3rd Annual
Austin Conference on Integrated Systems & Circuits
2008 Publication

KEYNOTE ADDRESS **Dr. Necip Sayiner,** ***President, CEO, and Board Member*** ***Silicon Laboratories***	
Track 1: DIGITAL DESIGN-1 **Session Chair:** ***Mike Seningen***	Track 2: ANALOG/BIO **Session Chair:** ***Ka Leung***
1-1 Design Automation and Verification Methodology Challenges of the Intel Atom Processor ***Rajesh Gupta***	***2-1*** ***Invited Paper:*** Biosensor Microarrays in CMOS ***Arjang Hassibi***
1-2 Reducing Flip-Flop Power for DSP Design ***Bassam Mohd,*** *Martin Saint-Laurent****, Paul Bassett, and Shahid Imam***	2-2 Rail to Rail Fully Differential Sample and Hold Based on Clocked Diffrential Difference Amplifier using Resistive Local Common Mode Feedback ***Jaime Ramirez-Angulo, Clara Lujan-Martinez, Carlos Rubia-Marcos, Ramon G. Carvajal, and Antonio Lopez-Martin***
1-3 Adaptive Voltage Tuning for Dual-Vdd ASICs ***Stephen Bijansky and Adnan Aziz***	2-3 Buck-Boost Converter Based Power Conditioning Circuit for Low Excitation Vibrational Energy Harvesting ***Arvindh Rajasekaran, Abhiman Hande, and Dinesh Bhatia***
Track 3: CAD-1 **Session Chair:** ***Michael Solka***	Track 4: COMPUTER ARITHMETIC **Session Chair:** ***Earl Swartzlander***
3-1 Modeling of NBTI-Induced PMOS Degradation under Arbitrary Dynamic Temperature Variation ***Bin Zhang and Michael Orshansky***	***4-1*** ***Invited Paper:*** Automated Multiplier Design ***K'Andrea C. Bickerstaff and Earl Swartzlander***

3-2 ELIAD: Efficient Lithography Aware Detailed Router with Compact Post-OPC Printability Prediction ***Minsik Cho, Kun Yuan, Yongchan Ban, and David Z. Pan***	4-2 Full Adder Evaluation and Selection for a Parallel Multiplier ***Mike Spear, Gaurav Tuteja, and Earl Swartzlander***
3-3 COOLER - A Fast Multiobjective Fixed-Outline Thermal Floorplanner ***Debarshi Chatterjee, Theodore Manikas, and Igor Markov***	4-3 Dual Vdd Design Optimization of Fast Multipliers ***Eun Jung Jang, Earl Swartzlander, and Jebediah Keefe***
3-4 Dynamic Compaction with Recursive Learning for Delay Test ***Zheng Wang and Duncan M. H. Walker***	4-4 A Hybrid Approach to Last Stage Addition for Wallace and Dadda Multipliers ***James Haydn Nelson, Eiman Ebrahimi, Mahnaz Sadoughi, and Earl Swartzlander***
TUTORIAL Fractional-N PLL ***Dr. Axel Thomsen,*** ***Silicon Labs***	
KEYNOTE ADDRESS **Dr. Jason Rhode,** ***President and CEO*** ***Cirrus Logic***	
Track 5: DIGITAL DESIGN-2 **Session Chair:** ***Adnan Aziz***	Track 6: ANALOG AND RF **Session Chair:** ***TR Viswanathan***
5-1 ***Invited Paper:*** A Sub 1W to 2W Low Power IA Processor for Mobile Internet Devices in 45nm Hi-K Metal Gate CMOS ***Gianfranco Gerosa, Steve Curtis, Mike D'Addeo, Bo Jiang, Belliappa Kuttanna, Feroze Merchant, Binta Patel, Mohammed Taufique, Haytham Samarchi, and Christopher Weaver***	***6-1*** ***Invited Paper:*** Receivers Design: Case Studies ***Hesam Amir-Aslanzadeh and Edgar Sanchez-Sinencio***
5-2 A High Speed 128-Point Fast Fourier Transform Circuit for OFDM Systems ***Tung-Yeh Wu and Jacob Abraham***	6-2 Feed-Forward Interference Suppression for Broadband Systems ***Xin Wang and Ranjit Gharpurey***

5-3 Hardware Trojan Modeling and Detection Techniques ***Daniel G. Saab, Fatih Kocan, and Jacob Abraham***	6-3 A Linear Transconductor using Series-Connected CMOS Quad ***Venkatesh Acharya, Bhaskar Banerjee, and TR*** *Viswanathan*
5-4 ***Invited Paper:*** Migration of Cell Broadband Engine from 65nm SOI to 45nm SOI ***Osamu Takahashi*** ***(Scott Cottier presenting)***	6-4 Indirect Compensation Techniques for Three-Stage CMOS Op-Amps ***Vishal Saxena, Jacob Baker, and TR Viswanathan***
Track 7: CAD-2 **Session Chair:** ***Michael Orshansky***	Track 8: SYSTEMS & ARCHITECTURES **Session Chair:** ***Mattan Erez***
7-1 3D Resistance Extraction with Lithographic and Scattering Effect ***Ying Zhou, Zhuo Li, and Weiping Shi***	8-1 Power Analysis of a Path-Based Perception Branch Predictor ***Justin Friesenhahn, Lizy Kurian John, and Mark McDermott***
7-2 Accelerating Statistical Static Timing Analysis using Graphics Processing Units ***Kanupriya Gulati and Sunil Khatri***	8-2 Hardware / Software Tradeoffs in Multicore Architectures ***Steven Guccione***
7-3 Autonomous Optical Proximity Correction: The New Frontier of Design for Manufacturing? ***Shanhu Shen, Peng Yu, and David Z. Pan***	8-3 Workload Slicing for Detailed Pre-Silicon Power Estimation ***Hassan Al-Sukhni, James Holt, David Lindberg, and Michele Reese***
TUTORIAL Holistic Coupling of Manufacturing and Design ***Dr. Sani Nassif,*** **IBM**	
TUTORIAL Out of Order Superscalar Architecture ***Dr. Derek Chiou,*** **University of Texas at Austin**	
TUTORIAL Constraint Solving for Functional Verification ***Dr. Andreas Kuhlmann,*** **Cadence Berkeley Labs**	

Power Analysis of a Path-Based Perceptron Branch Predictor

Justin J. Friesenhahn, Lizy Kurian John, and Mark McDermott
Department of Electrical and Computer Engineering
The University of Texas at Austin
{freeze, ljohn, mcdermot}@ece.utexas.edu

Abstract

Dynamic branch prediction is a technique that has increased the performance in modern microprocessors. From a power perspective, it is important to consider the tradeoffs when comparing alternatives. Conventional dynamic branch prediction includes one-bit and two-bit counter-based predictors. More recently, perceptron-based branch predictors have proven effective for high prediction accuracy. Using analytical techniques, this paper studies a 65 nm area, timing, and power analysis comparing a baseline gshare predictor to a path-based perceptron predictor. With these predictors, the path-based perceptron predictor shows a 28% area increase, a 9% prediction delay decrease, and an 8x power increase compared to the gshare predictor.

1. Introduction

Power is an important consideration when comparing design alternatives. When designing dynamic branch predictors, conventional techniques include and two-bit counter based approaches. For high prediction accuracy, perceptron-based approaches offer one alternative.

The conventional techniques involve predictors with one or multiple arrays of two-bit counters [1]. Created by Smith in 1981, the Smith predictor is one of the first two-bit counter based predictors. Yeh and Patt created the two-level predictor in 1991 [2]. McFarling created the gshare and tournament predictors in 1993 [1]. Other two-bit counter based predictors include the bimode, gskewed, agree, YAGS, alloyed, and path history predictors. Each of these predictors has a pattern history table (PHT) of two-bit counters used to make predictions.

For perceptron-based predictors, Jimenez and Lin introduced the global perceptron predictor in 2001 [3]. In 2002, Jimenez and Lin created the global/local perceptron. Jimenez created the path-based predictor in 2003 [4]. Seznec introduced the redundant-history predictor in 2004 that uses several variations of branch history to skew the indices [5]. Tarjan and Skadron created the gshare perceptron in 2005 which uses indexing similar to the two-bit counter gshare [6]. Jimenez combined the global perceptron and the path-based indexing to create the piecewise linear predictor in 2005 [7]. Tu, Chen, and John created the history-skewed predictor in 2007, which uses branch history to skew the indices [8]. For another enhancement, Jimenez also created the idealized predictor which uses the XOR to create the indices [9]. Created by Ninomiya and Abe in 2007, the path-traced predictor uses local history to skew the indices [10]. Each of these predictors includes a table of perceptron weights (TPW) used to make predictions.

The performance improvements of perceptron-based predictors includes a 13% to 33% misprediction reduction and a 2% to 19% IPC improvement compared to the conventional techniques [4, 11]. While the performance of perceptron-based predictors has been impressive, the implementation has its challenges. The previous work includes area and timing results for perceptron-based predictors [4, 8], but a predictor power analysis is not conducted.

Given the lack of research in this area, this paper includes a power analysis comparing a baseline two-bit counter based gshare predictor to a path-based perceptron predictor. The following sections cover the power analysis methodology, the predictor configurations, and the resulting power. The area and timing details for these implementations are also included.

2. Methodology

The power analysis methodology involves using analytical models to estimate the power required for each predictor [12, 13, 14, 15, 16]. The analysis is broken into standard-cell logic and memory arrays for a 65 nm 0.9 V technology. The logic power is calculated using a characterized 65 nm standard-cell library.

For the standard cells, Equation 1 below shows the gate power P_{gate}, gate leakage $P_{gate\text{-}leakage}$, SD leakage $P_{SD\text{-}leakage}$, interconnect power $P_{interconnect}$, glitch power P_{glitch}, and overall standard-cell logic power P_{logic} [12, 13].

$$P_{gate} = ½ \times switching \times activity \times C_{gate} \times freq \times V_{dd}^2 \quad (a)$$
$$P_{gate\text{-}leakage} = W \times percent\_on \times I_{gate\text{-}leakage\text{-}per\text{-}W} \times V_{dd} \quad (b)$$
$$P_{SD\text{-}leakage} = stacking \times W \times I_{SD\text{-}leakage\text{-}per\text{-}W} \times V_{dd} \quad (c)$$
$$P_{interconnect} = ½ \; switching \times activity \times l \times C_{per\text{-}l} \times freq \times V_{dd}^2 \quad (d)$$
$$P_{glitch} = 0.15 \times (P_{gate} + P_{interconnect}) \quad (e)$$
$$P_{logic} = P_{gate} + P_{gate\text{-}leakage} + P_{SD\text{-}leakage} + P_{interconnect} + P_{glitch} \quad (f)$$

Equation 1: Power (a) gate (b) gate leakage (c) SD leakage (d) interconnect (e) glitch (f) standard-cell logic

These equations are used for the standard-cell power estimates in this paper. For these equations, each standard cell has a unique input capacitance C_{gate}, *switching* factor, and *stacking* factor [12]. For the standard cells the input capacitances C_{gate} range from 1 fF to 8 fF, the *switching* factors range from 3% to 28%, and the *stacking* factors range from 0.25 to 1.00. The variables W_{min}, $I_{gate\text{-}leakage\text{-}per\text{-}W}$, $I_{SD\text{-}leakage\text{-}per\text{-}W}$, and $C_{per\text{-}l}$ are technology specific characteristics, and for the 65 nm technology used, W_{min} is 0.130 µm, $I_{gate\text{-}leakage\text{-}per\text{-}W}$ is 34 nA/µm, $I_{SD\text{-}leakage\text{-}per\text{-}W}$ is 15.6 nA/µm, and $C_{per\text{-}l}$ is 0.23 fF/µm. A *percent_on* of 50% is used for the leakage. The average wire length l of 57 µm is used for the interconnect between standard cells. The overall standard-cell logic power P_{logic} is calculated by adding each power component together.

For the memory array, a six transistor static random access memory (SRAM) is used for the area, power, and delay estimates [12]. Figure 1 below shows the layout and dimensions for the 65 nm SRAM bitcell.

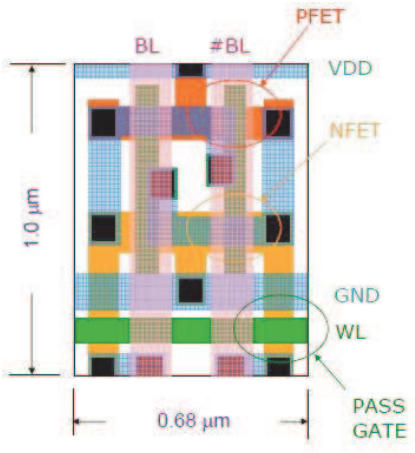


Figure 1: 65 nm SRAM bitcell [12]

These SRAM bitcells are unique for each technology, and the size of this 65 nm SRAM bitcell is 1.0 µm high by 0.68 µm wide [12]. The PFET transistors and NFET pass transistors are the minimum width W_{min} which is 0.130 µm. The pull-down NFET transistors have the width two times minimum 2 W_{min} which is 0.260 µm. Each SRAM cell can be placed directly side by side to create a memory array. To reinforce power, one extra cell is added every 16 bitcells to account for the added area.

Equation 2 below shows the memory-array power calculations for the wordline $P_{wordline}$, bitlines $P_{bitlines}$, sense-amps $P_{sense\text{-}amps}$, clock P_{clock}, leakage $P_{array\text{-}leakage}$, and overall array P_{array} [12, 13, 16].

$$P_{wordline} = ½ \times activity \times C_{wordline} \times freq \times V_{dd}^2 \quad (a)$$
$$P_{bitlines} = ½ \times activity \times C_{bitlines} \times freq \times \Delta V \times V_{dd} \quad (b)$$
$$P_{sense\text{-}amps} = ½ \times activity \times C_{sense\text{-}amps} \times freq \times V_{dd}^2 \quad (c)$$
$$P_{clock} = C_{clock} \times freq \times V_{dd}^2 \quad (d)$$
$$P_{array\text{-}leakage} = stacking \times W \times I_{array\text{-}SD\text{-}leakage\text{-}per\text{-}W} \times V_{dd} \quad (e)$$
$$P_{array} = P_{wordline} + P_{bitlines} + P_{sense\text{-}amps} + P_{clock} + P_{array\text{-}leakage} \quad (f)$$

Equation 2: Power (a) gate (b) gate leakage (c) SD leakage (d) interconnect (e) glitch (f) standard-cell log Memory array (a) wordline power (b) bitline power (c) sense-amp power (d) clock power (e) leakage power (f) array power

These equations are used for the memory-array power estimates in this paper. For the dynamic memory-array power, $P_{wordline}$, $P_{bitlines}$, $P_{sense\text{-}amps}$, and P_{clock} are determined by the *activity* factor, the amount of capacitance switched per access, frequency, and voltage [14, 16]. The *activity* factor of 100% is used since branch prediction is calculated every cycle whether used or discarded [1]. For the capacitance, the wordlines include metal at 0.23 fF/µm and gate capacitance at 1.8 fF/µm [12]. The bitlines include metal at 0.23 fF/µm and diffusion capacitance at 0.05 fF/µm. The sense-amps include 28 fF of capacitance [16] and the precharge circuits controlled by the clock include gate capacitance at 1.8 fF/µm [12]. The ΔV is calculated as 15% V_{dd}. The array leakage is determined by the SD leakage, the gate leakage is negligible. The *stacking* factor of 0.18 and SD leakage $I_{array\text{-}SD\text{-}leakage\text{-}per\text{-}W}$ of 20 nA/µm.

For both the standard-cell power and the memory-array power, the estimates in this paper use a frequency of 1 GHz and a voltage V_{dd} of 0.9 V. Tables of the 65 nm technology characteristics and standard cells are also shown in references [12] and [17].

3. Predictor Configurations

This section includes the gshare and path-based perceptron predictor configurations for a 16 KB hardware budget. For branch prediction, the SRAM bitcells form the memory array to store the values and the standard cells create the logic used to make the prediction and update the values.

Figure 2 below shows the gate-level implementation used for the gshare predictor in this paper.

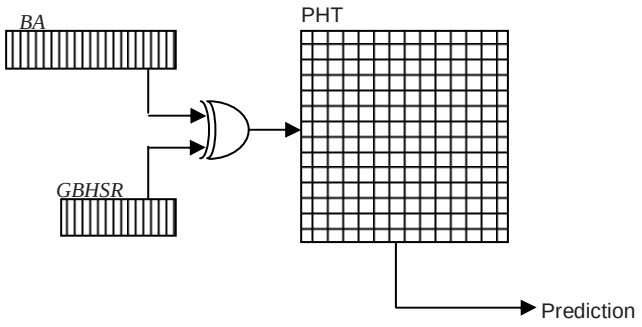


Figure 2: gshare predictor gate-level implementation

With a 16 KB hardware budget, the exclusive-OR (XOR) of 16 bits from the branch address *BA* and 16 bits of global branch history shift register (*GBHSR*) are used for the PHT index [1]. The PHT is an SRAM array that is 65,536 by 2 bits. The update logic, which is not shown above, includes logic to increment and decrement the two-bit counter and logic to control the write back to memory. The total logic includes 53 standard cells with an area of 515 μm^2 and a memory-array area of 181507 μm^2. The total prediction delay with this implementation is 763 ps.

For this gshare predictor, the 65,536 by 2 PHT is broken into four 4 KB memory sub-arrays. Figure 4 below shows the floorplan for each 4 KB sub-array.

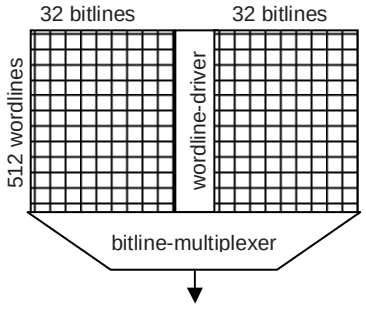


Figure 3: Floorplan for 4 KB sub-array

As shown, in each sub-array, there are 512 wordlines and 64 bitlines. The wordline driver is placed in the middle of the wordlines with half of the bitlines on each side [12, 13]. The bitline multiplexer selects 2 bits from the bitlines. This floorplan requires 16 address bits: 9 bits to select the wordline, 5 bits to select the bitlines, and 2 bits to select the sub-array. Combining four sub-arrays of 4 KB makes the top-level 16 KB memory array which is 65,536 by 2 bits. This approach reduces the dynamic power and speeds up the access time, but there is added delay to select the appropriate sub-array.

For the path-based perceptron predictor, the branch path history determines the TPW indices, and the prediction is ahead pipelined [4]. Figure 4 below shows the gate-level implementation used in this paper.

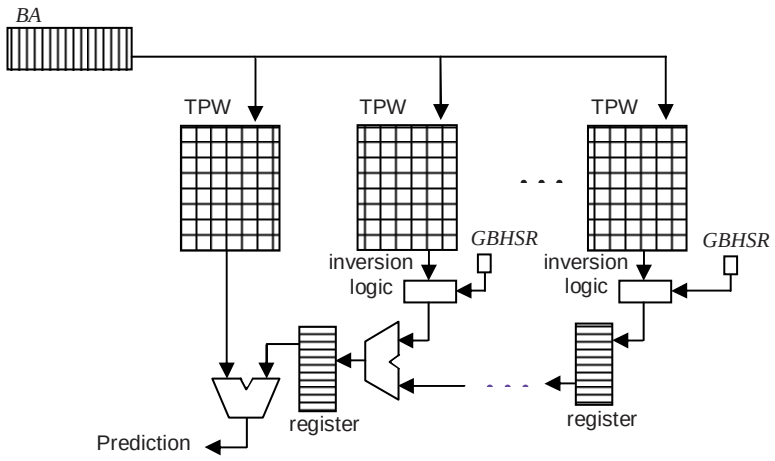


Figure 4: Path-based perceptron predictor gate-level implementation

Each perceptron weight is accessed from a separate table of perceptron weights (TPW) [4]. For a 16 KB hardware budget, there are $h + 1$ TPWs, 31 in this case, and each TPW is 512 by 8 bits. The index for each TPW is 9 bits from the branch address *BA*. When the vector weights *w* are accessed, the sign is inverted or remains the same with the inversion logic using the *GBHSR*. The addition of weights is ahead pipelined, and the total is accumulated over *h* cycles using a separate adder for each stage, 30 adders for this predictor. The final prediction is made using the bias weight *bw* from the TPW on the left and the accumulated vector weight *w* total to generate the prediction. The TPW updates occur with logic to increment and decrement the weight values and logic to control write back to memory. The total logic includes 5041 standard cells with an area of 42267 μm^2 and a

memory array area of 191125 µm^2. With the ahead pipeline, the total prediction delay with this implementation is 692 ps.

