Instrumentation-based Tool for Latency Measurements

Pekka Pääkkönen
VTT Technical Research Centre of Finland
Kaitoväylä 1
90571, Oulu, Finland
Pekka.Paakkonen@vtt.fi

Jarmo Prokkola
VTT Technical Research Centre of Finland
Kaitoväylä 1
90571, Oulu, Finland
Jarmo.Prokkola@vtt.fi

Ali Lattunen
VTT Technical Research Centre of Finland
Tekniikankatu 1
33101, Tampere, Finland
Ali.Lattunen@vtt.fi

ABSTRACT
Software has to be tested from functional and performance viewpoints in order to create products, which fulfill customer demands. The need for testing has led to the development of a plethora of testing tools. Performance measurement of SW latencies on local and distributed SW platforms hasn't yet been completely solved, which is the research problem of this paper. In particular, GPS-based time synchronization and performance of the proof-of-concept has been concentrated on. The approach is to instrument the SW implementation under study, and to collect measurement data with the presented tool. The results indicate a resolution of ~590 ns, which can be achieved with high performance reference clocks. CPU processing can be kept lower than 5% even with a high event transmission rate. In addition, the presented GPS synchronization method can be used for other purposes such as data packet time-stamping in network monitoring solutions.

Categories and Subject Descriptors
D.2.5 [**Testing and Debugging**]: Distributed debugging.

General Terms: Measurement, Performance, Experimentation, Verification.

Keywords: Event-based, debugging, SW instrumentation, latency, time synchronization, GPS.

1. INTRODUCTION
The need to validate functional or performance requirements of SW has been fulfilled with many commercial tracing tools. SW functionality under study may consist of one or several modules, which have events of interest to the test personnel. In addition, the modules may be executed in the same or different computing systems with heterogeneous SW platforms (Unixes, Windows, Symbian etc.). Currently a plethora of tools exist for SW performance measurements for different SW platforms.

Different parameters of SW processing may be interesting to the users of testing tools. For example CPU processing share, memory usage and power consumption may have value to the user. The focus of this paper is on the measurement of SW latencies on local and distributed computing systems, which is based on instrumentation of SW. The contribution is presentation of a proof-of-concept, which is aimed to be used for measurement of simple and complex events of interest. The main focus of the concept is in the method for time synchronization with the GPS (Global Positioning System), which is needed for distribution support. In addition, performance results of the implemented tool for the Linux Operating System (OS) are presented. The results indicate that ~590 ns resolution can be achieved with high performance system clocks, and CPU processing can be kept lower than 5% even with a large event transmission frequency.

The structure of the paper is as follows. Related work for SW event tracing is described in Chapter 2. The proposed measurement concept is presented in Chapter 3. In particular, motivation, requirements and main features of the tool are provided. Proof-of-concept and performance results of the tool are presented in Chapter 4. The tool is analyzed in terms of the stated requirements and compared to existing research in Chapter 5. Conclusion of the work is presented in Chapter 6. The final chapter discusses the results and possibilities for improvements.

ICPE'11, March 14–16, 2011, Karlsruhe, Germany.

2. RELATED WORK
Previous work related to the subject of this paper deals with capturing of time, and synchronization of clocks for distributed systems, and development of measurement tools.

2.1 Time and synchronization
When the SW of interest resides in different computing systems and time is measured, synchronization between the clocks is important for validity of the results. These problems have been studied decades ago [8]. In the system defined, distributed processes have separate clocks, which can be used for synchronization and ordering of events with the presented algorithms.

Clock synchronization for parallel embedded systems has been designed [2]. It presents a model for synchronizing clocks between processor nodes with the reference clock of a global server. Drifting of the clock in the nodes of the centralized system is corrected with synchronization pulses.

Most current clock synchronization techniques for One-Way Delay (OWD) measurements have been compared [5]. It was noticed that Network Time Protocol (NTP) doesn't provide enough accuracy for industrial applications. GPS and IEEE 1588/Precision Time Protocol (PTP) offer high accuracy (sub-microseconds), but have limitations in effort to use and dependency of LANs for high accuracy.

PTP is based on the exchange of timing messages between distributed nodes, and provides up to 30 ns clock accuracy with an implementation based on the latest version of the standard (2008) [20]. The problem is performance in wireless networks where accuracy is in the range of few to hundreds of

microseconds or even worse, and may depend on the layer of SW where the measurements are performed on [21][22]. In addition, a clock with higher accuracy needs support from HW.

PC-based system clocks can also be applied for timing purposes. Accuracy of the system clock is dependent on the quartz crystal it is made of, and thus its accuracy depends on temperature. Accuracy and drifting of the clock over time were found to be modest in modern microprocessor architectures [3]. However, the resolution is constrained by frequency of the crystal to around 1μs [25]. As an alternative to system clocks of the operating system, CPU clock cycle register (TSC) based clock has been proposed, which offers 1 ns resolution on Real-Time Linux [25]. Standard Linux kernel provides also support for high resolution timers (Performance API/PAPI) [26] [28]. In Windows OS, system performance counters can be used easily. Based on the tests made by the authors, the resolution is dependent on the HW. In many computers, the counter frequency is 3.58 MHz, providing ~ 280 ns resolution. However, in some newer computers, the frequency is the same as the processor clock speed, providing thus very high resolution. In reality, however, processor load, and the timestamp request decrease the achievable resolution.

2.2 Measurement tools

Tools have been developed for measurement of processing delays based on instrumentation of SW. ARM (Application Response Measurement) [13] standard is based on this approach. It defines an API to a library for measurement of application response delays. The library is linked to the measured SW, which executes the needed transactions with calls to the API. The approach has been successfully verified in a business case [14], where performance data has been gathered in a multi-radar air-situation display system.

NetLogger follows a similar approach as ARM [15]. The toolkit provides an API for logging application events to a file, and provides APIs for different programming languages. A Real-time Collector receives the events and aggregates the data for the purpose of visualization.

TAU performance measurement system [17] also follows the SW instrumentation-based approach. The main difference between ARM and NetLogger is that different instrumentation alternatives are supported (source, preprocessor, compiler, wrapper library, binary etc.) with TAU. In addition, TAU has been integrated for programming environments with IDE plug-ins [18].

