# A Systematic Method For Functional Unit Power Estimation in Microprocessors*

Wei Wu, Lingling Jin, Jun Yang
Department of Computer Science
University of California, Riverside, CA 92521

Pu Liu, Sheldon X.-D. Tan
Department of Electrical Engineering
University of California, Riverside, CA 92521

## ABSTRACT

We present a new method for mathematically estimating the active unit power of functional units in modern microprocessors such as the Pentium 4 family. Our method leverages the phasic behavior in power consumption of programs, and captures as many power phases as possible to form a linear system of equations such that the functional unit power can be solved. Our experiment results on a real Pentium 4 processor show that power estimations attained as such agree with the measured power very well, with deviations less than 5% only.

**Categories and Subject Descriptors:** C.4[Performance of Systems]

**General Terms:** Measurement, Experimentation

**Keywords:** Power Estimation, Performance Counter, Microprocessor

## 1. INTRODUCTION

Power density has become one of the major constraints on attainable processor performance as integrated circuits enter the realm of nanometer technology. The exponential increase in power density leads to rapid growth in chip temperature, which jeopardizes the reliability and the lifetime of the processor. Efficient runtime regulation of operating temperature through dynamic thermal management therefore becomes imperative. The prevailing studies on thermal managements rely heavily on cycle-level simulators [8] which is neither accurate enough nor efficient.

An alternative approach is to perform online monitoring and estimation of the processor temperature [3, 4, 5]. The temperature is first obtained through a runtime component-wise power estimation of the processor, and second an analytical thermal modeling that computes temperature based on the power estimations. This approach is governed by the two main elements – the quality of the power estimation and the efficiency of the thermal modeling – keeping in mind that both are performed at runtime. While the latter has been addressed in [4, 5], the former remains challenging as all the schemes fall back on *empirical* component power inputs that are only good for showing the power trend rather than a real power value. Since the temperature is derived from power, a less accurate power input would cripple the thermal modeling, devaluating the entire online thermal tracking scheme.

Recently, a mechanism for estimating the component and the total power of a *real* microprocessor has been proposed [1, 2]. The idea is

that each component unit power can be approximated and tuned until the summation matches with the total power that can be measured experimentally. The main obstacle of this approach, however, is that the searching for component unit power values is still empirical, which almost always involves manual tunings. In this paper, we introduce a novel method for finding the component unit power. Our new method incorporates a statistical method that solves for a vector of unit power from a system of linear equations established through experimental measurements. We tackle the difficulties in establishing a good matrix equation, and develop a $K$-means based method that can search a good set of solutions successfully. The set of unit power values solved using our method is more trustworthy than that obtained through experiential fitting. It will also benefit software simulators by providing them with close-to-reality component power values. Our experience evidenced that our new method delivers very good runtime power estimations on two Pentium 4 processors and can be applied to different microprocessors.

## 2. PROBLEM STATEMENT

### 2.1 Problem Statement

The "components" we target to refer to the functional units (FU) in a processor. For example, Fig. 1 shows the block level floorplan of the FUs in the Willamette core. There are 22 FUs in total, divided by dashed lines as shown in the figure. The powers for these 22 FUs are the unknowns we target to solve. Here a power value is the *active per access* power of an FU. This includes the dynamic power, the clock power and the leakage power. The total processor power during an execution of a program is expressed as:

$$\sum_{i=1}^{i=22} AF_i \times P_i + P_{idle} = P , \qquad (1)$$

where $P_i$ is the active power for each FU. $AF_i$ is the activity factor, $P_{idle}$ is idle power when no programs are loaded onto the processor and $P$ is the total processor power. The activity factor is the number of times the FU is used during a unit time. It can be obtained through programming the *performance counters* available in many modern processors. The total processor power and the $P_{idle}$ can be measured from the CPU power lines externally. Hence, once we can attain all the $AF_i$s and the $P$, multiple instances of equation (1) can be created through running different programs so that a system of equations can be established where $P_i$s are unknowns and the rest are known.