For the path-based perceptron predictor with a 16 KB hardware budget, there are 31 memory arrays each 512 B. Figure 5 below shows the floorplan for each 512 B memory array.

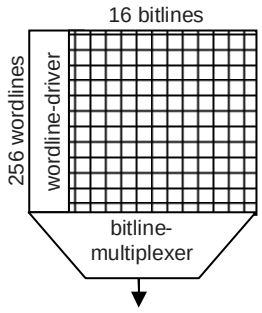


Figure 5: Floorplan for a 512 B array

In each 512 B memory array, there are 256 wordlines and 16 bitlines. The wordline driver is placed to the left of the wordlines. The bitline multiplexer selects 8 bits from the bitlines. This floorplan requires 9 address bits: 8 bits to select the wordline and 1 bit to select the bitlines.

4. Results

This section includes the power analysis results separated into standard-cell power and memory-array power. A power comparison of the total power is also shown in this section.

The standard-cell power includes gate power P_{gate}, gate leakage $P_{gate\text{-}leakage}$, SD leakage $P_{SD\text{-}leakage}$, interconnect power $P_{interconnect}$, glitch power P_{glitch}. The gshare predictor has 53 standard cells and the path-based perceptron predictor has 5041 standard cells leading to an 81x area increase. With a 1 GHz frequency *freq*, a 100% *activity* factor, and the average *switching* factor, the gate power P_{gate} is 0.0169 mW for the gshare predictor and 1.1112 mW for the path-based perceptron predictor. Using a W_{min} of 0.130 µm, *percent_on* of 50%, and $I_{gate\text{-}leakage\text{-}per\text{-}W}$ of 15.6 nA/µm, the gate leakage power $P_{gate\text{-}leakage}$ is 0.0001 mW for the gshare predictor and 0.0055 mW for the path-based perceptron predictor. Using the *stacking* factors, W_{min} of 0.130 µm, and $I_{SD\text{-}leakage\text{-}per\text{-}W}$ of 34 nA/µm, the SD leakage power $P_{SD\text{-}leakage}$ is 0.0028 mW for the gshare predictor and 0.1754 mW for the path-based perceptron predictor. Using a 100% *activity* factor, an average wire length l of 57 µm, a $C_{per\text{-}l}$ of 0.23 fF/µm, and a 1 GHz frequency *freq*, the interconnect power $P_{interconnect}$ is 0.0346 mW for the gshare predictor and 3.3196 mW for the path-based perceptron predictor. The glitch power P_{glitch} is 15% of the gate and interconnect power: 0.0077 mW for the gshare predictor and 0.6646 mW for the path-based perceptron predictor. Table 2 below summarizes the standard-cell power results.

Component	Power Used (mW)	
	gshare	Path-based perceptron
Gate Power	0.0169	1.1112
Gate Leakage	0.0001	0.0055
SD Leakage	0.0028	0.1754
Interconnect Power	0.0346	3.3196
Glitch Power	0.0077	0.6646
Standard-Cell Power	0.0622	5.2763

Table 2: Power comparison of standard-cell implementation

The overall standard-cell power P_{logic} is 5.2763 mW for the path-based perceptron predictor compared to 0.0622 mW for the gshare predictor. The standard-cell area increase of 81x leads to an 83x increase in standard cell power P_{logic}. For each predictor the gate power P_{gate} and interconnect power $P_{interconnect}$ are the dominant components. Looking into these components, the difference between predictors is the capacitance. The overall gate capacitance C_{gate} is 327 fF for the gshare predictor and 22111 fF for the path-based perceptron predictor. The overall interconnect capacitance is 695 fF for the gshare predictor and 66087 fF for the path-based perceptron predictor.

The memory-array power includes wordline power $P_{wordline}$, bitline power $P_{bitlines}$, sense-amp power $P_{sense\text{-}amps}$, clock power P_{clock}, leakage $P_{array\text{-}leakage}$. The dynamic memory-array power is determined by the amount of capacitance switched per access, in this case the wordline, bitlines, and the precharge circuit controlled by the clock [14, 16]. The array leakage is determined by the SD leakage, the gate leakage is negligible.

For each access, the gshare predictor accesses one sub-array within the memory array, and the path-based perceptron predictor accesses 31 memory arrays [1, 4]. Using the 100% *activity* factor and a 1 GHz frequency *freq*, the wordline power $P_{wordline}$ is 0.0082 mW for the gshare predictor and 0.1274 mW for the path-based perceptron predictor. The power for the bitlines $P_{bitlines}$ is 0.2354 mW for the gshare predictor and 1.8243 mW for the path-based perceptron predictor. The sense-amp power $P_{sense\text{-}amps}$ is 0.0113 mW for the gshare predictor and 0.3515 mW for the path-based perceptron predictor. The clock power P_{clock} is 0.3882 mW for the gshare

predictors and 0.7521 mW for the path-based perceptron predictor. The SD leakage is reduced to 20 nA/um in the memory array, and the leakage power $P_{array\text{-}leakage}$ is 0.2123 mW for the gshare predictor and 0.2057 mW for the path-based perceptron predictor. Table 3 below summarizes the results of these power calculations for both predictors.

Component	Power Used (mW)	
	gshare	Path-based perceptron
Wordline Power	0.0082	0.1274
Bitline Power	0.2354	1.8243
Sense Amp Power	0.0113	0.3515
Clock Power	0.3882	0.7521
Array Leakage Power	0.2123	0.2057
Memory Power	0.8554	3.2610

Table 3: Power comparison of memory-array implementation

The overall memory power P_{array} is 3.2610 mW for the path-based perceptron predictor compared to 0.2123 mW for the gshare predictor. The 5% array area increase leads to a 3x increase in memory power P_{array}. This increase is caused by accessing 31 memory arrays as opposed to one sub-array. The bitline power $P_{bitlines}$ is the dominant component for the path-based perceptron predictor, but the clock power P_{clock} is the dominant component for the gshare predictor. This is due to the capacitance difference for each configuration. The path-based perceptron predictor has a dominant capacitance with the bitline capacitance $C_{bitlines}$, 969 fF per 512 B memory array, making the bitline power $P_{bitlines}$ dominant. For the gshare predictor, the clock capacitance C_{clock} is dominant, 120 fF per 4 KB sub-array.

Combining the standard-cell power P_{logic} and the memory power P_{array}, the path-based perceptron predictor has a total power of 8.5373 mW compared to the gshare predictor with a total power of 0.9176 mW, shown below in Table 4.

Component	Power Used (mW)	
	gshare	Path-based perceptron
Standard-Cell Power	0.0622	5.2763
Memory Power	0.8554	3.2610
Total Power	0.9176	8.5373

Table 4: Power comparison of final results

For the path-based perceptron predictor, the dominant power component is the standard-cell power compared to a dominant memory array power for the gshare predictor. The increase in standard-cell power for the path-based perceptron predictor is caused by the prediction computation, 30 adders plus additional logic, compared to 16 XOR gates for the gshare predictor. Both predictors have the same hardware budget of 16 KB, but there is a substantial difference in memory power. Since the gshare predictor memory array can be split into four sub-arrays, the memory-array power is reduced from one large memory array [12]. For the path-based perceptron predictor, each memory array is smaller than the gshare predictor sub-array, but all 31 memory arrays must be accessed each cycle. Overall, the path-based perceptron predictor shows an 8x power increase compared to the gshare predictor.

5. Conclusion

This paper included a 65 nm area, timing, and power analysis comparing a baseline gshare predictor to a path-based perceptron predictor. Analytical techniques were used to provide estimates, and the power analysis was grouped into standard-cell power and memory-array power. For the path-based perceptron predictor the dominant component is the standard-cell power, compared to the baseline gshare predictor where memory power is the dominant component. Looking at the total power, the results show an 8x power increase for the path-based perceptron predictor compared to the gshare predictor. Considering an area increase of 28%, the extra power increase comes from the added computation required for perceptron-based predictors. As expected, the standard cell area increase of 81x leads to an 83x increase in standard cell power. For the memory array, there is a 5% increase in array area yet there is an increase of 3x in memory power. This is caused by the path-based perceptron predictor accessing 31 memory arrays per cycle compared to the gshare predictor accessing one memory sub-array.

When considering performance, the path-based perceptron predictor has a 13% to 33% misprediction reduction and a 2% to 19% IPC improvement compared to the gshare predictor [4, 11]. For these performance benefits, based on these findings, the computation heavy path-based perceptron predictor leads to an 8x power increase and a 28% area increase, and use of the gshare predictor is recommended when targeting low power. Given the path-based perceptron predictor has a misprediction reduction, an IPC improvement, and a 9% prediction delay decrease, the total program runtime is reduced which also reduces total energy. Future work involves comparing total energy of sample programs for each predictor.

References

[1] J.P. Shen and M.H. Lipasti, *Modern Processor Design: Fundamentals of Superscalar Processors*, New York: McGraw-Hill, 2005.

[2] T-Y. Yeh and Y.N. Patt, "Two-Level Adaptive Training Prediction," *International Symposium on Microarchitecture*, 1991.

[3] D.A. Jimenez and C. Lin, "Neural Methods for Dynamic Branch Prediction," *ACM Transactions on Computer Systems Vol. 20 No. 4*, pp. 369-397, November 2002.

[4] D.A. Jimenez, "Fast Path-Based Neural Branch Prediction," *IEEE Proceedings of the 36th International Symposium on Microarchitecture*, pp. 243-252, December 2003.

[5] A. Seznec, "Revisiting the Perceptron Predictor," pp. 1-21, May 2004.

[6] D. Tarjan and K. Skadron, "Merging Path and gshare Indexing in Perceptron Branch Prediction," *ACM Transactions on Architecture and Code Optimization Vol. 2 No. 3*, pp. 280-300, September 2005.

[7] D.A. Jimenez, "Piecewise Linear Branch Prediction," *Proceedings of the 32nd International Symposium on Computer Architecture*, June 2005.

[8] J. Tu, J. Chen, and L.K. John, "Hardware Efficient Piecewise Linear Branch Predictor," *20th International Conference on VLSI Design*, January 2007.

[9] D.A. Jimenez, "Idealized Piecewise Linear Branch Prediction," pp. 1-4.

[10] Y. Ninomiya and K. Abe, "A3PBP: A Path Traced Perceptron Branch Predictor Using Local History for Weight Selection," *Journal of Instruction-Level Parallelism*, pp. 1-18, May 2007.

[11] J.J. Friesenhahn, "High Performance Perceptron-Based Branch Prediction," *University of Texas at Austin Master's Report*, December 2007.

[12] M. McDermott, G. Gerosa, and S. Sullivan, "VLSI II Lecture Notes," Spring 2007.

[13] N.H.E. Weste and D. Harris, *CMOS VLSI Design*, Boston: Pearson Education, 2005.

[14] S.M. Kang and Y. Leblebici, *CMOS Digital Integrated Circuits*, New York: McGraw-Hill, 2003.

[15] A. Chandrakasan, W.J. Bowhill, and F. Fox, *Design of High-Performance Microprocessor Circuits*, IEEE Press, 2000.

[16] B. Toronyi, "Power Modeling," *University of Texas at Austin Master's Report*, December 2005.

Power Analysis of a Path-Based Perceptron Branch Predictor

Justin J. Friesenhahn, Lizy Kurian John, and Mark McDermott

Overview

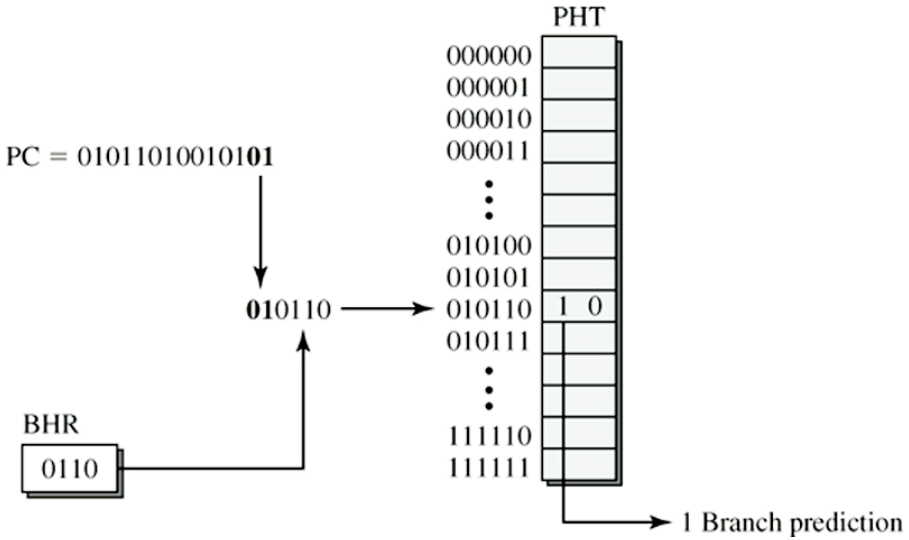
- Motivation
- Methodology for evaluating predictors
- Predictor implementations
- A comparison of the results

Types of Branch Predictors

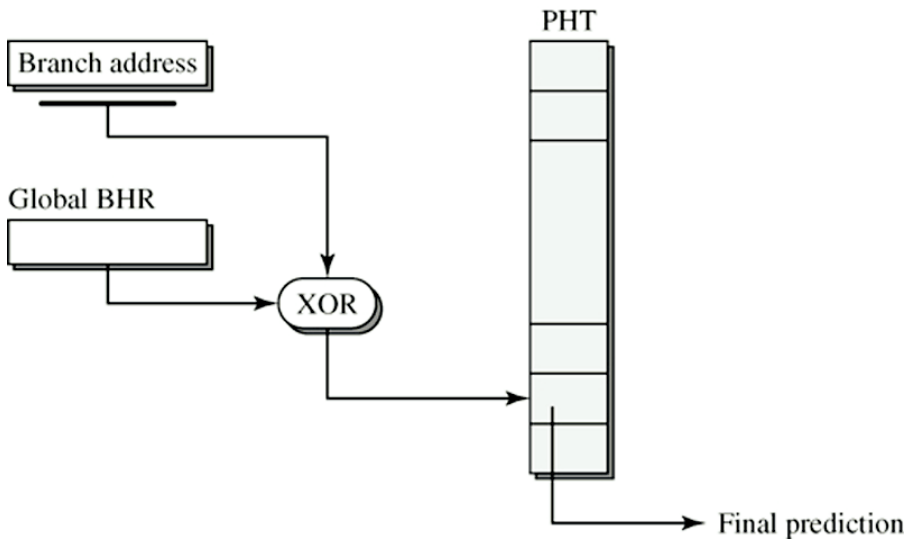
- Conventional predictors
  - Examples: Smith, two-level, gshare
  - Pattern history table (PHT): 2-bit counters
  - Prediction typically made with most significant bit

- Perceptron-based predictors
  - Table of perceptron weights (TPW): 8-bit weights
  - Prediction made by sign of combined weights

Two-Level [1]



gshare [1]



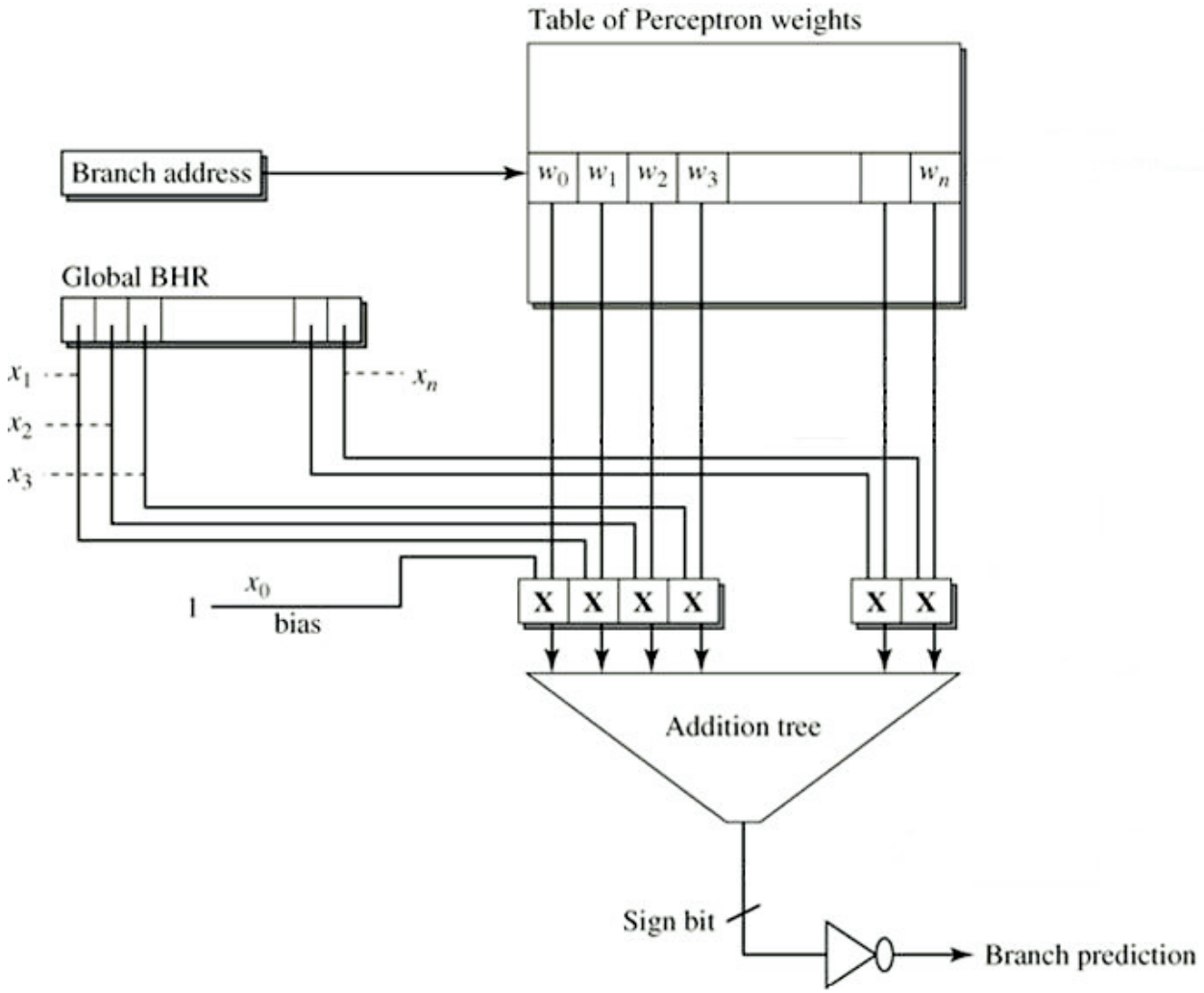
Perceptron Prediction Algorithm

$$y = bw + \sum_{i=1}^{h} GBHSR[i]w[i]$$

$y \geq 0$ (positive) predict taken

$y < 0$ (negative) predict not taken

Perceptron Predictor [1]