In addition, other performance debugging tools have been developed. Accuracy of high resolution timestamps on Linux platform was measured by the authors of [3] for the development of the Linux Trace Toolkit. It enables the tracing of program execution flow without instrumentation based on timestamps embedded to the kernel [4].

A system has also been developed for the performance assessment of Embedded HW/SW Systems [1]. The solution is to use a separate HW-system, which is plugged into the system under study. The system consists of an ASIC (for timestamp collection), a serial bus and a set of probe-chips, which are connected to the bus for event collection. The purpose of the tool is to enable real-time performance measurement of HW/SW systems according to the specified performance indexes.

In addition, there are plenty of commercial debuggers available, which enable tracing of program flow on the target HW. An example is Mentor's Majic (www.mentor.com), which enables tracing of SW via the standardized JTAG interface to the target HW.

Tools are available for estimating CPU/memory consumption of SW on target HW in run-time. A web-based toolkit has been proposed for tracking performance of applications in grid platforms, which can be used for visualization of CPU, memory usage and network bandwidth [10]. A method has been developed for monitoring CPU and heap memory usage of Java-based software components (bundles) on OSGi platform [11].

Finally, tools have been developed for integration of SW development with performance testing processes [9]. In the presented approach performance tests are specified, implemented and executed during the SW development process in order to track critical problems early in the process. Performance tests were implemented by inserting timestamps at the desired measurement points, and data was collected with JUnit test software.

Methods for simulating software performance without target HW have been proposed in order to aid decision making in SW/HW design. The existing approaches have been reviewed, and a new solution has been proposed, which targets micro-architecture issues with source code instrumentation [12].

2.3 The contribution

Although there exists plenty of techniques for measurement of SW performance, a contribution is missing, which presents the integration of SW instrumentation approach with GPS-based clock synchronization for measurement of processing delays on local and distributed systems. In addition, performance of the approach hasn't been properly studied (to the best of the author's knowledge). The afore-mentioned issues are the main contribution of this paper.

The basic synchronization principle of this paper is based on VTT's earlier work on data packet time-stamping synchronization. Qosmet tool was developed for passive monitoring of Quality of Service (QoS) in communications [7]. In order to perform accurate OWD measurements, an analogous synchronization problem is faced as when measuring SW process performances. For this, a special GPS synchronization driver was developed for accurate time-stamping of packets in remote machines. The driver has been developed for Windows OS and uses system performance counters. The synchronization accuracy is better than 50 μs.

3. CONCEPT

Motivation of the developed concept is described in Section 3.1, which is followed by the statement of requirements in Section 3.2. The main features of the tool are described in Section 3.3. In particular, architecture, measurement of different events, API of the developed Event-library and GPS-based time synchronization is described.

3.1 Motivation

The presented concept is aimed to be implemented by a tool, which can be used for latency measurement of SW related events on target HW. An event is defined here as any processing performed by the SW implementation(s) over local or distributed computing systems. Latency characterizes performance of the event on the target platform. The expected latency depends at least on the HW/SW platform, real-time requirements and use

case related to the event(s). Thus, the goal is to enable performance measurement of events, which have value for the end-user.

3.2 Requirements

In order to have a tool for measurement of different events, which is usable on different SW platforms and programming environments, the following requirements have been defined:

Req. 1: Reduce time for measurement and analysis: In addition, measurement data should be easily visualized for the purpose of analysis.

Req. 2: Measurement of different events: The functionality of interest may be simple such as parsing of XML data. In addition, there may be multiple events related to each other, which are of interest e.g. encoding and transmission of XML document to the end point. Such an event is defined as a chained event. Measurement of both types of events must be supported.

Req. 3: Support local and distributed computing systems: Measurement of events on local and distributed computing systems should be supported.

Req. 4: Independency on programming language, SW process/platform and operating system: The tool should be independent of the process/platform it is executed on and it should support as many programming languages as possible in order to be usable in heterogeneous test environments and SW platforms.

3.3 Main features

3.3.1 Measurement of different events

In order to fulfill Req. 1, a unique identifier is used for mapping to the event of interest. An example would be to map 'XML message decoding delay' user level concept to events with ID=1. In order to identify different measurements of an event, a sequence number is applied for each measurement.

A simple event has a start and end time associated with it, and latency of the event is calculated based on the difference. A complex chained event is comprised of multiple steps, each of which is a component of the total latency. The event types consist of the following data:

Simple-event: identifier, sequence-number, start-time, stop-time

Chained-event: identifier, sequence-number, step-number, time

3.3.2 Architecture

Figure 1 describes architecture of the tool on a local computing system. The tool is comprised of event producers, an EventLibrary and a Collector. An event producer is any SW process, which sends events to the Collector. The Event Library enables transmission of events from event producers to the Collector. The role of the Collector is to process the received events for the purpose of visualization. The visualization tool may be co-located with the Collector or may be implemented as a separate tool.

In the example of Figure 1, functionality 1 is under study. It is measured with a simple event in SW process X, which has been implemented with C-programming language. The event producer transmits start and stop time of the simple event, and associates them with a unique identifier (ID=1). The event is sent via the Event Library to the Collector. The former provides an Application Programming Interface (API) to event producers. As soon as the event is received by the Event Library, time is captured. Any accurate clock can be used as the reference clock for time-stamping. The Event Library associates time with the event, and sends the event to the Collector. In addition, the Event Library associates a sequence number with the event for the purpose of identification.

Figure 1 provides also an example for inter-process measurements. In this case the functionality of interest is chained (ID=2), and starts in process X (step 1), continues in process Y (steps 2-3) and finally completes in process X (step 4). The chained event is identified (ID=2).

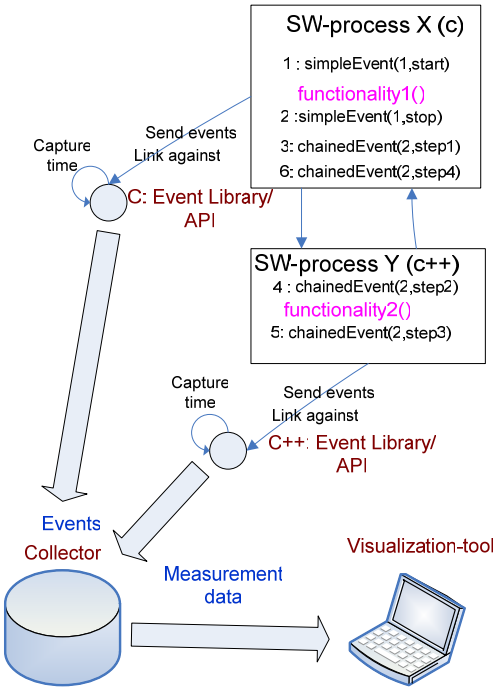

Figure 1. System architecture of the tool.

