not functioning due to a few remaining implementation issues. Nevertheless, a significant amount of functionality exists in this release, therefore we have made it available and will update the code as bugs are identified and fixed.

A.1 Sim-iPAQ Distribution Components

Platform Simulator - The platform simulator is derived from SimpleScalar version 3.0 located at www.simplescalar.com. It includes ARM instruction emulation, ARM MMU support, and I/O models for the ARM iPAQ real-time clock, interrupt controller, serial devices, FLASH and DRAM memory, and the OS timer.

Platform Console - The platform console provides serial terminal emulation. It connects to the platform simulator and allows the user to issue bootloader and Linux command-line commands to the simulator.

ARM Bootloader - The ARM bootloader is installed into memory at simulator initialization time. It decompresses the kernel and initializes the filesystem.

ARM Linux Kernel - The ARM Linux Kernel provides operating system functionality and is decompressed by the bootloader at initialization.

ARM Linux Root Filesystem - The ARM Linux Root Filesystem provides a number of standard utilities in the root filesystem that are available after the ARM Linux Kernel boots on the platform simulator. A few examples of these standard utilities are ls, diff, and mount.

A.2 Building the Simulation Environment

The following sections will describe how to build the various components of the Sim-iPAQ simulation environment. One thing to note before beginning is that this release of Sim-iPAQ has only been tested on RedHat Linux version 8.0 for x86. However, it will probably work on any little-endian platform provided that the build uses GNU GCC for the compilation.

A.3 iPAQ Platform Model

The Sim-iPAQ platform model, located in the "sim-ipaq/" directory of the download archive, must be configured before it can be built. In addition to the normal SimpleScalar configuration parameters found in the README file, the variable LINUX_PATH in the Makefile must be set to the root location of the Linux build. Once the normal configurations are made and the Makefile is configured as above, the platform simulator, called sim-ipaq, is built with the following command:

make

The build processor will also compile the Platform Console, conveniently called console.

A.4 iPAQ Bootloader, ARM Linux Kernel, and Root Filesystem

Before beginning with this step, it should be noted that pre-built versions of the iPAQ Bootloader, ARM Linux Kernel, and Root Filesystem are provided in the Sim-iPAQ distribution, thus you may skip this step unless a custom kernel or filesystem is required.

If a custom environment is desired, the first thing that will be needed is an ARM cross compiler to build the bootloader and ARM Linux Kernel. One such cross compiler is available for SimpleScalar at the following location: <http://www.simplescalar.com/v4test.html>.

The iPAQ Bootloader, located in the "linux-build/bootldr" directory, is a free ARM-based bootloader distributed by Compaq. It provides a variety of debug functions, plus Linux kernel decompression, and root filesystem initialization. To build the bootloader execute the following command in the bootloader directory:

```
make
```

This will produce the file "bootldr.bin", which is an ELF binary format bootloader, in the format expected by the Sim-iPAQ platform simulator. See the README files for details on the commands supported by the bootloader. Additional documentation is available by executing the "help" command at the bootloader prompt.

The ARM Linux Kernel, located in the directory "linux-build/linux", has a complete kernel build. A large number of build options are available for the kernel which can be seen in the README file located in the above linux-build/linux directory. The kernel has been pre-configured with the options expected by the Sim-iPAQ platform simulator. To build the ARM Linux Kernel, execute the following command in the kernel directory:

```
make zImage
```

This will create "zImage.bin", which is a compressed ARM Linux Kernel with devices compiled in to match the devices supported by the Sim-iPAQ platform simulator.

The ARM Linux Root Filesystem provides a minimal filesystem available to users once the ARM Linux Kernel boots on the Sim-iPAQ platform simulator. The root filesystem is loaded into simulated FLASH memory as a compressed-RAM filesystem (CRAMFS). The first step to building a compressed-RAM filesystem is to build the filesystem build utility "mkcramfs", which is located in the directory "linux/kernel/scripts/cramfs/". Build this utility with the following command in the CRAMFS directory:

```
make
```

Next, assemble a filesystem, on the local host filesystem, with exactly the same ARM binaries and permissions desired on the CRAMFS. To create the CRAMFS filesystem, execute the following command:

```
linux/kernel/scripts/cramfs/mkcramfs init-2-56 init-2-56.cramfs
```

Where "init-2-56" is the top-level directory of the local representation of the filesystem to create, and "init-2-56.cramfs" is the name of the file that will contain the compressed filesystem.

A.5 Running the IPAQ Platform Model

To run the IPAQ platform model, first run the platform simulator, located in the "sim-ipaq/" directory, with the following command:

```
sim-ipaq linux-boot
```

The argument "linux-boot" indicates that the platform simulator should initiate a standard Linux boot sequence. The standard boot sequence accesses files in the directory specified by the build parameter LINUX_PATH. The sequence first reads the bootloader executable "bootldr.bin",

then the compressed Linux ARM Kernel "zImage.bin", and finally the root filesystem "init-2-56.cramfs".