Motivation

- Published papers showing performance benefits for perceptron-based predictors

  Example: 16 KB hardware budget

  – Misprediction reduction of 13% to 33%

  – IPC performance improvement of 2% to 19%

- Lack of research including power analysis of perceptron-base predictors

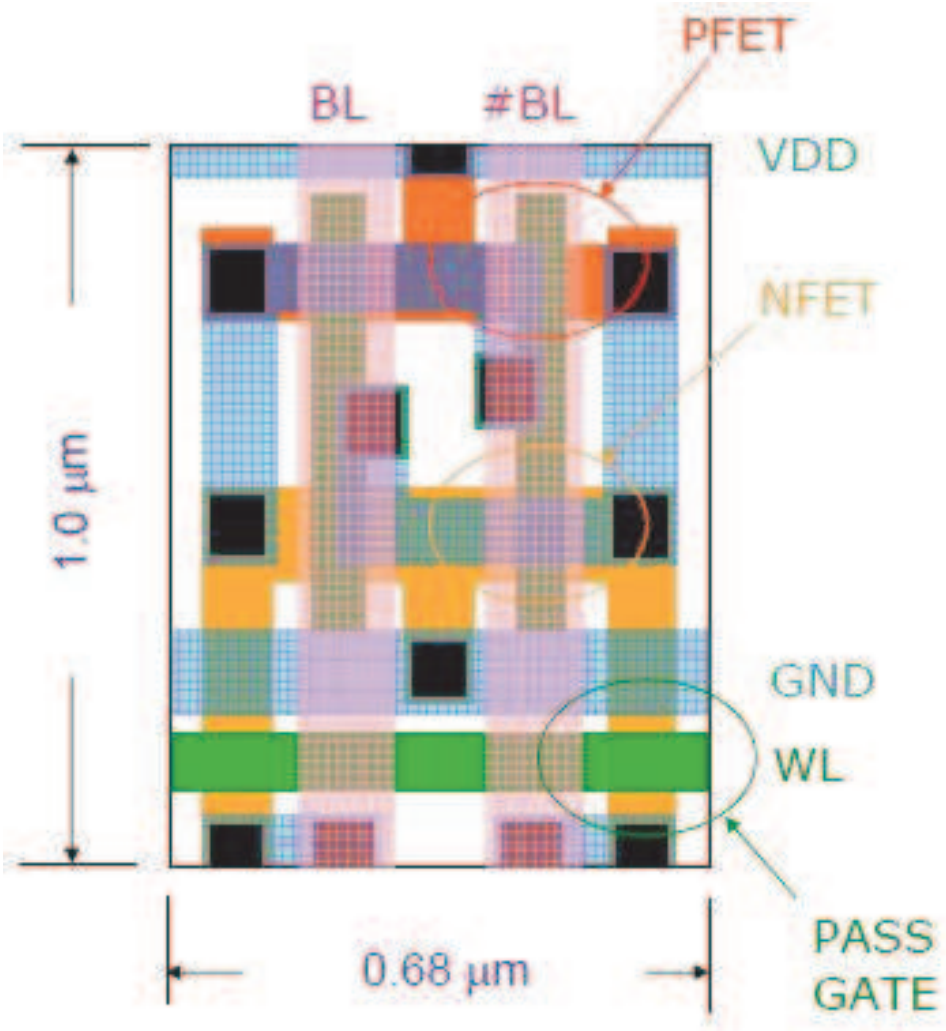
Evaluation of Predictors

- Implemented baseline gshare and path-based perceptron predictors with 16 KB hardware budget
- Measured area, prediction delay, and power
- Compared path-based perceptron results to baseline gshare predictor

Methodology

- 65 nm technology
  - Standard cells for control logic
  - SRAM array for memory
- Voltage: 0.9 V
- Frequency: 1 GHz
- Activity Factor: 100%

65 nm Bitcell



Power Analysis Components

- Standard-cells
  - Gate and interconnect power
  - Gate and SD leakage power
  - Glitch power
- Memory arrays
  - Wordlines, bitlines, sense-amps, clock power
  - Array leakage power

Standard Cell Power Equations

$$P_{gate} = ½ \times switching \times activity \times C_{gate} \times freq \times V_{dd}^2$$

$$P_{gate\text{-}leakage} = W \times percent\_on \times I_{gate\text{-}leakage\text{-}per\text{-}W} \times V_{dd}$$

$$P_{SD\text{-}leakage} = stacking \times W \times I_{SD\text{-}leakage\text{-}per\text{-}W} \times V_{dd}$$

$$P_{interconnect} = ½ \times switching \times activity \times l \times C_{per\text{-}l} \times freq \times V_{dd}^2$$

$$P_{glitch} = 0.15 \times (P_{gate} + P_{interconnect})$$

Memory Array Power Equations

$$P_{wordline} = \tfrac{1}{2} \times activity \times C_{wordline} \times freq \times V_{dd}^2$$

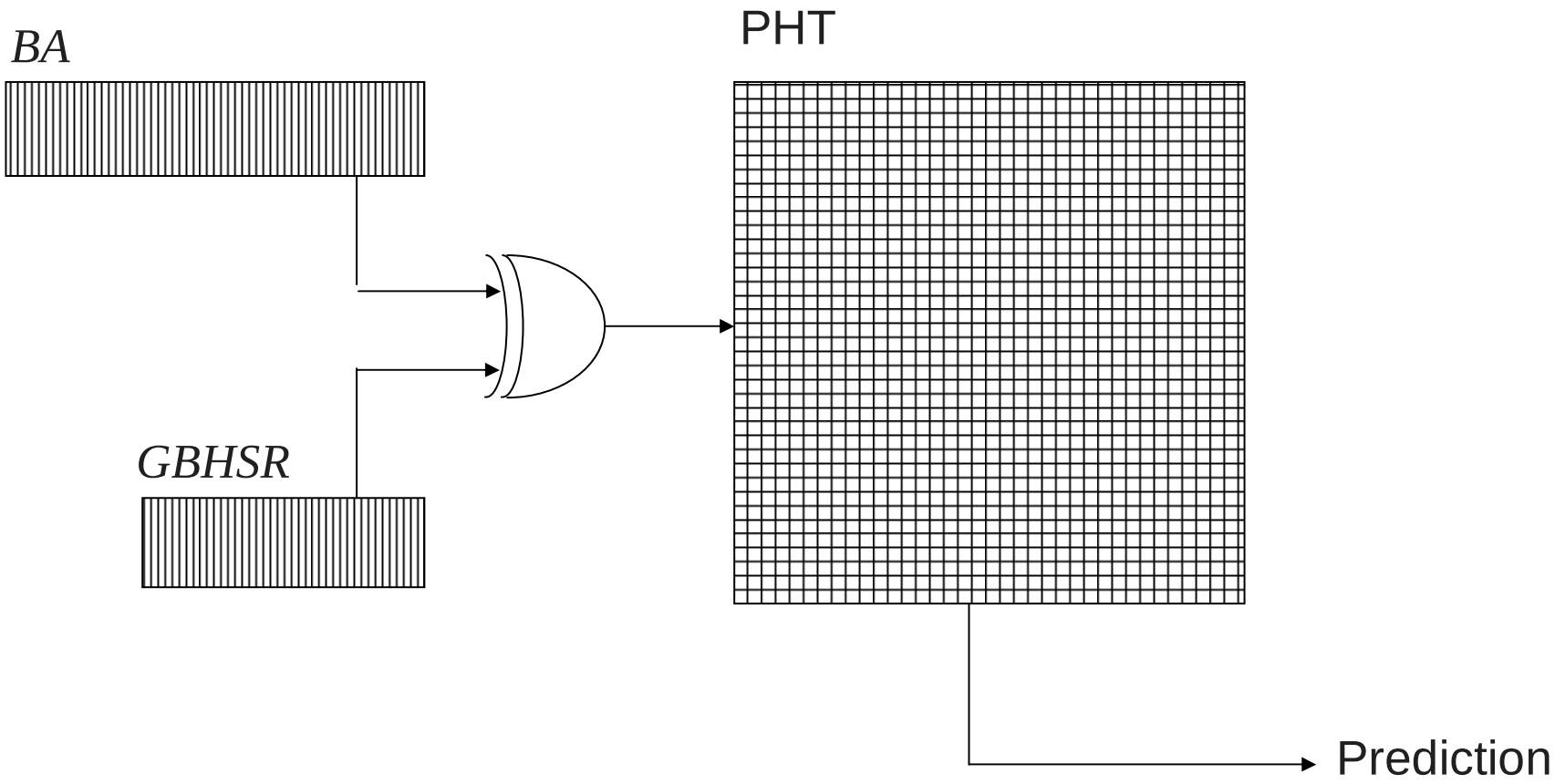
$$P_{bitlines} = \tfrac{1}{2} \times activity \times C_{bitlines} \times freq \times \Delta V \times V_{dd}$$

$$P_{sense\text{-}amps} = \tfrac{1}{2} \times activity \times C_{sense\text{-}amps} \times freq \times V_{dd}^2$$

$$P_{clock} = C_{clock} \times freq \times V_{dd}^2$$

$$P_{array\text{-}leakage} = stacking \times W \times I_{array\text{-}SD\text{-}leakage\text{-}per\text{-}W} \times V_{dd}$$

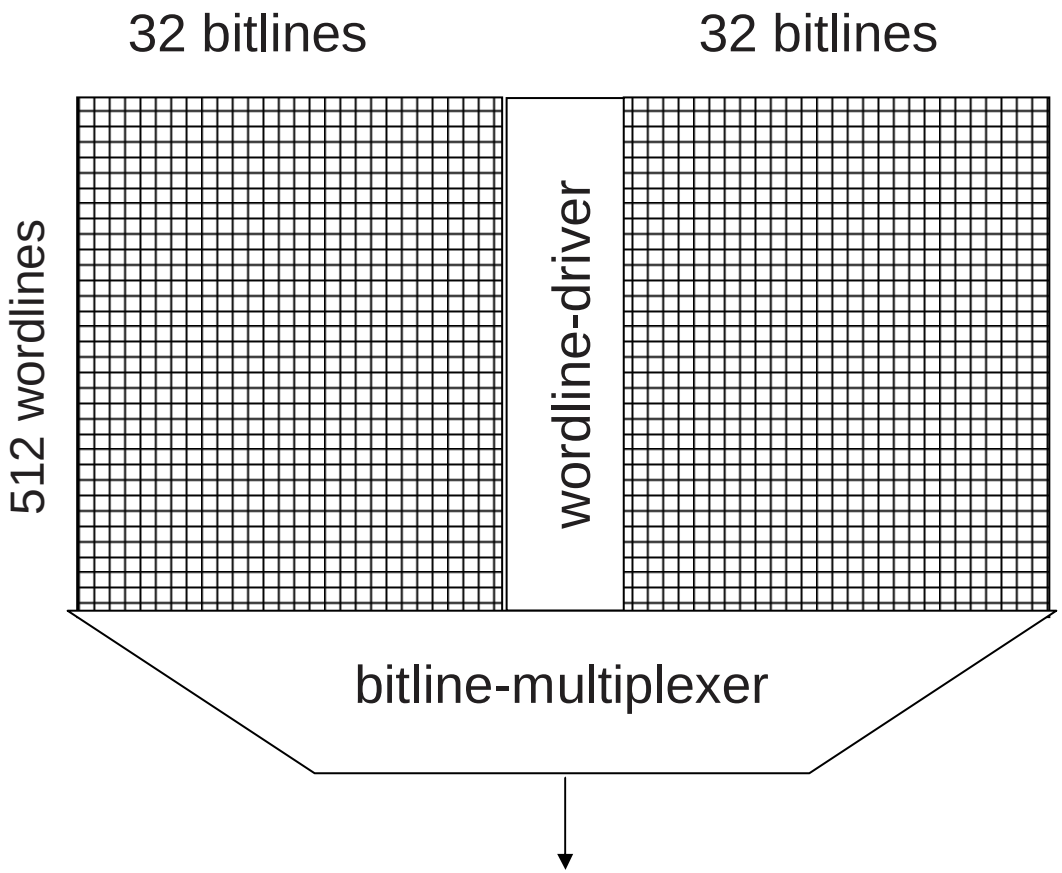
Baseline gshare Predictor



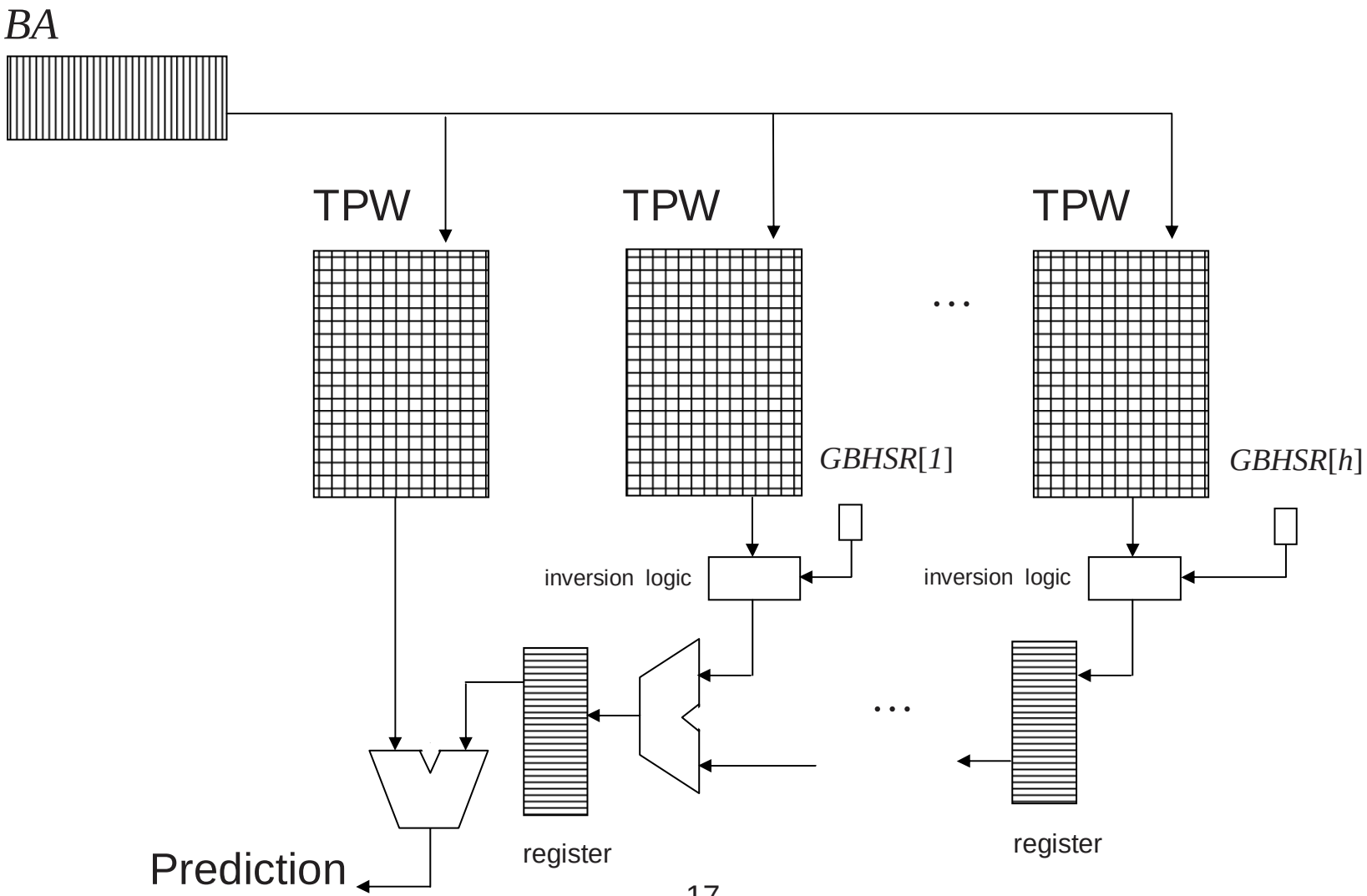
Baseline gshare Implementation

- Standard cells
  - XOR gates
  - Shift register
- Memory configuration
  - Total 16 KB
  - 1 x 4 x (4 KB sub arrays)

Baseline gshare 4 KB Sub-Array



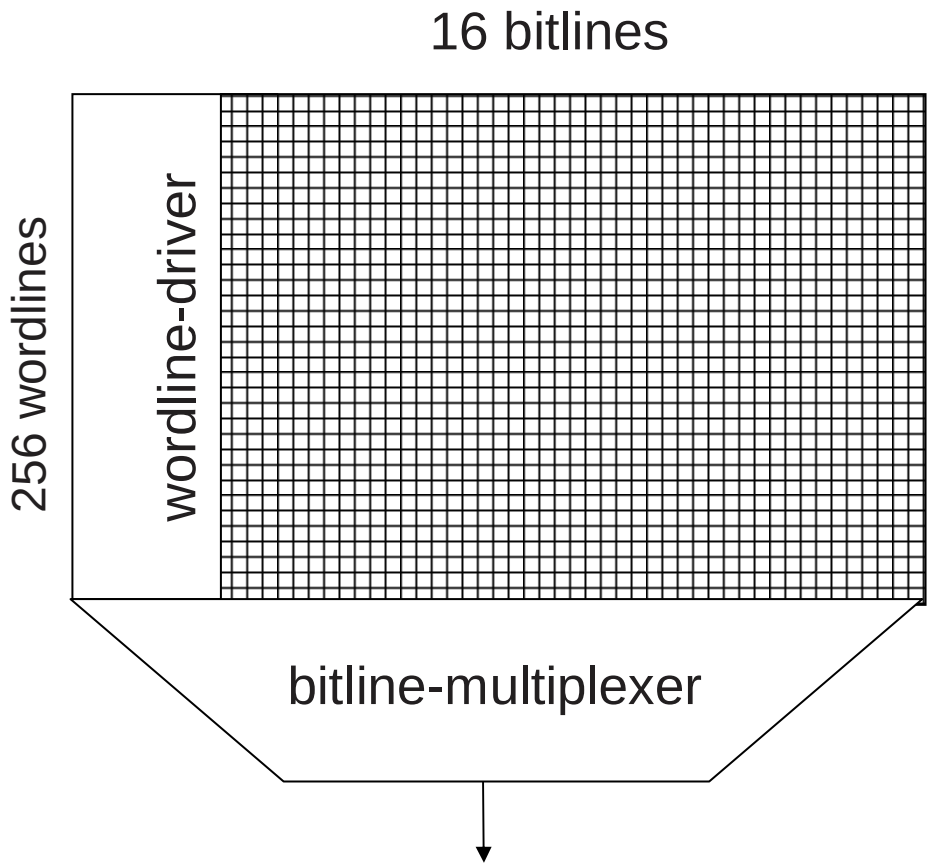
Path-Based Perceptron Predictor



Path-Based Perceptron Implementation

- Standard cells
  - Inversion logic
  - Shift register
  - Adders
  - Storage registers
- Memory configuration
  - Total ~16 KB
  - 31 x (512 B arrays)

512 B Array


16 bitlines
256 wordlines
wordline-driver
bitline-multiplexer

Area Comparison

Component	**gshare** (μm^2)	**Path-based Perceptron** (μm^2)
Standard Cells	515	42267
Memory Arrays	181507	191125
Total Area	182022	233392

28% increase

Prediction Delay Comparison

Component	**gshare** (ps)	**Path-based Perceptron** (ps)
Index Logic	55	0
Array Access	709	303
Prediction Logic	0	390
Total Prediction Delay	763	692

9% decrease

Power Comparison of Standard-Cell Implementation

Component	**gshare** (mW)	**Path-based Perceptron** (mW)
Gate Power	0.0169	1.1112
Gate Leakage	0.0001	0.0055
SD Leakage	0.0028	0.1754
Interconnect Power	0.0346	3.3196
Glitch Power	0.0077	0.6646
Standard-Cell Power	0.0622	5.2763

84x increase

Power Comparison of Memory-Array Implementation

Component	**gshare** (mW)	**Path-based Perceptron** (mW)
Wordline Power	0.0082	0.1274
Bitline Power	0.2354	1.8243
Sense Amp Power	0.0113	0.3515
Clock Power	0.3882	0.7521
Array Leakage Power	0.2123	0.2057
Memory Power	0.8554	3.2610

3x increase

Power Comparison of Final Results

Component	**gshare** (mW)	**Path-based Perceptron** (mW)
Standard-Cell Power	0.0622	5.2763
Memory Power	0.8554	3.2610
Total Power	0.9176	8.5373

8x increase

Results Compared to Baseline

- 28% area increase
- 9% prediction delay decrease
- Power
  - 84x standard cell power increase
  - 3x memory array power increase
  - 8x total power increase

Conclusion

- Advantages
  - Misprediction reduction of 13% to 33%
  - IPC performance improvement of 2% to 19%
- Disadvantages
  - Area increase of 28%
  - Total power increase of 8x
- Future work involves comparing total energy

References

[1] J.P. Shen and M.H. Lipasti, *Modern Processor Design: Fundamentals of Superscalar Processors*, New York: McGraw-Hill, 2005.

[2] T-Y. Yeh and Y.N. Patt, "Two-Level Adaptive Training Prediction," *International Symposium on Microarchitecture*, 1991.

[3] D.A. Jimenez and C. Lin, "Neural Methods for Dynamic Branch Prediction," *ACM Transactions on Computer Systems Vol. 20 No. 4*, pp. 369-397, November 2002.

[4] D.A. Jimenez, "Fast Path-Based Neural Branch Prediction," *IEEE Proceedings of the 36th International Symposium on Microarchitecture*, pp. 243-252, December 2003.

