INSTITUTO DE COMPUTAÇÃO
UNIVERSIDADE ESTADUAL DE CAMPINAS

Power Measuring Infrastructure for Computing Systems

L. Piga *R. Bergamaschi* *R. Azevedo*
S. Rigo

Technical Report - IC-11-09 - Relatório Técnico

March - 2011 - Março

Power Measuring Infrastructure for Computing Systems

Leonardo Piga Reinaldo Bergamaschi Rodolfo Azevedo Sandro Rigo

Abstract

The demand for electrical energy has increased making it more expensive. In computing, an environment highly dependent on energy, it is important to develop techniques which allow power savings. The evaluation of power-aware algorithms requires the measurement of actual computer power. This report presents a real power measurement framework. The framework is composed of a custom made board, which is able to capture the power consumption and is installed into a commodity computer, a data acquisition device that samples the measured values, and a piece of software that manages the framework. This work shows the steps taken to develop the framework and also presents two examples of its use. The first example power profiles a small matrix multiplication program and discusses performance and energy trade-offs. The second example uses the framework to characterize and model the power consumption of a web server delivering static web content.

1 Introduction

Energy efficient computing is a major design paradigm in the current days. It is used from power profiling a program to developing power models that are able to estimate computer power consumption. In these situations, it is necessary to know the power consumption of a computer. This report presents a real power measurement framework that can be used to measure the power consumption of commodity computers in order to create power models and evaluate power trade-offs. The power measurement framework does not interfere on the computer components, presents high sampling rate, and is straightforward to use. It is composed of a custom made board that measures the computer power by using current transducers, a data acquisition device (DAQ), which converts up to 400,000 samples per second, and a piece of software that controls the framework.

Modern processors support the concept of *Dynamic Voltage and Frequency Scaling* (DVFS), which could be exploited in order to optimize power and performance [1]. The power measurement framework is useful for understanding these power states and enables the development of more power-efficient applications, which reduces computation costs and impacts on the environment. For example, if a computation needs to be done in a period of time, it may be better to execute it the fastest as possible and change processor to a power-saving state in the remaining time or it might be better to run the computation in a slower but less power-hungry state.

In the case of Data Centers, reducing power decreases the heat dissipation of the computers which makes possible to allocate more computers using less space. Another possible usage of the framework is on characterizing computer power consumption of a real application. This enables the development of power models that use Performance Monitoring Counters (PMC) as proxy to estimate power utilization indirectly.

The rest of this report is organized as following: Section 2 presents the complete infrastructure, shows a custom made board, which uses current transducers and is placed inside the computer in order to measure real power. It also explains how to use the software developed to help power-profiling programs. Section 3 exemplifies the use of the infrastructure by power-profiling a matrix multiply procedure. Section 4 shows another example that uses the power measuring framework to characterize and model the power consumption of a web server delivering static web content. Finally, Section 5 presents the conclusions and possible uses of the framework.

2 Power Measuring Infrastructure

There are several methods to measure the power delivered to a given component. Some approaches use expensive intelligent power supplies or motherboards that have embedded power meters [14]. In [7, 18], multimeters connected in series with the circuit are used for current measurements. This approach might add noise into the circuit and not be suitable for high sampling rates.

We designed a measuring device similar to the one described in [15], which uses hall effect current transducers (LTS 25-NP [8]) in series with the power lines so as to convert current to measurable voltage. The transducers we used convert current into proportional values of voltage with accuracy of ±0.2% and linearity of less than 0.1%. Figure 1 shows the V-I characteristic of the transducer.

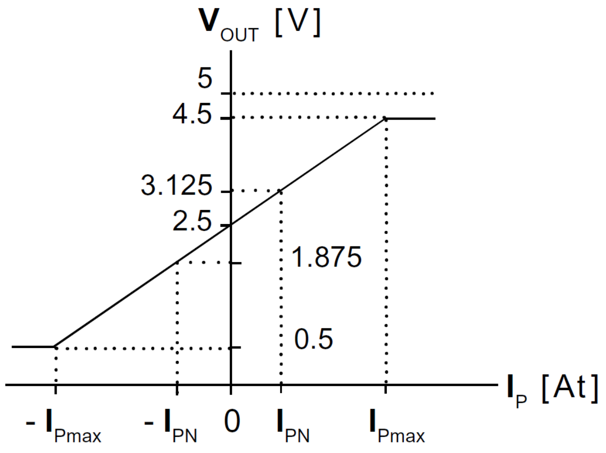


Figure 1: V-I Characteristic Curve

In the graph of Figure 1, I_{Pmax} is 80A. I_{PN} depends on the connection of the sensor sideline pins. There are three modes: in the first mode I_{PN} is 25A; in the second I_{PN} is 12V; and in the third I_{PN} is 8A. For further details on how the pins should be connected refer to the LTS 25-NP datasheet [8].

The sensors that measure the current operate on the third mode, since the maximum current that the connectors support is 6A. On the other hand, the sensors that measure hard-drive power operate on the second mode. The operating mode makes more precise the measurements in the region of $-I_{PN}$ and $+I_{PN}$, but does not limit the current to this interval. Moreover, values out of this interval are not observed during the experiments. By knowing the pins that should be monitored, as explained later, and having sensors to read the current, it is possible to put all together and to design a power measuring board.

Memory banks, CPU, and chipset are powered by the 3.3V, 5.0V and 12.0V rails. The hard-disk has an exclusive power connector. Thus, 15 current transducers (LTS 25-NP [8]) in series with the power lines are used so as to convert current to measurable voltage. Each ATX positive wire is connected in series to a sensor. Figure 2 sketches the board schematic that latter was sent to manufacturing.

