Defence Research and Development Canada | Recherche et développement pour la défense Canada

DEFENCE | DÉFENSE

Monitoring and tracing of critical software systems

State of the work and project definition

By alphabetical order:

R. Charpentier
M. Couture
DRDC Valcartier

M. Dagenais
M. Desnoyers
P. M. Fournier
G. Matni
École Polytechnique de Montréal

D. Toupin
Ericsson, Montréal

Defence R&D Canada – Valcartier
Technical Memorandum
DRDC Valcartier TM 2008-144
December 2008

Canada

Monitoring and tracing of critical software systems

State of the work and project definition

By alphabetical order:

R. Charpentier
M. Couture
DRDC Valcartier

M. Dagenais
M. Desnoyers
P. M. Fournier
G. Matni
École Polytechnique de Montréal

D. Toupin
Ericsson, Montréal

Defence R&D Canada – Valcartier

Technical Memorandum
DRDC Valcartier TM 2008-144
December 2008

Principal Author

Mario Couture

Defence Scientist

Approved by

Guy Turcotte

SdS Section Head

Approved for release by

Christian Carrier

Chief scientist

This work was done between December 2007 and March 2008 under the work unit 15BZ10.

Abstract

This document presents a summary of the results of the tutorial/workshop “Monitoring Distributed Multi-core IT Systems for Optimization and Security”, which was held in January ’08 at Ericsson’s Montreal office. The aim of this event was to identify important problems and challenges that are experienced by researchers and industry in this domain, as well as potential avenues of research and development that may lead to concrete solutions for critical operations.

A brief review of the state of the art in this domain is presented as an introduction. The main results of the January ’08 tutorial/workshop are then presented. These results will eventually be used in the definition and set-up of R&D projects designed to put forward potential solutions for critical operations. The annexes contain complete information relative to the January ’08 event, a brief description of tools that are currently used in this domain, as well as a glossary.

Résumé

Ce document présente un résumé des résultats qui ont été obtenus lors du tutoriel/atelier « Monitoring Distributed Multi-core IT Systems for Optimization and Security », lequel a eu lieu dans les bureaux d’Ericsson à Montréal en janvier 2008. L’activité visait à identifier les problèmes et défis importants que les chercheurs et l’industrie rencontrent dans ce domaine, ainsi que des pistes de recherche et de développement potentielles pouvant mener à des solutions concrètes pour les opérations critiques.

Une brève revue de l’état de la technologie de pointe dans ce domaine est d’abord présentée en introduction. Les résultats principaux de l’activité de janvier 2008 sont ensuite présentés. Ces résultats seront éventuellement utilisés lors de la définition et de la mise en œuvre de projets de R et D visant à pousser plus loin des solutions potentielles pour les opérations critiques. Le lecteur trouvera dans les annexes l’information complète relative à l’activité de janvier 2008, une brève description des outils utilisés dans ce domaine ainsi qu’un glossaire des termes utilisés.

This page intentionally left blank.

Executive summary

Monitoring and tracing of critical software systems: State of the work and project definition

M. Couture; M. Dagenais; D. Toupin; R. Charpentier; G. Matni; M. Desnoyers; P.M. Fournier; DRDC Valcartier TM 2008-144; Defence R&D Canada – Valcartier; December 2008.

Introduction:

Nowadays, most military operations have to rely on sophisticated software and hardware infrastructures. The ever-increasing complexity of these systems along with disruptions from the environment in which they are used makes it extremely challenging to ensure system reliability. These systems have complex networked topologies, and each computer may now employ central processing units that have evolved from simple processors to symmetric or asymmetric multi-processors (SMP/ASMP), non-uniform memory access (NUMA) and more recently multi-core (SMP/ASMP on a single chip) systems.

These systems are becoming exceedingly difficult to analyze, debug and tune. In particular, the process of monitoring and tracing such distributed systems demands highly sophisticated tools to ensure continuous optimization and robustness assurance. Moreover, many problems (often related to timing) only show up under real computing loads, when the hardware and software are interacting in real time. The state of the art in this field shows that tool improvements are currently needed to extract precise, globally ordered debugging and performance data while minimizing the overhead on the systems under test.