[5] A. Seznec, "Revisiting the Perceptron Predictor," pp. 1-21, May 2004.

[6] D. Tarjan and K. Skadron, "Merging Path and gshare Indexing in Perceptron Branch Prediction," *ACM Transactions on Architecture and Code Optimization Vol. 2 No. 3*, pp. 280-300, September 2005.

[7] D.A. Jimenez, "Piecewise Linear Branch Prediction," *Proceedings of the 32nd International Symposium on Computer Architecture*, June 2005.

[8] J. Tu, J. Chen, and L.K. John, "Hardware Efficient Piecewise Linear Branch Predictor," *20th International Conference on VLSI Design*, January 2007.

[9] D.A. Jimenez, "Idealized Piecewise Linear Branch Prediction," pp. 1-4.

[10] Y. Ninomiya and K. Abe, "A3PBP: A Path Traced Perceptron Branch Predictor Using Local History for Weight Selection," *Journal of Instruction-Level Parallelism*, pp. 1-18, May 2007.

[11] J.J. Friesenhahn, "High Performance Perceptron-Based Branch Prediction," *University of Texas at Austin Master's Report*, December 2007.

[12] M. McDermott, G. Gerosa, and S. Sullivan, "VLSI II Lecture Notes," Spring 2007.

[13] N.H.E. Weste and D. Harris, *CMOS VLSI Design*, Boston: Pearson Education, 2005.

[14] S.M. Kang and Y. Leblebici, *CMOS Digital Integrated Circuits*, New York: McGraw-Hill, 2003.

[15] A. Chandrakasan, W.J. Bowhill, and F. Fox, *Design of High-Performance Microprocessor Circuits*, IEEE Press, 2000.

[16] B. Toronyi, "Power Modeling," *University of Texas at Austin Master's Report*, December 2005.

Standard Cell Library

INV	NAND3	NOR3	AOI4	OAI3	MUX2	IMUX4	AND4	OR4
BUF	NAND4	NOR4	XOR2	OAI4	IMUX2	AND2	OR2	DFFR
NAND2	NOR2	AOI3	XNOR2	TRIINV	MUX4	AND3	OR3	

CELL	FANOUT 1 DELAY (ps)	FANOUT 2 DELAY (ps)	FANOUT 3 DELAY (ps)	FANOUT 4 DELAY (ps)	FANOUT 5 DELAY (ps)	FANOUT 6 DELAY (ps)	FANOUT 7 DELAY (ps)	FANOUT 8 DELAY (ps)
INV	24.55	33.44	40.63	46.82	52.33	57.66	63.09	68.5
BUF	38.87	43.92	48.58	53.02	57.39	61.76	67.1	72.58
NAND2	40.43	46.75	52.22	57.65	63.14	68.63	74.06	79.55
NAND3	54.44	60.21	65.94	71.64	77.31	82.91	88.52	94.07
NAND4	71.96	78.19	84.2	90.11	96.01	101.8	107.5	113.2
NOR2	41.06	47.85	53.95	59.96	65.76	71.68	77.47	83.27
NOR3	61.2	67.55	73.69	79.88	85.85	91.98	98.02	104
NOR4	87.34	93.95	100.6	106.9	113.3	119.6	125.8	132.1
AOI3	52.7	58.94	64.9	70.95	76.77	82.65	88.51	94.3
AOI4	62.95	69.16	75.08	81.1	86.99	92.88	98.76	104.6
XOR2	54.53	60.66	66.65	72.62	78.54	84.38	90.29	96.09
XNOR2	61.04	67.09	73.06	79.01	84.82	90.75	96.55	102.4
OAI3	52.08	58.31	64.33	70.37	76.27	82.18	88.06	93.9
OAI4	60.76	66.85	72.93	78.99	84.8	90.85	96.69	102.6
TRIINV	46.77	52.9	58.69	64.33	70.01	75.54	81.16	86.69
MUX2(sel)	34.75	39.09	44.05	48.15	51.7	55.29	58.88	62.49
MUX2(data)	26.15	30.67	34.65	38.28	41.64	44.78	47.74	50.51
IMUX2(sel)	43.88	49.32	54.69	60.27	65.88	71.61	77.34	83.13
IMUX2(data)	55.8	64.32	72.84	81.37	89.89	98.39	106.9	115.4
MUX4(sel)	67.34	75.22	83.11	91.01	98.93	106.9	114.8	122.9
MUX4(data)	53.48	59.43	65.66	72.06	78.73	85.57	92.65	99.87
IMUX4(sel)	83.96	98	111.4	124.6	137.6	150.5	163.4	176.2
IMUX4(data)	96.8	109.7	122.4	135.1	147.7	160.3	172.9	185.5
AND2	62.01	70.9	78.09	84.28	89.79	95.12	100.55	105.96
AND3	75.57	84.46	91.65	97.84	103.35	108.68	114.11	119.52
AND4	93.16	102.05	109.24	115.43	120.94	126.27	131.7	137.11
OR2	62.49	71.38	78.57	84.76	90.27	95.6	101.03	106.44
OR3	82.13	91.02	98.21	104.4	109.91	115.24	120.67	126.08
OR4	108.17	117.06	124.25	130.44	135.95	141.28	146.71	152.12

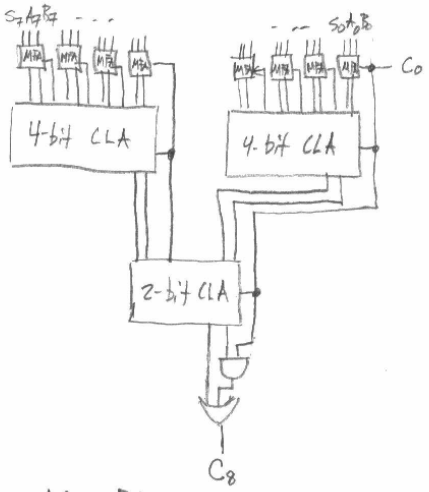
65 nm Standard Cell Characteristics

CELL	AREA (um²)	*utilization* factor	UTILIZED AREA (um²)	GATE CAP (fF)	*stacking* factor	*switching* factor
INV	1.2	80.0%	1.50	1	1.00	28%
BUF	2.4	80.0%	3.00	2	1.00	28%
NAND2	2.9	75.0%	3.87	2	0.50	12%
NAND3	4.1	70.0%	5.86	4	0.33	8%
NAND4	6.8	65.0%	10.46	6	0.25	6%
NOR2	2.9	75.0%	3.87	2	0.50	9%
NOR3	4.1	70.0%	5.86	5	0.33	7%
NOR4	6.8	65.0%	10.46	8	0.25	6%
AOI3	9.0	70.0%	12.86	4	0.50	6%
AOI4	11.0	65.0%	16.92	5	0.33	6%
XOR2	6.9	75.0%	9.20	7	0.33	9%
XNOR2	7.2	75.0%	9.60	7	0.25	9%
OAI3	13.0	70.0%	18.57	4	0.50	6%
OAI4	17.0	65.0%	26.15	5	0.50	6%
TRIINV	4.2	80.0%	5.25	4	0.50	15%
MUX2	4.2	70.0%	6.00	2	0.63	3%
IMUX2	6.6	70.0%	9.43	4	0.63	3%
MUX4	8.3	60.0%	13.83	6	0.50	3%
IMUX4	11.0	60.0%	18.33	9	0.50	3%
AND2	3.2	75.0%	4.27	3	0.75	12%
AND3	4.7	70.0%	6.71	4	0.67	9%
AND4	6.1	65.0%	9.38	6	0.63	7%
OR2	3.4	75.0%	4.53	4	0.75	9%
OR3	4.7	70.0%	6.71	5	0.67	9%
OR4	6.2	65.0%	9.54	5	0.62	9%
DFFR	8.1	55.0%	14.73	8	0.64	12%

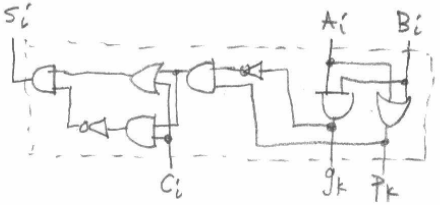
65 nm Technology Characteristics

Parameter	Value
V_{dd}	0.9 V
W_{min}	0.130 µm
NFET I_{Dsat}	1380 µA/µm
PFET I_{Dsat}	630 µA/µm
$I_{gate\text{-}leakage\text{-}per\text{-}W}$	15.6 nA/µm
$I_{SD\text{-}leakage\text{-}per\text{-}W}$	34 nA/µm
$I_{array\text{-}SD\text{-}leakage\text{-}per\text{-}W}$	20 nA/µm
C_{gate}	1.8 fF/µm
$C_{diffusion}$	0.05 fF/µm
Poly $R_{per\text{-}sq}$	35 ohms/sq
Metal $C_{per\text{-}l}$	0.23 fF/µm
Metal $R_{per\text{-}sq}$	0.0977 ohms/sq
Average Wire Length l	57 µm

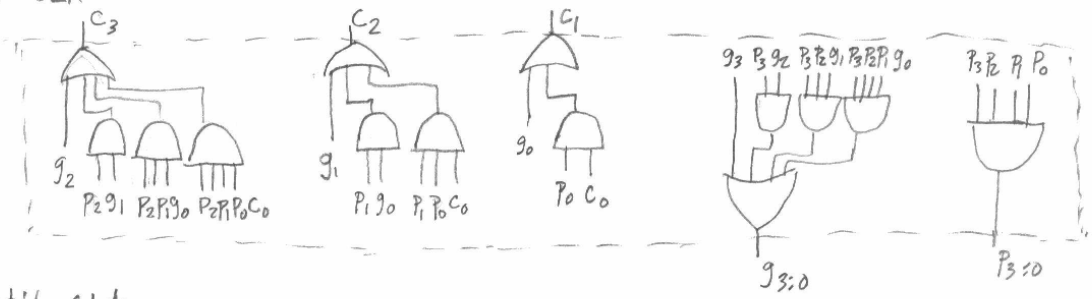
8-Bit Carry Look Ahead Adder Implementation



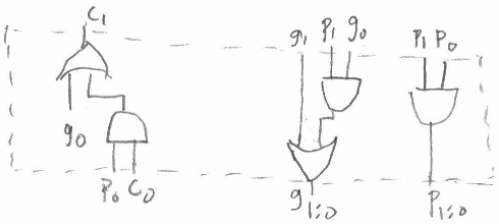
MFA



4-bit CLA



2-bit CLA



Hardware / Software Trade offs in Multicore Architectures

Steven A. Guccione
Cmpware, Inc.
Austin, TX (USA)
Steven.Guccione@cmpware.com

Abstract - For over three decades microprocessor architectures have used increasingly large numbers of transistors to improve performance. With the advent of multicore microprocessors, it is not obvious how best to use transistor budgets. The choice is between a small number of complex cores or a larger number of simpler cores. This paper investigates the specific trade offs between using a hardware floating point unit as opposed to software emulation of floating point using a larger number of cores.

I. Introduction

The history of microprocessor architecture has been one of making use of ever increasing numbers of transistors to provide ever increasing levels of performance. Over the course of several decades the number of transistors available to microprocessor architects could reliably be expected to double approximately every 18 months.

While the number of available transistors increased by several orders of magnitude over this period, the performance gains achieved were much smaller. New transistors added to maturing processor architectures achieved smaller and smaller incremental performance gains over time until finally large numbers of transistors did little to increase performance for most applications.

Near the beginning of the new millennium a significant change occurred in mainstream microprocessor architecture. New transistor budgets were no longer used to increase performance of individual microprocessors; instead multiple processor cores were implemented on a single device. This change occurred for a variety of reasons, but the effect on performance was dramatic. Suddenly doubling the number of transistors could, at least in theory, double the performance of the device.

This was a situation that had not existed for decades. While mainstream microprocessor architects quickly embraced this multicore approach to utilizing new transistor budgets, the process has typically made use of existing modern processor architectures for the individual cores. Since it is clear that the last several generations of architectural enhancements added little in performance to these single core architectures, it seems obvious that re-deploying these transistors to increase the number of cores, while reducing the size of the cores themselves, should result in an increase in overall performance. What is not clear is how far back should the clock be turned in redeploying transistors in this manner.

This paper will focus on a very early hardware addition to the basic microprocessor architecture: the floating point unit. Issues concerning the redeployment of floating point unit transistors to increase the number of cores in a multicore device is explored. Software emulation of floating point operation as well as similar techniques are examined.

II. Multicore Architectures

All modern desktop and server microprocessor devices today are multicore devices. It is expected that this trend will continue for some time, with the number of cores continuing to increase, approximately doubling with every new generation of microprocessor.

This shift to multiple core microprocessors has occurred for a variety of reasons, which are interrelated and happened to occur at approximately the same time in history. First, the increase in clock speed and the continued shrinking in transistors resulted in a situation where power consumption reached and even exceeded 100W per device. The trend in power consumption for high performance microprocessors was perhaps the most visible and urgent reason for the move to multicore. With multiple cores, the core voltages and clock speed could be reduced, resulting in significant power savings. Two lower power, but lower performing, cores could operate at a combined performance level above that of a more powerful single core.

A second reason for the move to multicore was the plateauing of microprocessor clock speeds. While related to power consumption, the ability of modern CPU architectures to support clock speed above approximately 4 GHz was also a function of the

underlying silicon technology. In order to continue to provide performance gains in new generations of microprocessors, other techniques besides faster clocks were going to have to be employes. Multicore was a convenient method of boosting total processor device performance without having to raise clock speeds.

Finally, one significant problem with modern microprocessor design was the cost. Design and verification of a large microprocessor was beginning to exceed the available resources of even the largest corporations. Multicore also addressed this problem by reusing a single smaller core multiple times in the same design.

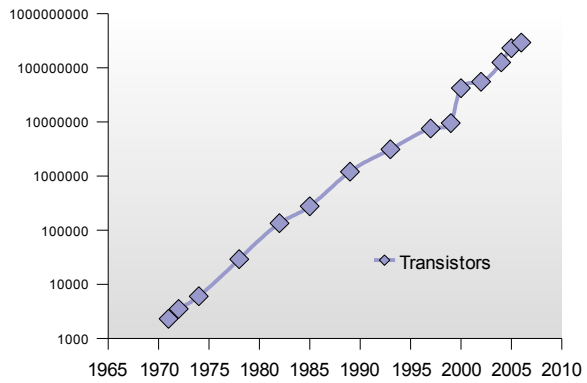


Figure 1: Transistor budgets for microprocessors.

As Figure 1 indicates the number of transistors available to microprocessor architects continues to grow, even as the ability to produce traditional single core microprocessor devices became less and less feasible. After approximately the year 2000, multicore became the sole method of continuing to keep microprocessor performance on the established 30 year track.

III. FPGA CPUs

It is clear that the later generations of microprocessor devices used their transistor budgets to provide fairly minimal increases in performance. In fact, much of the performance gains in modern microprocessor devices came from increases in clock speed associated with smaller transistors, not from the use of larger numbers of these smaller transistors. While these performance gains were important in the highly competitive marketplace, the equation changed significantly with the move to multicore.

While in early microprocessors, a doubling in the number of transistors could result in a doubling of performance, only very small gains could be expected in recent generations. But with mucticore promising a doubling in performance for a doubling in transistor count, a re-evaluation of existing multiprocessor architectures may be in order. Such an investigation is potentially complicated due to the long history, lack of published information and potential effects of rapidly changing technology.

Fortunately, Field Programmable Gate Array (FPGA) technology presents an environment remarkably similar to the world of microprocessor design circa 1985. Simple 32-bit RISC processors are beginning to find their way in common usage and hardware floating point units are just beginning to be explored.

Component	*LUTs*
CPU	800
FP Add	1,312
FP Multiply	1,380
FP Mac	2,772
FPU support*	850

Table 1: FPGA core sizes.

Table 1 gives a list of microprocessor core components for FPGAs measured in the number of 4-input Look Up Tables (LUTs) used in each component. LUTs are the basic building block used by FPGAs to construct circuits. This provides a very good measure of the relative circuit size of these elements.

While there is currently no information available on a complete floating point unit Arithmetic and Logical Unit (ALU) in an FPGA, these numbers indicate that a Multiply-Accumulate unit (MAC) is over three times the size of the CPU core itself. In a multicore environment, the choice of hardware for such a system would be a single CPU with a single MAC or approximately five CPUs. In fact, the choice is likely to involve a larger number of cores. The choice may be between 200 CPU plus MAC cores and 1,000 CPU cores.

IV. Floating Point Performance

With a hardware design choice of a single CPU plus a single FPU versus five CPUs, some measurement of performance of these competing approaches is required. In this example, the goal is to first give an estimate of hardware floating point performance versus software emulation of floating point. Such a system may offer a wide variety of size versus performance points. While floating point hardware often takes multiple cycles and has the ability to pipeline certain operations, it is conservatively assumed that each floating point operation is independent and takes place in a single

cycle. These two approaches were simulated on the Cmpware CMP-DK multicore simulator [8]. The results of these simulations are shown in Table 2.

```
int main(int argc, char *argv[]) {
      int  i;
   float  a = 10.5;
   float  b = 3.25;
   float  c = 0.0;

   for (i=0; i<1000; i++)
      c = c + (a * b);

   }  /* end main() */
```

Figure 2: The multiply-accumulate code.

Table 2 indicates that unoptimized floating point software emulation is approximately 28 times slower than the floating point hardware.

While this result is disappointing, attempts to use the compiler to optimize the code results in a factor of two improvement in performance over the unoptimized floating point emulation. Unfortunately it also results in a factor of three increase in performance for the floating point hardware architecture. Overall, the optimized code for the software emulation of the floating point operations is 41 times slower than the hardware. These results are also shown in the graph in Figure 3.

	Instr. Executed	*Instr. Executed (-O3)*
FP Hardware	12,024	4,015
FP SW (unpacked)	83,026	15,073
FP Software	336,254	165,010

Table 2: Floating point hardware performance versus floating point emulation performance.

Clearly even having five CPU cores operating in parallel is no match for a factor of 41 increase in performance for floating point hardware. These naive results are, however, somewhat unexpected. Clearly floating point arithmetic, particularly a multiply-accumulate operation, is not over 40 times more efficient than a software implementation. So investigation into the underlying implementation is in order.

Both implementations used the same exact source code shown in Figure 2, which was a simple loop repeated 1,000 times containing a multiply-accumulate operation. This is a simple benchmark, but one that contains a high ratio of floating point operations, in order to draw a fair comparison. Both were compiled with the Gnu gcc compiler version 4.2.2 using the standard libraries.

Upon closer investigation, it was revealed that the floating point emulation libraries perform a substantial amount of work keeping the data in a rigid floating point format. This format is the standard IEEE format with a single bit for the sign, eight bits for the exponent and the remaining 23 bits for the mantissa. A final extra leading mantissa bit is always assumed to be '1'.

These four fields are 'unpacked' before each floating point operation, giving the integer arithmetic units access to the individual fields for the calculations. In addition, after each operation is complete, the data is 'packed' back into the IEEE format. This pack / unpack combination is performed on each data item for each floating point calculation performed.

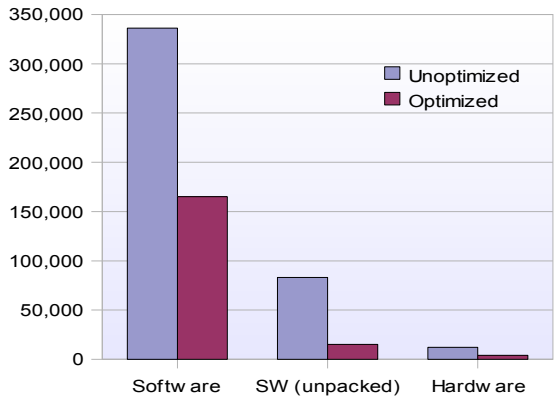


Figure 3: Floating point performance.

This packing and unpacking of data adds nothing to the computation and is used simply to keep data in a format favored by the floating point hardware, which is not in use. An alternative is to unpack the data once at the beginning of the calculation and pack it back again into the standard format when it is required by other hardware or routines such as output.

The middle row in Table 2 gives the results for this 'unpacked' calculation. Unlike the naive implementation from the standard libraries, this version is approximately 7 times slower than the floating point hardware implementation in the unoptimized version, and 3.75 times slower than the floating point hardware implementation in the optimized version.

This result indicates that over 95% of the time in the emulation routines is spent packing and unpacking data.

