Compact Trace Generation and Power Measurement in Software Emulation

Fabian Wolf, Judita Kruse, Rolf Ernst
Institut für Datenverarbeitungsanlagen, 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 avoid development cost. Evaluation boards provide accurate timing results as long as the main architecture parameters match the target hardware system. For larger processors, this is often not the case since the cache and main memory architectures might differ. Another problem is the lack of observability of the software execution. Pin-Out versions of processors with improved observability are expensive (so are in-circuit emulators) and not always available, and on-chip processor test support requires software adaptation. A particular problem arises when trying to verify the running time bounds of embedded software such as required for hard real-time systems. Here, formal analysis approaches have been proposed which require segment-wise execution of a program under investigation. Another problem is the accurate analysis of processor power consumption for different execution paths.

The paper presents an approach to fast acquisition of compact timed execution traces with instruction cycle accurate power samples on commercial evaluation kits. Global system modeling abstracts the environment to a set of parameters that is included in the software under investigation for segment-wise, real-time execution. Trigger points write source code line numbers and energy samples to the address and data bus where they are read by a logic state analyzer. Experiments show that the application of trigger points avoids the acquisition of long, complete traces on sophisticated, dedicated prototyping platforms as in previous work while more accurate execution time and power consumption can be delivered.

Keywords: Software Emulation, Program Trace Generation, Software Timing and Power Measurement

1. INTRODUCTION

Embedded system running time and power analysis are required for validation and design space exploration of embedded systems. Due to the increasing amount of software on reusable target architectures, fast and accurate software running time and power analysis play an important role in VLSI system design while rapid prototyping needs early physical execution. Test patterns can instrument the program under investigation for execution while static approaches isolate executable program segments from the program.[1,2] We investigated the possibility of using commercial evaluation kits and a logic state analyzer with network access for fast and accurate acquisition of embedded system software execution time and power consumption.

For the verification of hard real-time systems, formal analysis approaches require segment-wise, real-time execution and measurement of the program under investigation. The original cache behavior has to be preserved and acquired because it has a major impact on execution time. 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 sophisticated communication of the complete processor under investigation with its environment results in unacceptable emulation times. In these approaches, major changes to the instrumented program invalidate its original cache behavior and power consumption while measurement results are only available for single instructions or the complete program. In the presented approach, beforehand system modeling[3] can deliver the relevant parameters of the surrounding components that are integrated as far as possible at compile time. This allows real-time execution and measurement of the instrumented program or isolated program segments while relevant information is stored in a very compact trace. An off-the-shelf processor evaluation kit and very little hardware for power measurement are needed. Digital values are stored by the instrumented program under test itself. A logic state analyzer with a network connection to the development platform is used to measure execution time and power consumption in a very efficient way that abstracts the board hardware to a processor simulator.

This paper is organized as follows: Section 2 reflects previous work. Section 3 introduces the code instrumentation of programs before section 4 explains our approach to measurement of execution time. Section 5 explains our approach to power measurement. Section 6 presents experiments with a commercial evaluation kit before we conclude in section 7.

2. PREVIOUS WORK

Processor evaluation kits with PCI bus interfaces are state-of-the-art in embedded system software emulation using desktop personal computers. They are often preferred to simulation since they avoid the necessity to include processor models and develop models for the embedded system environment. Accurate processor models are often hard to get and environment model development extends design time and cost. Most software development environments are tool suites with built-in debuggers that run on a host PC or work station and communicate with the evaluation kit. The compilers generate special executables for debugging. This is a cost effective approach, but it lacks the capability of detailed software execution analysis, such as the detection and analysis of ”hot spots” in the program execution, i.e. parts of a program which are executed frequently or consume high energy. Moreover, evaluation board memory architecture is often different from the target architecture. To overcome the different drawbacks, several research groups have presented different prototyping and measurement approaches with real-time execution of the program under investigation.

The dedicated BACH address tracing platform[4] is used to collect an Intel 486 specific trace. It collects all access addresses while source line numbers are needed for software emulation. Traces get long and are hard to back annotate to the source code of the program under investigation. Caches are switched off during tracing, so no relevant performance metrics can be estimated. The system also lacks a time base for the trace. As the time tracing hardware was specially built for the Intel 486 it is hard to be used for other processor types such as RISCs. Another approach with a dedicated architecture[5] called TimerMon instruments the code with write references to a special trace memory including event ID and timestamp for timing measurement when executing the program under investigation. A time base is present, but caches are not regarded in this approach. No power measurement is part of these previous approaches while both need sophisticated, dedicated hardware to store the traces.

Experiments with SPARC and DSP evaluation platforms[6] can be done in a very efficient way. Execution time and power consumption of single machine instructions are determined by executing them in a loop. An analog ampere meter is used to measure the average current in the board supply line for power measurement. The metrics are used in an instruction timing addition approach[1] extended to an instruction energy addition approach. No remote access for starting the measurement or automatic evaluation and exploitation of the measurement results is possible. The supply current of the complete board is measured, so the power consumption of the processor cannot be separated from that of the memories and the other components on the board. No instruction cycle accurate measurement is possible because the system lacks a time base and the possibility to store measured values after a trigger definition. Instruction energy consumptions are measured in a loop instead of the real context of the program, thus the values are extremely inaccurate, especially when caches, pipelines and global register sets are present. This is a common case with RISC processors resulting in overlapping instruction execution. As the system also lacks the possibility to store a power trace, the measurement of a complete program or program segment is not possible.