In the Figure 2, the red lines are the sensor's outputs. The connectors from the power supply (i.e. ATX, CPU 2x2 pins and HD) are plugged on the power measuring board. The board outputs are connected to the respective computer devices. Also, the board has a power supply input to power the sensors on. Currents flow across the board passing through the sensors. The sensors measure the currents and output their values. Figure 3 shows a picture of the board. The board dimensions are about $30cm \times 10cm$. If you want to reproduce this board, be careful about the rail thickness. We had to apply extra tin to the rails in order to support high current peaks.

The next step consists on calibrating the sensors in order to increase the accuracy of the measurements, it is done as following: insert into the sensors known values of currents; annotate the sensor output values; after annotating the values, fit 15 curves (one for each sensor). For this process, a precise current source (Minipa MPL 3303M), which is able to generate currents from 0A to 3A, is used.

The curves are on the form $i_j = a_j * AI_j - b_j$. Table 1 shows the parameter for each sensor. The sensors support current values that may be in the intervals $[-I_{Pmax}, -I_{PN}]$ and $[I_{PN}, I_{Pmax}]$, however these values are not observed along a large set of experiments. Hence, the equations are for the interval $[-I_{PN}, I_{PN}]$.

j	a_j	b_j	j	a_j	b_j
0	20.151	50.130	8	20.187	50.209
1	13.399	33.390	9	13.341	33.175
2	13.340	33.259	10	13.264	33.015
3	13.369	33.305	11	13.320	33.123
4	13.317	33.101	12	13.396	33.293
5	13.329	33.165	13	13.302	33.073
6	13.493	33.586	14	13.435	33.476
7	13.456	33.553			

Table 1: Sensors' parameters

ATX specification [11] expects the power supply to output five voltage values: 3.3V, 5.0V, 12V, low-power -12V, and 5V$_{SB}$. This section explains each of the 24-pin ATX12V

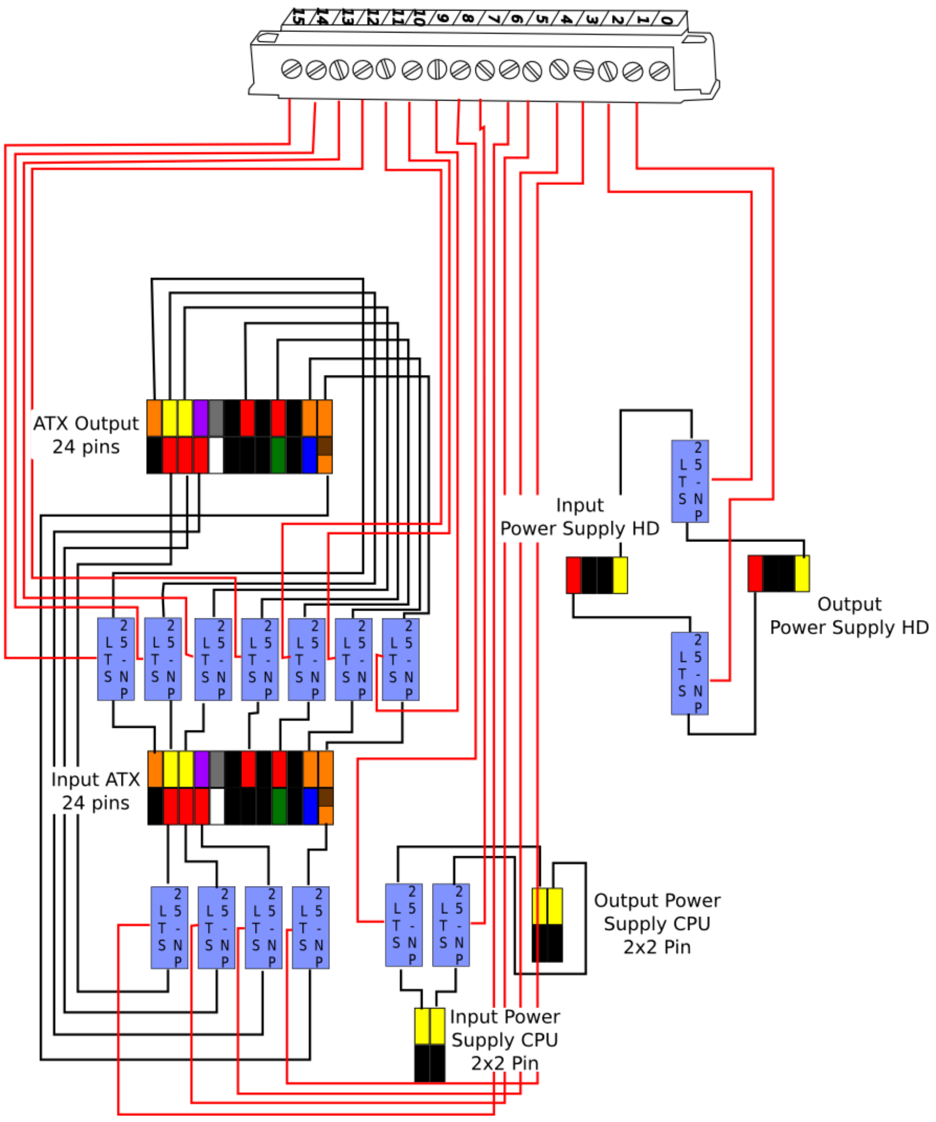


Figure 2: Board Schematic Sketch

2.x power supply connectors. It also describes the 4-pin auxiliary connector.