3.3.2.1 Event library API

The event library offers the following simple API for the event-producers:

sendSimpleEvent(event-identifier,startStop);

sendChainedEvent(event-identifier,step);

The purpose is to define a very simple API, which can be implemented with different programming languages.

3.3.2.2 Collection of events

The events produced by the instrumented SW are recorded by the Collector, and thus a method has to be defined for communication. The interface between the Collector and the EventLibrary has not been defined in the architecture. In practice any suitable communication method may be used. For example on a local computing system shared memory, loopback sockets or Web Services may be used. For distributed systems firewalls and applied Network Address Translators (NAT) between the collector and the library limit the selection of communication protocols.

3.3.2.3 Time-stamping of captured events

When an event is captured in the Event-library, time is associated with it. Time may be captured with any accurate clock in the Event Library. Candidates are at least system clock of the OS,

GPS, NTP and PTP/IEEE 1588. The choice of the clock depends mostly on the requirement of time accuracy and resolution, mode of measurements (local/distributed) and accuracy provided by system clocks of the OS. For example, many Windows based OSs suffer from low resolution system clock, providing only millisecond level accuracy. Thus, system performance counters must be used for high resolution. In the local mode it is possible to rely on the local reference clocks. In the distributed mode clocks must be synchronized between the platforms by using e.g., GPS/NTP/PTP.

3.3.3 Distribution: GPS-based time synchronization

In order to achieve synchronization of time on distributed SW platforms, the usage of GPS is defined. Retrieval of accurate time-stamps from GPS device is not straightforward. Cheap GPS receivers do not output accurate time. Thus, GPS devices equipped with Pulse-Per-Second (PPS) functionality are needed. PPS enabled devices output RF pulses at even seconds via a BNC output in contrary to the standard GPS protocol messages (e.g. NMEA/proprietary GPS data). GPS devices with PPS functionality are equipped with an internal clock, which is synchronized to the UTC time received from satellites.

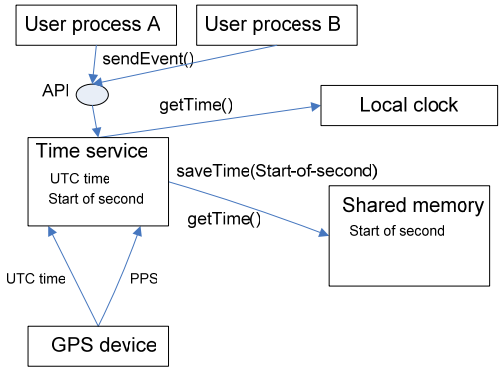

Figure 2. Time synchronization based on GPS device equipped with PPS functionality.

Retrieval of UTC time and PPS signal from the GPS device for time-stamping purposes in the user space is described in Figure 2. Time service stores UTC time and Start of second, which are used for calculation of accurate time to be associated with the captured event. UTC time is the current time without fraction, and it is synchronized with the PPS signal in the GPS receiver. Start of second corresponds to the beginning of each second, which is retrieved from the local clock (system clock, system performance counter, etc.). It is used for calculation of fraction to be concatenated with UTC time.

UTC time is received from the GPS device with standard protocol messages and is saved in Time service after GPS device has configured a position and time fix. At each PPS signal, time is captured from the local clock and saved as Start of second. At the same moment, UTC time is incremented by one second. The procedure enables a running clock in Time service.

When the user of Time service calls sendEvent(), current time is received from the local clock, and Start of second is subtracted from it in order to get fraction for the current time. Finally, UTC time is concatenated with the fraction in order to get an accurate time-stamp, to be associated with the captured event.

When SW is executed in different processes and linked to the Event-library, different instances of Time service are executed. In order to enable synchronization of time, Start of second is saved into shared memory. When multiple instances of Time service get the PPS signal from the GPS device, the first process updates Start of second from local system clock to the shared memory (saveTime()). The other processes get Start of second from the shared memory (getTime()), and thus stay synchronized.

An alternative option for implementation (as done in Windows environment in [7]), is to run the functionality of Time service as a driver in kernel. In this case, several user processes can ask timing information from the single driver entity. The kernel-mode operation also provides better accuracy than the user-mode.

Synchronization of time with GPS between distributed nodes depends naturally on the accuracy provided by the GPS device. Trimble Lassen iQ GPS manual reports up to 50 ns accuracy for the rising edge of PPS, which is synchronized with UTC time. Delay for the signal propagation from GPS device via the adapter to the laptop should be constant [3]. However, the internal delay of the interrupt handler in PC is variable and dependent on the processing load in non-real-time operating systems. For a PC (Pentium III) the delay has been measured to vary between 8-50 µs, but with special PPS capture cards the delay can be reduced to tens of nanoseconds [27]. Here it is assumed that propagation delay in the PC is the same at both ends of the distributed system, and thus compensated.

4. VALIDATION

The presented concept described in the previous chapter has been validated with a real prototype system, which is described in this Chapter. In addition, performance tests of clocks are presented. HW architecture of the prototype is depicted in Section 4.1.1 and SW architecture is described in Section 4.1.2. Implementation of different event types is presented in Section 4.1.3, and capturing of time in Section 4.1.4. Finally, results of performance tests are described in Section 4.2.

4.1 Proof of concept

4.1.1 HW architecture

The HW architecture of the prototype system is presented in Figure 3. The used GPS-terminal is Trimble Lassen IQ. It uses USB for communicating location and UTC time data to the receiving device (laptop). It also provides PPS output with a separate BNC connector (1-PPS). A converter was constructed in order to transform BNC signal into serial format. PPS pulse is output to the CTS pin of the serial line connector (RS-232).