Setting up a good system of equation (1) turns out to be critical in finding accurate FU powers. A good system should consider as many power consumption scenarios as possible. Moreover, since the total power is measured externally, there are white noise and errors introduced into $P$. Therefore, we need a way to establish good equations and statistical techniques to eliminate the noise and errors from the measurement.

### 2.2 Power Measurement Setup

To obtain the total power and the activity factors, we use the experimental setup as in [1] to measure the total power of the processor and
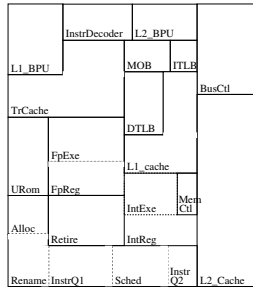
**Figure 1: Floorplan of P4 Willamette core.**

to log information (shown in Fig. 2). An Agilent Digital Multimeter (DMM) 34401 is used to read the total current of CPU from the clamp-on ammeter and transfer the data through a RS232 digital port into a logging machine. We then compute the powers based on the logged information. The logging time interval is 0.4 second.
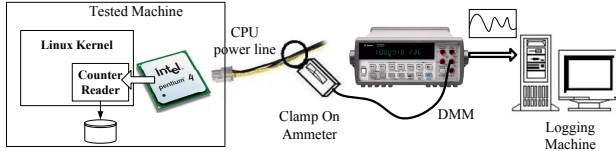


**Figure 2: Experimental setup.**

The activity factors of the FUs are obtained using the *performance counters* that can be predefined and retrieved at runtime. The Pentium 4 supports totally 45 hardware events and 18 physical performance counters [9]. When deriving the activity factor for each FU, we need to understand correctly the actual activity each counter is accumulating. Some *AF*s can be determined by a single counter, while others may need to combine several counters. In Fig. 1 the blocks divided by solid lines are assigned with different counters and the sub-blocks separated by dashed lines share a common counter.

## 3. NEW METHOD FOR UNIT POWER ESTIMATION

In this section, we present our new method of estimating the unit power of functional units of microprocessors. We take a two-step approach in this process.

- In the first step, we design a set of simple programs (microbenchmarks), each of which exercises only a small subset of FUs, to obtain relatively rough values for the $P_i$s.
- In the second step, we use those initial $P_i$s to bound and guide the searching for much more accurate results incorporating the $K$-means based method on a set of significantly more complex programs. The first step is described next.

### 3.1 Rough Estimation Using Microbenchmarks

The main purpose of designing the microbenchmarks is to have a good initial approximation of the $P_i$s for each FU. Each program is an infinite loop with only one instruction type covering only a few performance counters (or FUs) so the set of microbenchmarks covers the complete set of counters and FUs. The advantage of designing such simple programs is that each consumes nearly a constant power. Theoretically each point in the power trace and the counter trace can form an equation (1). However, due to the possible noise in the measured power and the rotation effect in counter collection [1], using one point from an entire trace to form one equation would incur significant errors. It is much better to take the average over the entire trace so that the noise and errors are canceled as much as possible. Hence, having constant power and counter values can help to obtain more accurate $P_i$s. The collection of all microbenchmarks enable us to form a system of equations as

$$\mathbf{M}_{AF}\mathbf{P_i} + P_{idle} = \mathbf{P} \qquad (2)$$

where $\mathbf{M}_{AF}$ is the activity factor coefficient matrix of vector $\mathbf{P_i}$. $\mathbf{P}$ is the vector of measured powers for each microbenchmark. Our goal here is to find a solution that best satisfies all the equations created. We used the least square linear regression method in Matlab to solve equation (2), and searched for the best solution such that:

$$\sum_{all\ equations} (\sum_i AF_i \times P_i - P)^2 \qquad (3)$$

is minimum. This criterion helps to reduce the accumulative error of the computed power and the measured power. We plotted the two powers in Fig. 3 for all the microbenchmarks we ran. As we can see, the computed $P_i$s are fairly close to the measured power.
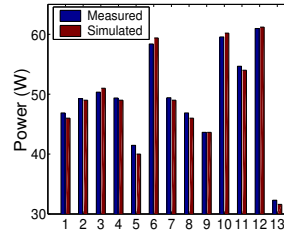
 