This document focuses on these monitoring and tracing challenges. Its aim is to identify and prioritize relevant R&D efforts that could yield solutions and answers to military and public security requirements. A brief overview of the state of the art in this domain is first presented as a basic introduction. A summary is given of the results of the January ’08 tutorial/workshop “Monitoring Distributed Multi-core IT Systems for Optimization and Security” (Mon/Trac Event, 2008). This summary also describes important problems and challenges that are encountered by academic researchers and industry in this domain, as well as potential R&D threads that may lead to concrete solutions for critical operations. The conclusion presents a summary of results and makes recommendations for pursuing the next phases of this R&D effort. The annexes contain complete information relative to the January ’08 tutorial/workshop, a brief description of tools that are currently used in this domain, as well as a glossary.

Results:

The work done by international experts during the January ’08 tutorial/workshop led to the identification of six main areas of R&D (called **R&D threads**, briefly described below) for which R&D efforts should be deployed in the coming years.

1. **Adaptive fault probing**: inserting or activating low-overhead probes in a live real-time software system.

2. **Multi-trace handling**: synchronizing events from numerous – possibly huge – traces collected on distributed multi-core systems.

3. **Multi-level trace analysis and correlation**: abstracting low-level events, correlating events from multiple related traces, and quantifying differences.

4. **Automated fault identification**: defining languages and structures for building symptom catalogues.

5. **System health measurement**: defining and testing system health metrics, and studying the possible activation of protection, adaptation and optimization services.

6. **Trace-directed modelling**: relating the events to a model of the traced system in order to provide higher-level answers such as decomposing the time taken for a request or estimating the benefits of adding resources (e.g. disk or memory).

Future plans

Following the January ’08 tutorial/workshop, recommendations are made regarding the next steps of this R&D effort. They are:

1. **Feasibility studies**: literature reviews relative to each R&D thread (and associated technologies) should first be completed. A limited number of technological options should then be selected and studied more in depth in order to identify the critical information that will aid the definition of an R&D project. Costs, risks and probable duration of the R&D efforts should be addressed (among other topics).

2. **R&D project to be proposed for funding**: selected solutions should be the subject of further, more in-depth R&D efforts. These efforts should take the form of a 3-year R&D project having six main work breakdown elements (WBEs), one for each of the six R&D threads. Collaboration opportunities between governmental, academic and industrial organizations should first be defined. The aim is to form a robust and integrated team of professors, students (MSc and PhD) and other key experts having complementary areas of expertise. Based on all this theoretical and technical information, the team should then complete the project proposal and submit it for funding.

Sommaire

Monitoring and tracing of critical software systems: State of the work and project definition

M. Couture; M. Dagenais; D. Toupin; R. Charpentier; G. Matni; M. Desnoyers; P.M. Fournier; DRDC Valcartier TM 2008-144; R & D pour la défense Canada – Valcartier; Décembre 2008.

Introduction ou contexte :

De nos jours, la plupart des opérations militaires doivent s’appuyer sur des infrastructures sophistiquées tant logicielles que matérielles. La complexité toujours grandissante de ces systèmes, combinée aux perturbations provenant de l’environnement dans lequel ils sont utilisés, rend leur fiabilité extrêmement difficile à garantir. Ces systèmes ont des topologies de réseaux complexes et chaque ordinateur peut maintenant être fait d’unités centrales de calcul qui ont évolué : des processeurs simples aux processeurs multiples symétriques ou asymétriques (SMP/ASMP), avec accès non uniformes à la mémoire (NUMA), et plus récemment, aux systèmes multi-cœurs (SMP/ASMP sur une simple microplaquette).

Ces systèmes deviennent extrêmement difficiles à analyser, déboguer et ajuster. En particulier, le processus de monitorage et de traçage de tels systèmes répartis nécessite des outils hautement sophistiqués pour assurer leur optimisation continue et leur robustesse. De plus, plusieurs problèmes (souvent liés à la synchronisation) ne deviennent apparents que lorsque ces systèmes font l’objet d’une utilisation intense, lorsque le matériel interagit en temps réel avec le logiciel. La technologie de pointe dans ce domaine montre que les outils utilisés doivent être améliorés afin de permettre l’extraction de l’information de débogage et de performance, d’une façon globalement ordonnée et précise, tout en minimisant les coûts en ressources des systèmes.