Early, the motherboard was energized by one 20-pin connector. In modern systems, an ATX power supply provides two connectors for the motherboard: a 4-pin auxiliary connector supplying additional power to the CPU, and a main 24-pin power supply connector, an extension of the original 20-pin version. These connectors are also known as “Molex Mini-fit Jr.”. Figure 4(a) depicts them. The orange pins (1, 2, 12, and 13) provide 3.3V. The red ones (4, 6, 21, 22, and 23) supply 5V. The pins colored as yellow have 12V on their outputs. Three pins have specific purpose:

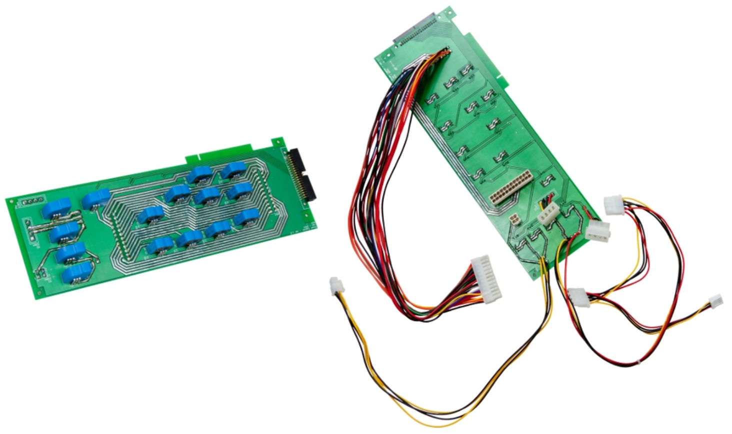


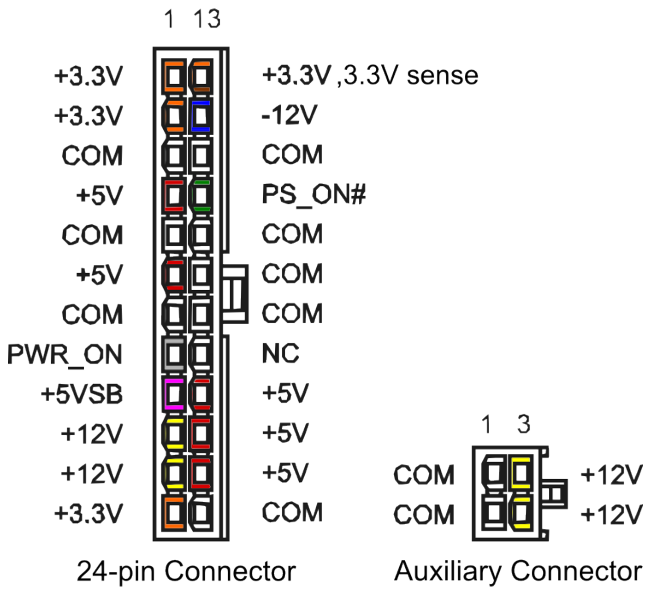
Figure 3: Picture of the Board

- **+5VBS**: This wire is a standby voltage. It can be used to provide power to circuits that need power even when the power supply is off. Features such as Wake-up on LAN can be implemented using it.
- **PS_ON#**: This pin, also called “Power ON”, is a signal from the motherboard which, once connected to ground, turns the power supply on.
- **PWR_OK**: This pin is used to indicated that the power supply outputs have already stabilized. When the voltages are above a under-voltage thresholds it signals high, when below, it changes to low.

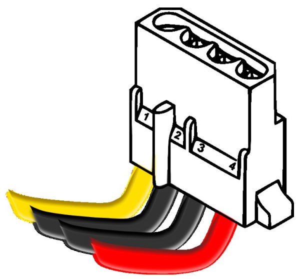
In order to supply power to the the hard-drives (HD), the “Molex 8981 Series Power Connectors” are used. Figure 4(b) shows this connector. The wire identified by number one provides 12V, number two and three are connected to ground and number four provides 5.0V.

The most important components of a computer are memory, CPU, chipset, and HD which are connected to 3.3V, 5.0V and 12.0V from the Molex connectors. Therefore, by measuring the current flowing across these wires, it is possible to calculate the computer power. As shown in Figure 5, the power supply outputs are connected to the board, and the board outputs are attached to the respectively computer components (i.e. CPU, motherboard and HD).

The board is installed into the computer (plugged into a PCI slot). ATX, HD, and CPU plugs from the power supply are connected to the board’s inputs. The outputs are connected to the power plugs of the HD, CPU and motherboard. The LTS 25-NP transducer outputs are analog. It is necessary to acquire and store these measurements. The sensor’s outputs are attached to a 16-bit data acquisition system (National Instruments NI USB–6212 [10]) that is capable of acquiring 400K samples per second across all channels. Since we need 15



(a) ATX 2.0 Connector



(b) Molex 8981 Series

Figure 4: Power Supply Connectors

channels to be sampled, the actual sampling rate is about 25K per second. However, if it is necessary, it is possible to sample only one channel at 400K. The data acquisition device is connected to another computer that, in turn, executes a monitoring application responsible for summarizing the acquired power information. Figure 5 illustrates this setup.

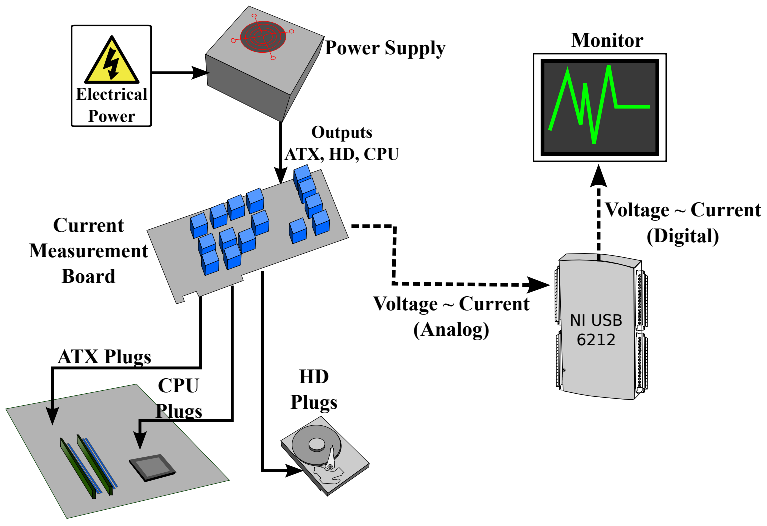


Figure 5: Power Measuring Infrastructure

The DAQ (i.e. NI USB–6212 device) has 16 input ports used to plug the signals to be sampled, these inputs are named AI_x where x is in the interval [0, 15]. We use the inputs from 0 to 14. Table 2 shows the mapping among power supply pins which is being sampled and the power supply connector.

