Segment-Wise Timing and Power Measurement in Software Emulation

Fabian Wolf, Judita Kruse, Rolf Ernst
Institut für Datentechnik und Kommunikationsnetze, Technische Universität Braunschweig
Hans-Sommer-Str. 66, D-38106 Braunschweig, Germany
Tel: +49 531 391 3728, Fax: +49 531 391 4587
Email: $\{wolf|kruse|ernst\}$@ida.ing.tu-bs.de

Abstract

Evaluation boards are popular as prototyping platforms in embedded software development. They often are preferred over simulation to avoid modeling effort and simulation times as well as over complete hardware prototypes to reduce development cost. A particular problem arises when trying to verify the running time bounds of embedded software. The paper presents the use of a commercial evaluation board for timed trace acquisition for program segments with instruction cycle accurate power measurement on commercial evaluation kits.

1. Introduction

For formal software running time and power analysis approaches, programs can be partitioned into blocks like instructions, basic blocks or program segments. Program path analysis can deliver possible execution sequences of these blocks where longer sequences lead to higher analysis accuracy because of overlapping instruction execution. Timing and power can be measured instead of assuming worst cases [9] where the instruction sequence is not known.

The execution time and power consumption of the isolated blocks or segments can be determined by simulation using a host trace and cost tables for the single instructions [9]. While this first approach is fast but very inaccurate, cycle true simulation delivers more accurate results. This is very slow especially for power consumption. Segment-wise software emulation and measurement allows real-time execution, but it introduces other problems. The original cache behavior has to be preserved and acquired because it has a major impact. Cycle accurate power consumption needs to be measured without changing the execution context of the instructions under investigation. These requirements are impossible to meet with most existing approaches where communication of the processor with its environment via the debugger invalidates the original behavior.

Global system modeling [11] can deliver program input parameters while path analysis [9] can identify executable program segments. We use a processor evaluation kit and a logic state analyzer to measure execution time and power consumption. We extend our approach in [10] with the extraction of statistics about instruction power consumptions required by recent approaches [2, 5] and automatic back annotation to formal analysis [9].

Previous work is reviewed in section 2. Section 3 reflects formal software analysis. Section 4 and 5 explain our approach to timing and power measurement. Section 6 presents experiments before we conclude in section 7.

2. Previous Work

Evaluation kits with PCI bus access are state-of-the-art in embedded software emulation. They are often preferred to simulation since they avoid the necessity to include processor models or develop models for the embedded system environment. Accurate processor models are often hard to get and environment model development extends design time and cost. Special executables for debugging lack detailed software analysis, such as the detection of parts of a program which are executed often or consume high energy.

The BACH system [1] collects access addresses. Traces get long and are hard to back annotate. Another approach [4] instruments the code with write references to a trace memory with a timestamp. No power measurement is part of these approaches and both need dedicated hardware. Instruction level power measurement can be done with an ampere meter [6]. The board supply current is measured while the instruction under investigation is executed in a loop instead of the real program. Power consumption of the components cannot be separated. No instruction cycle accurate measurement is possible because the system lacks a time base and the possibility to store measured values.

These research approaches cannot reference the source code. Only execution time and power consumption of either the complete program or single instructions on the complete board can be measured. Caches are not present or switched off during timing and power measurement.

Recent work in system level power estimation [2, 5] uses complex gate level simulation for single instruction power consumptions to integrate these values into higher level approaches. They lack the possibility to determine the basic values by accurate measurement instead of simulation.

3. Formal Software Analysis

3.1. Static Timing Analysis

Simulation of programs with test patterns might not deliver the real upper and lower execution time and power consumption bounds because the test patterns are not guaranteed to cover the best and worst cases for complex programs with input data dependencies [9].

When control structures are depending on input data, paths for best case and worst case execution can differ from the control flow given by the test patterns. This may be solved by path analysis. It finds input data independent program segments suitable for more accurate, path based timing and power determination across control structures. It stays conservative because it is a formal approach [9] that is also valid for caches [8].