Unfortunately, this type of code is very difficult for compilers to automatically optimize and currently has to be eliminated manually. However, the ability to achieve only a factor of 3.75 slowdown in a multicore environment that has five times the number of cores appears to support the idea that even for floating point hardware, more cores represent a higher performance implementation than special purpose hardware.

V. Conclusions

As more and more desktop, server and embedded processors move to multicore architectures, a more thorough study of multicore architectural issues should be undertaken. Most significant are the combined questions of how many cores should a design use and how large should those cores be.

The CPU cores used in today's designs appear to arise more from convenience than any other measurable factor. The most recent generation of cores is typically replicated in a device with little indication of other alternatives being explored. This is problematic because the microprocessor core of today has undergone a very long process of development involving many orders of magnitude increase in size and performance. The changes to these cores along the way had a single goal: to improve *serial performance* in a *single* core environment.

This focus on improving serial performance was understandable for single core devices. It preserved the programming model and kept legacy software operating on subsequent generations of devices. But once the break with uniprocessing has been made and the programming model broken, many of the architectural decisions made to create these cores come into question.

In this paper we explored some very early decisions in microprocessor history: the addition of a floating point unit. This occurred somewhere in the middle of this history of microprocessor development, approximately halfway between the original Intel 4004 and today's quad core devices.

Close examination indicates that even for simple code with very high proportions of floating point operations, the use of special purpose hardware for floating point operations may not be warranted. The ability to perform such operations in software using simpler integer operations appears to be more efficient in a multicore environment.

Of course, multicore architectures are new and there is much to be learned. In particular software issues are more important in multicore devices than in traditional single core architectures. The ability to parallelize software across a large number of cores will be the key to successfully exploiting the potential high performance of multicore devices. As work on tools, algorithms, and programming techniques progress, the parameters which define emerging multicore architectures will become more sharply defined.

But the situation is somewhat unusual. Multicore devices of many types are already in production, even without basic software support. A large swath of the semiconductor industry is now dependent on significant breakthroughs in application level software which already appear to be overdue.

REFERENCES

[1] “The Future of Microprocessors”, David Patterson, U. California at Berkeley, June 2001. http://www.cs.berkeley.edu/~pattrsn/talks/NAE.ppt

[2] Kunle Olukotoun and Lance Hammond, "The Future of Microprocessors", ACM Queue, Volume 3, Number 7, September 2005, pages 26-34.

[3] Michael J. Beauchamp, Scott Hauck, Keith D. Underwood, K. Scott Hemmert, "Architectural Modifications to Improve Floating-Point Unit Efficiency in FPGAs", International Conference on Field Programmable Logic and Applications (FPL) , 2006. Aug. 2006 Pages 1 - 6.

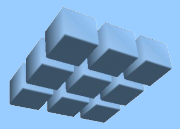
[4] Michael J. Beauchamp, Scott Hauck, Keith D. Underwood, K. Scott Hemmert, "Embedded floating-point units in FPGAs", Proceedings of the 2006 ACM/SIGDA 14th International Symposium on Field Programmable Gate Arrays (FPGA), Monterey, California, February 22-24, 2006, pages: 12 - 20.

[5] Krste Asanovic, Ras Bodik, Jim Demmel, John Kubiatowicz, Kurt Keutzer, Edward Lee, George Necula, Dave Patterson, Koushik Sen, John Shalf, John Wawrzynek, and Kathy Yelick, "The Landscape of Parallel Computing Research: The View from Berkeley 2.0", Manycore Computing Workshop, June 2007, http://science.officeisp.net/ManycoreComputingWorkshop07/Presentations/David%20Patterson.pdf

[6] Rey Archide, "The Microblaze v5.0 Soft-Processor Core: Optimized for Performance", Xilinx Embedded Magazine, November 2006, pages 18 - 21.

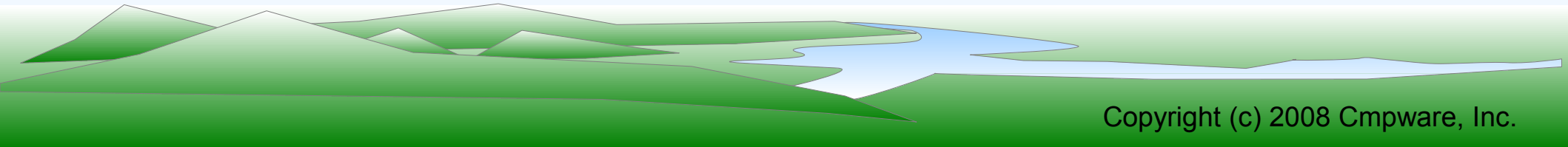
[7] "Intel Consumer Desktop PC Microprocessor History Timeline, http://www.intel.com/pressroom/kits/core2duo/pdf/microprocessor_timeline.pdf

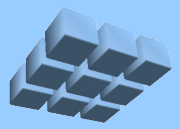
[8] Cmpware, Inc. http://www.cmpware.com/, 2008.



Hardware / Software Tradeoffs in Multicore Architectures

Steven A. Guccione
Cmpware, Inc.

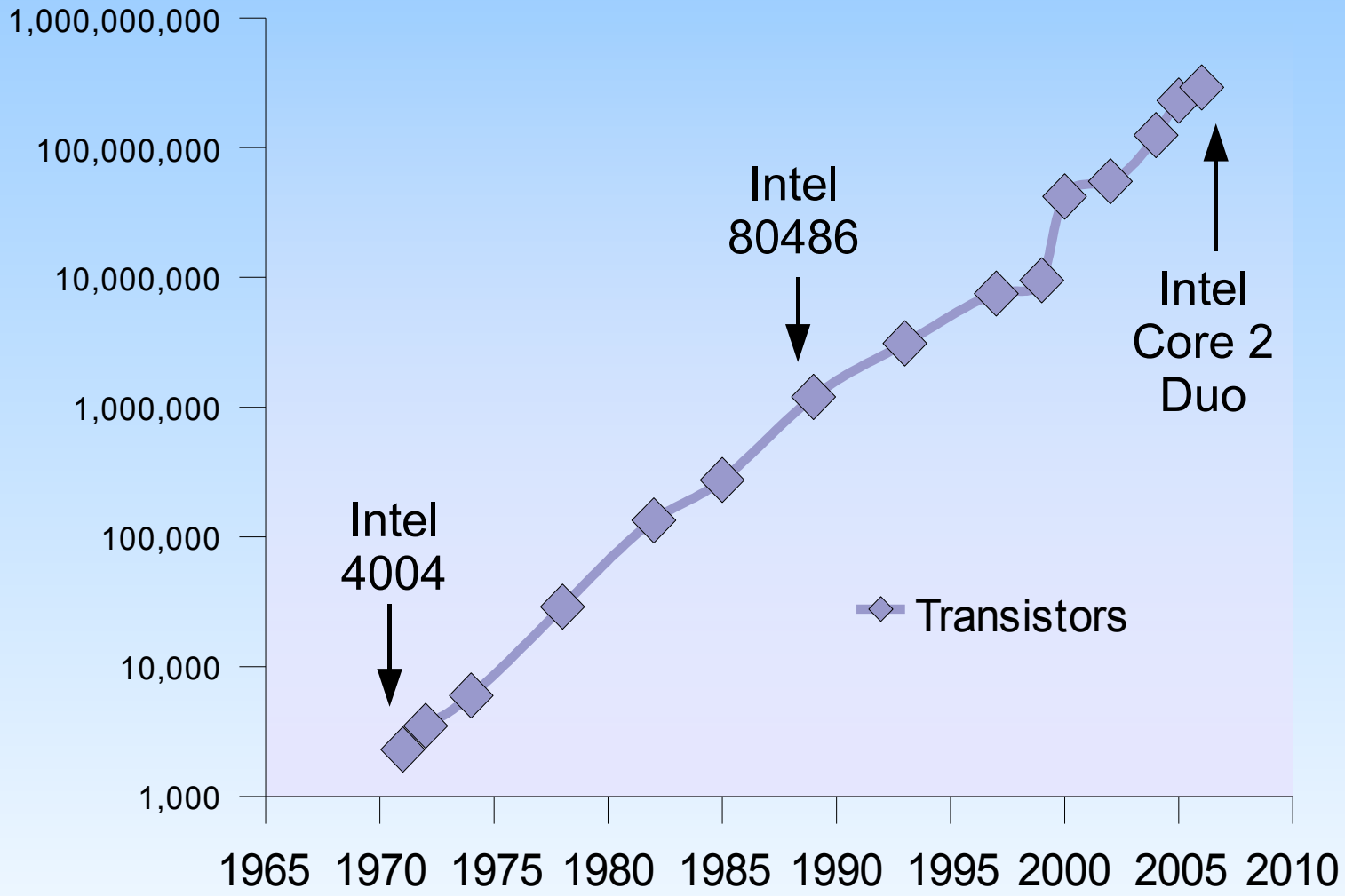


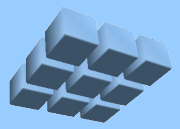


CPU Transistor Budgets

- Approx. 1B transistors available (2008)
- 30+ years of *Moore's Law*
  - 8 -> 16 –> 32 –> 64 bit CPUs
  - Floating Point
  - Superscalar
  - Caches, buffers, caches and more caches
- Decreasing 'ROI' for new transistors
  - Little performance boost in last generation of single core CPUs

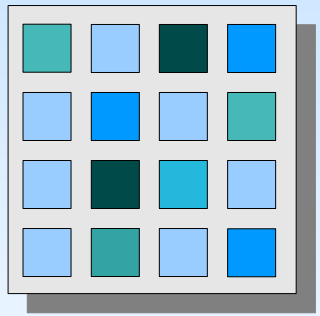
Transistors Per Year

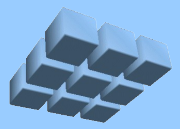




CPUs in the New Millennium

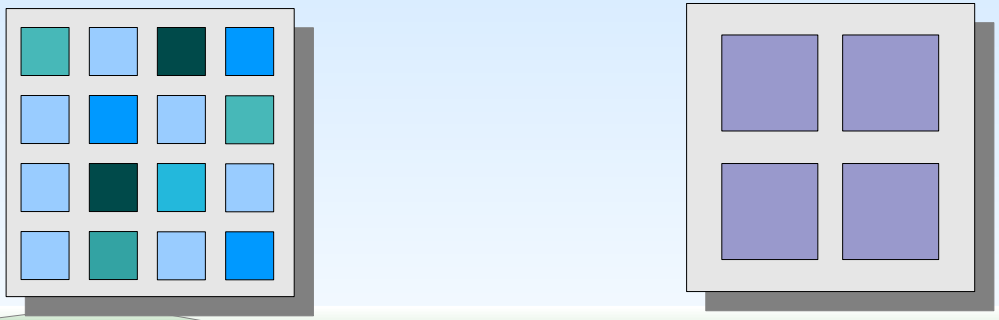
- Single core architectures
  - Clock speeds plateauing (4 GHz)
  - Heat dissipation problems (100+ W)
  - Design costs ($$$)
- Multicore architectures
  - Increase performance
  - Reduce power consumption
  - Simplify design



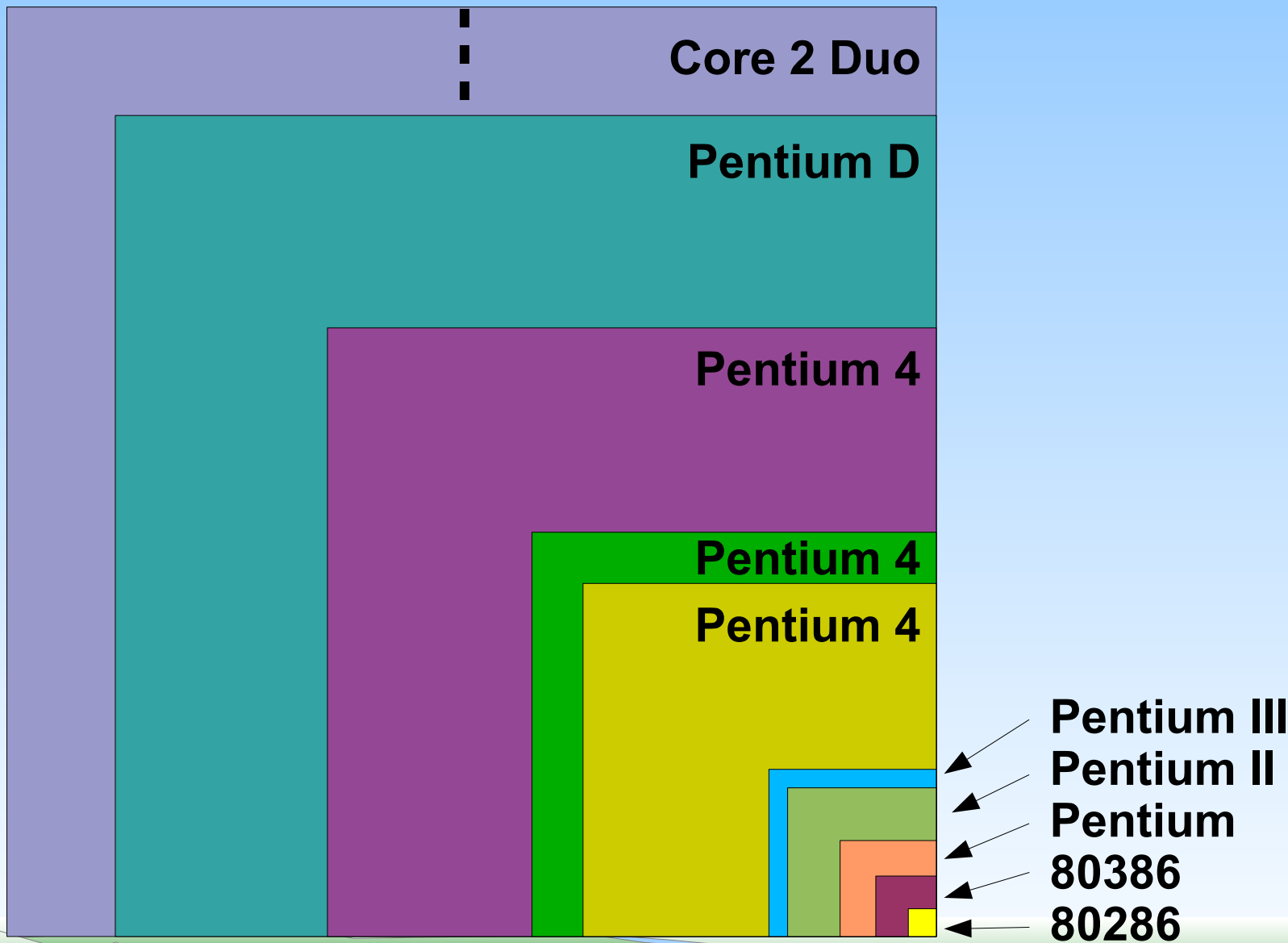


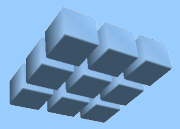
Multicore Architecture

- A new set of design parameters:
  - Core size
  - Number of cores
  - Core functionality
  - Core to core communcation
- **Q**: *Do you want dual Pentium 4 or 1,000+ 80386 cores?*



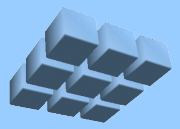
Relative CPU Sizes





Massively Multicore

- Smaller cores provide higher total MIPS
  - Using thousands of CPUs in parallel challenging
  - Software and tools issues
  - Memory and IO bandwidth questions
  - Application dependent
- 1,000 cores breaks software -- *but so does dual core*
- **Q**: How small should a core be?



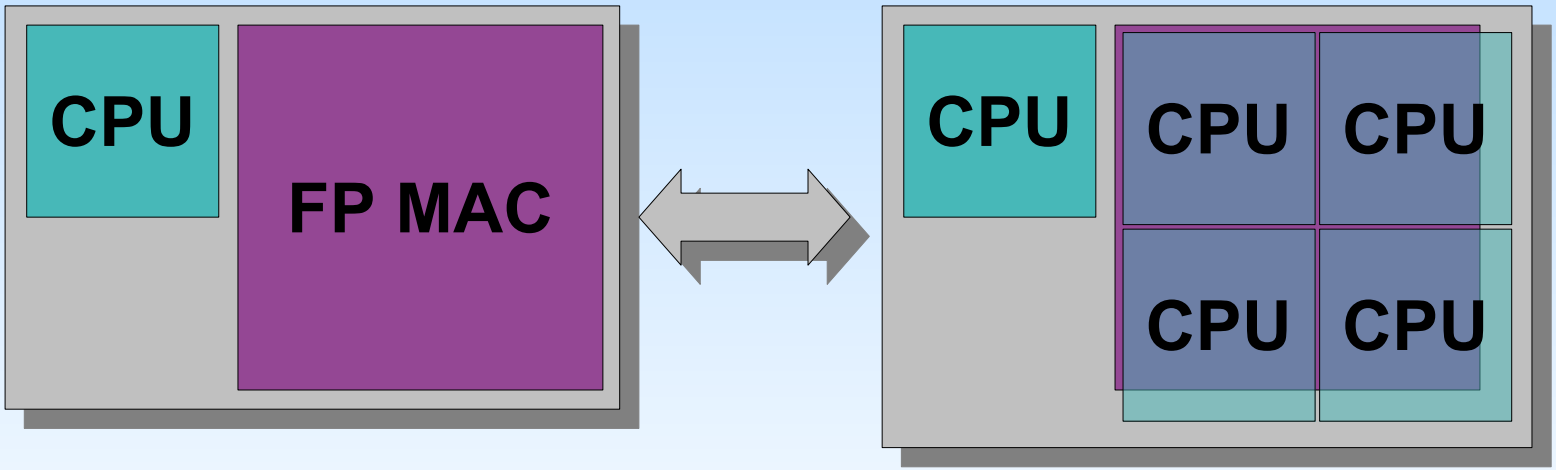
FPGA CPUs

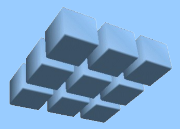
- Currently simple 1980s-style RISC CPUs
- Beginning to implement floating point
- Some multicore experimentation

Component	*LUTs*
CPU	800
FP Add	1,312
FP Multiply	1,380
FP Mac	2,772
FPU support*	850

Relative FPGA Core Sizes

- FP MAC almost 4x larger than CPU
- **Q**: CPU + MAC or 5x CPU?
- **Q**: *50x CPU + MAC or 250x CPU?*





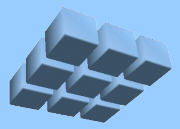
SW vs. HW FP

- Simple floating point code
- Run on *Cmpware PowerPC* simulator
- HW FP instructions vs. soft FP emulation

```
int main(int argc, char *argv[]) {
    int  i;
    float  a = 10.5;
    float  b = 3.25;
    float  c = 0.0;

    for (i=0; i<1000; i++)
       c = c + (a * b);

    }  /* end main() */
```