These research approaches cannot automatically reference the line numbers of the source code. Often only execution time and power consumption of either the complete program or single instructions on the complete board under investigation can be measured while the delivered trace only contains an address sequence without execution time, power consumption, cache behavior or references to program segments in the source code. Caches that have a substantial effect on execution time and power consumption are either not present or switched off in the presented approaches. This can be overcome by using fast cache simulation[7] but none of the reviewed approaches can deliver accurate results for execution time and power consumption of program segments on more complex processor hardware.

3. CODE INSTRUMENTATION AND PROGRAM PARTITIONING

The software under investigation usually contains control structures that depend on input data while measurement needs an execution of the program. For this purpose, a test pattern representing one possible input data set can be given leading to one execution and according measurement of the program that is only valid for the test pattern. This enables us to use our approach for the complete program. Segment-wise execution time and power consumption are needed as a base for static validation of real-time embedded software,[2] so every segment needs to be executed. We need to mark the beginning and the end of a complete program or a program segment to measure its execution time and power consumption. This is done with **trigger points**. These points mark the source code to be measured and can be recognized with the logic state analyzer connected to the processor bus. The beginning and the end of the program segment under investigation is marked. Trigger points are implemented by store instructions to non-cached memory spaces. Details are explained in section 4.

Test pattern might not reflect best or worst cases of running time or power consumption. Static analysis explores feasible and infeasible execution paths[1,2] and designer interaction.[8] Program segment execution time and power consumption can be measured with our approach. In the following, we present different possibilities for the instrumentation of the complete program or the program segments under investigation.

3.1. Instrumentation of the Complete Source Code

For timing and power measurement on the evaluation kit, the program can be executed as a whole using test patterns or any set of input data. Where control structures are depending on input data, paths for best case and worst case execution may differ from the control flow given by the test patterns. This may be solved by path analysis[8,2,1] which is beyond the scope of this paper. As long as test patterns cover every program segment, we mark the beginning and the end of a program segment with a trigger point. Execution of the program segment permits the measurement of its execution time and power consumption that is a base for the calculation of the worst case and best case execution time or power consumption bounds of the feasible paths. This results in the upper and lower bounds for the execution time and power consumption of the complete program.

3.2. Instrumentation of Isolated Program Segments

If program segments are not reached in the execution of the instrumented program because test patterns do not cover all potential execution paths, while the standalone execution of the missing program segment is possible, they can be extracted and be executed separately. Therefore, they are loaded to the base address they would be mapped to when the complete program is given to the linker. The map file delivers this information. The used variables must be initialized to avoid, for example, divide-by-zero effects. A standard data flow analysis[9] provides the variables to be declared and initialized to their start values to make physical execution of the program segment possible.

3.3. Simulation of Program Segments

The program segment or complete program can be simulated as a reference solution using given input data and a cycle true processor model,[10] which can exactly deliver processor execution time or power consumption. This can be any well established, off-the-shelf processor simulator provided by the processor vendor. As an example for cycle true processor simulation, that delivers the execution time or power consumption of the program, a StrongARM simulator core is combined with the DINERO III cache simulator[7] delivering both instruction and data cache behavior. Source codes have been recompiled to one simulator. Architecture modeling regarding execution time is given by the Cygnus simulator while the energy dissipation model[6,11] is directly based on the execution time in a first approach. Compared to the evaluation kits, simulation is slow, inaccurate or the simulator might not be available at processor release time, so the application of software emulation is feasible.

3.4. Execution on the Evaluation Kit

After the code under investigation has been cross-compiled for the processor on the evaluation kit it is downloaded to its memory. This is the only down link communication, the execution runs in real-time during the measurement. If different processes need to be measured at once, a real-time operating system, RTOS, that allows unrestricted memory access should be installed on the evaluation kit.

4. TIMED TRIGGER POINT TRACE

When programs or program segments can be executed by code instrumentation or path extraction, the trace and the resulting execution time and power consumption of the executed code can be obtained by measurement. Previous approaches store the full address sequence of data and address bus which leads to memory consuming program traces. We reduce the trace of the program by using **trigger points**.

4.1. Compact Trace Generation

Trigger points mark the beginning and the end of a program or a program segment that we need to measure the execution time for. Relevant program segments can be determined with the SYMTA path analysis tool suite.[2,1] If we only need the timing for the segment, we trigger a clock that is using the system clock of the board when we pass the first trigger point and stop it when we pass the last one. Intermediate program behavior is not needed. The trigger points have to contain information about the source code lines the program segments begin and end at to permit a back annotation of the timing to the program under investigation. In our approach, several steps are needed to implement a fully automated measurement and back annotation of results for execution time and power consumption to the source code.

1. Trigger points need to be inserted into the source code. To preserve the program structure with automatic trigger point insertion, every block belonging to a control structure needs to be set in curly braces { }. This is done in a recursive descend. After that, trigger points are automatically inserted into the code. A table with source line cross references is built for later back annotation.

2. A trigger point is implemented by a store of the source code line information to a defined trigger address in a non cached part of the memory space. An architecture specific example is given in figure 1. Unlike previous work, this method allows to use the processor cache. The store instructions are instrumented as inline assembler lines to the C source code. Assembler instruction and trigger address can easily be changed to various architectures while the trigger point locations have to be a conservative selection with respect to pipeline and cache behavior as well as the global register allocation.