In this approach, the input data independent program segments are classified as *Single Feasible Paths (SFP)*. They can contain several basic blocks. As control flow in these program segments is input data independent, an execution of the program segment which requires segment-wise measurement delivers the one path through this program segment as well as the according execution time and power consumption. This is also valid for context dependent execution of such a program segment when input data that control flow depends on is given by so called *process modes* [3].

After the measurement of execution time and power of the different program segments a formal approach can be used to calculate the overall best and worst cases for the execution time and the power consumption. It is independent of test patterns while it is guaranteed to be conservative regarding the real bounds.

3.2. Software Power Analysis

According to [6], program segment power analysis can use the same techniques as program segment timing analysis. Program segments are executed using a tracing tool that delivers the instruction sequence. The sum of the instruction execution energies and instruction transition energies represents the overall power consumption of the program segment. While measurement already delivers more reliable results for the program segments, more accurate metrics for the energy consumptions of the single instructions and the instruction transitions can be derived.

3.3. Code Instrumentation

For program segment execution, input patterns must be selected such that all program segments are executed at least once. In a reasonable program test, the test patterns should have this property. When a program segment is not reached in execution it is extracted and executed separately. We mark the beginning and the end of a program segment under investigation to measure its execution time and power consumption. This is done with *trigger points*. Trigger points around the program segments enable us to calculate the static timing or power interval after measurement.

3.4. Requirements to Measurement

Software power consumption is usually given per instruction or basic block. As instruction execution can be dependent on input data [9] and adjacent instructions, one single value per instruction cannot be correct. Every single execution cycle needs to me measured to add to the instruction or basic block power consumption intervals. The cycle true measurement facilities have to be kept cost efficient and easy to use and access. Retargeting the concepts and software and even parts of the hardware to different target architectures should be possible.

4. Timed Trigger Point Trace

4.1. Compact Trace Generation

A trigger point is implemented by a store of the source code line number to a trigger address in a non cached part of the memory space shown in figure 1. Unlike previous work, this method allows to enable the processor cache. The trigger points are instrumented as inline assembler lines while their locations have to be a conservative selection with respect to pipeline and cache behavior as well as the global register allocation. This means the execution of the segment must not overlap with the trigger point which can be ensured by a location before and after the program segment in the source code and a flush of pipeline, caches and registers with every trigger point.

During execution, trigger points are found by monitoring the address bus with a logic state analyzer. The line numbers of the trigger points with logic state analyzer time stamps are stored instead of full address traces. The development platform reads the results from the logic state analyzer via a simple `ftp` session to back annotate the results. The execution time of the whole program and the program segments with their execution counts can be achieved.

4.2. SPARClite Example

We have implemented our methodology using a commercial SPARClite evaluation kit. The whole measurement is software controlled including board startup, logic analyzer configuration, trigger point insertion, compilation, downloading, execution and back annotation.

```
                DATA(31..0) = fileID(31..24) & line(23..0)
23 __asm__ volatile ("sta %0,[%r1]%2" : : "(0xa000018), "rJ" (8), "I" (7));
24 printData();           ADR(31..0) = base(27..0) & ASI(31..28)
25 __asm__ volatile ("sta %0,[%r1]%2" : : "(0xa00001a), "rJ" (8), "I" (7));
```

Figure 1. Trigger points for SPARClite

The implementation of the trigger points is shown in figure 1. The address bus value contains the trigger address recognized by the logic state analyzer. The data bus value consists of a file identifier and the source code line number. A hardware overview for SPARClite is shown in figure 2. Details about this experiment are presented in [7].