After reading the FLASH ROM components into simulated FLASH RAM, the platform simulator will connect to the terminal emulator. The terminal emulator is the user's access point to the Linux simulation, providing a means for entering command lines to the boot-loader and Linux shells. The platform console is a front-end to the serial device emulator. To start the platform console, enter the following command in a separate window:

```
console -s script-boot.txt
```

This will initiate a platform console connection to the running sim-ipaq simulator, and run an initial set of bootloader commands, listed in the file "script-boot.txt". These commands are required to initialize the Linux kernel memory and CRAMFS filesystem. Once the commands complete, the Linux kernel will boot, after which the user can enter additional commands from the platform console window.

References

- [1] D. Brooks et al., "Wattch: A Framework for Architectural-Level Power Analysis and Optimizations," *Proc. 27th Int. Symp. on Computer Architecture (ISCA27)*, May 2000.
- [2] N. Vijaykrishnan, et al., "Energy-Driven Integrated Hardware-Software Optimizations Using SimplePower," *Proc. 27th Int. Symp. on Computer Architecture*, May 2000.
- [3] G. Cai et al., "Architectural Level Power/Performance Optimization and Dynamic Power Estimation," *Cool Chips Tutorial in conjunction with the 32nd Int. Symp. on Microarchitecture*, Nov 1999.
- [4] S. Wilton et al., "An Enhanced Access and Cycle Time Model for On-Chip Caches," *Western Research Laboratory Research Report 93/5*, July 1993.

- [5] H. Mehta et al., "Energy Characterization based on Clustering," *Proc. 33rd Design Automation Conf.*, June 1996.
- [6] T. Austin et al., "SimpleScalar: An Infrastructure for Computer System Modeling," *IEEE Computer*, Vol. 35, pp. 59-67, Feb 2002.
- [7] B. Geuskens, et al., "Modeling Microprocessor Performance," Kluwer Academic Publishers, 1988.
- [8] The MOSIS Service. <http://www.mosis.com>.
- [9] Berkeley Predictive Technology Model, <http://www-device.eecs.berkeley.edu/~ptm/interconnect.html>.
- [10] P. E. Landman et al., "Activity-Sensitive Architectural Power Analysis," *IEEE Transaction on CAD of Integrated Circuit and Systems*, Vol. 15, No. 6, June 1996
- [11] P. E. Landman et al., "Architectural Power Analysis: The Dual Bit Type Method," *IEEE Transaction on VLSI Systems*, Vol. 3, No. 2, June 1995.
- [12] Z. Brazilai et al., "HSS: A High-Speed Simulator," *IEEE Trans. on CAD/ICAS*, July 1987.
- [13] L. T. Wang et al., "SSIM: A Software Levelized Compiled-Code Simulator," *Proc. 24th Design Automation Conf.*, June 1987.
- [14] M. K. Gowan et al., "Power Considerations in the Design of the Alpha 21264 Microprocessor," *Proc. of 35th Design Automation Conf.*, June 1998.
- [15] J. Montanaro, et al., "A 160-MHz, 32-b, 0.5-W CMOS RISC Microprocessor," *IEEE Journal of Solid-State Circuits*, Vol 31, Nov 1996.
- [16] K. Roy and S. Prasad, "Low-Power CMOS VLSI Circuit Design," *Wiley Interscience publication*, 2000.
- [17] M. R. Guthaus et al., "MiBench: A Free, Commercially Representative Embedded Benchmark Suite," *Proc. IEEE 4th Annual Workshop on Workload Characterization*, Dec 2001.
- [18] Standard Performance Evaluation Corporation. <http://www.specbench.org>.
- [19] Intel 875 Chipset Datasheet — Platform Design Guide, <ftp://download.intel.com/design/chipsets/datashts/25252703.pdf>.
- [20] Microstrip Impedance Calculator, <http://www.emclab.umr.edu/pcbtlc2/microstrip.html>
- [21] Intel 875 Chipset Datasheet, <ftp://download.intel.com/design/chipsets/datashts/25252501.pdf>.
- [22] A. Bellaouar et al., "Low-Power Digital VLSI Design: Circuit and Systems," *Kluwer Academic Publishers*, 1996.
- [23] K. Ghose and M. Kamble, "Reducing Power in Superscalar Processor Caches using Subbanking, Multiple Line Buffers and Bit-line Segmentation," *Proc. Int. Symp. on Lower Power Electronics & Design*, Aug 1999.