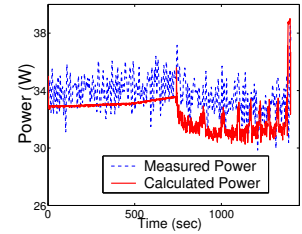**Figure 3: The computed power and the measured power for microbenchmarks match very well.**

**Figure 4: The inaccuracy of using the power results from the microbenchmarks on a SPEC2K program `vpr`.**

However, when we plug in the above initial $P_i$s into the equations such as the SPEC2K benchmarks, we observed large deviations of the computed power from the measured power. Fig. 4 shows such an instance. In this program, the computed power is less than the measured power, indicating that some initial $P_i$s are overly under-estimated and need to be refined. In this next section, we explain the reasons and propose a new method to mitigate this problem.

### 3.2 Fine Tuning the Unit Powers of FUs

The main reason for the inaccuracy from using the microbenchmarks is because they cannot cover all possible scenarios of power consumption. For example, we used the "add" instruction in a microbenchmark to exercise the integer ALU. However, the ALU has other arithmetic and logic operations such as "sub" "shift" as well. Those consume different powers than the "add" operation. Similarly, different binary inputs produce different power consumption in FUs. Hence, it is more efficient to adopt a comprehensive set of benchmarks that are sophisticated enough to show vastly different cases. We choose the SPEC2K benchmark suite for this purpose.

When experimenting with large programs, it is advantageous to consider as many scenarios as possible since large and complex programs exhibit a variety of behavior. Fortunately, recent works have shown that the behavior of programs are neither completely homogenous nor totally random, and they can be categorized into *phases* [7]. Each phase represents a unique behavior in terms of, e.g., IPC, cache miss rates, etc. A program can enter the same phase at different times of execution. We identified similar phase behavior for the collected counters and the measured power values. As an example, Fig. 5 shows clear phases of program `gzip` in SPEC2K. Since each phase represents a different power characteristic, our goal here is to cover as many phases as possible in order to establish a more complete set of equations for solving the component power.

**Identifying counter and power phases.** To achieve our goal, the first task is to identify different phases in the counter traces. In other words, we need to cluster the points into groups such that each group is a phase containing *similar* points and points in different groups are different. The similarity is defined according to specific requirements of the clustering. For example, the program phase studied in [7] uses
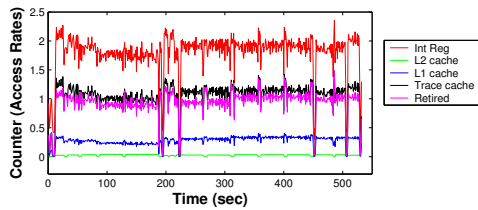
**Figure 5: The phase behavior of counters for `gzip`.**

the Manhattan distance between two *basic block vectors* (BBVs) (the usage frequencies of the basic blocks) as the similarity metric. The smaller the distance, the more similar two BBVs and similar BBVs belong to the same phase. Therefore, we need to develop our own metric suitable for finding similar counter points.

We use the *K*-means [6] clustering method which is a simple global optimization method for finding *k* independent sets. Once the points in those traces are clustered properly, we create one equation for one cluster only because similar points within a cluster do not give more information and different clusters represent different power characteristics. Next, we will explain the similarity metric we developed which is used as the distance function in the *K*-means algorithm, followed by the description of the algorithm itself.