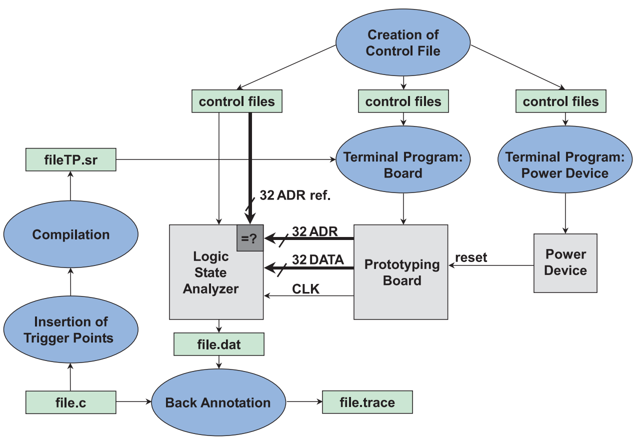

Figure 2. SPARClite evaluation kit

4.3. Limitations

For conservativity, no compiler optimization across trigger points is allowed, pipelines are flushed and the caches are reset at trigger points. Code motion of or across the trigger point has to be avoided, too. As only two trigger points are needed for a measurement of a program segment anywhere in the program, this is not considered to be a serious restriction because local optimization of the program segment is possible. In general, very few trigger points are inserted so nearly the full optimization potential is preserved.

Each trigger point implies a timing overhead when the address is written to the bus. This overhead is subtracted from the measured timing. While the shifting source code line numbers caused by trigger point insertion are taken care of in a table, the shifting addresses of the object code may be realigned to the instruction cache by inserting *nop's*. The data cache is not influenced. Even though the trace is compact, we can only analyze as many trigger points as the memory of the logic state analyzer can store. Compared to dedicated time tracing platforms, this off-the-shelf memory is easy to extend.

5. Timed Energy Samples

5.1. Hardware Setup

Processor power is measured with a shunt resistor in its power supply as proposed in [6]. For the calculation of the power consumption the voltage across the resistor is measured. We have to ensure that it does not reduce the supply voltage below the specified input level. We keep this voltage small by choosing a small resistor value, but then it is much too small for the input level needed for the analog-to-digital conversion. An analog amplifier generates the according input voltage with respect to value and offset for the analog-to-digital conversion. This delivers the power consumption at one instant in a processor cycle while power consumption during a cycle can vary a lot due to switching effects near the clock edges. For this reason, we integrate the power consumption over one clock cycle which is sampled. Trigger points give the source code reference.

5.2. Continuous Sampling

For segment-wise measurement, the power consumption for every cycle between the trigger points is read with the logic state analyzer. The address bus can optionally be read for more accurate back annotation to get a trace of power samples between the trigger points.

As at least eight bit are needed for every sample, so the trace length is limited by the logic state analyzer memory depth. For this reason, power measurement for the segment should only be started at the beginning of the program segment under investigation using according trigger points.

5.3. Power Statistics

Statistics for instruction power consumption extend [10]. The OP-code and its power consumption can be measured to derive the statistics on the host when caches are switched off. This delivers accurate metrics for instruction power consumptions that are very important in system level power estimation or low power software approaches [2, 5].

Trigger points are only needed to select the segment the metrics for the instructions are measured in. For pipelined execution, the instruction under investigation is dealt with several cycles. The execute-stage has been chosen as a reference. We have to keep in mind that cycle energies of other instructions in the different pipeline stages add to the measured energy for the whole core.

5.4. SPARClite Example

An implementation is using the SPARClite evaluation kit shown in figure 3. A 0.1 Ω shunt resistor in the power supply delivers an analog voltage of 10 to 40 mV. The MAX436 operational amplifier with 275 MHz GBP is used as the differential amplifier. The power integrator uses a 39 Ω resistor, a 10 pF capacitor and an AD8012 operational amplifier with 350 MHz GBP. The same operational amplifier is used for the preamplification followed by an NSCLC425 operational amplifier with 1.9 GHz GBP for the main amplification. An ADOP27GS decouples amplification from the analog-to-digital conversion which is using an AD9054A eight bit converter with 135 mega samples per second.