- [24] R. Preston et al, "Design of an 8-wide superscalar RISC microprocessor with simultaneous multithreading", *ISSCC Digest and Visuals Supplements*, Feb 2002.
- [25] M. Hrishikesh, N. Jouppi, K. Farkas, D. Burger, S. Keckler, and P. Shivakumar. The optimal logic depth per pipeline stage is 6 to 8 FO4 inverter delays. *Proc. the 29th Int'l Symp. on Computer Architecture*, May 2002.
- [26] S. Manne et al., "An Industrial Perspective on Low Power Processor Design," *Cool Chips Tutorial in conjunction with the 32nd Int. Symp. on Microarchitecture*, Nov 1999.
- [27] Kim, T. Austin, T. Mudge, and D. Grunwald. Challenges for architectural level power modeling. in *Power Aware Computing*,(R. Melhem and R. Graybill eds.), Kluwer Academic Publishers: Boston, MA, 2001.

Publications

1. Nam Sung Kim, T. Austin, D. Blaauw, T. Mudge, K. Flautner, J. Hu, M. Irwin, M. Kandemir, N. Vijaykrishnan. Leakage Current: Moore's Law Meets Static Power. *Computer*, vol. 36, no. 12, Dec. 2003, pp. 65-77.
2. D. Ernst, N. Kim, S. Das, S. Pant, T. Pham, R. Rao, C. Ziesler, D. Blaauw, T. Austin, T. Mudge, and K. Flautner. Razor: A low-power pipeline based on circuit-level timing speculation. 36th Ann. IEEE/ACM Symp. Microarchitecture (MICRO-36), Dec. 2003, pp. 7-18. [received best paper award]
3. N. Kim, K. Flautner, D. Blaauw, and T. Mudge. Circuit and microarchitectural techniques for reducing cache leakage power. *IEEE Trans. VLSI*, 2003.
4. N. Kim, D. Blaauw, and T. Mudge. Leakage power optimization techniques for ultra deep sub-micron multi-level caches. *Proc. Int. Conf. of Computer Aided Design (ICCAD-2003)*, San Jose, CA, Nov. 2003, pp. 627-632.
5. N. Kim and T. Mudge. Microarchitecture for a low power register file with reduced register ports. *Proc. of the Int. Symp. on Low Power Electronics and Design (ISLPED)*, Seoul, Korea, Aug. 2003, pp. 384-389.
6. K. Flautner and T. Mudge. Vertigo: Automatic performance-setting for Linux. *Proc. of the 5th Operating Systems Design and Implementation (OSDI)*, Dec. 2002, pp. 105-116.
7. D. Blaauw, S. Martin, T. Mudge, K. Flautner. Leakage current reduction in VLSI systems. *Jour. of Circuits, Systems, and Computers*, 11(6), 2002, pp. 621-636.
8. N. Kim, K. Flautner, D. Blaauw, and T. Mudge. Drowsy instruction caches: Leakage power reduction using dynamic voltage scaling and cache sub-bank prediction. 35th Ann. IEEE/ACM Symp. Microarchitecture (MICRO-35), Nov. 2002, pp. 219-230.
9. S. Martin, K. Flautner, D. Blaauw, and T. Mudge. Combined dynamic voltage scaling and adaptive body biasing for lower power microprocessors under dynamic workloads. *Proc. Int. Conf. of Computer Aided Design (ICCAD-2002)*, San Jose, CA, Nov. 2002, pp. 721-725.
10. K. Flautner, S. Reinhardt, and T. Mudge. Automatic performance setting for dynamic voltage scaling. *ACM Jour. Wireless Networks*, vol. 8, no. 5, Sep. 2002, pp. 507-520.

11. K. Flautner, N. Kim, S. Martin, D. Blaauw, T. Mudge. Drowsy Caches: Simple techniques for reducing leakage power. Proc. of the 29th Ann. Int. Symp. on Computer Architecture, Anchorage Alaska, May 2002, pp. 148-157.
12. N. Kim, T. Austin, and T. Mudge. Low-energy data cache using sign compression and cache line bisection. 2nd Annual Workshop on Memory Performance Issues (WMPI). In conjunction with the 29th Ann. Int. Symp. on Computer Architecture, Anchorage Alaska, May 2002.
13. N. Kim, T. Austin, T. Mudge, and D. Grunwald. Challenges for architectural level power modeling. in Power Aware Computing, (R. Melhem and R. Graybill eds.), Kluwer Academic Publishers: Boston, MA, 2001.
14. M. Guthaus, J. Ringenberg, D. Ernst, T. Austin, T. Mudge, and R. Brown. MiBench: A free, commercially representative embedded benchmark suite. IEEE 4th Annual Workshop on Workload Characterization, (held in conjunction with 34th Ann. IEEE/ACM Symp. Microarchitecture, Austin, TX), Dec. 2001, pp. 3-14.
15. K. Flautner, S. Reinhardt, and T. Mudge. Automatic performance setting for dynamic voltage scaling. Proc. 7th Ann. Int. Conf. On Mobile Computing and Networking (MOBICOM), Rome, Italy, July 2001, pp. 260-271.
16. T. Mudge. Power: A first class design constraint. Computer, vol. 34, no. 4, April 2001, pp. 52-57.