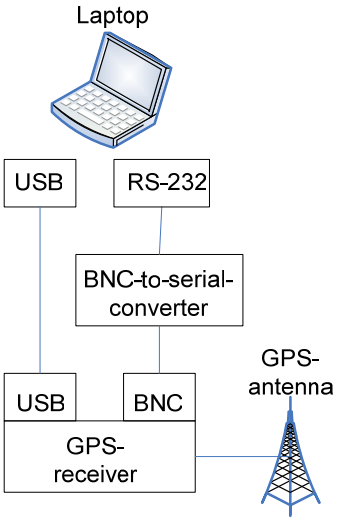

Figure 3. HW architecture of the tool.

4.1.2 SW architecture

The main features of the concept were implemented with C++ for the Linux OS running on the laptop. Architecture of the implementation has been depicted in Figure 4. The tool is comprised of an executable Collector and dynamically shared library (EventLibrary). Only one Collector is executed on a SW platform. However, multiple EventLibraries can be executed, and the Collector gathers events from each over shared memory. This enables measurement from multiple processes.

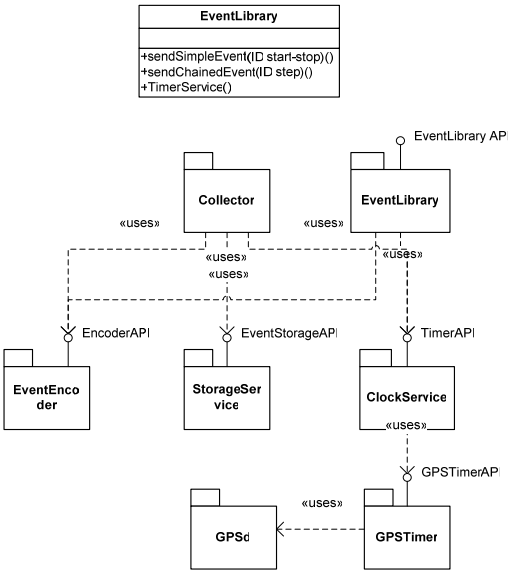

Figure 4. SW architecture of the tool.

EventEncoder is used for encoding and decoding of events to be exchanged over shared memory between the Collector and the EventLibrary. ClockService encapsulates the retrieval of time from the different clock services. StorageService processes and saves the captured events to a file for the purpose of visualization.

GPSTimer implements the algorithm (described in Section 3.3.3) for getting accurate time from GPS. It communicates with the GPS Daemon, which is executed in the background for accessing the GPS receiver over USB.

4.1.3 Implementation of event types

The mapping of events between the EventLibrary and Collector has been described in Figure 5. The events are exchanged between the EventLibraries and the Collector over shared memory. The System V Linux implementation of shared memory was applied for Inter-Process Communication (IPC).

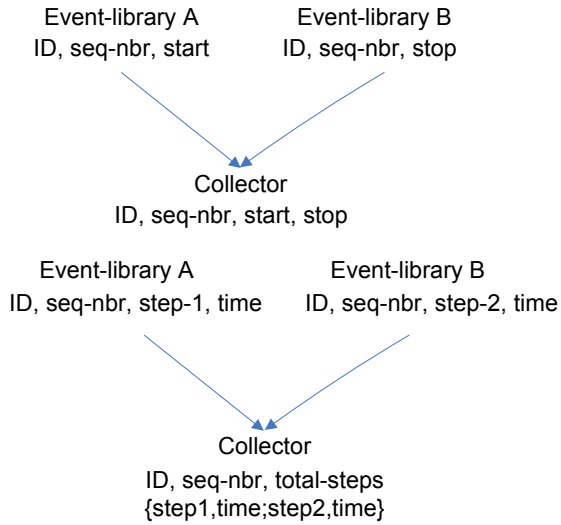

Figure 5. Mapping of events with the tool.

4.1.4 Capturing of time

Time is captured immediately after the user calls the EventLibrary API. Subsequently the event is passed to the EventEncoder, which adds the event into a queue for encoding and transmission to the Collector. It is critical to capture time quickly, and to return from the EventEncoder, since processing in the library shouldn't skew results of the event under study.

ClockService uses timers offered by the Linux OS or PAPI library and gets accurate time from GPSTimer. Functions gettimeofday() and clock_gettime() are used as local system clocks by the module.

GPSTimer reads UTC time over shared memory from GPSd. It uses separate sysctl() function calls for getting Start of second signals over the serial line based on PPS output of the GPS receiver.

4.2 Performance results

4.2.1 Resolution of reference clocks

Resolution of reference clocks was tested with the tool. The laptops under testing were Dell Latitude D400 (Slow CPU-(1.6GHz Pentium M) and Acer Travelmate 8371 (Fast CPU-Intel Centrino 2). Ubuntu 9.10 operating system with 2.6.31 Linux kernel had support for the Performance API (PAPI) clock on both laptops.

In the tests two consecutive calls to the clock API was executed. A short delay (1 ms) was included after the function calls. The test procedure included 100'000 iterations, and it was executed three times.

The results of the experiments are described in Table 1. System clock resolution based on the PC clock crystal is ~ 1 µs. PAPI

timers can be used for reaching a much higher resolution for time-stamping as was expected. The resolution depends on the CPU, which is used as a time source.

Table 1. Average resolution and confidence interval (99%) of different reference clocks. The values are provided as nanoseconds.

Clock	Resolution	Conf-(99%)	
		min	max
clock_gettime()	1148	1095	1201
gettimeofday()	1125	1118	1131
PAPI-Slow CPU	140	138	141
PAPI-Fast CPU	49	48	50

In order to achieve high resolution with the tool, processing during capturing of time has to be minimized. When processing was minimized, resolution of the tool decreased up to ~ 0.5-1.3 µs compared to the reference clock (see Table 2). The experiments were performed similarly as the tests for reference clocks.

Table 2. Average resolution and confidence interval (99%) of resolution with the tool for different reference clocks. The values are provided as nanoseconds.

Clock	Resolution	Conf-(99%)	
		min	max
clock_gettime()	2217	2213	2220
gettimeofday()	2465	2461	2469
PAPI-Slow CPU	811	809	813
PAPI-Fast CPU	585	583	586

4.2.2 Overhead and error

Overhead and error was measured by using PAPI as the reference clock for the tool. The goal was to study how much usage of the tool affects the measured event. The processing length of the event under measurement was modified and delays were measured with simple events and without the tool (as reference). The event of interest was execution of a for()-loop, where the number of loops was modified. Overhead and error of the tool were compared to the reference.