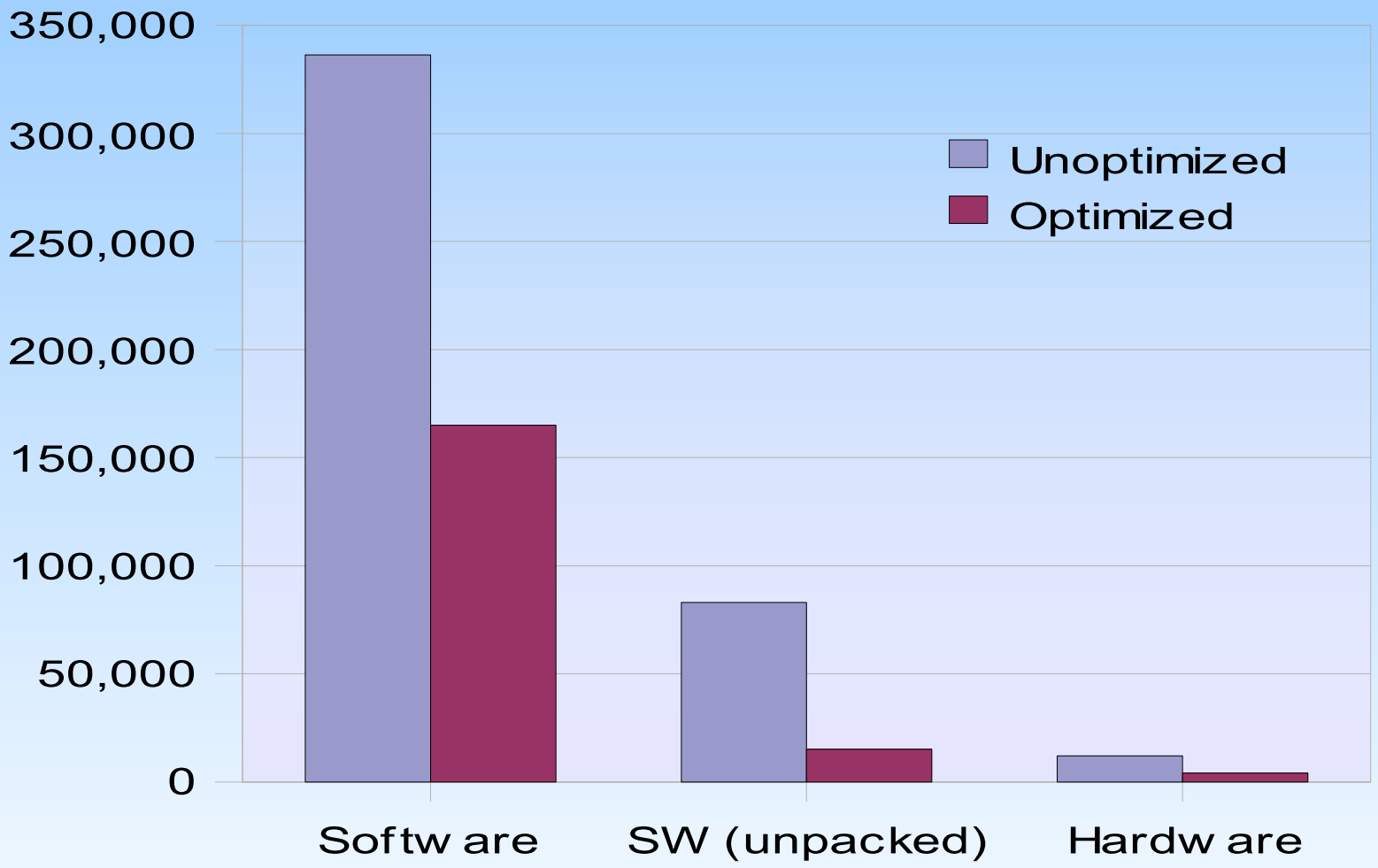


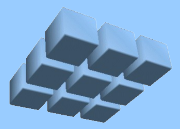
HW vs. SW Floating Point

- SW emulation 30x – 40x slower than HW
- ... but most of the time spent packing / unpacking numbers into floating point format
- 'Unpacked' FP only 3.75x slower than HW

	Instructions Executed	***Instructions Executed (-O3)***
FP Hardware	12,024	4,015
FP SW (unpacked)	83,026	15,073
FP Software	336,254	165,010

HW vs. SW Floating Point

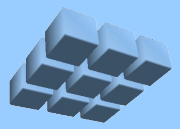




FP HW in Multicore

- SW FP 3.75x slower FP execution than HW
- ... but permits 5x CPU cores
- Opens up new optimization opportunities
- Performance available for non-floating point applications
- Other applications likely to have an even lower mix of FP / non-FP instructions

==> *Software FP 'wins' in multicore*



Conclusions

- CPU + FPU 5x size of CPU
- FPU software emulation optimized to 3.75x speed of HW FPU
- FP software beats FP hardware in multicore
  - Depends on instruction mix
  - Depends on ability to parallelize application
  - Depends on I/O and system parameters
- More simple cores favored
- SW beats special purpose HW in multicore

Workload Slicing for Detailed Pre-silicon Power Estimation

Hassan Al-Sukhni, James C. Holt, David Lindberg, Michele Reese
{Hassan.Alsukhni, Jim.Holt, David.Lindberg, Michele.Reese} @freescale.com
Freescale Semiconductor, Inc.
7700 W. Parmer Ln, Austin, TX 78729

Abstract

Detailed pre-silicon analysis of new features incorporated in high performance microprocessors often requires the use of RTL gate-level models because accurate higher-level models of such features are not available. While the results of such studies can be very valuable, this approach is both slow and complex, resulting in extreme constraints on the maximum size of workloads that can be studied. These issues are exacerbated when the object of a study are specific new features of a larger design, requiring that instructions in the workload target the features under evaluation. Although micro-benchmarks can sometimes be used for this purpose, it is also desirable to examine real-world workloads. Thus, a methodology for using detailed RTL models with realistic workloads requires workload sampling techniques. Existing workload sampling techniques do not completely satisfy the requirements of this methodology because (1) they are restricted to using predefined workload metrics to identify representative samples of the workload, but these metrics may not encompass the requirements of a highly-targeted study, and (2) small samples do not sufficiently warm up architectural and micro-architectural state such that measured results will mirror those of the sample in the context of the full workload. For these reasons, a more flexible and complete technique is required for extraction of representative samples of workloads that exercise specific features of the microprocessor. In this paper, we present Workload Slicing Flow as a set of tools that enables the creation of representative workload slices satisfying a set of metrics and constraints while providing sufficient warmup. The use of the flow is illustrated by selecting slices for power-characterization of the floating-point unit of a research microprocessor. The selected slices are 5% of the size of statistical samples of the original workload, and result in power estimates within 4% of the full workload power estimates..

1. Introduction

Power consumption is becoming a first-order design concern of microprocessors. As such, power estimation is needed during the early design stages to evaluate alternative design options, and during later design stages to validate that the power targets and constraints have been met for a specific application domain. This concern with microprocessor power is a direct result of the increasing demand for lowering the power consumption of systems due to several factors that include: (1) reliability (2) cost of packaging and cooling, and (3) longer operation-time-between-charges of battery-operated personal computing devices and wireless communication systems [1].

Models with different levels of abstraction are used to characterize microprocessors during the design cycle. Low-level models of the microprocessor are often used to characterize power consumption of new features because of the lack of higher-level models for the new features, or because higher-level models do not provide sufficient accuracy. Figure 1 illustrates typical abstraction levels at which power consumption can be characterized. The power characterization tools for low-level models of complex designs such as high-performance microprocessors are very costly to run in terms of both time and space. Therefore, the stimuli that can be used at these levels are highly targeted to specific microprocessor features and are subject to severe size and instruction mix constraints [2]. This paper presents *workload slicing* as a technique to generate effective workloads to meet these constraints.

Workload slicing uses statistical sampling to find large samples representative of execution phases of applications (*segments)*, and then and then employs user-specified workload metrics and constraints to each segment to produce a set of short representative samples of the segment (*slices)*, which collectively represent the segment for a given response *metric* (for example floating point unit power consumption, see Section 2). A slice includes initial state information, such as the state of the general purpose registers, and sufficient memory initialization of code and data sections. Furthermore, an effective slice will comprise considerations for both architecture and micro-architecture warm-up considerations.

Several techniques for reducing workload size have been proposed, including control flow modeling [3], benchmark subsetting [4], and synthetic workload

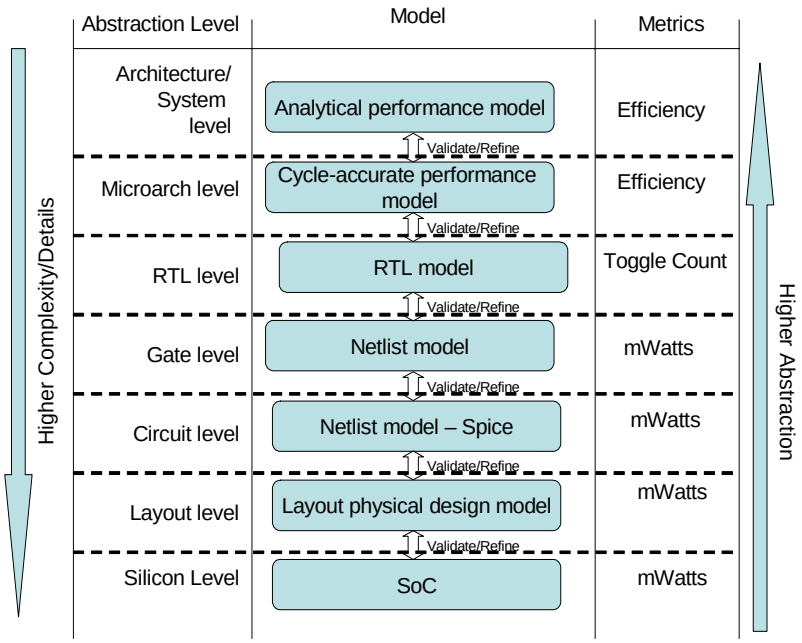


Figure 1 Power estimation abstraction levels

extraction [5]. Unfortunately, these approaches are not sufficient for slicing because they result in workloads that are still too large for this purpose, or because they require special simulators or hardware-generated traces. Such approaches could be used to provide a set of initial workloads for further reduction, but they cannot eliminate the need for slicing. Two exisiting techniques that can be used to generate small-sized stimulus for power characterizations are micro-benchmarks, and benchmark sampling [2, 6-8].

Micro-benchmarks are good for targeted characterization and testing of specific functionality of new features [9]. However, micro-benchmarks do not scale well to higher-level models because they usually are not sufficiently representative of workloads of interest, and hence cannot confidently be used forestimating power consumption of large interesting applications.

Sampling techniques have been successfully used in architectural and micro-architectural design exploration studies [10]. Tools like SimPoint [7] use basic-block execution counts to identify phases of program execution, and after that, use a clustering algorithm to find a set of representative samples that can be executed in lieu of the program. In contrast, ad-hoc sampling techniques such as fast-forwarding, reduced input sets, etc. have been demonstrated to be less effective in producing samples that are representative of their programs [4, 8, 11].

Existing sampling techniques suffer two main problems for power characterization and estimation using low-level models. The first is that the available sampling tools do not offer flexibility in specifying workload-oriented metrics of interest that the samples should exhibit. For example, characterizing the power consumption of a floating point unit in a microprocessor requires workloads that include floating point instructions. Samples of a program from regions of code that do not include such instructions may not be of interest for this characterization. A sampling tool that provides flexibility in specifying the sampling metrics and constraints is necessary for detailed characterization of specific features of a hardware design.

The second problem with existing statistical sampling techniques is that they assume that the sample size is large enough to allow the initial state of the micro-architecture to be ignored; as a result these techniques do not adequately account for warm-up of samples which are created with strict size constraints [3, 5, 8, 12]. Our own experiments confirm that the micro-architecture state can influence the characterization to a large degree, considering the workload length constraint on stimulus used for power characterization with low-level models (as illustrated in Section 3).

The next section explains the slicing approach and slicing flow. Section 3 presents experiments to illustrate the micro-architecture warm-up requirements for workload slicing. Finally, Section 4 summarizes this paper, and gives future work suggestions and ideas.

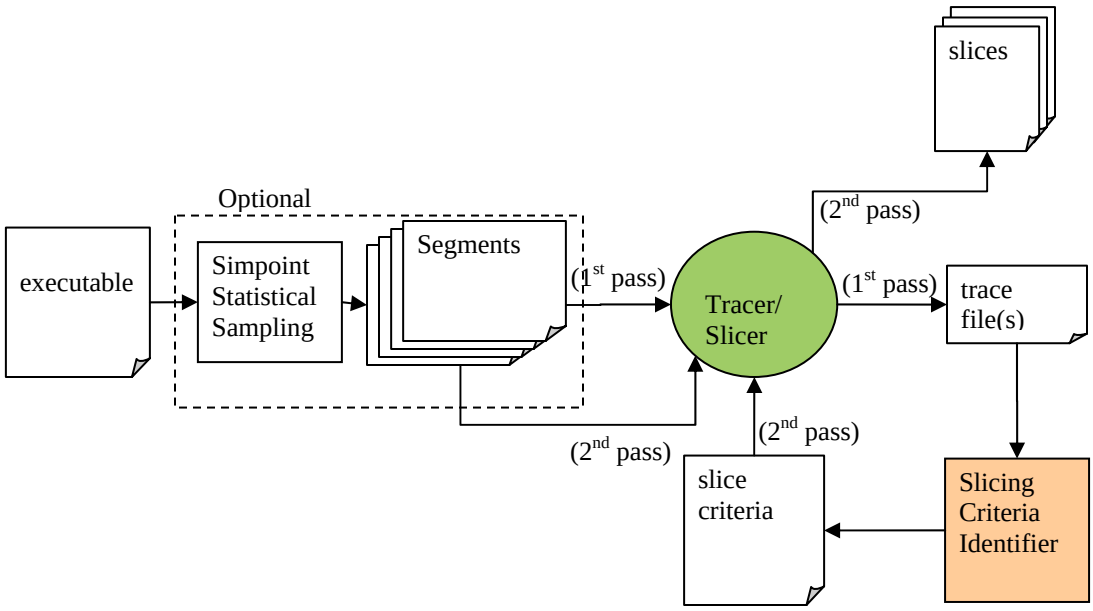


Figure 2 Workload slicing flow.

2. Approach

Workload slicing is carried out by processing the workload in two passes. Figure 2 illustrates the workload slicing flow, and the tools used in the flow. The first pass identifies slicing criteria, while the second pass produces the slices. In the first pass, the workload executable is traced using an Instruction Set Simulator (ISS) controlled by the Tracer/Slicer shown in Figure 2. The ISS has been augmented with simplistic models of certain micro-architecture elements. Thus, in addition to tracing the instructions execution of the workload, the Tracer uses the augmented functional model to annotate the resulting traces with additional information that is used by other tools of the flow (for example cache hit/miss information). The traces from this pass are then fed to a tool referred to in Figure 2 as Slicing Criteria Identifier (SCI), which identifies slicing points based on user-supplied workload metrics and constraints. In the second pass, the workload is run again to generate slices based on the identified criteria. This is done by the Slicer, which can also generate micro-architecture warm-up prelude if needed (discussed in Section 2.3).

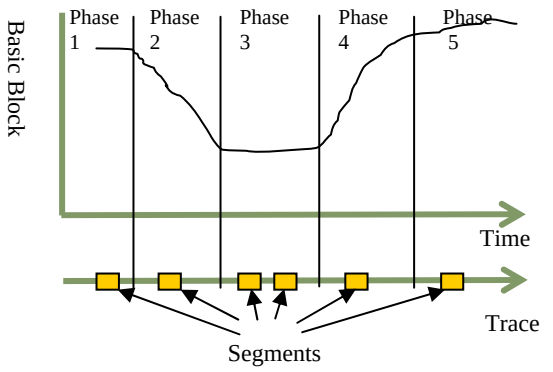
Workload slicing comprises three main tasks: segmenting, slicing, and micro-architecture warm-up prelude generation. These tasks are discussed in the following sections.

2.1. Segmenting

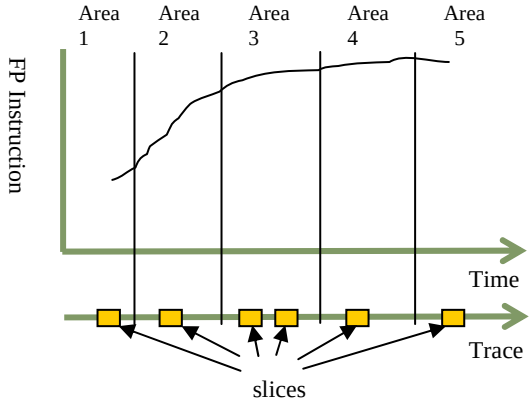
The first task in the slicing process is statistical sampling, illustrated in Figure 3(a). Statistical sampling is an optional task and is used to reduce the workload size for the subsequent tasks. Simpoint is used to find a set of segments that represent the different phases of the workload based on the execution counts of its basic blocks [7]. Each segment is associated with a weight that indicates the percentage of its basic blocks execution to the overall workload basic blocks execution. The weights associated with each segment can be used later to estimate the workload's overall metric of interest (e.g. its total power consumption) from each segment's metrics.

2.2. Slicing

The second task, illustrated in Figure 3(b), involves using the SCI to identify representative slices within the segments. The slices represent the segments based on specific workload metrics, and satisfy a set of constraints (such as a size constraint). Since this paper illustrates the use of the slicing flow for estimating floating point power consumption using RTL-level simulations, our slicing criteria included the density of floating point instructions in the code as a workload metric, a constraint to slice at loop boundaries, and a maximum slice size constraint.



(a) **Segmenting**: Using SimPoint to make segments representing phases of execution



(b) **Slicing**: Using a slicing tool to generate a set of slices to represent a segment

Figure 3 Segmenting and slicing illustration.

2.3. Micro-architecture warm-up

The final slicing task is to generate the identified slices from the segments, and add the necessary warm-up of the micro-architecture to the slices. This task is carried out by the Tracer shown in Figure 2, and it consists of three sub-tasks: initial state warm-loading, cache warming, and branch-predictor warming. These sub-tasks are explained next.

2.3.1. Initial state warm-loading

Many simulation test benches provide the ability to specify initial state values for the different architected resources, like registers and memory. The values of such resources at the beginning of the slice are collected at the slicing point by the Slicer, and are output in a format understood by the test bench as part of the final slice. If the test bench does not provide resource warm-loading facilities, then code execution can be used to initialize the required resources to the desired initial state.

2.3.2. Cache warming prelude

When a workload is segmented or otherwise reduced in size, differences in cache behavior between the original and reduced workload can introduce CPI errors as high as 15% [8, 12]. Consequently, cache warming may be necessary to ensure that a slice will produce appropriate metrics upon its execution. Cache warming may not always be necessary (and was not significant in our studies, see Section 3.2.1), but when it is, a prelude can be added to the slice to cover addresses that would hit in the context of full workload execution. These addresses would be collected from the cache activity annotations that the Tracer added to the trace in the first pass. The prelude could take the form of testbench directives, or it could be code included within the slice to prefetch data and instructions (supported by most architectures).

2.3.3. Branch-predictor warming

Branch predictor warming may also be necessary for micro-architectures that support speculative execution [3, 8]. Our results indicate that the need for such warmup can be exacerbated by constraints on the length of slices that can run on low-level models, which does not result in sufficient execution time to allow the performance effects of an initially cold branch predictor to be amortized (Section 3.2.2). The branch predictor is likely to make different predictions for branches contained within a slice when the slice is run in isolation as compared to when the same sequence of instructions is run as part of the full workload. This difference in predictions can result in speculatively executing different sequences of instructions during running the slice.

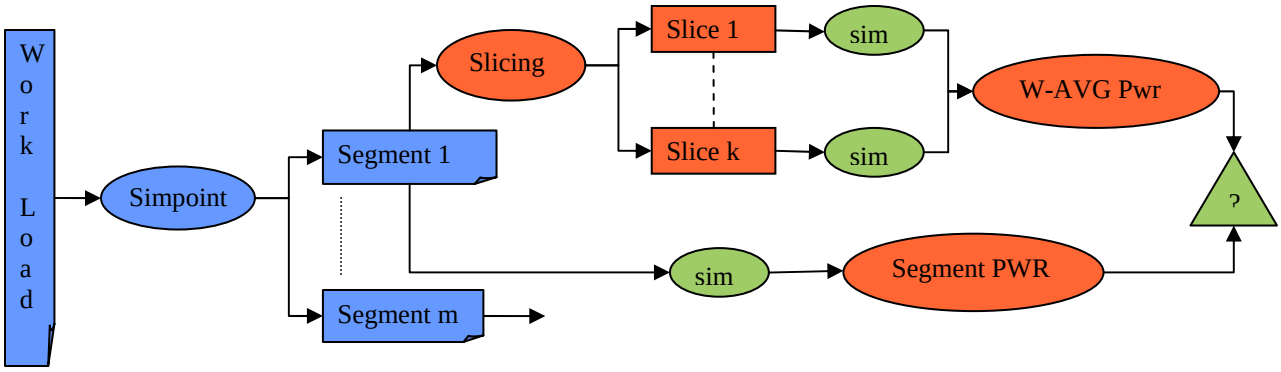


Figure 4 Workload slicing validation methodology.

Identifying branch predictions during the tracing pass requires a detailed model of the micro-architecture. Such a detailed model can slow the tracing task significantly. In addition, modern branch predictor hardware is complex and uses several tables, which are difficult to preload. One alternative to the warming of branch predictors is to execute prelude code for preloading branch target buffers, and using hints in branch instructions to direct the branch predictor based on an approximate model of the branch predictor that can be used with the tracer. Alternatively, in this research, we study the effect of slicing size on branch prediction, and we identify minimum slicing size for a given quality metric (defined in Section 3).