Ce document est directement lié aux défis de monitorage et de traçage. Son but est d’identifier et prioriser les efforts pertinents de R et D qui pourraient déboucher sur des solutions pour répondre aux besoins des militaires et de la sécurité publique. Une brève revue de la technologie de pointe dans ce domaine est d’abord présentée comme introduction de base. Un sommaire des résultats obtenus lors du tutoriel/atelier « Monitoring Distributed Multi-core IT Systems for Optimization and Security » (Mon/Trac Event, 2008) de janvier 2008 est ensuite présenté. Les problèmes et défis importants qui sont rencontrés par les chercheurs académiciens et l’industrie dans ce domaine, ainsi que les voies potentielles de recherche et développement pouvant mener à des solutions concrètes pour les opérations critiques sont ensuite décrits. La conclusion présente un sommaire des résultats et propose des recommandations pour la conduite des phases suivantes de cet effort de R et D. Le lecteur trouvera dans les annexes l’information complète relative au tutoriel/atelier de janvier 2008, une brève description des outils qui sont couramment utilisés dans ce domaine ainsi qu’un glossaire définissant les termes en usage.

Résultats :

Les travaux effectués par les experts internationaux, lors du tutoriel/atelier de janvier 2008 ont permis l’identification de six sous-domaines de R et D (appelés **options de R et D**) pour lesquels

des efforts de R et D devraient être déployés dans les prochaines années. Ils sont listés et brièvement décrits dans les lignes suivantes :

1. **“Adaptive fault probing”**: Insertion et activation de sondes ayant peu d’effets dans les systèmes de type « temps réel », incluant les sections relatives au contexte des interruptions.

2. **“Multi-trace handling”**: La synchronisation d’activités provenant de plusieurs traces d’exécution (possiblement énormes) qui ont été recueillis sur des systèmes répartis, lesquels sont peut-être multi-cœurs.

3. **“Multi-level trace analysis and correlation”**: Abstraction d’activités de bas niveau, corrélation d’activités provenant de traces multiples reliées, quantification des divergences.

4. **“Automated fault identification”**: Définition de langages et structures permettant la mise au point de catalogues de symptômes.

5. **“System health measurements”**: Définition et vérification de mesures relatives à l’état de santé du système, étude de l’activation de moyens de protection, services d’adaptation et d’optimisation.

6. **“Trace-directed modelling”**: Relier les événements survenant dans un système à un modèle de ce dernier dans le but de fournir des réponses de plus haut niveau, telles que la décomposition temporelle de requêtes ou l’estimation des bénéfices attendus de l’ajout de ressources (par exemple : disque dur, mémoire).

Perspectives :

À la suite du tutoriel/atelier de janvier 2008, des recommandations concernant les prochaines étapes de cet effort de R et D ont été faites. Elles sont :

1. **Études de faisabilité** : dans un premier temps, les revues de la littérature relative à chaque option de R et D et technologies associées devraient être complétées. Un nombre limité de solutions technologiques devrait ensuite être sélectionné et étudié plus en profondeur dans le but d’identifier l’information critique qui va faciliter la définition d’un projet de R et D. Entre autres, les coûts, risques et durées probables des efforts de R et D devraient être décrits.

2. **Projet de R et D à être proposé pour financement** : les solutions sélectionnées dans les études de faisabilité devraient faire l’objet d’efforts de R et D plus approfondies. Ces efforts devraient prendre la forme d’un projet de R et D de trois années ayant six sous-éléments principaux, chacun d’eux correspondant à une des six options de R et D. Les possibilités de collaboration entre les organisations gouvernementales, universitaires et industrielles devraient premièrement être identifiées. Le but est de former une équipe solide et intégrée de professeurs, d’étudiants (M.Sc. et Ph.D.) et autres experts-clés dont l’expertise est complémentaire. Utilisant cette information théorique et pratique, cette équipe devrait compléter la proposition de projet et la soumettre pour financement.

Table of contents

Abstract …….. i
Résumé ….... i
Executive summary iii
Sommaire v
Table of contents vii
List of figures ix
List of tables x
Acknowledgements xi
1 Monitoring and tracing of critical software systems 1
1.1 Introduction 1
1.2 Goals, contexts and scopes of this document 1
1.3 Used methodology 2
1.4 How to use this document 4
2 Overview of the state of the art on monitoring and tracing 5
2.1 Acquired data – Their types and forms 5
2.2 Data providers – The Methods 7
2.2.1 Sampling 7
2.2.2 Automatic and manual static source-level instrumentation 8
2.2.3 Static and dynamic binary instrumentation 9
2.2.4 Emulation 10
2.3 Data acquisition – How to keep the acquired data 11
2.3.1 Writing to disk 11
2.3.2 Sending to the network 11
2.3.3 Circular memory buffers in overwrite mode 11
2.4 Data analysis and visualization 12
2.4.1 On-line versus off-line analysis 12
2.4.2 Visualization tools 13
2.4.3 Some potential problems with analysis of execution traces 14
2.5 Frameworks – Integrating the tools 14
3 January ’08 Tutorial/workshop – Overview of results 16
3.1 Initial drivers of the event 16
3.2 Identified long-term challenges 16
3.3 R&D threads and associated concepts and technologies to be investigated 17
4 Concluding remarks and recommendations 21
4.1 Recommendations and guidelines for the next steps 21
4.1.1 Six R&D threads 21