DAQ Input	**PSU Connector**	**Pin**	**Voltage**
AI_0	HD	4	5.0V
AI_1	ATX CPU	4	12.0V
AI_2	ATX24	21	5.0V
AI_3	ATX24	13	3.3V
AI_4	ATX24	1	3.3V
AI_5	ATX24	11	12.0V
AI_6	ATX24	23	5.0V
AI_7	ATX24	4	5.0V
AI_8	HD	1	12.0V
AI_9	ATX CPU	1	12.0V
AI_{10}	ATX24	6	5.0V
AI_{11}	ATX24	12	3.3V
AI_{12}	ATX24	22	5.0V
AI_{13}	ATX24	2	3.3V
AI_{14}	ATX24	10	12.0V

Table 2: Pin Mappings

2.1 Monitoring software

This section presents the application software that runs on a remote computer identified as "Monitor" in Figure 5. The application that interacts with the data acquisition system works based on TCP/IP messages and listens on port 6790. The application is composed of seven main steps. Figure 6 illustrates the control flow. The first step is the initialization which opens a TCP/IP socket and listens for connections. Only one client is allowed at a time. The second step waits for `Start` messages. The third step is responsible for recognizing the measuring type mark, which determines the action taken when a sample is read. Then, the program flow is split up into two parts, one responsible for reading and processing the samples and other that waits for `Stop` messages. After receiving a *Stop* message, the reading process halts, the two parts are joint, and the application waits for the next message. If the next message is a `Start` the process restarts from the third step. If it is a `Quit` message, the application executes the termination steps and halts.

A starting message is followed by a measuring type mark. It determines how the samples should be stored. Some experiments need only a summarized result such as mean, minimum value, maximum value, others may need to store all samples. Currently, the application has four ways of processing the samples:

1. Read the samples, convert the sampled current values into power values, group them

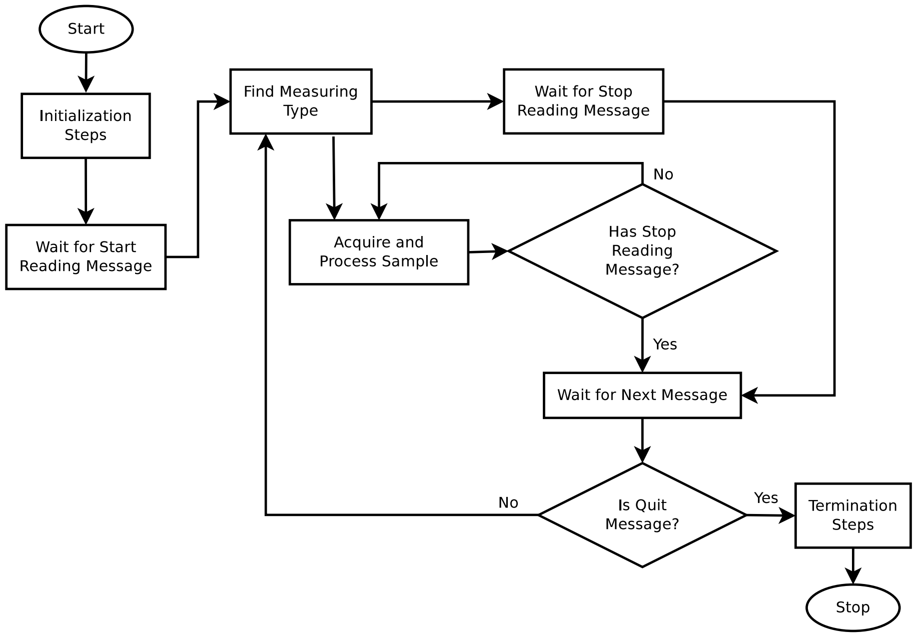


Figure 6: Application Control Flow

into CPU, HD, and miscellaneous components power, and write them into a binary file.

2. Read the samples, convert the sampled current values into power values, **group them into CPU, HD, and miscellaneous components power**, compute the average, checks for the minimum and the maximum values. After receiving a `Stop` message, these statistics are sent to the client over TCP/IP messages.

3. Read the samples, convert the sampled current values into power values, **for each sensor**, compute the average, checks for the minimum and the maximum values. After receiving a `Stop` message, these statistics are sent to the client over TCP/IP messages.

4. Read the samples, convert the sampled current values into power values, store the latter into a big buffer. After receiving a `Stop` message, the buffer is sent to the client over TCP/IP.

These operation modes are sufficient for most of the experiments that we need. For the last mode, we developed a converter that enables the signals to be displayed in a waveform. The program we used to do this operation is the GTKWave [2]. Figure 7 shows a period of time when the computer is in idle mode. If we observe the CPU signals, we can see that it displays bursts of power in intervals of 4ms. This corresponds to the system tick clock cycle which is 250Hz. This mode is interesting for using when seeking patterns along the time line.