The results are provided in Figures 6 and 7 and cover 100'000 iterations from three test cases. The following metric was used for estimation of overhead and error:

$Events$ = Number of events in a test case

Len_{tool} = Total length of a test case with the tool

Len_{ref} = Total length of a test case in the reference

$Len_{event,i}$ = Measured length of an event in test iteration (i)

$AvgLenEvent_{tool} = \frac{Len_{tool}}{Events}$ = Average length of an event with the tool

$AvgLenEvent_{ref} = \frac{Len_{ref}}{Events}$ = Average length of an event in the reference

$AvgMeasuredLenEvent_{tool} = \frac{\sum_{i=1}^{Events} Len_{event,i}}{Events}$ = Average length of an event, which is measured with the tool

$$Overhead = \frac{100 * (\text{AvgLenEvent}_{tool} - \text{AvgLenEvent}_{ref})}{\text{AvgLenEvent}_{ref}}$$

$$Error = \frac{100 * (\text{AvgMeasuredLenEvent}_{tool} - \text{AvgLenEvent}_{ref})}{\text{AvgLenEvent}_{ref}}$$

The tests were executed with the slow and fast CPUs mentioned before.

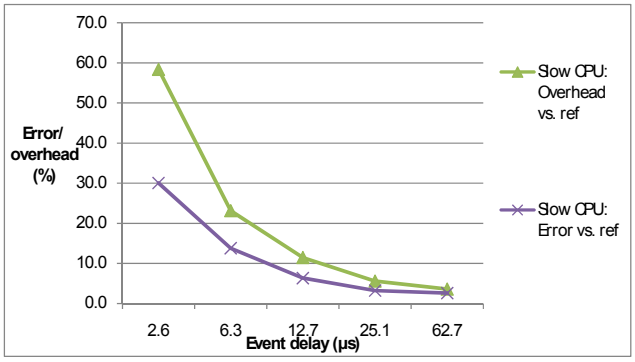

Figure 6. Overhead and error produced by the tool, when experiments were conducted with the slow CPU.

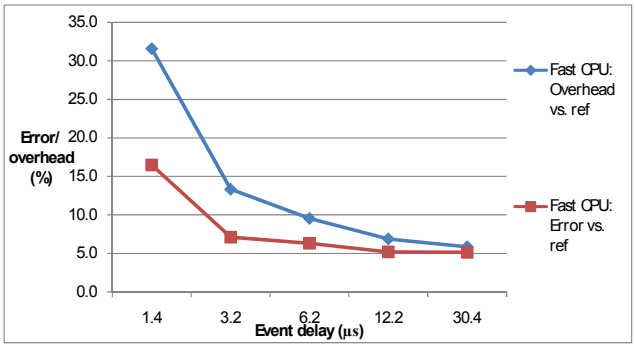

Figure 7. Overhead and error produced by the tool, when experiments were conducted with the fast CPU.

Based on the results it can be seen that overhead and error caused by the tool depends on the CPU power. With the slow CPU error is less than 10%, when the event to be measured is longer than ~ 13 µs. With the fast CPU the same level of error is achieved, when the event is longer than ~ 6 µs.

4.2.3 CPU consumption of the tool

CPU processing caused by execution of the tool was measured. The test procedure consisted of execution of a simple event measurement after expiration of an application timer. The length of the timer was 1 ms. In the tests the frequency of shared memory access for transmission of events in the tool was modified in order to discover the effect on CPU power consumption.

CPU processing was measured with top Unix-tool, as a function of application timer length. The results are provided in Figure 8, and average CPU consumption is calculated from three test cases, where 100'000 events were transmitted with the slow CPU.

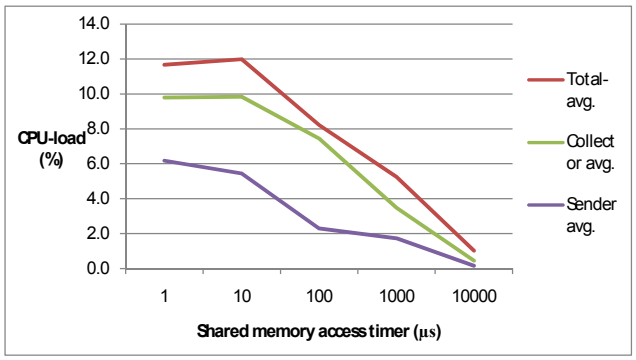

Figure 8. CPU-processing caused by the tool as a function of shared memory access. 'Total avg.' is the average of total CPU-processing. Average CPU-consumption caused by the Collector, and Sender (EventLibrary) is also described.

From the results it can be noticed that CPU-processing is lower than 10%, when shared memory is not accessed more frequently than 0.1 ms.

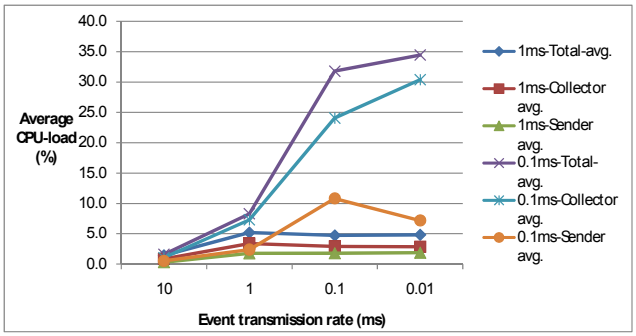

Figure 9. CPU-processing caused by the tool as a function of event transmission rate. The curves describe processing for different rates of shared memory access.

Figure 9 describes CPU consumption as a function of event transmissions triggered by an application timer, when shared memory access rate is kept constant (0.1 ms/1 ms). The test procedure is similar as described above.

When shared memory is accessed with a rate of 1 ms, CPU-processing stays lower than 5% even with a high message transmission rate (100'000 messages/second). Messages are queued at the sender, because transmission rate is higher than the memory access rate. When shared memory access is higher (0.1 ms), CPU-processing increases, because messages are transmitted over shared memory with a higher rate than in the previous case.

5. ANALYSIS OF THE TOOL

In this chapter the tool is evaluated in terms of the stated requirements. Finally, performance of the implementation is evaluated and the tool is compared to existing commercial solutions and published work.

5.1 Reduce time for measurement and analysis

The biggest value of the tool for the user should be reduced time between measurement setup and analysis phases achieved with automated measurements. Usage of the tool consists of the following steps:

1.) Identify events of value. Allocate identifiers for the events.