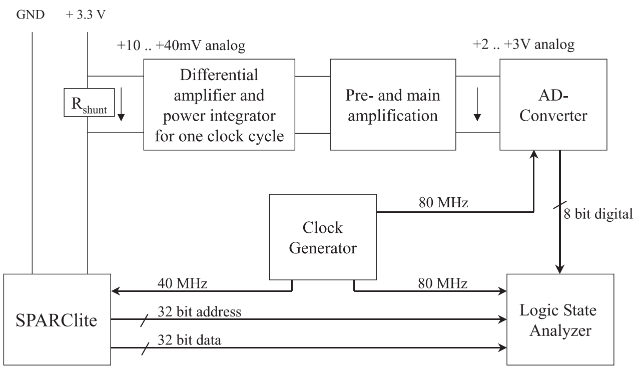

Figure 3. SPARClite power measurement

The amplification, offset and integration have to be fine tuned to permit a coverage of the full voltage range by the eight bit of the analog-to-digital conversion to achieve a wide range of digital values. The clock consists of a Philips X05860 80 MHz clock for the measurement and a 74F109N flip-flop which divides the clock to 40 MHz for the processor. The discussion of the analog circuit and its accuracy are considered to be beyond the scope of this paper.

5.5. Limitations and Generality

A precision shunt resistor for very high frequencies has been used, so the operation frequency is limited by the active components. 80 MHz clock frequency has been achieved with very cost efficient components which is way beyond the operation frequencies of previous approaches [6]. Higher frequencies can be achieved with more expensive components where measurement speed has to be traded for cost efficiency.

As eight bit for every cycle are needed when power consumption is measured, we might get problems with the logic state analyzer memory size. A sensible setting of trigger points at the start of the segment under investigation enables us to achieve metrics with the limited logic state analyzer memory. The power consumption does not need to be sampled from the start of the program. This is provided by none of the previous approaches.

The application of trigger points, logic state analyzer and segment-wise measurement is a general methodology that is not limited to the SPARClite example. Software and hardware are easy to adopt to different evaluation kits.

6. Experiments

6.1. Segment-Wise Measurement

The execution time and the power consumption of a bubble sort algorithm instrumented with one set of input data have been measured. The source code is shown in figure 4. Trigger points are specified at the beginning and the end of the program. An additional trigger point is inserted in the outer `for`-loop to demonstrate segment-wise measurement.

```
 1:  #define NUM 5
 2:
 3:  int a[NUM]={34,25,36,5,38};
 4:
 5:  main()
 6:  {
 7:    int i,j, tmp;
 8:
 9:    /* %TP : program: BEGIN */
   TP: 10, previous TP: 0
       total: n=1 t=15753600ns W=12993706nWs
10:    for (i=0;
11:         i< (NUM-1);
12:         i++)
13:    /* %TP : program: SIMPLE */
   TP: 14, previous TP: 10
       total: n=1 t= 2000.00ns W=1642nWs
   TP: 14, previous TP: 14
       total: n=3 t=13275.00ns W=10899nWs
14:      for (j=0;
15:           j < NUM;
16:           j++)
17:       if (a[i] < a[j])
18:         {
19:           temp = a[i];
20:           a[i] = a[j];
21:           a[j] = temp;
22:         }
23:  /* %TP : program: END */
   TP: 24, previous TP: 14
       total: n=1 t=1900.00ns W=1560nWs
24:  }
```

Figure 4. Source code with results

Execution time and power consumption at the first trigger point give the values for the program start. The execution time between the trigger points 10 and 24 in figure 4, i.e. the timing for the execution of the complete program without startup overhead is 1900 ns plus 13725 ns resulting in 15625 ns. The power consumption is 1560 nWs plus 10899 nWs resulting in 12459 nWs. Two values for TP 14 exist because the first iteration of the loop and the next iterations have different previous trigger points.

6.2. Instruction Statistics

The energy in nWs consumed by the processor core for each instruction is shown in table 1. The first column shows the mnemonic of the instruction followed by the maximum, the minimum and the average energy consumed by the processor core when the instruction is in the execute stage of the pipeline. It includes the energies consumed by adjacent instructions in the other pipeline stages. The last column contains the number of iterations of the program segment these statistics are derived from.