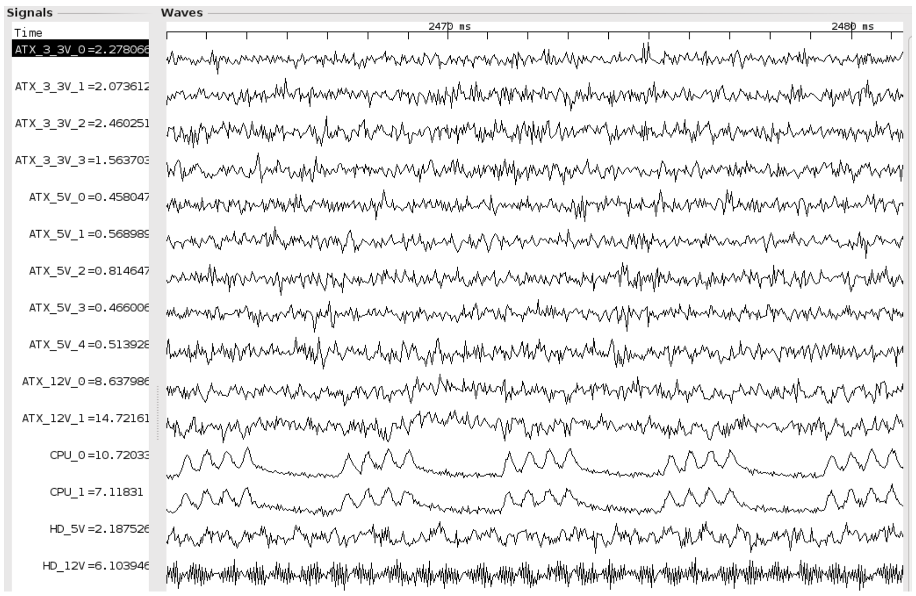


Figure 7: Idle Sample Waveform

3 Profiling a program

This section shows an example on how to use the framework in order to profile a program. It does multiplication of two square matrices. The program is parallelized using OpenMP [16] and the energy consumption for a variable number of threads is measured. For these experiments, we use a computer equipped with an Intel Core 2 Quad Q6600 with 8GB of memory and a serial ATA hard disk of 7200rpm. The operating system is the CentOS Linux 4.5 running Linux kernel version 2.6.31. The program is compiled using gcc 4.1.1 using the `-fopenmp` flag.

The elements of the input matrices are 32-bit wide. The program loads the matrices from disk to memory storing the rows side by side in memory. The pseudo-code used to multiply the matrices is given by Procedure 1. The resulting matrix whose elements is 64-bit is stored back on disk.

Since there is no data dependence among the loop iterations (lines 4 to 6), they can be computed in parallel and the final result can be calculated by adding the values of the `element` variable of each parallel computation. In C, the OpenMP API provides an easy method to do this using the directive `#pragma omp parallel for reduction(+:element)` before the for loop. This directive splits up the loop in a provided number of parallel computation. The OpenMP API provides a mechanism to dynamically control the number of threads that a loop uses on a computation without the need to recompile the program. This can be done by setting the `OMP_NUM_THREADS` environment variable. The default value is the computer number of cores.

Before entering `MULTIPLY-SQUARE-MATRIX` procedure, the program sends a `Start` message to the power measure framework software and sets it to the second experiment mode

Procedure 1 MULTIPLY-SQUARE-MATRIX(A, B, n)

```
for $x \leftarrow 0$ to $n$ do
    for $y \leftarrow 0$ to $n$ do
        element $\leftarrow 0$
        for $i \leftarrow 0$ to $n$ do
            element $\leftarrow$ element + A$[x * n + i]$ + B$[i * n + y]$
        end for
        C$[x * n + y] \leftarrow$ element
    end for
end for
return C
```

type, which calculates average powers for CPU, HD, and miscellaneous components. In addition, it saves the starting time. After computing the product, it sends a `Stop` message, receives the power summary, and calculates the elapse time and the energy consumption of the computation.

The experiment consists on multiplying two matrices of 2048×2048 elements using from one to six threads. In order to increase accuracy, it is repeated 20 times for each number of threads. Thus, the final measurements for each number of thread are the average values of the 20 runs.

In order to compare the behavior of each measure, we normalized the values with respect to the execution using one thread. Figure 8 shows the normalized energy consumption, the normalized power dissipation, and the normalized time. Using five or more threads decreases the average power. If an user needs to minimize average power dissipation, these computations would be the best choice. However, we can see that the computations which use three and four threads are the most energy saving choices.

This is a simple experiment that illustrates the use of the board and how it can be used to find trade-offs among power, energy, and performance. More variations can be tried, for example, trying to change power-saving states of the processor such as P-states[1] and finding the best configuration for a given restriction.

The board has some limitations, for example, it is not possible to profile procedures that takes approximately less than one second to execute because the communication overhead and the DAQ setup time will not guarantee that the measure actually refers to that procedure. Also, it is not possible to analyzed power consumption of an isolated component. It is very difficult to insure which power supply rail powers a certain component because motherboard manufactures do not disclose this information.

4 Power Modeling and Characterization of a Web Server

This section presents a complete example of using the power infrastructure in order to create a power model of a Web Server delivering static content. The model uses the number

[1]Each P-state designates a different operating frequency and voltage of the processor

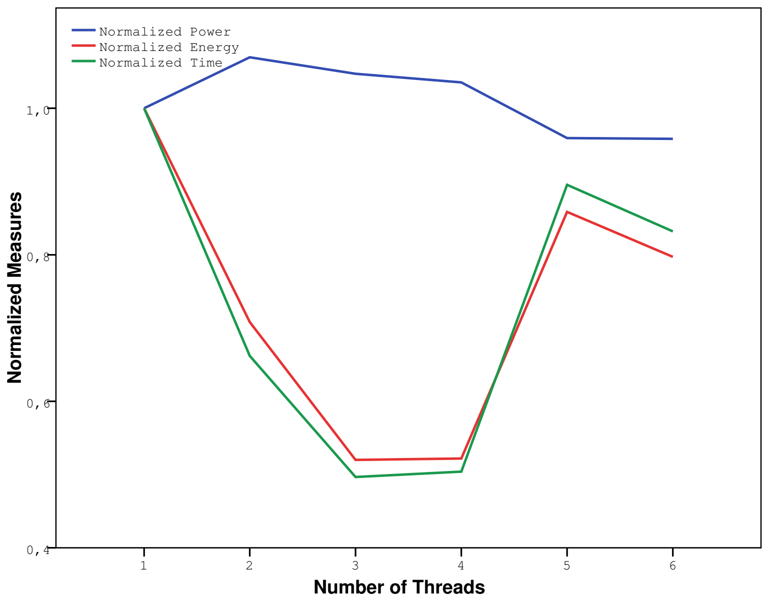


Figure 8: Power, Energy, and Time Normalized Measurements

of retired instructions per second to estimate the power consumption. Since commodity servers do not have any real power measuring device, indirectly estimation is needed when a power-aware load balance algorithm needs the power information of a server.