**Similarity in counter vectors.** The data points we cluster here are the counter vectors extended by one more dimension for the measure power. Intuitively, if two vector points are similar, their corresponding components should be quite close to each other in value. The commonly used Manhattan distance for two vectors can quantify this similarity. However, the distance alone is not sufficient since we also care for the *correlation* of two values for each individual counter as well. For example, if the first components of two vectors are the same, but the second components have opposite differences from the first component even with the same distance, we do not consider them similar. To make it clear, let us look at the following example where three vectors, A, B and C, are shown, each vector being eight-dimension.
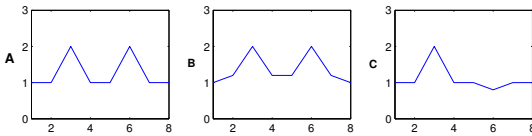


**Figure 6: The difference between the *distance* and the *correlation*.**

In Fig. 6, the x-axis shows the components of the vectors, and the y-axis shows the corresponding values. The distance from A to B and A to C are the same. However, A and B are strongly correlated as the curves show a similar shape. Whereas A and C are not correlated since the second spike in C is in negative direction compared with that in A. During the clustering, A and B should belong to the same cluster, but C does not.

The Manhattan distance between vectors *a* and *b* is defined as:

$$D_{manhattan}(a,b) = \sum_{i=1}^{n} |a_i - b_i| \; ,$$

where *n* is the dimension. The correlation of two vectors is statistically defined by the "correlation coefficient" (CC) as:

$$\begin{cases} CC(a,b) = \frac{Cov(a,b)}{\sqrt{Cov(a,a)Cov(b,b)}} \\ Cov(a,b) = n \sum_{i=1}^{n} a_i * b_i - \sum_{i=1}^{n} a_i \sum_{i=1}^{n} b_i \end{cases}$$

The CC is a value in [-1, 1], and it indicates the correlation between two vectors as follows:

$$\begin{cases} |CC| > 0.8 & \text{linearly correlated} \\ |CC| < 0.3 & \text{not correlated} \\ CC > 0 & \text{positively correlated (one increases,} \\ & \quad \text{the other also increases)} \\ CC < 0 & \text{negatively correlated} \end{cases}$$

Before defining our similarity metric, we need to pre-process the counter trace as different counters may differ in values by orders of magnitude. In order to avoid one counter taking over other counters numerically, we need to first normalize every component of a vector. We define that the similarity of two vectors is high when their Manhattan distance is small and their CC is large. Hence, our similarity metric for two vectors **v** and **w** is defined as:

$$Simi(\mathbf{v}, \mathbf{w}) = CC(\mathbf{v}, \mathbf{w}) \times (1 - D_{manhattan}(\mathbf{v}, \mathbf{w})') \; .$$

where $D_{manhattan}(\mathbf{v}, \mathbf{w})'$ equals to the original distance divided by total dimension number, such that it was projected to the interval of 0 to 1. Hence, when two vectors are same, the similarity value between them would be maximum, as their distance is zero and CC is 1. We now explain how the similarity metric is used in the *K*-means algorithm.

**The *K*-means algorithm.** The *K*-means clustering algorithm is an iterative optimization process. It clusters data points into *K* disjoint subsets. Each subset has a centroid which is usually the mean of the points in that cluster. The objective of the iteration is to minimize the total distance of points to their corresponding centroid. In our case, we replace this distance function with our similarity metric, and iteratively maximize:

$$\sum_{j=1}^{K} \sum_{x \in cluster_j} Simi(x, \varphi_j) \; , \tag{4}$$

where $\varphi_j$ is the centroid vector of the $j^{th}$ cluster. The algorithm starts with a random selection of *K* vectors as the initial *K* centroids. Then it iterates the following steps until convergence:

1. Put each vector into a cluster with the maximal similarity metric between the vector and the cluster centroid.

2. Recompute the centroid of each cluster. The centroid is the average of all the members in the cluster.

The above steps are repeated until all clusters are stable. Notice that we have assumed the number of clusters, *K*, is predetermined. However, finding an appropriate *K* is critical to obtaining a good clustering. To see what value is good for *K*, we varied it and measured the minimal $Simi(x, \varphi_j)$ for all $j = 1 \cdots K$. We use Gzip as an example, and the results are shown in Fig. 7.
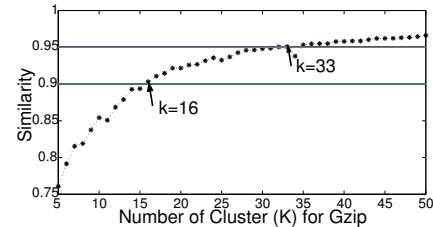