3. During execution, the trigger points are found by monitoring the address bus with a logic state analyzer. The associated data is stored, containing the current source line number. The line numbers of the trigger points with corresponding logic state analyzer time stamps are stored instead of full traces. Address and data bus can either be measured at extra connections at the processor pins or by connecting the probe to a bus slot of the evaluation board. The time base for the offset between the trigger points is calculated by the logic state analyzer using the board clock, not by dedicated hardware like in previous approaches.

4. The development platform reads the results from the logic state analyzer via a simple `ftp` session. This raw data can be evaluated in many ways. Execution time of the whole program and execution counts plus execution time of program segments between trigger points can be achieved. An example of an instrumented program and the resulting intermediate logic state analyzer format can be seen in figure 6. Best case, worst case, first or last executions of the program segments marked by trigger points can be computed as well as the averages.

4.2. Exploitation of Trigger Points and Back Annotation

In a first step, we have to mark the source code lines that we want to measure the execution time and power for. This needs a file denominating these lines in ascending order separated by `NEWLINE`s, called *definition file*. We define a format for a *trigger point comment* that supports the direct determination of the source lines.

`/*` *TPcommentID: TPdefinition1, TPdefinition2, ..* `*/`

TPcommentID: `%TP`
TPdefinition : TPname: TPkind
TPname : string
TPkind : `BEGIN | SIMPLE | END`

A trigger point comment is composed of one or more *trigger point definitions*, each of which consists of a *trigger point name* and a *trigger point kind*. Any string constant can be chosen for a trigger point name while all trigger points with the same name are collected in a *trigger point class*. The first trigger point of a class will be marked as `BEGIN`, the last as `END` and the others as `SIMPLE`. But it is also possible to set all trigger points of a class to the kind `SIMPLE`. The trigger point comment should exactly be written in the line *before* the line of interest. Only for an `END` point it makes sense to set it `behind` the last instruction of interest. A tool reads the source code with trigger point comments and creates the definition file, according to the coded information and the given trigger point class. It is possible to create the definition file with other tools or even manually. As we insert additional source code into a program, we have to preserve the consistency of the program. Control structures like `if-else` permit the writing of single dependent C instructions without applying braces. In the case of triggering on such single dependent instructions, we have to insert additional braces, because now there are two instructions depending on the same control structure, namely the inline assembler line and the instruction of interest. We set braces around every instruction block using a recursive descent.

The next step is the insertion of the inline assembler lines. The insertion is done by reading the prepared C file and by writing the architecture specific inline assembler lines to the line numbers, which are denominated in the definition file. The cross references of the original, the prepared and the trigger point file line numbers are stored in a table file. Finally the executable program is compiled by a standard C compiler. We automatically generate different control files for the logic state analyzer and the power device. Other control files are needed as input files for the terminal program of the evaluation kit. The measurement starts with a complete reset of the board by the software driven power control. The executable program is downloaded to the evaluation kit and the setup program containing the trigger sequence is loaded to the logic state analyzer. After program execution the logic state analyzer is stopped, and the results are transfered from the analyzer to the host.

At last the results are evaluated on the host. As the same trigger point can be reached on different paths, it has to be analyzed in relation to the others. If there are more iterations, for example when measuring a loop, the results are given for the total number of executions. They are enumerated for the first and the last, the worst and the best cases. It is possible to mark a special iteration, too. Finally the setup overhead at program start as well as the number of trigger points passed are listed.

4.3. Example: Timing Measurement for a SPARClite RISC Processor

Execution time and power consumption of program segments have been measured on a commercial SPARClite evaluation kit by inserting trigger points. The Cygnus C compiler has been used for assembly code generation. In the current setup we are using DRAMs which prevents us from separating cache misses from memory refresh as we cannot access the memory controller, but this can be overcome by analyzing the refresh strategy or selecting SRAMs when choosing an evaluation kit.

After program completion the results are back annotated to the source code. The whole measurement is software controlled including board startup by switching on its power supply from the network, trigger point insertion, compilation, downloading, execution and back annotation. An overview for the SPARClite evaluation kit and its setup is shown in figure 2. The terminal program only writes the instrumented executable program to the board, which is a fast one way communication. The program is executed in real-time before the results are read from the logic state analyzer memory.

The implementation of the trigger points for the definition of the beginning and the end of a program segment or the complete program for SPARClite is shown in figure 1. The address word contains the trigger address that is recognized by the logic state analyzer. The data bus value consists of a file identifier to distinguish between different sources. It is followed by the source code line number that is also read by the logic state analyzer. For power measurement, this trigger point is extended in section 5. The access address as well as the logic state analyzer software can easily be modified for an adaption to different evaluation kits.

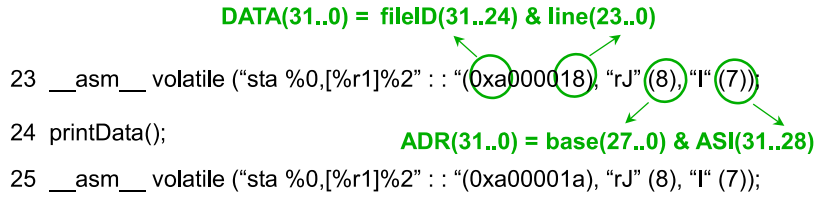

Figure 1. Trigger points for SPARClite emulation boards

Any commercial evaluation kit with bus access can be used for this purpose. Our current work in the prototyping area focuses on software controlled execution time and power measurement on a StrongARM evaluation kit. The automation abstracts the evaluation kit to the same level as a software simulator for the architecture modeling of a program segment.