Table 1. SPARClite core energy [nWs] with the instruction in the EX pipeline stage

mnemonic	maximum	minimum	average	iterations
add	12.549	7.935	9.770	36
addx	10.307	9.876	10.163	3
and	11.601	10.436	10.824	3
b	10.005	9.833	9.890	3
bge	10.264	9.272	9.588	6
bl	11.256	10.781	10.939	3
ble	9.186	7.331	8.625	9
call	9.833	9.401	9.545	3
cmp	29.411	8.970	12.069	23
ld	11.773	7.762	9.843	61
mov	29.411	8.108	11.206	31
nop	29.627	8.237	11.792	24
or	12.420	7.504	10.353	15
sethi	11.989	7.762	10.121	16
sll	12.204	8.237	9.934	26
srl	10.566	9.315	9.775	9
st	10.393	7.418	8.764	9
sub	11.299	8.841	9.631	6
xor	11.816	10.264	10.781	3

The averages are very near to the minimum values that are very close to each other. The maximum values can differ a lot which may be caused by other instructions in the pipeline or data processing, but this only happens in few cases. These more accurate metrics can be used for system level approaches as in [6], [5] or [2]. Designing a power supply or battery for the worst case using these data would lead to unnecessary system cost. The average values or segment-wise measurement should be used in this case.

7. Conclusion

Segment-wise, real-time timing and power measurement is used for formal program analysis. The presented approach to fast and compact trace acquisition including power samples uses off-the-shelf evaluation kits. Measurement using an off-the-shelf logic state analyzer, automatic evaluation and back annotation are completely transparent to the designer working with an electronic design automation tool set. The tool set can easily be adapted to other evaluation kits.

References

[1] J. Flanagan, B. Nelson, J. Archibald, and K. Grimsrud. BYU address collection hardware. In *International Conference on Modeling Techniques and Tools for Computer Performance Evaluation*, pages 128–137, 1992.

[2] T. Givargis, F. Vahid, and J. Henkel. Instruction-based system-level power evaluation of system-on-a-chip peripheral cores. In *Proceedings of the IEEE International Symposium on System Synthesis*, Madrid, Spain, 2000.

[3] M. Jersak, D. Ziegenbein, F. Wolf, R. Ernst, F. Cieslok, J. Teich, K. Strehl, and L. Thiele. Embedded system design using the SPI workbench. In *Proceedings of the SigVHDL/ECSI Forum on Design Languages*, Tübingen, Germany, Sept. 2000.

[4] S. Lee, M. Lee, S.-L. Min, and C.-S. Kim. Timermon: A time-tracing hardware for instrumenting real-time software. In *RTCSAS*, 1998.

[5] T. Simunic, L. Benini, G. Micheli, and M. Hans. Source code optimization and profiling of energy consumption in embedded systems. In *Proceedings of IEEE International Symposium on System Synthesis*, Madrid, Spain, 2000.

[6] V. Tiwari, S. Malik, and A. Wolfe. Instruction level power analysis and optimisation of software. *Journal of VLSI Signal Processing*, pages 223–238, 1996.

[7] F. Wolf and R. Ernst. Software timing and power estimation of telecom systems. ESPRIT MEDIA published report, Technical University of Braunschweig, 1999.

[8] F. Wolf and R. Ernst. Data flow based cache prediction using local simulation. In *Proceedings of the IEEE High Level Design Validation and Test Workshop*, pages 155–160, Berkeley, USA, Nov. 2000.

[9] F. Wolf and R. Ernst. Intervals in software execution cost analysis. In *Proceedings of the IEEE/ACM International Symposium on System Synthesis*, pages 130–135, Madrid, Spain, Sept. 2000.

[10] F. Wolf, J. Kruse, and R. Ernst. Compact trace generation and power measurement in software emulation. In *Proceedings of the SPIE International Symposium on Microelectronics and Assembly*, pages 97–108, Singapore, Nov. 2000.

[11] D. Ziegenbein, R. Ernst, K. Richter, J. Teich, and L.Thiele. Combining multiple models of computation for scheduling and allocation. In *Codes/CASHE*, USA, 1998.