**Figure 7: The variations of similarity with *K*.**

According to our definition of the similarity metric, the larger the *K* the better the clustering and the larger the similarity metric. This is because when *K* is large, the points within a cluster are closer to each other so they are more similar to the centroid. As we see from Fig. 7, after $K = 16$, the similarity is higher than 0.9, and the first time it exceeds 0.95 when $K = 33$. However, the similarity does not grow linearly with *K*. After $K = 33$ the increase levels off. Therefore, we choose this value as our *K* since it is sufficiently good.

Once we found 33 phases for the counter and power traces collected for one program, we can set up an equation for every phase and the collection of all phases in all programs forms an *over-determined linear system* as in (2) where the number of equations is more than the number of unknowns. Fig. 8 shows the reordered traces of those in Fig. 5 when we put all the points belonging to the same phase together. The phases are separated by dotted lines in the graph. As we can see, the

counters in each phase are fairly flat, analogous to those microbenchmarks. Therefore, the representative counter and power values can be obtained by averaging the points within a phase. In fact, it is the equation for the centroid of the cluster.
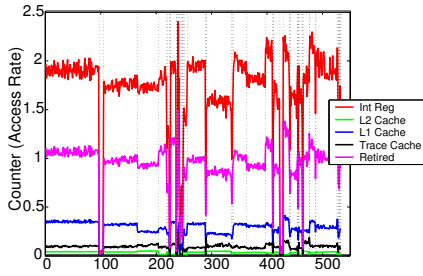


**Figure 8: The sorted traces showing the phases for `gzip`.**

We used the least square linear regression method to solve the established over-determined system of equations. For a linear regression problem, the more the number of equations the better the fitting. Also, it requires a good solution range to find the best fit. We use the solutions obtained from the microbenchmarks to create proper ranges. Such a method turns out to be quite effective. The results are presented next.

## 4. EXPERIMENTAL RESULTS

We have implemented this power estimation method on a Linux machine with a Pentium 4 Willamette processor. The Willamette core runs at 1.6Ghz with $0.18\mu$ technology. The core operating voltage is 1.75V and the typical power dissipation is 60.8W. The first and very important result we present is the comparison between our calculated power from dynamic counters and the measured power from the DMM (Fig. 9). It is not surprising to see that we are able to obtain very good power estimations as demonstrated by 10 SPEC2K programs. It is noteworthy that the program `vpr` shows a much better match compared with the solution obtained from running only the microbenchmarks as shown in Fig. 4.
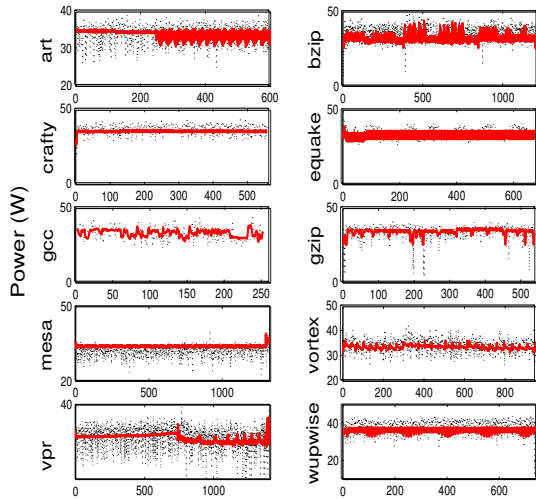