4.4. Limitations

The insertion of trigger points into the source code modifies the program under investigation. This implies some overhead and limitations regarding measurement precision caused by the insertion of trigger points.

- We want to keep the modifications conservative, so no global compiler optimization across trigger points is allowed, pipelines and registers are flushed and the caches are reset at trigger points. Compiler optimization shifting the code line of the trigger point has to be avoided, too.
- Each trigger point implies a timing overhead when the address is written to the bus. The number of cycles depends on the architecture. This overhead can easily be subtracted from the measured timing to get the real execution time.
- 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 through the insertion of store instructions for the trigger points have an unpredictable effect on pipeline and cache behavior. For this reason, we try to keep the modification through trigger point insertion as small as possible by inserting trigger points only at the beginning and the end of the program segment under investigation. Reducing the number of trigger points does not only help to keep traces as compact as possible but it also leads to higher measurement precision regarding the influence of the trigger points to the system under investigation.
- Another limitation is the memory depth of the logic state analyzer. Even though the trace is very compact, we can only analyze as many trigger points as the memory of the logic state analyzer can store. In comparison to dedicated time tracing platforms, this off-the-shelf memory is easy to extend.

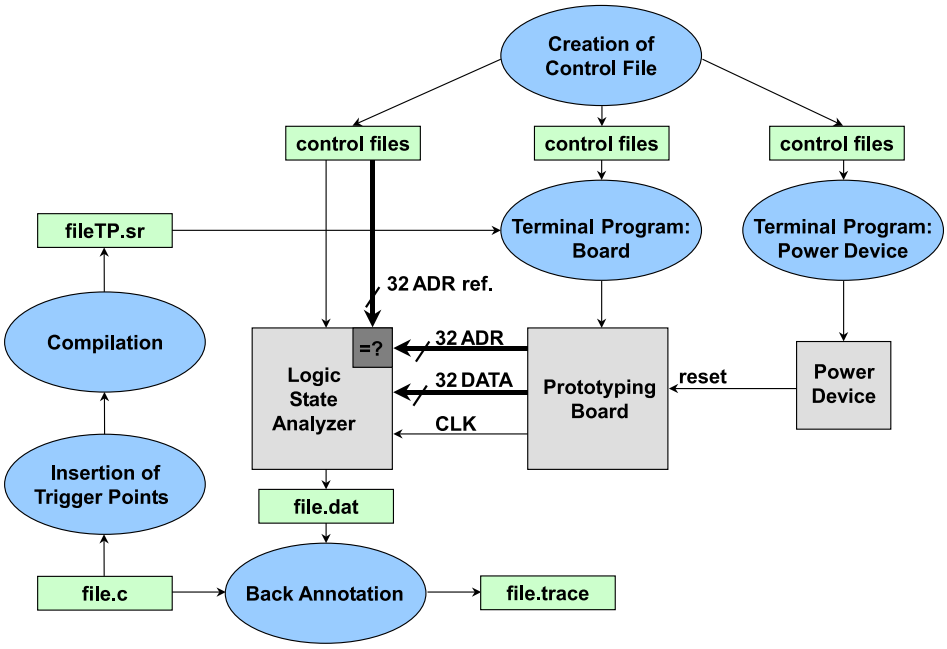

Figure 2. SPARClite evaluation kit with logic state analyzer and host functionality

5. TIMED ENERGY SAMPLES

After successful implementation of the compact time tracing we extended our approach to power measurement. This extension can deliver the energy consumption of program segments or a small complete program. The instrumentation, the preparation and the insertion of trigger points into the program to mark the beginning and the end of a program segment under investigation are the same as introduced in section 4.

5.1. Hardware Setup

First of all, we need to select core and bus frequency we want to measure at because both influence the power consumption of board and processor. Then, a decision whether to measure the power consumption of the complete evaluation kit including processor, memories and interfaces or of the isolated processor has to be made. Measurement of the complete board power consumption is far easier as we only have to access the power connector. For isolated power measurement of the processor we have to access its on-board power regulator output current.

Power is measured by inserting a shunt resistor into the power supply line. For on-board access, the resistor has to be mounted into the power supply under investigation. For the calculation of the power consumption the voltage across the resistor is measured as we can see for SPARClite in figure 3. The setup is valid for most evaluation kits. We have to ensure that the voltage across the resistor does not reduce the supply voltage below the specified minimum input level of board or processor. This means we have to keep this voltage as small as possible 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. As amplification and analog-to-digital conversion have to be done at processor speed, we need very fast components and limit the measurement to eight bit which is accurate enough in a first approach.

This kind of measurement delivers the power consumption at one point 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 a complete clock cycle which is sampled. This means we get one value for the energy consumed in the cycle with each activation of the analog-to-digital conversion. The logic state analyzer reads the digital output in parallel to the address bus, so a reference to the source code is given that can be used for sophisticated exploitation of instruction level power consumption. As we get one sample for every cycle, the influence of adjacent instruction cycles can be investigated. In other words, supply

current results in a voltage across the resistor which represents the instantaneous power consumption. This is integrated to achieve the energy consumption of an instruction. In the following, the term *power consumption* is used because it represents all these metrics, actually meaning the energy consumption in the cycle.