Previous works on power modeling use Performance Monitoring Counters (PMC) as a proxy to estimate CPU power. In [5], Bellosa showed that CPU power correlates to floating point operations, L2 cache references and memory references. His work was one of the first to propose the use of PMCs to create an energy-aware scheduler.

Some authors introduced models for estimating the power consumption of the whole system. Bohrer *et al.* [7], which focused on Web servers, proposed such a model. The authors conducted the power characterization considering a hypothetical support to DVFS (Dynamic Voltage and Frequency Scaling). They developed a Web server simulation tool which can predict the power consumption based on Web requests and CPU cycles. The adopted workload is generated from LOG files and static content. Such workload is not compatible with modern Web content, which rely heavily on dynamic content for rendering the pages. Thus, one can see that considering solely the CPU cycles is not sufficient for obtaining accurate power values when dynamic content is present.

This work follows the same path, the Web server used in the experiment is a commodity server with an Intel Core 2 Quad Q6600 with 8GB of memory and a serial ATA hard disk of 7200rpm. The operating system is the CentOS Linux 4.5 running Linux kernel version 2.6.31. The Web server software is Apache 2.0.52 [9].

HTTP requests to the Web Server are done having the number of retired instructions per second and the computer power consumption recorded. The model is for Web Servers delivering static web content, thus the assumption is that as more instructions are retired per second as more power is consumed. In order to create the points to build the model, we vary the request sizes and the number of simultaneous requests.

The points are fit and a linear regressor is used to deliver a linear model that relates instruction retired per second to power consumption. Then, a second set of points is created in order to test the accuracy of the model. The CPU task is homogeneous on a Web Server delivering static web content. The server needs to attend a request, look for the file on the disk and do the file transfer. As shown in [7], memory and disk power can be modeled as a constant value, and the major power variation is due to CPU power. Thus, instruction retired per second is a high quality parameter to be used as a power proxy.

To choose the typical file sizes, we study the 1998 World Cup Web site LOGs [3]. This LOG is composed of 250 compressed files representing 92 days of the web site access. Some of them are in the interval when the event was taking place. We study the typical request file size distribution. The LOG files recorded 13 types of files, HTML, image, audio, video, among others. The graph in Figure 9 shows the requests for different file sizes grouped in intervals of 1kB. The graph on the left-hand side corresponds to the HTML file requests, the one on the right-hand side relates to image files. A complete analysis can be found in [4]. Note that the graphs are on logarithm scale in order to improve low values visualization.

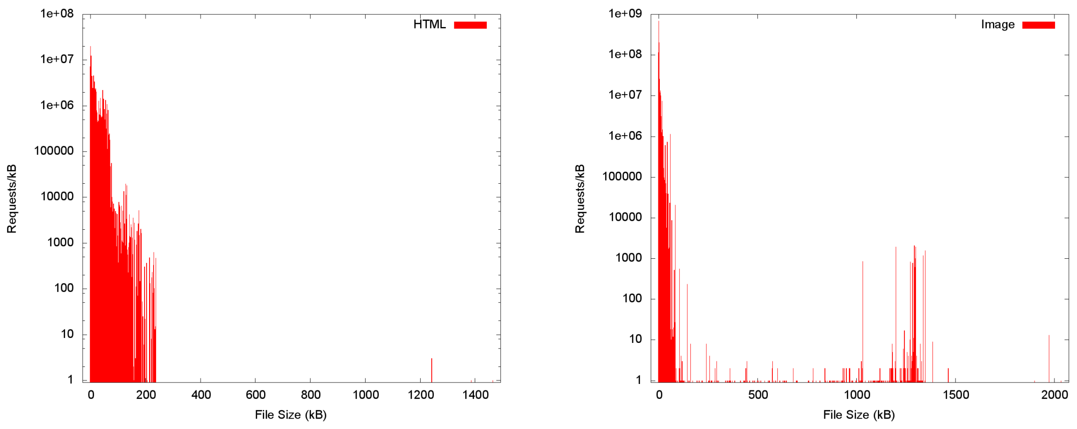


Figure 9: Requests per kB for HTML and image files

The graphs show that HTML file sizes are concentrated on a range that varies from 0kB to 200kB. For image files, due to small images used to decorate the web pages, such as borders, this is also true. However, for image files, there are a secondary group in the range of 1MB and 1.5MB. By knowing the typical file size, we create 10 files in the range from 0 to 100kB, 5 in the range from 100kB to 200kB and 3 in the range 1MB to 1.5MB. These files are requested by a client.

The requests are done while performance counter values and power measurements are collected. In order to vary the server load, concurrent requests are done. For this characterization, single requests, 10, 100, 1000, and 5000 concurrent requests are done. In this way, we are able to measure the computer working on different CPU loads.

The Performance Monitoring Counters (PMCs) are read by the `perf` [17] utility. All collected information is stored on a remote computer disk to be used by the model generator.

The process of gathering these values does not have a significant impact during power measurement [6, 12, 13]. The graphs in Figure 10 plots the server total power against billions of instructions per second for P0 and P1 CPU states.