2.) Instrument the SW implementation with function calls to the EventLibrary. Compile and link the implementation.

3.) Start the Collector and execute test cases with the instrumented SW.

4.) Extract data from the measurement file for the purpose of analysis with a visualization tool (Excel/Matlab/proprietary tool etc.).

Alternatively testing could be performed without the tool or with a SW/HW tracer. If the tool is not used, determination of the correct reference clock, synchronization between processes and calculation of statistics becomes problematic and requires effort. If SW/HW tracer is used, SW doesn't have to be instrumented. However, calculation of statistics from trace files and increased CPU processing may create problems.

Instrumentation of SW could potentially be shortened with binary-based instrumentation as in the TAU approach [17]. In this case the executable code is instrumented instead of the source code, and compiling and linking of SW would be avoided.

5.2 Measurement of different events

The most important solution related to the proposal in previous research is ARM [13] [24]. API calls of ARM and our approach are compared in Figure 10. The main difference is in the usage of event identifiers and amount of function calls between the application and the library. ARM defines an identifier for the application and measured transactions. In addition, it uses correlators for mapping different transactions together. This adds complexity to the API. Our approach is kept as simple as possible. Event-producer provides the identifier of the measured event to the EventLibrary, which reduces the amount of function calls.

The obvious downside of our proposal is the need to keep identifiers unique in the applications, which use services of the EventLibrary.

Our approach defines the chained event for measurement of interrelated events. When the measurement files are integrated in the Collector, the user gets a unified view of the processing between different steps in the chained event. ARM uses correlators for the same purpose. ARM doesn't specify how correlators are mapped together between libraries, which are executed in different processes. In the proposed approach, responsibility is left for the user to apply the same identifier in different instances of an EventLibrary, when different steps of interrelated events are measured with chained events.

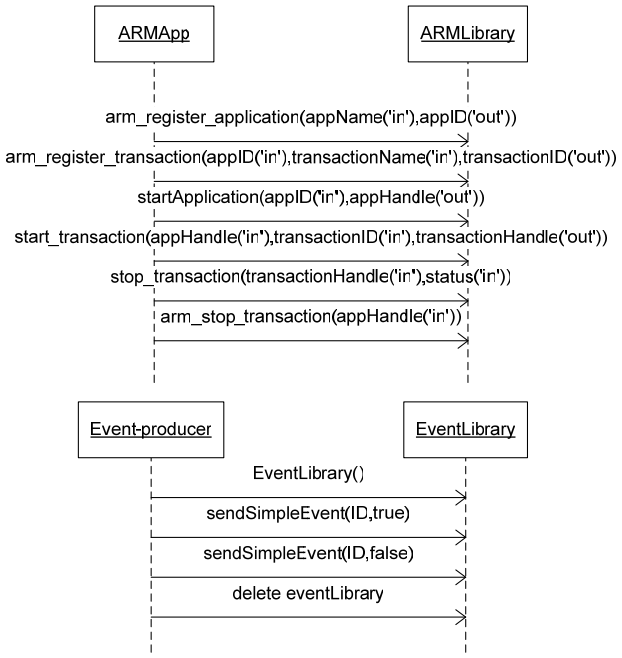

Figure 10. Comparison of API calls between ARM and the proposal.

5.3 Support local and distributed computing systems

If measurements are performed on a local SW platform, system clocks provided by the OS or high performance timers can be applied. For distributed platforms clock synchronization is achieved with GPS, and thus each SW platform has to support GPS with PPS functionality. Distribution could also be supported with other clock synchronization techniques such as PTP.

In the distributed mode clock synchronization requires that PPS signals from the OS are synchronized between multiple processes. The Linux OS notifies each of the processes, which has called sysctl(), when a PPS is detected. Thus, only the first process should update Start of second to shared memory, which is used as the reference for all the EventLibraries executed in multiple processes.

5.4 Independency on programming language, SW process/platform and operating system

In order for the tool to be usable on different SW platforms and operating environments, portability issues have to be taken into account. In the current architecture one Collector is executed on each SW platform, which collects data from all EventLibraries executed on the same SW platform. The data saved on multiple SW platforms can be merged by any of the Collectors. At the moment measurement files have to be manually transferred to a centralized location for merging.

The usage of the EventLibrary API is transparent to the application, even if measurements are performed on multiple SW processes. However, sequence numbers are unique to the process EventLibrary is executed in. The user is responsible that events of a chained event always occur in the same order.

EventLibrary API has been developed for Java, Javascript, C and C++ programming languages. When API of the extension API is called, the native implementation of EventLibrary is executed. Support may also be provided for other languages such as Python.

The current full implementation exists for Linux, but it should be possible to be ported also for other OSs. The critical parts in porting are access to shared memory, support for GPS drivers and API, and accurate timers provided by the OS. As stated earlier, for Windows, a GPS time-stamping driver has been implemented for providing synchronization. Thus, the most difficult part for Windows already exists.

5.5 Performance

The results indicate that system clocks of Linux achieve accuracy in the range of 1 microsecond. The results are within the same range of resolution, which has been reported earlier for system clocks [25]. PAPI achieves higher resolution (~ 80 – 140 ns) due to usage of CPU counters as a time source. The resolution depends on the power of the CPU as expected. However, nanosecond resolution for CPU based clocks wasn't achieved on Linux. Other OSs (e.g. Windows) may reach higher resolution as was reported in earlier in the related work.

It was noticed during measurements that processing should be minimized, when the timestamp is captured. In the current implementation captured time is saved into a circular buffer (pre-created C++ object). With this approach overhead of capturing is ~ 0.5 – 1.3 µs, and highest resolution achieved with PAPI ~590 ns.

Tests were executed in order to find out the level of error and overhead caused by execution of the tool. It seems that error and overhead depends on the HW the tool is executed on. Error was lower than 10% with a high power laptop, when length of the measured event was longer than 6 µs. Inaccuracy is most probably caused by overhead of the tool, which becomes significant with very short events. The tests described an extreme, where a very short event was measured in a loop.

Tests indicated that CPU-processing reduces, when the frequency of memory access is reduced for transmission of events, which was expected. With a transmission rate of 1 ms, the tool produces up to ~ 5% CPU load. It can be seen that CPU-processing doesn't significantly increase, even if event transmission frequency is increased above the frequency of shared memory access. In such a case the events are queued in the sender implementation, which doesn't lead to excessive CPU load.