5.2. Single Shots

Trigger points lead to a measurement of the power consumption of the cycle the instrumented instruction is executed in. This means there is absolutely no reference to the program under investigation because we measure the power consumption of the inserted trigger point. This is still necessary to be able to subtract its overhead. For measurement of single cycles, the sampling of power has to be delayed at least one cycle beyond the trigger point, but its influence is still present. Measurement of single cycles does not make much sense in energy measurement for program segments or complete programs. It is only needed to measure the power consumption of specific instruction sequences[6] to build up a table based approach.

5.3. Continuous Measurement

For program segment measurement, at least one sample of the integrated power consumption per clock cycle is needed. Measurement is started and terminated with a trigger point like in trace acquisition. Between the trigger points only the sampled power consumption for every cycle is read with the logic state analyzer. The address bus can optionally be read for more accurate back annotation. This means that we get a trace of power samples between the trigger points, so we can assign the power consumption to a program segment. It delivers very accurate power consumption measurements compared to previous approaches while very little hardware overhead is needed. As we directly measure the power consumption of the instructions in the environment they are executed in instead of loops, their original power consumption is delivered. The same trigger points are used for timing and power measurement, so the same mechanisms for the result calculation like in section 4.2 can be used.

While single shots are needed for a table based approach, continuous power measurement can be applied for design space exploration of different implementations of program segments regarding their coding influence to power consumption.

5.4. Example: Power Measurement for SPARClite

We implemented the power measurement approach using the SPARClite evaluation kit that we already applied for compact timed trace generation. Figure 3 shows the hardware setup to measure the power consumption. The cycle energies measured between the trigger points are added to the total energy consumption for the trigger points. These values can be exploited the same way as the execution times that we explained in 4.2. The core frequency, being twice the bus frequency, is generated from the processor itself. This means we need two clocks, one for the bus clock of the processor and one for the measurement which must be the same as the core frequency.

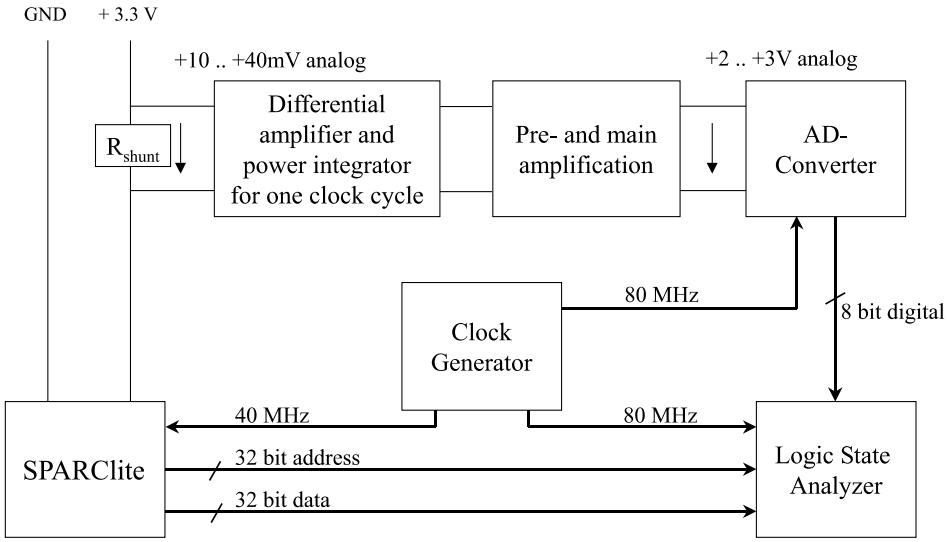

Figure 3. Power measurement setup for SPARClite

In order to permit an 80 MHz operation of the power measurement, fast components for amplification, integration and analog-to-digital conversion are needed: A precision 0.1 Ω shunt resistor in the power supply lines delivers an analog voltage of 10 to 40 mV. This ensures correct operation of all components on the board because the remaining supply voltage is still way above the minimum. The MAX436 operational amplifier with 275 MHz gain bandwidth product 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 to gain a level suitable for analog-to-digital conversion. An ADOP27GS decouples amplification from the analog-to-digital conversion which is using an AD9054A eight bit converter with 135 mega samples per second for fast operation. The amplification, offset and integration have to be finetuned 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 measured values. The clock generator 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.