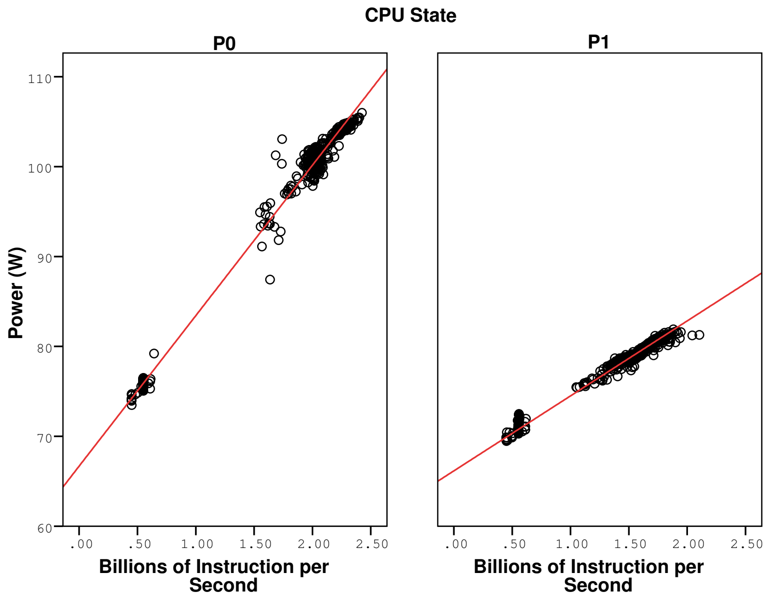


Figure 10: Power model for Static Content

The graphs in Figure 10 show that the points are placed in a linear way. Thus, a linear model is created for each CPU state using a linear regressor. The models are of form $P = a \cdot BIPS + b$, where P is the server total power, a and b are constants, and $BIPS$ is the number of retired instructions (in billions) per second. Table 3 shows these values.

Parameter	P0 State	P1 State
a	16.731	8.336
b	66.691	66.165

Table 3: Web Server Power Model

From Table 3, we can see that the constant b have similar values for both states. This happens because when CPU is idle, it is executing an infinite loop of halt instructions, which places the CPU in an energy-saving state. In addition, it is possible to observe that in P1 state the processor uses less power, although, the performance is not the same.

The next step is to test the model quality. We create a second set of measurements and compare with the model values. The model displays an average relative error of $(1.0 \pm 0.5)\%$. The maximum error is 7.7%. Hence, our model is able to estimate the computer power running a web server delivering static web content in an accurate way.

5 Conclusion

Power-profiling has become an important tool to optimize power and energy consumption of a specific task. An instrument that precisely measure real power consumption is imperative when doing these analysis. This report showed an infrastructure that is able to profile pieces of programs, programs, and even benchmarks.

The framework is composed of a custom made board which uses current transducers to measure real power. A data acquisition is used to convert analog sensor measurements into digital samples. A piece of software was developed to enable communication with the framework. It allows the insertion of test probes in anything ranging from a piece of code to a complete benchmark. Four measuring modes are already available, and more could be easily implemented.

The report also exemplified the use of the framework power-profiling a matrix multiplication routine when multiplying 2048×2048 elements matrices. The experiments showed the trade-offs among time, power dissipation, and energy consumption. Usually, when comparing project decisions, programmers only takes into account the amount of time spent on a task. This board enables them to take into account power and energy consumption as well.

Another use is to derive power models that make possible the use of performance counters in order to be used as proxy for power estimation. We show an example where a web server delivering static web content was characterized and modeled. The model could estimate the server power displaying a relative error of about 1%.

This power measuring framework will be used in more characterization and will help to create power-aware load balance algorithms for web servers. As a future work, we will characterize a web server delivering dynamic web content. And use the results for a Data Center simulator.

Appendix

A Monitoring Software Communication

This sections gives further details about how to use the monitor application. We show pieces of C code that can be used to communicate to the monitor application. We explain each of these code sections.

For communicating to the monitor application you need to include the headers that enable TCP/IP communication. In addition we need the `string.h` header in order to format the messages. The following code does this.

```c
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>
```

Next, we create a structure that stores the summarized power results. In this example we illustrate the usage where the CPU, HD, and miscellaneous components average, minimum, and maximum power are returned by the monitor application. The following structure is used to record the power values.

```c
typedef struct _powers {
  double cpu;
  double hd;
  double others;
} Power;
```

For opening a TCP/IP section to the monitor application, you need to create a TCP/IP socket, and connect to the computer that runs the monitor. The following code can be used to do this.

```c
1    // Communicates with the power measuring framework
2    portno = 6790;
3    sockfd = socket(AF_INET, SOCK_STREAM, 0);
4    if (sockfd < 0)
5      error("ERROR opening socket");
6    // Server name
7    server = gethostbyname("arvoredo");
8    if (server == NULL) {
9      fprintf(stderr,"ERROR, no such host\n");
10     exit(0);
11   }
12
13   bzero((char *) &serv_addr, sizeof(serv_addr));
14   serv_addr.sin_family = AF_INET;
15   bcopy((char *)server->h_addr,
16         (char *)&serv_addr.sin_addr.s_addr,
17         server->h_length);
18   serv_addr.sin_port = htons(portno);
19   if (connect(sockfd, (struct sockaddr *) &serv_addr,sizeof(serv_addr)) < 0)
20     error("ERROR connecting");
```

In the code above, `arvoredo` is the hostname of the computer that runs the application. The variable `portno` is initialized with 6790 which is the port used to the monitor application. Lines 13 to 18 initializes the socket structures. Finally, line 20 is used to open the connection.