4.1.2 Feasibility studies 22
4.1.3 R&D project to be proposed for funding 22
References 23
4.2 Referenced papers and thesis 23
4.3 Referenced Web sites 23
Annex A .. The “Monitoring Distributed Multi-core IT Systems for Optimization and Security” workshop 27
A.1 Scientific steering committee 27
A.2 List of attendees 27
A.3 List of presentations – Day one 30
A.4 Eclipse Tracing Monitoring Framework BOF – Day one 32
A.4.1 Participants 32
A.4.2 Discussions (Chairman’s personal notes) 32
A.5 System Level Target Data Collection BOF – Day one 33
A.5.1 Participants 33
A.5.2 Discussions (Chairman’s personal notes) 33
A.6 List of presentations – Day two 35
A.7 Concluding panel (attendee’s personal notes) – Day two 37
A.8 List of presentations – Day three 38
A.9 Some pictures of the event 40
Annex B .. Overview of selected tools 41
B.1 Data providers 41
B.1.1 DTrace 41
B.1.2 SystemTap 42
B.1.3 Valgrind 43
B.2 Frameworks 44
B.2.1 Eclipse Framework 44
B.2.2 Intel VTune 45
B.2.3 LTTng 46
B.2.4 QNX Momentix 47
B.2.5 WindRiver Workbench 49
B.2.6 Zealcore System debuger 52
Glossary 53
Distribution list 69

List of figures

Figure 1. The seven steps of the methodology. 3

Figure 2. Observations and corresponding types of analysis. 6

Figure 3. Four categories for grouping data analysis. 13

Figure 4. From left to right: D. Toupin, M. Dagenais and R. Charpentier. 40

Figure 5. Attendees of the workshop. 40

List of tables

Table 1. List of attendees (January ’08 tutorial/workshop). 27

Acknowledgements

The authors of this document would like to thank Ericsson for the financial and hosting support of the January ’08 tutorial/workshop. They would like to thank all the participants for the quality of the presentations that allowed for the development of a very comprehensive understanding of the state of the art in the field of tracing and monitoring.

This page intentionally left blank.

1 Monitoring and tracing of critical software systems

1.1 Introduction

Nowadays, most military operations have to rely on sophisticated software and hardware infrastructures. The ever-increasing complexity of these systems along with disruptions from the environment in which they are used makes it extremely challenging to ensure system reliability. These systems have complex networked topologies, and each computer may now employ central processing units that have evolved from simple processors to symmetric or asymmetric multi-processors (SMP/ASMP), non-uniform memory access (NUMA) and more recently multi-core (SMP/ASMP on a single chip) systems.

These systems are becoming exceedingly difficult to analyze, debug and tune. In particular, the process of monitoring and tracing such distributed systems demands highly sophisticated tools to ensure continuous optimization and robustness assurance. Moreover, many problems (often related to timing) only show up under real computing loads, when the hardware and software are interacting in real time. Efficient and effective monitoring and tracing tools are needed to extract precise and globally ordered monitoring, debugging and performance data while minimizing the overhead on the systems under test. The tools must handle, for example, the potentially millions of significant events per minute that may be found on a 64-processor system running at several GHz.

A tutorial/workshop was held in January 2008 (Mon/Trac Event, 2008) in order to survey the tracing and monitoring field. Several of the most advanced players in the areas of advanced communications, information management and computer security discussed the state of the art regarding current tools, associated problems and unmet needs. The most promising avenues for solutions were identified with the guidance of key industrial, governmental and academic researchers in the field. Representatives from Defence R&D Canada at Valcartier, Enea, Ericsson, Freescale, IBM, MontaVista, Nokia, Rational, Red Hat, Oracle, Wind River and ZealCore were present.