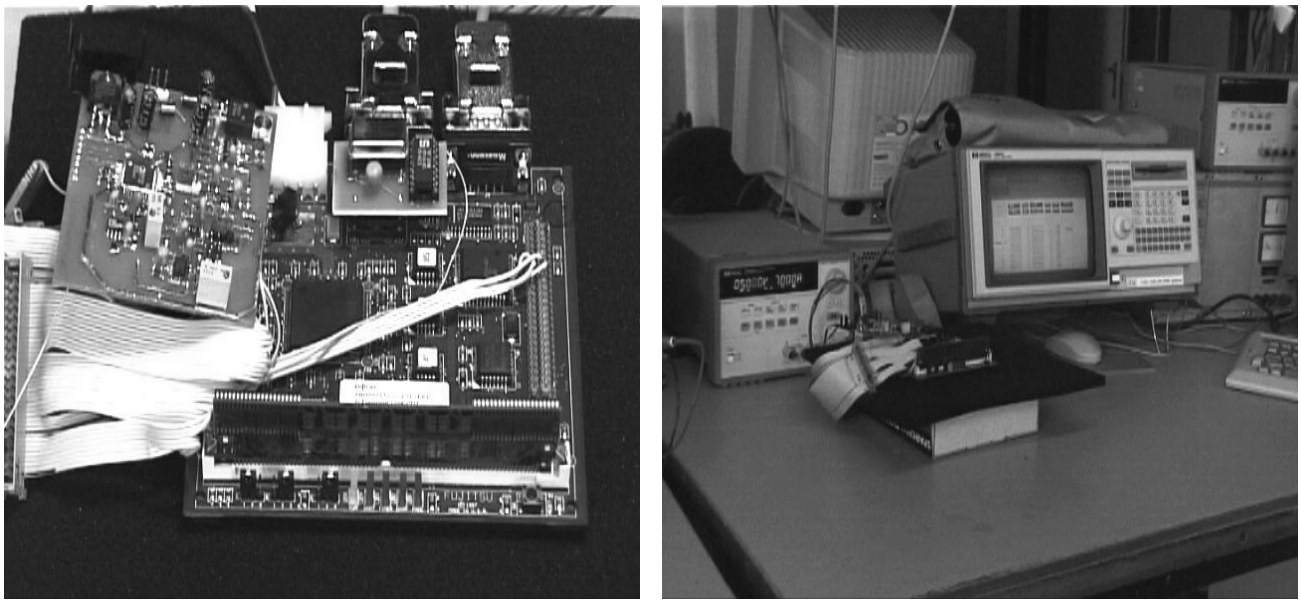

Figure 4. SPARClite evaluation kit with measurement facilities

Figure 4 shows the SPARClite evaluation kit with the measurement facilities and the logic state analyzer which is connected to the local network. The small PCB on the left contains the power measurement while only the evaluation kit and the logic state analyzer are needed for the trace acquisition. Figure 5 shows the frequency diagram of the whole power measurement circuit. The amplification is nearly constant until we reach the maximum frequency of the circuit which is between 80 and 90 MHz that we can see in figure 5.

5.5. Limitations

For the trigger points, the same drawbacks apply as for the acquisition of the program segment execution time. The following problems are additionally introduced by power measurement.

- If the power consumption of single cycles shall be measured, trigger points have a particular influence because the change in the program is happening very near to the cycle under investigation. These local effects can be overcome by continuous measurement of the segment with the instruction under investigation and the address bus. This delivers a reference to the instruction while the trigger point can be inserted some cycles in advance. For most RISC processors, there is no possibility to read the processor and pipeline status, so there may be no exact coherence between the power trace and the assembler program while pipelined execution also influences the results for the power consumption of single instructions.

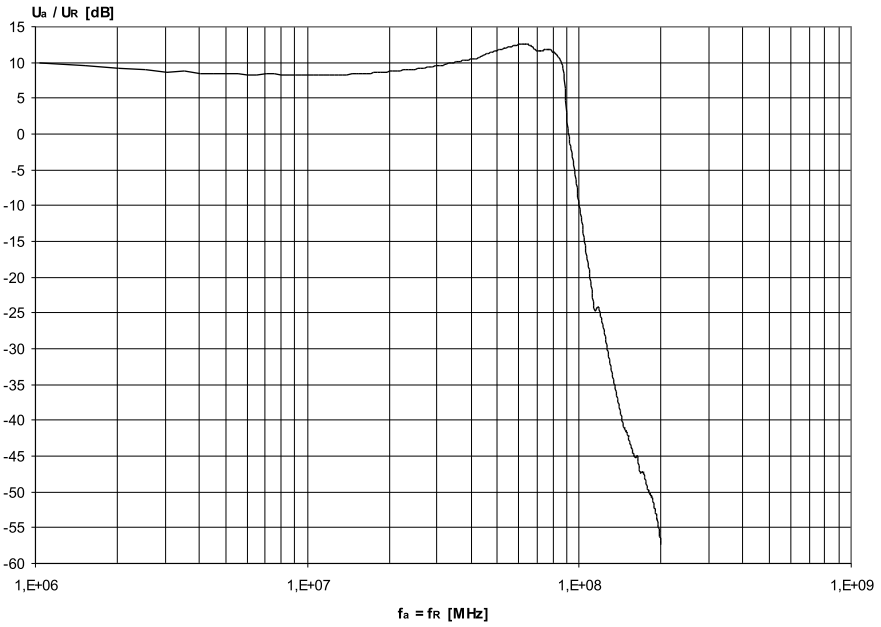

Figure 5. Frequency diagram of the power measurement

- All the measurement hardware is dependent on the processor speed of the board because the time constant for the integration has to be set with a resistor and a capacitor on the board. These have to be changed to adopt the system to other processor speeds. It is no serious restriction because the change is easy to do while execution time and power consumption for other processor speeds can directly be calculated from the speeds we have done our measurements at.
- As eight bit for every instruction cycle are needed when power consumption is measured, we might run into problems with the logic state analyzer memory size. It is more severe than for pure trace acquisition where only the trigger points need to be stored. This problem may be overcome with source code partitioning using path analysis[2] and measurement of shorter program segments that allow to fit the power trace into the logic state analyzer memory.

5.6. Table Based Power Estimation Tool Suites

When execution times and power consumptions of single machine instructions as well as timing or power penalties for transitions between certain instructions have been determined by measurement, these values can be used for the tool based prediction of execution time and power consumption.[6] This is done in an instruction cost addition $(ICA)^2$ approach. Then, traces are identified running the code on a host processor while the instruction execution cost like execution time or power consumption is taken from a table delivered by beforehand measurement. The problems for the measurement of single shots apply.