5.6 Comparison to related work

The contribution of this paper is presentation of GPS-based time synchronization method for an instrumentation-based delay measurement tool. In addition, similar performance results of the approach haven't been discovered by the authors. However, similar approaches have been published and this work is compared here for the purpose of argumentation.

ARM [13] proposes a very similar instrumentation-based approach. As described in Section 5.2, our approach is simpler and requires less function calls between the application and the tool. In addition, ARM doesn't specify any time synchronization method, which is left open for implementation of the specification. TAU [17] is also based on instrumentation, but similar contributions on time synchronization weren't found either. NetLogger [16] applies a more verbose approach as it dumps timestamps of events directly to an instrumentation file. The results indicate that overhead of the

solution is low (5% CPU load) with a frequency of 10000 events/second. The results are similar as in our approach, if the frequency of shared memory access is set lower than 1 ms.

PTP [19] offers also an approach for time synchronization, and it could be applied with the proposed tool. The problem is accuracy, which deteriorates in wireless networks.

Some other proposals are also worth of comparison. Source code instrumentation proposal [12] aims at SW performance estimation before the target HW is available with a simulator. In comparison our approach is instead focused on SW performance on target SW/HW platform. The authors of [9] presented measurements of SW latency during the SW performance process, but the implementation was manually instrumented without automation. The SW/HW tracer described in [1] is close to the concept proposed in this paper. The system under study has to be instrumented, and measurement is carried out with HW probes. Only the interesting events are measured, as in the approach presented in this paper. The downside of the presented approach [1] is dependency of the measurement HW and lack of implementation details provided. Mentor's solution is also HW-based, and provides detailed tracing of the program flow in the system. The downside is lack of support for distributed systems, and possibly overhead produced by the measurement method. The approaches for measuring CPU and memory usage [10] [11] of SW components can be considered as complementary to the tool proposed in this paper. Qosmet [7] has the initial implementation of the time synchronization method, but aims at different goals (data network monitoring) than the presented tool.

6. CONCLUSION

This paper presented a proof-of-concept for latency measurements, which is based on instrumentation of SW. The generic goal is to enable latency measurement of events on local and distributed systems. Latency characterizes performance of SW, which is valuable to the test person. HW/SW architecture and implementation were presented as a proof-of-concept. The solution was analyzed in terms of the stated requirements, and performance of the proposal was studied. In addition, a method for time synchronization with the GPS was presented. The presented time synchronization method can also be used independently for different purposes, such as data packet time-stamping for network monitoring [7].

The presented tool supports the measurement of simple and complex events, the measurement and analysis processes can be automated, and the method incurs low overhead to the event(s) under study. The tool supports various programming languages and inter-process communication. It is argued to have a simpler API than competing solutions based on the ARM specification. The presented time synchronization method has been implemented for Windows and Linux, which indicates portability for different operating systems. Experiments showed that a resolution of ~590 nanoseconds can be achieved with high performance clocks (PAPI) on Linux SW platform, which is lower than optimal. It is caused by processing related to the saving of the captured timestamp. It was discovered that measurement error becomes significant, when very small events (smaller than 15 µs) are captured. The level of error depended on the HW the tool was executed on. CPU consumption of the approach should remain lower than 5%, if the rate of shared memory access for event transmission is kept lower than 1 ms.

7. DISCUSSION

Achievement of optimal resolution remains as an open issue. The problem is minimization of processing related to the saving of timestamps. In the current solution data related to the event was saved to a pre-initialized data structure. When this minimal operation was performed in between time-stamping, the optimal resolution of time reduced significantly.

Nanosecond level resolution of time wasn't achieved on Linux platform. The performance is partly dependent on the CPU of the computing platform, the tool is executed on. Also frequency scaling of the CPU may affect performance [28]. In addition, higher resolution may be achieved with other operating systems (e.g. Windows).

GPS based time synchronization should not be affected by drifting of time, when compared to a solution based on system clocks. In our approach reference time for time-stamping from system clock is fetched during each second, and thus drifting doesn't have an effect. The delay and variance of the PPS signal from serial port to user space wasn't measured. However, this should be taken into account, because it has an effect on accuracy. The delay of the signal may be reduced with special PPS measurement HW [27].

The proposed approach for time synchronization requires investment to GPS HW. The tool could also be used with PTP based technology. However, high accuracy with PTP also requires investment to HW [5]. In some use case scenarios, where high accuracy is not needed for measurements, PTP could be used as reference time without HW support [22].

Portability of the concept for mobile terminals is very challenging. GPS chip sets in current mobile phones have been optimized for price and PPS is not usually supported. Thus, accuracy for time synchronization can be difficult to be achieved without PPS only based on UTC time.

The SW mentioned in this publication: The developed tool, Windows' timestamping driver, and Qosmet, are property of VTT. Do not hesitate to contact the authors for more information.

8. ACKNOWLEDGEMENT

The author acknowledges Daniel Pakkala (VTT) for providing funding for the work and Tommi Aihkisalo (VTT) for testing of the tool.

9. REFERENCES

[1] Calvez J.P, Pasquier O. 1995. Performance Assessment of Embedded Hw/Sw Systems. In *Proc. of the International Conference on Computer Design: VLSI in Computers and Processors* (Austin, Texas, USA, October 02-04, 1995). ICCD'95. IEEE, New York, NY, 52-57. DOI=http://dx.doi.org/10.1109/ICCD.1995.528790.

[2] Fleury M., et al. 1997. Design of a clock synchronization sub-system for parallel embedded systems. In *Proc. of the IEE Proc. Comput. Digit. Tech*., 144, 2 (March, 1997), 65-73. DOI= http://dx.doi.org/10.1049/ip-cdt:19971155.

[3] Marouani H., Dagenais M.R. 2005. Comparing high resolution timestamps in computer clusters. In *Proc. of the Canadian Conference on Electrical and Computer Engineering* (Saskatoon, Canada, May 01-04, 2005). IEEE, New York, NY, 400-403. DOI= http://dx.doi.org/10.1109/CCECE.2005.1556956.

[4] Linux Trace Toolkit, URL: “http::/www.opersys.com/LTT”.