To start monitoring the computer which the board is installed, it is needed to send a start message over the TCP/IP connection. The section of code bellow can be used to send the start message, setting the monitor application to summarize power of the CPU, HD, and miscellaneous components average.

```c
void sendStartMessage(int sockfd) {
  int ack = 0;
3   int n;
4   char startMessage[] = "#startSumMon\r\n";
5   char restartMessage[] = "#restart\r\n";
6
7   n = write(sockfd, startMessage, strlen(startMessage));
8   if (n < 0)
9     error("ERROR␣writing␣to␣socket");
10
11  n = read(sockfd,(char*) &ack,4);
12  if (n < 0)
13    error("ERROR␣reading␣from␣socket");
14
15  while (ack == 0xDEAD) {
16    error("Restarting");
17    n = write(sockfd,restartMessage,strlen(restartMessage));
18    if (n < 0)
19      error("ERROR␣writing␣to␣socket");
20    n = read(sockfd,(char*) &ack,4);
21    exit(2);
22  }
23 }
```

Line 4 creates a string that is sent to the monitor application, it contains the start keyword followed by `SumMon`, which defines the experiment type. As described in section 2.1, we have developed four experiment types, each one is set by a different keyword following the start message.

The data acquisition device may experience error during a experiment. Thus, after sending a start message, the monitoring application sends back an answer that indicates if it is able to start, thus line 11 reads this message and stores into `n`. Lines 15 to 20 check if the device is not able to sample the power values. If this is the case, it resets the device by sending a restart message. After executing an experiment, a stop message should be sent. This can be done by following the same steps as for sending the start message.

The next step that follows the sending of the stop message is to get the results. The code bellow illustrates how to get the average values of the board.

```
#define AVG_HD_POS 32
#define AVG_CPU_POS 40
#define AVG_OTHERS_POS 48

Power* getAvgPowers(int sockfd) {
  char* buffer;
  Power* avgPowers;
  int bufferSize = (112 + 42*3*4);
  int n;

11    avgPowers = (Power*) malloc(sizeof(Power));
12    buffer = (char*) malloc(sizeof(char) * bufferSize);
13
14    n = read(sockfd, buffer, bufferSize);
15    if (n < 0)
16       error("ERROR reading from socket");
17
18    avgPowers->hd = *((double*) &buffer[AVG_HD_POS]);
19    avgPowers->cpu = *((double*) &buffer[AVG_CPU_POS]);
20    avgPowers->others = *((double*) &buffer[AVG_OTHERS_POS]);
21
22    free(buffer);
23    return avgPowers;
24 }
```

When running the experiment that summarized the computer component power values, the result is sent over a buffer which has average, minimum, and maximum power values. The average values are stored between the 32nd byte and the 55th byte. This is done from lines 18 to 20.

References

[1] Advanced Configuration and Power Interface Specification. online. `http://www.acpi.info/spec.htm` [Accessed on 9 August 2010].

[2] GTKWave. online. `http://gtkwave.sourceforge.net/` [Accessed on 16 February 2012].

[3] The Internet Traffic Archive. 1998 world cup web site access logs. online. [Accessed on 11 March 2009].

[4] Martin Arlitt and Tai Jin. Workload characterization of the 1998 world cup web site. Technical report, Hewlett Packard, September 1999.

[5] Frank Bellosa. The benefits of event–driven energy accounting in power-sensitive systems. In *EW 9: Proceedings of the 9th workshop on ACM SIGOPS European workshop*, pages 37–42, New York, NY, USA, 2000. ACM.

[6] Ramon Bertran, Marc Gonzalez, Xavier Martorell, Nacho Navarro, and Eduard Ayguade. Decomposable and responsive power models for multicore processors using performance counters. In *ICS '10: Proceedings of the 24th ACM International Conference on Supercomputing*, pages 147–158, New York, NY, USA, 2010. ACM.

[7] Pat Bohrer, Elmootazbellah N. Elnozahy, Tom Keller, Michael Kistler, Charles Lefurgy, Chandler McDowell, and Ram Rajamony. The case for power management in web servers. pages 261–289, 2002.

[8] LEM Components. Current transducer lts 25-NP data sheet.

[9] The Apache Software Foundation. Apache HTTP. online. [Accessed on 11 March 2009].

[10] National Instruments. Bus-powered m series multifunction daq for usb - 16-bit, up to 400 ks/s, up to 32 analog inputs, isolation data sheet, 2009.

[11] Intel Corporation. *ATX Specification Version 2.2*.

[12] Canturk Isci and Margaret Martonosi. Runtime power monitoring in high-end processors: Methodology and empirical data. In *MICRO 36: Proceedings of the 36th annual IEEE/ACM International Symposium on Microarchitecture*, page 93, Washington, DC, USA, 2003. IEEE Computer Society.

[13] Russ Joseph and Margaret Martonosi. Run-time power estimation in high performance microprocessors. In *ISLPED '01: Proceedings of the 2001 international symposium on Low power electronics and design*, pages 135–140, New York, NY, USA, 2001. ACM.

[14] J.H. Laros, K.T. Pedretti, S.M. Kelly, J.P. Vandyke, K.B. Ferreira, C.T. Vaughan, and M. Swan. Topics on measuring real power usage on high performance computing platforms. In *Cluster Computing and Workshops, 2009. CLUSTER '09. IEEE International Conference on*, pages 1–8, 31 2009-Sept. 4 2009.

[15] Chris D. Lucer and Chakravarthy Akella. Power profiling for embedded applications. White paper, January 2009. `http://edc.intel.com/Link.aspx?id=1061`.

[16] OpenMP Architecture Review Board. *OpenMP Application Program Interface Version 3.0*, May 2008.

[17] Red Hat Inc. Performance counters for linux, 2010.

[18] John Zedlewski, Sumeet Sobti, Nitin Garg, Fengzhou Zheng, Arvind Krishnamurthy, and Randolph Wang. Modeling hard-disk power consumption. In *FAST '03: Proceedings of the 2nd USENIX Conference on File and Storage Technologies*, pages 217–230, Berkeley, CA, USA, 2003. USENIX Association.