3. Experiments

The purpose of this section is to study three issues related to workload slicing: 1) The validity of power estimates using small size slices by defining a power estimate accuracy metric and measuring it experimentally, 2) studying the effects of cache and branch predictor warming, and 3) finding the minimum slicing size, based on the defined metric of power estimation accuracy.

3.1. Methodology

In order to evaluate workload slicing, four of the EEMBC automotive benchmarks [13] are used to estimate the power consumption of the floating-point unit in a research microprocessor. The benchmarks are: a2time01, basefp01, matrix01, and tblook01. All of these benchmarks were compiled using gcc 3.4, and the default iteration count for each benchmark. The benchmarks were compiled to run without an operating system, such that system calls were replaced with a no-op. The research microprocessor models an out-of-order pipeline that can issue and complete three instructions per cycle. It includes 32Kbytes separate L1 instruction and data caches, with a 1Mbytes unified L2 cache.

Each benchmark was run on a functional model of the microprocessor to generate a trace of the benchmark. Each benchmark's trace was analyzed by Simpoint to generate 1-million-instruction segments, as depicted in Figure 4. Each segment was sliced by the SCI to generate a set of slices $S=\{s_1,s_2, \ldots,s_k\}$, of equal sizes. The experiment included a set of trials using slicing size constraints of 50, 100, 500, 1000, 5000, and 10000 instructions. For each trial, the criteria given to the SCI included slicing at loop boundary (where possible), minimum density of floating point instructions, and a slicing size. Each slice is associated with a weight that represents the instructions count fraction of each slice out of the segment (for example a slice may represent multiple iterations of a loop body). The set of weights $W=\{w_1,w_2,\ldots,w_k\}$ satisfies the equation:

$$\sum_{i=1}^{k} w_i = 1$$

An RTL gate-level model of the floating-point unit was used to collect bit-switching information as code executed on the core. This information was then used by a power-estimation tool to find the power consumption of the floating-point unit. Each segment's entire trace of instructions was run on the gate-level model to generate switching information, which was used by the power-estimation tool to find the segment's power numbers, *SP*. The segment power, SP, is assumed to be the reference power in this experimental section for the corresponding segment. The power of the slices, $P=\{p_1,p_2, \ldots,p_k\}$, was collected using the same approach.

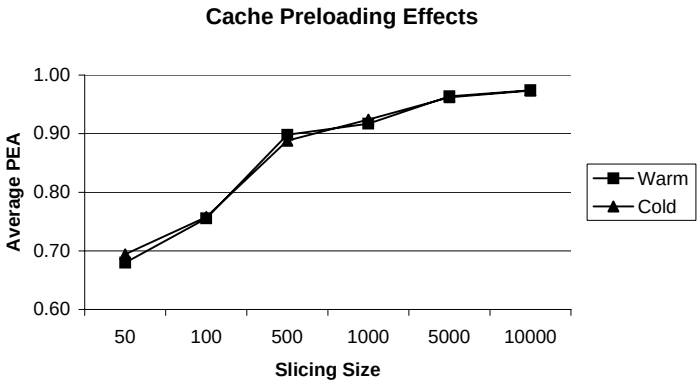


Figure 5 Cache preloading effect on power estimation accuracy.

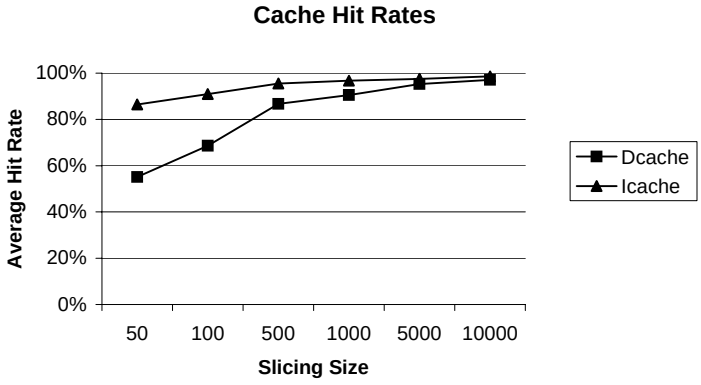


Figure 6 Cache preloading effect on power estimation accuracy.

The segment's estimated power, *EP*, is computed as a weighted sum of the set *P* using the weights of the set *W*.

$$EP = \sum_{i=1}^{k} p_i w_i$$

3.2. Results

3.2.1. Cache Warming Effects

Power estimation accuracy (PEA) is defined as a metric to evaluate the quality of the power estimation of each segment from its slices. PEA is defined as follows:

$$PEA = \frac{EP}{SP} = \frac{\sum_{i=1}^{k} p_i w_i}{SP}$$

Figure 5 depicts the average PEA over all segments of all benchmarks at the different slicing sizes for two configurations: warm, and cold. In the warm configuration, both the L1 data and instruction caches were pre-loaded before executing each slice or segment code. In the cold configuration, the code was run without pre-loading the L1 caches. In both configurations, the L2 cache was pre-loaded. On one hand, Figure 5 indicates that preloading the caches has little effect on the power estimation accuracy for the studied benchmarks. On the other hand, independent of cache warming effects, PEA improves significantly as the slicing size increases, until it flattens at about 0.97 at the slicing size of 5000 instructions.

To illustrate why cache preloading did not affect the PEA, the average hit rates (using cold caches) for all the slices of each slicing size are shown in Figure 6. This figure indicates that the hit rate increases significantly as the slicing size increases, up to slicing size of 5000, and then flattens. The large difference in the hit rate is not reflected in the PEA values in the previous figure, which suggests that the cache misses do not affect the execution of instructions. This can happen if the instructions per cycle (IPC) is low even if the caches were warm.

Figure 7 shows the average IPC of all the slices at each slicing size when the caches are preloaded. The IPC at the lower slicing sizes is significantly less than it is at the higher slicing sizes. This figure supports the intuition that the cache miss penalty is masked by other IPC lowering effects in these configurations of benchmarks and hardware. The following section illustrates how speculation hides the cache effects.

3.2.2. Speculative Execution Effects

As the cache preloading is not the reason for the poor PEA at the lower slicing sizes, the speculative execution effects are studied in this section. The branch prediction rate of each segment, referred to as *GBPR*, is compared to the branch prediction rates of the slices representing the segment. The error in the branch prediction rate of each slice, called *EBPR*, is defined as follows:

$$EBPR = \frac{GBPR - SBPR}{GBPR}$$

where, *SBPR* is the branch prediction rate of the slice,

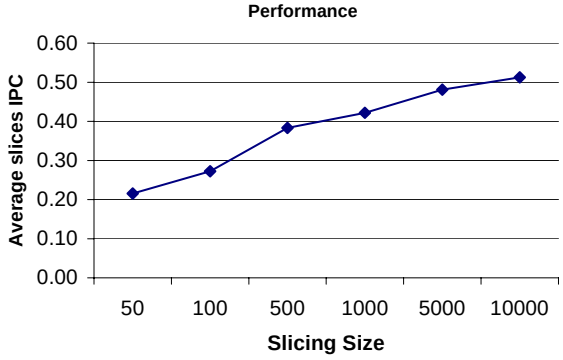


Figure 7 Average IPC of preloaded caches.

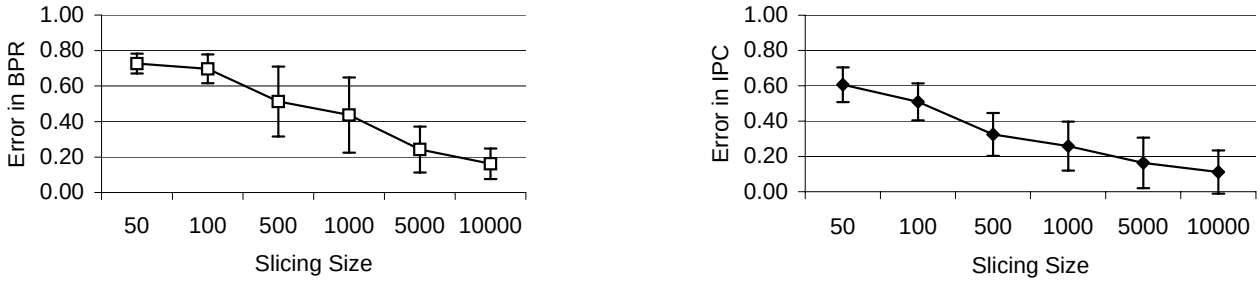


Figure 8 Relative difference in BPR and IPC.

and *GBPR* is the branch prediction rate of the segment represented by the slice. Similarly, the error in IPC of a slice, called EIPC, is defined as:

$$EIPC = \frac{GIPC - SIPC}{GIPC}$$

where, *SIPC* is the IPC of the slice, and *GIPC* is the IPC of the segment represented by the slice.

Figure 8 depicts the average EBPR and average EIPC over all the slices of all the segments of the studied benchmarks. The figure also depicts the error bars in both averages using the standard deviation. The figure indicates a strong correlation between the error in BPR, and the error in IPC for the slices at the different slicing sizes. It also suggests that as the slicing size increase, both error values decrease significantly, until the slicing size reaches 5000, where the error change becomes less significant as the slicing size increases to 10000 instructions.

The large EIPC of the smaller size slices can be attributed to the large EBPR. The reason for this difference in EBPR is that the branch predictor needs to see each branch several times before it can correctly predict it, in addition to correctly predicting its branch target. At the lower slicing size, the branch predictor sees each branch a few times only and as a result does not have a chance to learn it. As the slicing size increases, the branch predictor accuracy improves significantly as it sees the same branch repeatedly. However, after the slicing size reaches the 5000 instruction size, new branches get introduced in the code, and the improvement in the prediction accuracy becomes more difficult, and the prediction accuracy approaches that of the full segment.

3.2.3. Power Estimation Accuracy (PEA) of individual segments

Finally Figure 9 depicts the PEA of the individual segments from each benchmark for slicing sizes of 5,000 and 10,000 instructions. The first observation in this figure is that the benchmark matrix01 exhibits 6 phases using 1-million-instruction sampling size, while the rest of the benchmarks exhibit 3 phases. This agrees with the tasks carried out by these benchmarks.

The second observation is that the power estimation accuracy of the individual segments ranges from about 0.96 to 1.00 (accuracy within 4%) at 5,000 instruction slicing size, and that it is improved by about 0.02 when increasing the slicing size to 10,000 instructions. Thus, the total number of instructions executed using the representative slices is about 5% of the total instructions in each segment, and accuracy of the power estimates using slices was within 4% of the power estimate for the entire segment.

4. Summary

Power estimation and characterization of new features of high-performance micro-processors is becoming a first-order design concern. The ability to perform power estimation and characterization is limited by available models of these features. These models present constraints on the size of stimulus that can be used. Highly targeted studies place additional constraints on workloads, for example instruction content.

Micro-benchmarks and benchmark statistical sampling are not sufficient to produce representative workloads for power estimation, because of scalability, or the lack of flexibility in the metrics and constraints that the sampling tools provide. Workload slicing can be used effectively to produce representative small slices that can be run in lieu of long workloads.

In addition to the requirement for flexibility in the criteria used for slicing (both metrics and constraints), it was illustrated that micro-architecture warm-up is necessary for short slices intended to be used in estimating the power consumed by the long benchmarks. Two main warm-up problems were discussed: cache warming, and speculative execution due to branch predictor warm-up. The benchmarks used in this research to characterize the floating-point unit of a research microprocessor are compute bound. As such, cache warm-up did not have a significant effect on the power estimation accuracy from slices. Conversely, the

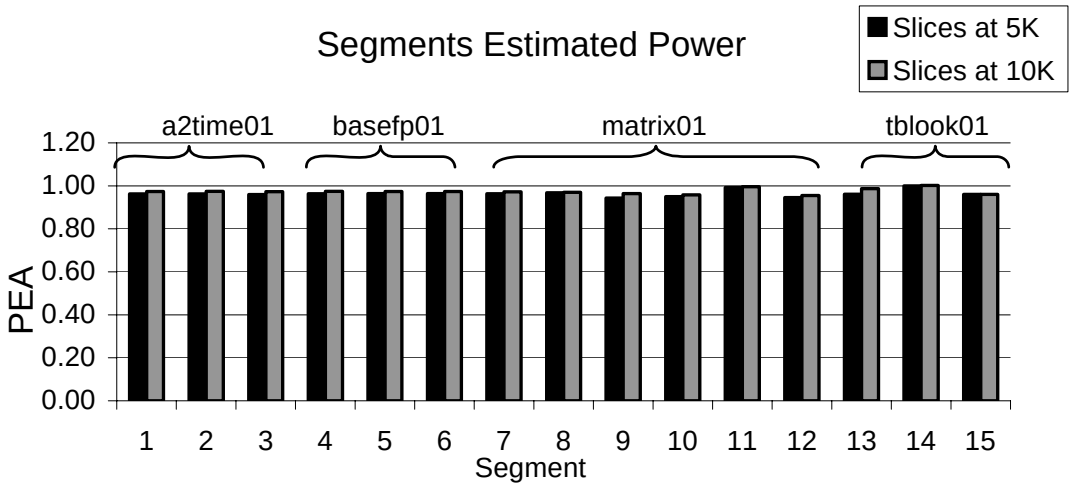


Figure 9 Segments' power estimation accuracy (PEA).

warm-up of the branch predictor has a significant effect on the power estimation accuracy in small-sized slices. It was found that slices of sizes around 5,000 instructions or larger are sufficient to hide the error in power estimation due to executing instructions on the wrong path, for the studied benchmarks.

The experiments presented in this paper illustrated that a set of slices comprised of about 5% of the code contained in a set of segments representing the different phases of the benchmarks was sufficient for estimating the power consumption to within 4% accuracy. This result is an encouraging one, motivating us to pursue evaluating workload slicing for other types of benchmarks.

Experiments are needed to further evaluate this proposed approach. The presented experiments did not examine other types of workloads, such as memory-bound benchmarks. Accordingly, more workloads of different domains need to be considered to further establish confidence in the approach.

References

[1] E. Macii, M. Pedram, and F. Somenzi, "High-level Power Modeling, Estimation, and Optimization," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 17, pp. 1061-1079, 1998.

[2] R. Todi, "SPECLite: Using Representative Samples to Reduce SPEC CPU2000 Workload," presented at IEEE International Workshop on Workload Characterization, 2001.

[3] L. Eeckhout, R. H. Bell, B. Stougie, K. De Bosschere, and L. K. John, "Control Flow Modeling in Statistical Simulation for Accurate and Efficient Processor Design Studies," presented at 31st Annual International Symposium on Computer Architecture, 2004.

[4] J. J. Yi, R. Sendag, L. Eeckhout, A. Joshi, D. J. Lilja, and L. K. John, "Evaluating Benchmark Subsetting Approaches," presented at IEEE Symposium on Workload Characterization, San Jose, CA, 2006.

[5] R. H. Bell and L. K. John, "Efficient Power Analysis using Synthetic Testcases," presented at International Symposium on Workload Characterization, 2005.

[6] T. Diep and J. P. Shen, "VMW: A Visualization-Based Microarchitecture Workbench," *IEEE Computer*, vol. 28, pp. 57 - 64, 1995.

[7] E. Perelman, G. Hamerly, M. V. Biesbrouck, T. Sherwood, and B. Calder, "Using SimPoint for Accurate and Efficient Simulation," presented at 2003 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems, San Diego, CA, 2003.

[8] R. E. Wunderlich, T. F. Wienisch, B. Falsafi, and J. C. Hoe, "Statistical Sampling of Microarchitecture Simulation," *ACM Transactions on Modeling and Computer Simulation*, vol. 16, pp. 197-224, 2006.

[9] R. Desikan, D. Burger, and S. W. Keckler, "Measuring experimental error in microprocessor simulation," presented at 28th Annuan International Symposium on Computer Architecture, 2001.

[10] H. F. Al-Sukhni, J. C. Holt, and D. A. Connors, "Improved Stride Prefetching Using Extrinsic Stream Characteristics," presented at IEEE International Symposium on Performance Analysis of Systems and Software, Austin, TX, 2006.

[11] J. J. Yi, S. V. Kodakara, R. Sendag, D. J. Lilja, and D. M. Hawkins, "Characterizing and Comparing Prevailing Simulation Techniques," presented at 11th International Symposium on High Performance Computer Architecture, Austin, TX, 2005.

[12] Y. Luo, L. K. John, and L. Eeckhout, "Self-monitored Adaptiv Cache Warm-Up for Microprocessor Simulation," presented at 16th Symposium on Computer Architecture and High Performance Computing, 2004.

[13] EEMBC, "The EEMBC Benchmark Suite," in *http://www.eembc.org*.

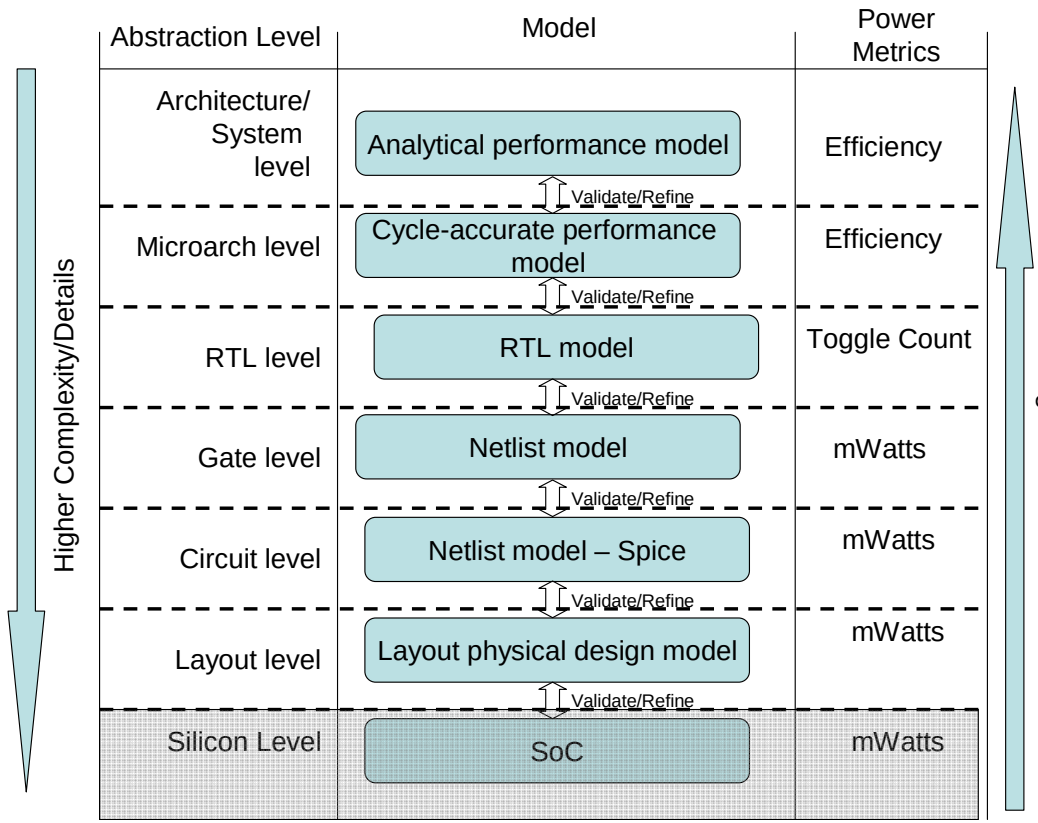
Workload Slicing for Detailed Pre-Silicon Power Estimation