5.7. Global Access

The possibility to connect the logic state analyzer with the measured execution time and power consumption metrics to the local network with simple `ftp` or `telnet` sessions allows a very flexible integration of the presented prototyping approach into any timing or power estimation tool suite. Software driven remote power control of logic state analyzer and evaluation kit as well as shell script driven measurement allow to simply specify where trigger points are set into the source code, start the measurement and read the results. This even permits Internet based measurement with a standard web browser running on top of the application assuming a security mechanism is implemented.

At present, the SPARClite evaluation kit and the logic state analyzer are part of the local network and can be accessed via Ethernet. Current work deals with the connection of experimental setups including a research platform and measuring devices to the Internet. The implementation of a web browser user interface is within the scope of this project, which provides an Internet based hardware and software design environment. Both scheduling techniques and security mechanisms are hot topics.

6. EXPERIMENTS

In the following, experiments for compact trace acquisition and results for timing and power measurement of program segments and complete programs are presented. Experiments for timing and power measurement are done in parallel because the measurement of execution time does not increase complexity when trigger points are added for power measurement.

6.1. Case Study: Bubble Sort Algorithm

In a case study, the execution time and the power consumption of a complete bubble sort algorithm instrumented with one set of input data have been measured, no best case or worst case bounds have been determined. The source code is shown on the upper left side of figure 6a. Trigger points are specified at the beginning and at the end of the program to demonstrate the measurement of the complete program. An additional trigger point is inserted in the outer `for`-loop to demonstrate multiple iterations across a trigger point. Results are back annotated to the source code where the trigger points are inserted. The different tables in figure 6 show the following: In *a*, the source code with the trigger point definitions is shown which is translated to a source code with additional braces { } and inline assembler instructions in *b*. In *c* we can see the trigger point definition file while the table in *d* keeps track of the shifting source code references. An intermediate format of the values measured with the logic state analyzer is given in *e*. The first column shows the trigger point number followed by trigger address, file ID, source code line number extracted from the trigger point and the timing offset in *states relative*. The source code with the results is shown in *f*. Back annotated results always reflect the relative values to the last trigger point. The number of iterations across the trigger point is given at the very beginning of the line followed by the execution time and the power consumption. 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 6f, i.e. the timing for the execution of the complete program without startup overhead is 1900 ns at TP24 plus 13725 ns at TP10 resulting in 15625 ns for the total execution time. The power consumption is 1560 nWs plus 10899 nWs resulting in 12459 nWs for the overall power consumption. Depending on the setting of trigger points, measurements for program segments like the inner `for`-loop are possible.

Further experiments for table based software power prediction as explained in section 5.6 and presented by Tiwari[6,12] are straightforward to implement while we focus on direct measurement of program segments because our prototyping platform and the measurement facilities are easy to operate due to the remote control.

6.2. Application in a Static Software Estimation Approach

In the following experiments, the SPARClite evaluation kit was integrated into a timing and power estimation tool suite.[2] No test patterns for code instrumentation are inserted but the conservative worst cases for execution time and power consumption are determined using path analysis. The tool suite first determines (in-)feasible paths in a program that can be executed on the evaluation kit. This delivers worst case execution time and power consumption of the program segments. The worst case path through the program is determined to deliver the overall worst case for execution time and power consumption.

Using this methodology, measurements for execution time and power consumption of program segments with the SPARClite evaluation kit were done as presented in table 1. Here we investigated execution time and power consumption of the top level of an ATM switch component. Table 1 gives the upper bounds for execution time and power consumptions of the operation administration and maintenance (OAM) component using different path analysis techniques. Simulation results for program segments using StrongARM are compared to our SPARClite example where the prototyping approach is using the compact trace acquisition and power measurement for the program segments. In the first approach by Malik,[8] measured program segments consisting of one basic block[9] each are shorter than in the following two approaches because of a missing exploitation of program paths. This means more trigger points and the according overheads regarding conservative approaches for pipeline and cache states are part of the program leading to higher overestimations. In the second approach by Ye and Ernst,[1] some basic blocks are joined to single feasible paths (SFP), so fewer trigger points are needed resulting in less overestimation. In the third approach by Wolf and Ernst,[2] even fewer trigger points are needed because of the exploitation of a more accurate path analysis resulting in fewer, but longer program segments containing context dependent paths (CDP) with less overhead. The exact case is determined by simulation or measurement of the complete program using worst case input data as test patterns. It serves as a reference to compare the overestimations of the conservative static approaches.

We notice that overestimations with the evaluation kit are higher than when using simulators, especially when more program segments are involved. The reason is the more conservative handling of program segments by using trigger points compared to the use of a cycle true processor simulator. This means the application of path analysis in conservative static approaches is very important when using evaluation kits for architecture modeling because it reduces the number of inserted trigger points and the according overhead. On the other hand, we have shown that our approach is suitable for segment-wise execution time and power measurement that is needed in static approaches.