[5] De Vito L., Rapuano S., Tomaciello L. 2008. One-Way Delay Measurement: State of the Art. *IEEE Transactions on Instrumentation and Measurement*. 57, 12 (December. 2008), 2742-2750. DOI= http://dx.doi.org/10.1109/TIM.2008.926052.

[6] Kannisto J. et al. 2005. Software and Hardware Prototypes of the IEEE 1588 Precision Time Protocol on Wireless LAN. In *Proc. of the 14th IEEE Workshop Local Metropolitan Area Networks* (Chania, Greece, September 18-21, 2005). IEEE, New York, NY. DOI= http://dx.doi.org/10.1109/LANMAN.2005.1541513.

[7] Prokkola J. et al. 2007. Measuring WCDMA and HSDPA Delay Characteristics with QoSMeT. In *Proc. of the International Conference on Communications,* (Glasgow, Scotland, June 24-28, 2007). IEEE, New York, NY, 492-498. DOI= http://dx.doi.org/10.1109/ICC.2007.87.

[8] Lamport L. 1978. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*. 21,7 (July, 1978), ACM, New York, NY, 558-565. DOI= http://dx.doi.org/10.1145/359545.359563.

[9] Johnson M..J. 2007. Incorporating Performance Testing in Test-Driven Development. *IEEE Software*. 24, 3 (May/June, 2007), 67-73, DOI= http://dx.doi.org/10.1109/MS.2007.77.

[10] Li H. et al. 2005. Design Issues of a Novel Toolkit for Parallel Application Performance Monitoring and Analysis in Cluster and Grid Environments. In *Proc. of the 8th International Symposium on Parallel Architectures, Algorithms and Networks* (Las Vegas, Nevada, December 7-9, 2005). IEEE, New York, NY. DOI= http://dx.doi.org/10.1109/ISPAN.2005.35.

[11] Miettinen T. et al. 2008. A Method for the resource monitoring of OSGi-based software components. In *Proc. of the 34th Euromicro Conference on Software Engineering and Advance Applications* (Parma, Italy, September 3-5, 2008). IEEE, New York, NY, 100-107. DOI= http://dx.doi.org/10.1109/SEAA.2008.24.

[12] Wang Z. et al. 2008. SciSim: A Software Performance Estimation Framework using Source Code Instrumentation. In *Proc. of the 7th International Conference on Software and Performance* (Princeton, New Jersey, USA, June 23-26, 2008). ACM, New York, NY, 33-42. DOI= http://dx.doi.org/10.1145/1383559.1383565.

[13] Application Response Measurement (ARM), Issues 4.0, Open Group, URL: http://www.opengroup.org/ management/arm.htm/.

[14] Engels K. and Heidger R. ARM instrumentation of the PHOENIX ATC System for Performance evaluation. In *Proc. of the Computer Management Group* (San Diego, CA, USA, December 2-7, 2007).

[15] Tierney B. et al., 2003. NetLogger: A toolkit for distributed system performance tuning and debugging. In *Proc. of the Eight International Symposium on Integrated Network Management* (Colorado Springs, USA, March 24-28, 2003). DOI: http://dx.doi.org/10.1109/INM.2003.1194164.

[16] Gunter et al. Log Summarization and Anomaly Detection for Troubleshooting Distributed Systems. In *Proc. of the 8th IEEE/ACM International Conference on Grid Computing (* Austin, Texas, USA, September 19-21, 2007). IEEE, New York, NY, 226-234. DOI= http://dx.doi.org/10.1109/GRID.2007.4354137.

[17] Shende S. S. Malony A. D. The TAU Parallel Performance System. *The International Journal of High Performance Computing Applications*, 20, 2 (May, 2006). IEEE/ACM, New York, NY, 287-311. DOI=http://dx.doi.org/10.1177/1094342006064482.

[18] Spear W. et al. Making Performance Analysis and Tuning Part of the Software Development Cycle. In *Proc. of the High Performance Computing Modernization Program Users Group Conference* (San Diego, CA, USA, June 15-18, 2009).

[19] Han J. and Jeong D. A Practical Implementation of IEEE 1588-2008 Transparent Clock for Distributed Measurement and Control Systems. *IEEE Transactions on Instrumentation and Measurement*, 59, 2 (February, 2010). IEEE, New York, NY, 433-439. DOI=http://dx.doi.org/10.1109/TIM.2009.2024371.

[20] Cooklev T. An Implementation of IEEE 1588 Over 802.11b for Synchronization of Wireless Local Are Network Nodes. *IEEE Transactions on Instrumentation and Measurement*, 56, 5 (October, 2007), 1632-1639. DOI= http://dx.doi.org/10.1109/TIM.2007.903640.

[21] Bang Y. et al. Wireless Network Synchronization for Multichannel Multimedia Services. In *Proc. of the 11th international conference on Advanced Communication Technology* (Phoenix Park, Korea, February 15-18, 2009). IEEE, New York, NY, 1073-1077.

[22] Correll K. and Barendt N. Design Considerations for Software Only Implementations of the IEEE 1588 Precision Time Protocol, In *Proc. of the IEEE-1588 Standard for a Precision Clock Synchronization Protocol for Networked Measurement and Control Systems*, NIST and IEEE, 2005.

[23] GPS Daemon, URL: http://gpsd.berlios.de/.

[24] Elarde J. V. and Brewster G. B. Performance Analysis of Application Response Measurement (ARM) Version 2.0 Measurement Agent Software Implementations. In *Proc. of the Performance, Computing, and Communications Conference* (Phoenix, Arizona, USA, February 20-22, 2000). IEEE, New York, NY, 190-198. DOI=http://dx.doi.org/10.1109/PCCC.2000.830318.

[25] Psztor A. and Veitch D. 2002. PC based precision timing without GPS. In *Proc. of the ACM SIGMETRICS international conference on Measurement and modeling of computer systems* (Marina del Rey, California, USA, June 15-19, 2002). ACM, New York, NY, 1-10. DOI= http://dx.doi.org/10.1145/511399.511336.

[26] Performance API project, URL: http://icl.cs.utk.edu/papi/overview/index.html.

[27] Smotlacha V. Measurement of Time Servers. CESNET Technical Report. 2001.

[28] Zaparanuks D. 2008. Accuracy of Performance Counter Measurements. Technical Report, University of Lugano.