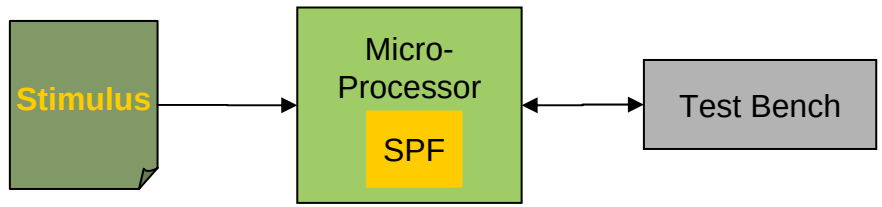
Hassan Al-Sukhni, James Holt, David Lindberg, Michele Reese

freescale™ semiconductor

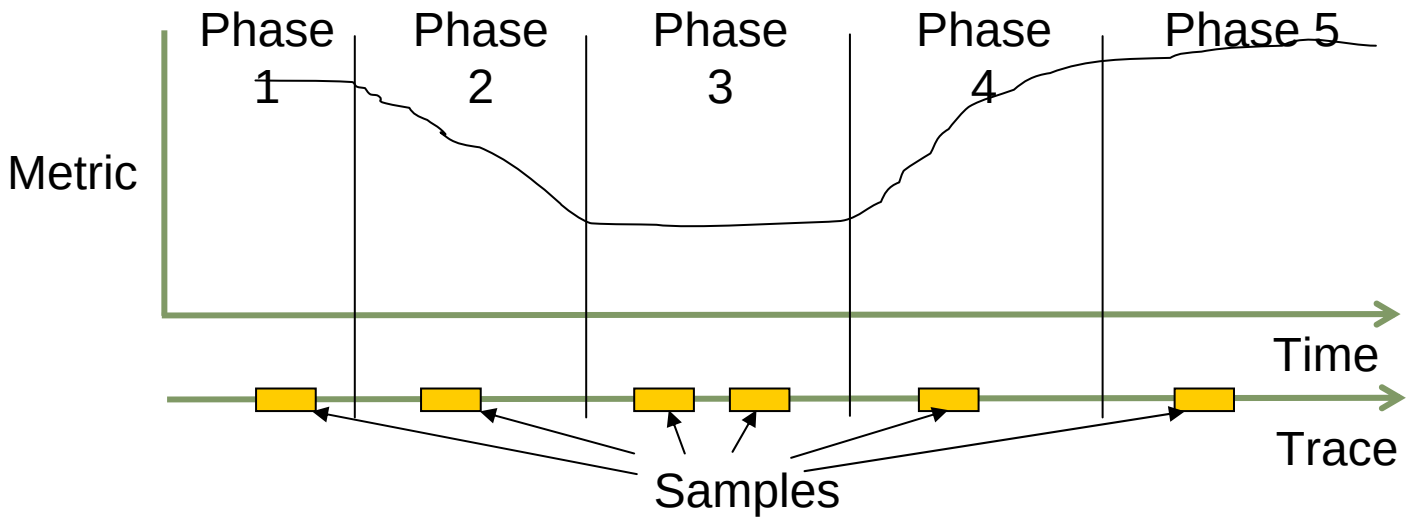
Introduction

- High performance microprocessors are very large designs that include several special purpose features (SPF) like, Speculation, multi-threading support, floating-point units, etc.
- High-level abstraction models of *power consumption* for new SPFs are not suitable for their characterization because of:
  - Availability
  - Poor accuracy
- Low-level models must be used
  - Slow
  - Large outputs
- Stimuli are constrained





Workload Sampling



Stimuli generation techniques

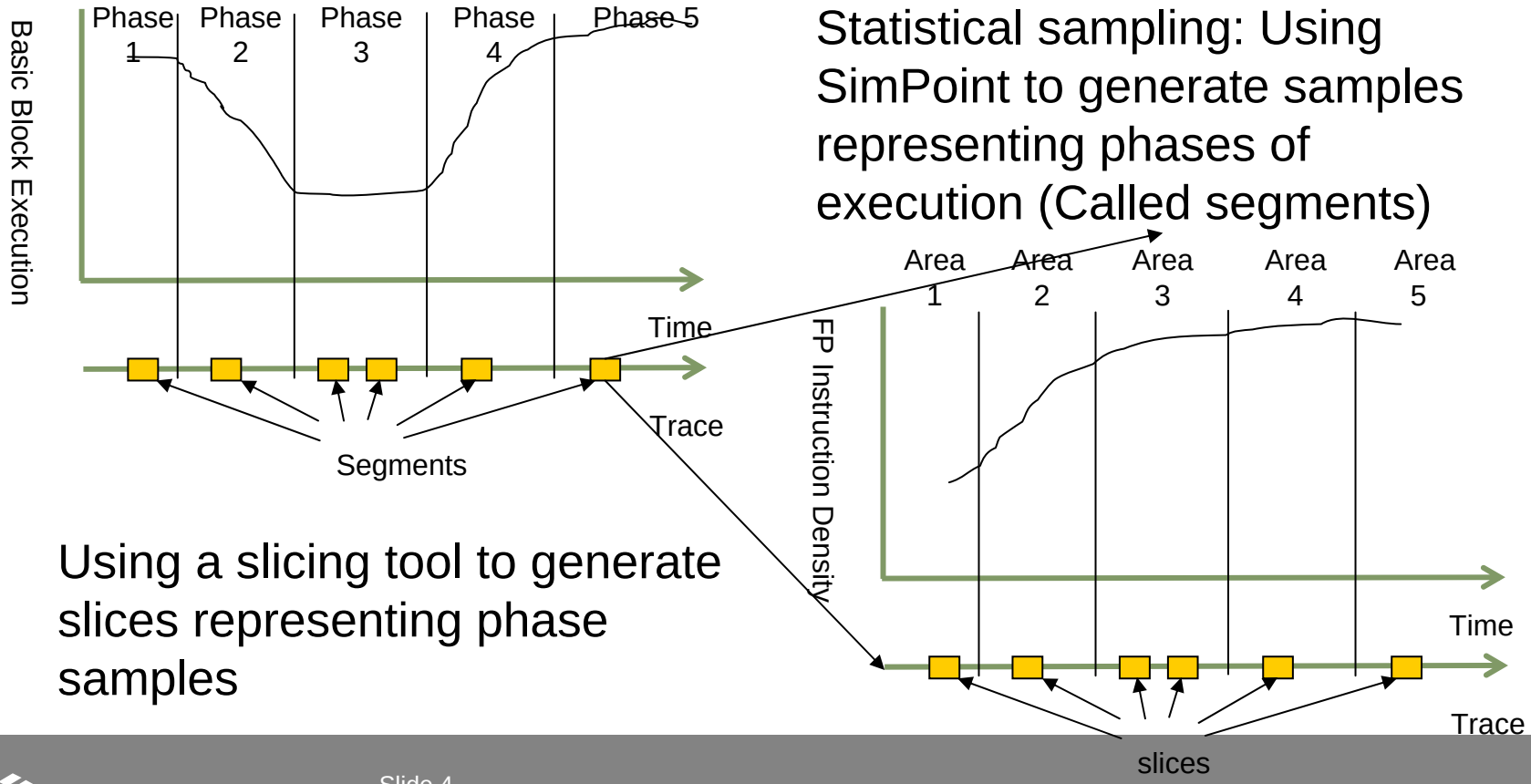
- Micro-benchmarks: Good for targeted testing, but not scalable to higher abstraction levels.
- Benchmarks' statistical sampling (e.g SimPoint): Designed for high-level abstraction models (Architecture/micro-architecture exploration)
  - Representative but large samples
  - Tools employ inflexible metrics
- Ad-hoc sampling
  - Lost effects of pre-sample code
  - Micro-architecture warming needed.

Motivation

- **Problem**: constraints on low-level pre-silicon simulation
  - sample size (few thousand cycles max)
  - warm-up code needed to account for:
    - > speculative execution effects
    - > micro-architecture state warming
  - feature-specific sampling metrics as opposed to micro-architecture independent sampling metrics, examples:
    - > instruction-based sampling (e.g. density of FP instructions)
    - > exercising specifics behavior of a Arch/uArch feature (e.g. high-miss L1 D-Cache miss rate, high branch prediction miss rate. Etc.)
- **Requirements**: Identify short representative code regions of a workload, called slices, that collectively represent the sample for a given set of metrics
- A slice includes:
  - The initial state, such as GPR values, etc, used to warm load the processor
  - Sufficient memory initialization of code and data sections
- Some verification test-benches have the ability to warm load processor state, and proper sizing of the slices can compensate for difficult to pre-load structures (as will be demonstrated).

Approach: A Two-Step Technique

Goal: Find a small set of representative slices that can be run in lieu of a large sample for power estimation

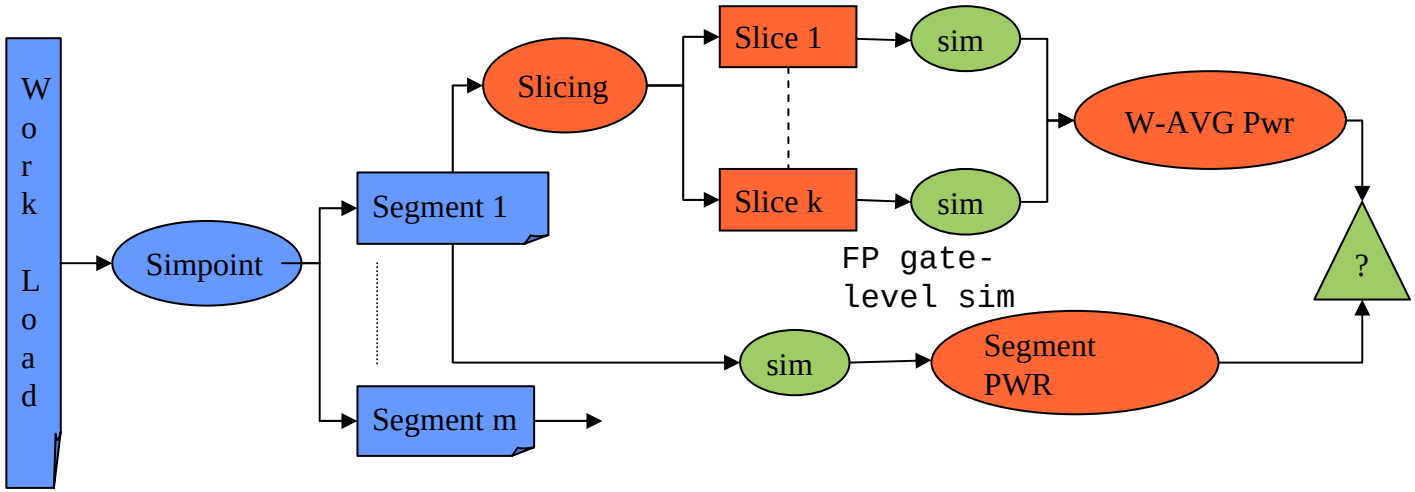


Slice Identification

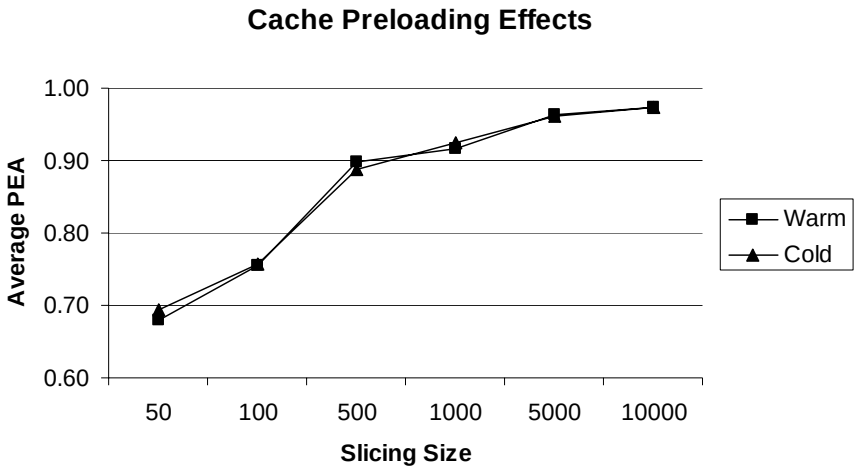
- Slices are identified based on:
  - trace data output by a tracing tool, and
  - slice constraints.
- Constraints can be metrics calculated from the trace data and/or constraints on those metrics.
- Prototype slicing tools were developed for identifying slices. Several primitive metrics are provided along with the ability to define more complex metrics. Constraints that the tools support include:
  - Metrics value
  - Length of slice
  - Eliminating overlapping slices
  - Loop iteration constraints

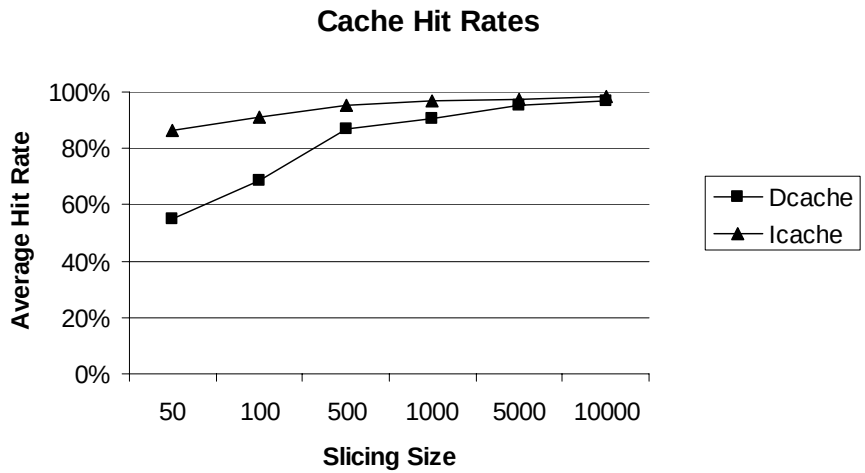
Experimental Methodology - EEMBC

- Research OoO 3-wide superscalar ~20-stage micro-processor with 7-stage FPU. Hybrid branch predictor, 32K I & D Caches, 1 MB UL2.
- 4 EEMBC automotive FP benchmarks, traced on functional simulator.
- Simpoint used to generate 1M instruction segments.
- Each segment sliced at sizes of (50,100,500,1000,5000) instructions, each slice is associated with a weight that represents its contribution.
- Gate-level simulation to collect bit switching that is used with a power estimation tool.



Results- Power Estimation Accuracy



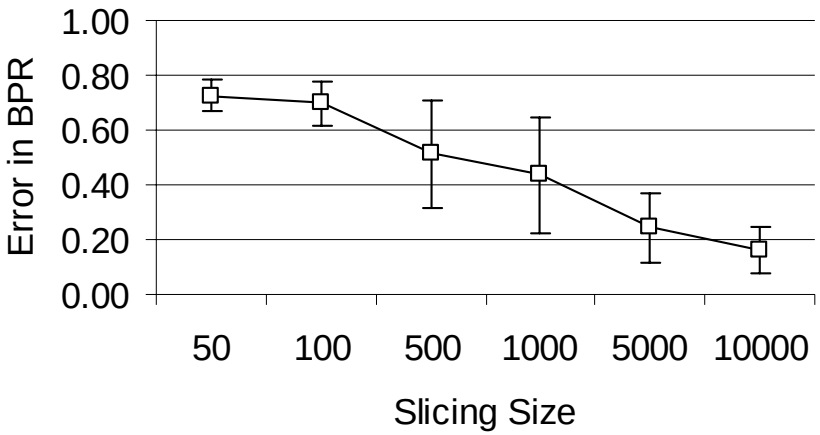


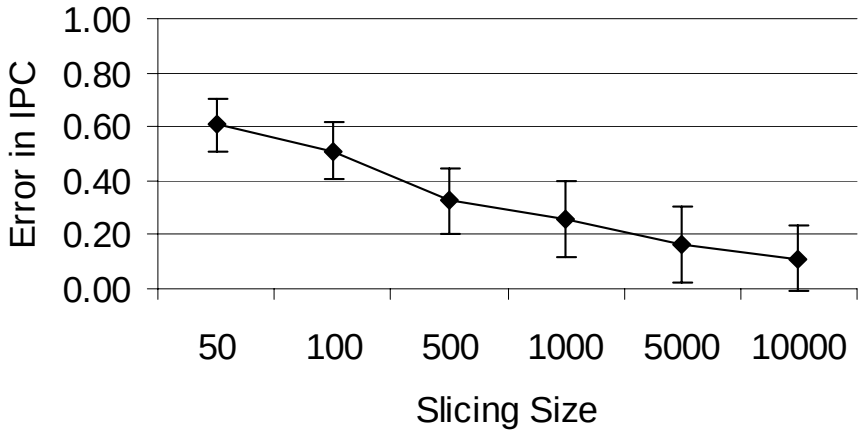
$$EP = \sum_{i=1}^{k} p_i w_i$$

$$PEA = \frac{EP}{SP} = \frac{\sum_{i=1}^{k} p_i w_i}{SP}$$

- Power estimation accuracy is poor at very short slices.
- Cache pre-loading does not improve power estimation accuracy.

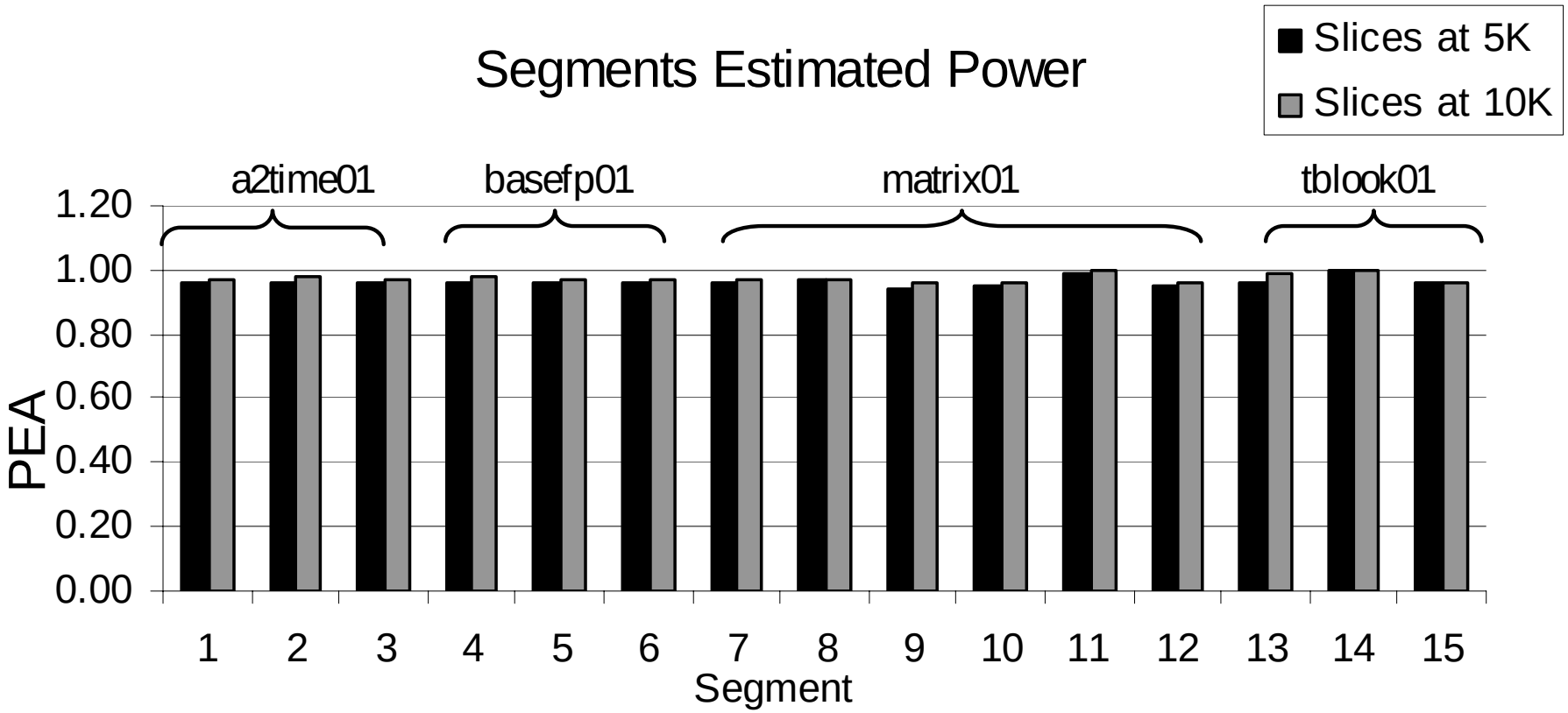
Branch Prediction Effects





- Error in IPC and power estimation highly correlated with the error in branch prediction rate between the slices and the segments they represent

Power Accuracy Details



Conclusions and Future Work

- Workload sampling for new features requires flexibility in sampling metrics.
- Representative slices as small as 5% of traditional workload sample sizes can be sufficient for the studied benchmarks, resulting in power estimates within 4% of the full samples' power.
- Illustrated that micro-architecture warm-up has significant effect on power estimation using small size slices.
- Careful sizing of the slices can account for micro-architecture warm-up effects on power estimation.

Backups

Workload Slicing Flow