a:Instrumented source code with comments

```
 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 */
10:    for (i=0;
11:         i< (NUM-1);
12:         i++)
13:    /* %TP : program: SIMPLE */
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 */
24:  }
```

b:Source Code with braces and trigger points

```
 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 */
10:    __asm__ volatile ("sta %0,[%r1]%2" :\
       : "r" (0x0000000a), "rJ" (8), "I" (7));
11:    for (i=0;
12:         i< (NUM-1);
13:         i++)
14:    /* %TP : program: SIMPLE */
15:     {
16:    __asm__ volatile ("sta %0,[%r1]%2" :\
       : "r" (0xa000000e), "rJ" (8), "I" (7));
17:      for (j=0;
18:           j < NUM;
19:           j++)
20:       {
21:        if (a[i] < a[j])
22:          {
23:            temp = a[i];
24:            a[i] = a[j];
25:            a[j] = temp;
26:          }
27:       }
28:     }
29:  /* %TP : program: END */
30:  __asm__ volatile ("sta %0,[%r1]%2" :
     : "r" (0x00000017), "rJ" (8), "I" (7));
31:  }
```

c:Trigger point definition file

```
 10
 14
 24
```

d:Table file with cross references

original	prepared	TP
1	1	1
2	2	2
3	3	3
4	4	4
5	5	5
6	6	6
7	7	7
8	8	8
9	9	9
10	10	11
11	11	12
12	12	13
13	13	14
14	15	17
15	16	18
16	17	19
17	19	21
18	20	22
19	21	23
20	22	24
21	23	25
22	24	26
23	27	29
24	28	31

e:Logic state analyzer intermediate format

Label Base	ADR Hex	FILE Hex	LINE No. Decimal	States Relative
1:	70000008	0	000000010	630144
2:	70000008	0	000000014	80
3:	70000008	0	000000014	382
4:	70000008	0	000000014	91
5:	70000008	0	000000014	58
6:	70000008	0	000000024	75

Time Printed: 29 Aug 2000 14:16:57

f:Source code with results

```
 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=15753600.00ns 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
       best : n=1 t= 1450.00ns W=1190nWs
       worst: n=1 t= 9550.00ns W=7841nWs
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 6. Source code with trigger points and intermediate logic state analyzer format

Group	SFP	CDP	StrongARM	StrongARM	SPARClite	SPARClite
Li/Malik[8]	0	0	11986 ns	1282 nWs	18425 ns	15127 nWs
Ye/Ernst[1]	3	0	11836 ns	1261 nWs	18125 ns	14880 nWs
Wolf/Ernst[2]	3	2	9505 ns	911 nWs	14870 ns	12208 nWs
Exact Bounds	-	-	9471 ns	903 nWs	13650 ns	11206 nWs

Table 1. Upper bound execution time and power for the OAM component

7. CONCLUSION

Segment-wise, real-time execution and timing and power measurement preserving the cache behavior of the program under investigation is required for an application in formal program analysis. The presented approach to fast and compact trace acquisition including execution time and power measurement with off-the-shelf evaluation kits meets these requirements. Avoiding full traces by using trigger points leads to much higher efficiency than in previous approaches while the one-way communication through the terminal program allows real-time execution of the program under investigation. Result measurement using an off-the-shelf logic state analyzer, automatic evaluation and back annotation abstract the hardware evaluation kit to the same level as a software simulator. It is completely transparent to the designer working with an electronic design automation tool suite and matches novel approaches to formal running time analysis based on program segment execution.

Execution time and power consumption of program segments and complete programs has been measured using a commercial SPARClite evaluation kit. It is generic and can easily be applied to any commercial evaluation kit with bus access. Current work focuses on experiments with execution time and power consumption prediction using instruction tables which can be a base for software power reduction. Internet access to the measurement facilities is also under investigation.

REFERENCES

1. W. Ye and R. Ernst, "Embedded program timing analysis based on path clustering and architecture classification," in *Proceedings International Conference on Computer-Aided Design (ICCAD '97)*, pp. 598–604, (San Jose, USA), 1997.
2. F. Wolf and R. Ernst, "Intervals in software execution cost analysis," in *Proceedings of the International Symposium on System Synthesis (ISSS 2000)*, (Madrid, Spain), 2000.
3. D. Ziegenbein, R. Ernst, K. Richter, J. Teich, and L. Thiele, "Combining multiple models of computation for scheduling and allocation," in *Proceedings Sixth International Workshop on Hardware/Software Co-Design (Codes/CASHE '98)*, pp. 9–13, (Seattle, USA), March 1998.
4. J. Flanagan, B. Nelson, J. Archibald, and K. Grimsrud, "BACH:BYU address collection hardware, the collection of complete traces," in *International Conference on Modeling Techniques and Tools for Computer Performance Evaluation*, pp. 128–137, 1992.
5. S. Lee, M. Lee, S.-L. Min, and C.-S. Kim, "Timermon: A time-tracing hardware for instrumenting real-time software," in *Proceedings of IEEE Real-Time Computing Systems and Applications Symposium*, 1998.
6. V. Tiwari, S. Malik, and A. Wolfe, "Instruction level power analysis and optimisation of software," *Journal of VLSI Signal Processing* , pp. 223–238, 1996.
7. M. Hill, "DINERO III Cache Simulator." http://www.ece.cmu.edu/ ece548/tools/dinero/src/, 1998.
8. Y.-T. S. Li and S. Malik, *Performance Analysis of Real-Time Embedded Software*, Kluwer Academic Publishers, 1999.
9. A. V. Aho, R. Sethi, and J. Ullman, *Compilers: Principles, Techniques and Tools*, Addison-Wesley, Reading, GB, 1988.
10. T. M. Conte and C. E. Gimarc, *Fast Simulation of Computer Architectures*, Kluwer Academic Publishers, 1995.
11. J. Montanaro, "A 160-MHz, 32-b, 0.5W CMOS RISC microprocessor," *IEEE Journal of Solid State Circuits* , pp. 1703–1714, November 1996.
12. V. Tiwari, S. Malik, and A. Wolfe, "Power analysis of embedded software: A first step towards software power minimisation," *IEEE Transactions on VLSI Systems* , 1994.