**Figure 9: The computed and measured total power**

The deviation of our computed power and the measured power is shown in Fig. 10. The average deviation rate is 4.5%. Here we do not term the deviation as error since the measured power may have white noise from the DMM. Also, there is no data available showing the runtime power consumption of the 10 SPEC2000 benchmarks. So we average the measured power corresponding to each counter cluster as a representative value for all the power trace in that cluster. The deviation is calculated as:

$$Deviation = \frac{\sqrt{\sum_{i=1}^{i=n}(Measured_i - Computed_i)^2}}{\sqrt{\sum_{i=1}^{i=n}(Measured_i)^2}} \times 100\% ,$$

where n is the number of clusters we found for each benchmark with at least 0.95 similarity in clustering.
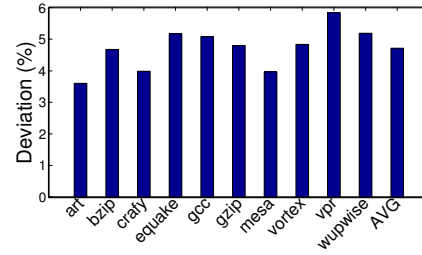


**Figure 10: Percentage of deviation for 10 SPECs benchmarks.**

Finally, we list the active power for each FU in Fig. 11. Note that the result obtained from solving equation (2) is per access FU power, which is the total FU energy consumed during one access divided by unit time. We need to further normalize the power value according to each FU's latency. According to Willamette's specification, L1 cache has two-cycle latency and L2 access lasts 7 cycles. We assumed all other latencies are one cycle, meaning that they are the per access energy consumed in one cycle.

| Function Units | Power (W) | Function Units | Power (W) |
|---|---|---|---|
| Bus Contrl | 11.0 | L2 Cache | 12.9 |
| 2nd level BPU | 7.7 | ITLB/Ifetch | 3.5 |
| L1 cache | 2.6 | MOB | 13.5 |
| Memory Control | 0.5 | DTLB | 6.1 |
| FP Exec. | 3.6 | Int Exec. | 0.5 |
| Instr Decode | 10.0 | Trace Cache | 2.5 |
| 1st Level BPU | 6.8 | Microcode ROM | 3.0 |
| Retirement | 2.5 | Idle Power | 22.3 |

**Figure 11: Active power (Willamette) for each FU.**

## 5. CONCLUSION

In this paper, we have described a new method for accurately estimating the active power of FUs in modern processors. The new method estimates the unit powers by solving the linear equations from statistically measured data. We used uncorrelated microbenchmarks to roughly estimate powers for functional units and K-means clustered power traces for fine tuning the unit powers. Our experiment results on Pentium 4 Willamette show that our power estimations agree with the measured power very well and the deviation errors are less than 5%.

## 6. REFERENCES
[1] C. Isci and M. Martonosi. Runtime power monitoring in high-end processors: methodology and empirical datar. In *the 36th Annual International Symposium on Microarchitecture*, pages 93–104, 2003.
[2] R. Joseph and M. Martonosi. Run-time power estimatino in high-performance microprocessors. In *International Symposium on Low Power Electronics and Design*, pages 135–140, 2001.
[3] K.-J. Lee and K. Skadron. Using performance counters for runtime temperature sensing in high-performance processors. In *the Workshop on High-Performance, Power-Aware Computing*, April 2005.
[4] H. Li, P. Liu, Z. Qi, L. Jin, W. Wu, S. X.-D. Tan, and J. Yang. Efficient thermal simulation for run-time temperature tracking and management. In *International Conference on Computer Design*, pages 130–133, 2005.
[5] P. Liu, Z. Qi, H. Li, L. Jin, W. Wu, S. X.-D. Tan, and J. Yang. Fast thermal simulation for architecture level dynamic thermal management. In *IEEE/ACM International Conf. on Computer-Aided Design*, pages 639–644, 2005.
[6] J. MacQueen. Some methods for classication and analysis of multivariate observations. In *Proc. Fifth Berkeley Symp. University of California Press 1*, pages 281–297, 1967.
[7] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder. Automatically characterizing large scale program behavior. In *The Tenth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 45–57, 2002.
[8] K. Skadron, M. Stan, W. Huang, S. Velusamy, K. Sankaranarayanan, and D. Tarjan. Temperature-aware microarchitecture. In *the 30th International Symposium on Computer Architecture*, pages 2–13, 2003.
[9] B. Sprunt. Pentium 4 performance-monitoring features. *IEEE Micro*, 22(4):72–82, 2002.