This document focuses on these monitoring and tracing challenges. Its aim is to identify and prioritize relevant R&D efforts that could yield solutions and answers to military and public safety requirements. A brief overview of the state of the art in this domain is presented first, followed by a summary of the results of the January '08 tutorial/workshop. This summary also describes important problems and challenges that are encountered by academics and industry in this domain, as well as potential avenues of research and development that may lead to concrete solutions for critical operations. The conclusion presents a summary of results and makes recommendations for pursuing the next phases of this R&D effort. The annexes contain complete information on the January '08 tutorial/workshop, a brief description of tools that are currently used in this domain, as well as a glossary.

1.2 Goals, contexts and scopes of this document

The context of this work is closely related to critical operations (telephony and military) involving complex systems such as those briefly described in Section 1.1. On-the-fly optimization and adaptation of these critical systems (through monitoring, tracing, analysis and control) is the long-term vision motivating this R&D investment. However, practical considerations expressed by the user community are not neglected. They include concerns such as providing the operator with an integrated, easy and readily understood views of software systems' health.

The main goals of this document are to:

- give an overview of the state of the art in monitoring and tracing;
- summarize the results that were captured at the January '08 tutorial/workshop;
- summarize the problems encountered by the end-user community and what they need to deal with them;
- identify potential solutions (R&D threads) that could address these problems; and
- make propositions regarding future R&D efforts.

1.3 Used methodology

The methodology used to define this R&D project consists of a suite of logically ordered steps (Figure 1), where each step builds on the results of the previous ones. The process ultimately aims to define R&D projects that will provide robust solutions for problems and needs that are currently encountered during operations. An overview of this methodology follows.

The formulation of requirements[1] (**1- Formulation of requirements**; Figure 1) starts with the identification and description of operational problems and needs (step 0). Needs and requirements are iteratively refined by scientists with the active contribution of representatives from the end-user community and technology specialists (two opposing arrows between steps 0 and 1).

Based on this work, a number of potential solutions are identified out of the scientific literature by scientists in step 2 (**2- Identification of potential solutions**). They typically require in-depth scientific investigations to be critically reviewed and validated.

The third step of the process consists in defining a state of the art[2] (**3- State-of-the-art; SOTA**) on identified problems, needs and, more specifically, potential solutions. The SOTA should identify, capture and integrate all relevant concepts, research, authors, organizations, technologies, processes, tools, etc., into an integrated, holistic and comprehensive view. The content of the SOTA should make it possible to validate the relevance and potential of the solutions identified.

In the next step (**4- Tutorial/workshop**[3]), the content of the SOTA is shared among many stakeholders (international experts, civilian and military partners of DRDC, academics, and

[1] A number of requirements were identified to address telephony and military operational problems and needs. A high-level overview of them is given in Section 1.1.
[2] A SOTA regarding current and potential monitoring and tracing technologies will be published this year.
[3] This document provides a description of results that were obtained in step (4), during the January '08 tutorial/workshop.

possibly industry). These people are invited to elaborate on pre-determined aspects of the solutions identified. The goals of the tutorial include:

1. spreading knowledge among stakeholders;

2. advancing our understanding of the problems and needs; and

3. providing complete answers to attendees' questions (particularly DRDC partners).

Just after the tutorial, a half-day workshop involving DRDC partners is held. Based upon acquired knowledge, participants are invited to have in-depth discussions regarding more specific problems and needs and the prioritization of strategically selected solutions. This step provides stakeholders all the information they need to make strategic decisions.

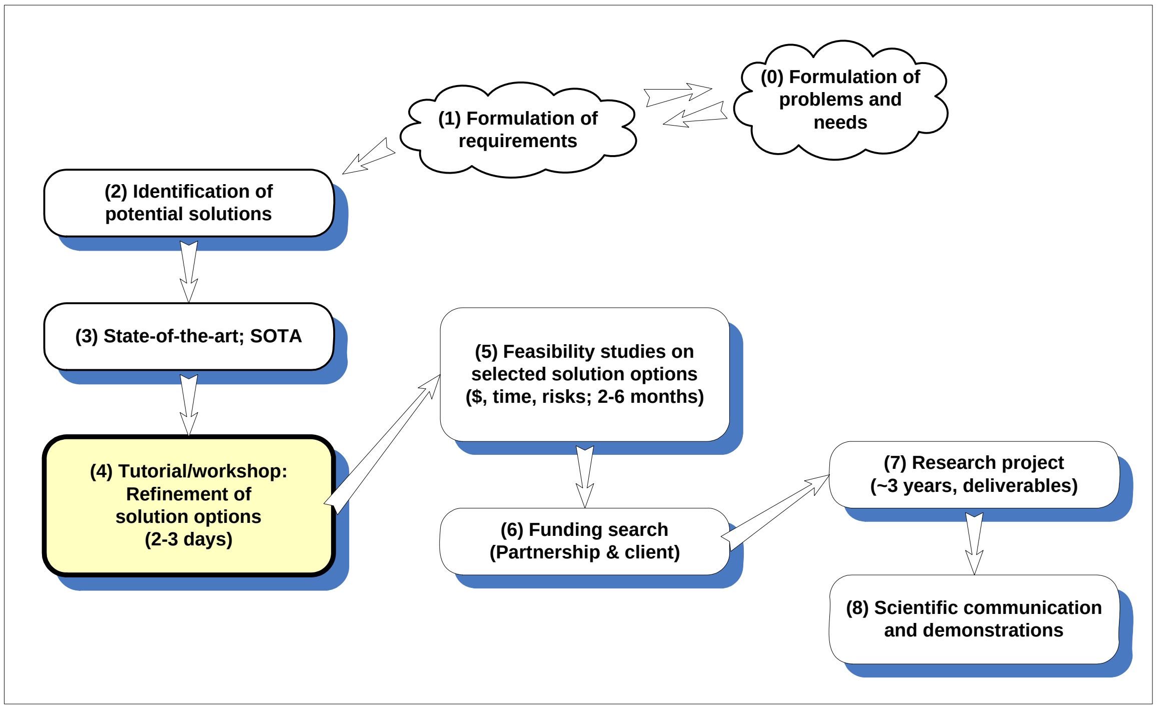
(0) Formulation of problems and needs
(1) Formulation of requirements
(2) Identification of potential solutions
(3) State-of-the-art; SOTA
(4) Tutorial/workshop: Refinement of solution options (2-3 days)
(5) Feasibility studies on selected solution options ($, time, risks; 2-6 months)
(6) Funding search (Partnership & client)
(7) Research project (~3 years, deliverables)
(8) Scientific communication and demonstrations

Figure 1. The seven steps of the methodology.

In step 5, feasibility studies are launched in order to estimate the cost, the risks and the time needed to study/develop preferred technological solutions. Identified R&D projects are then formally proposed for funding (step 6), and then executed (step 7), ultimately leading to field demonstrations and other forms of publication (step 8), which allow the end-user community to evaluate the results.

Iterative and incremental cycles may be needed to improve current solutions and find new ones. The process shown in Figure 1 may be cycled through many times; a different part of the solution would be studied in each cycle, thereby building on the results of preceding cycles.

1.4 How to use this document

This document describes the effort that was completed during step 4 of the methodology described above. Chapter 2 presents an overview of the SOTA that was defined in this domain. Chapter 3 presents the end-results of the January ’08 tutorial/workshop, and the Conclusion presents an overview of results and recommendations for the next steps.

The annexes contain complete information on the January ’08 tutorial/workshop event, a brief description of some selected tools that are used in this domain, as well as a glossary of technical terms.

It is worth noting at this point that the terms **software application** and **software system** are used in this document to refer to two things that are somewhat similar but that stand at two different conceptual levels. Software application (or program) refers to a piece of software while software system refers to a system that is made up of many pieces of software (the whole entity). Two examples will help clarify the difference between these terms.

- Example 1: the software application would be MS Word and the software system would be MS Word plus the MS XP operating system.
- Example 2: the software application would be a specific kernel module of the Linux operating system and the software system would be the Linux operating system itself (including all modules).

In this document, text that is written *in italics* indicates that it was borrowed verbatim from other sources. The reader is invited to consult the references for more information on these quotes.

2 Overview of the state of the art on monitoring and tracing

An overview of the state of the art in monitoring and tracing is presented in this chapter. It sets out the core information required to understand the upcoming work.

The field of computer science known as “monitoring and tracing” can be broken down into the following major components, each discussed in a specific section.

- **Data types** (Section 2.1): the types of data that are captured during monitoring and tracing.
- **Data providers** (Section 2.2): the basic mechanisms used to access (or acquire) the needed data.
- **Data acquisition** (Section 2.3): the process of writing captured data to a trace buffer, later to be written to disk or sent through the network.
- **Data analysis and visualization** (Section 2.4): once the captured data is available either through a file on disk or through a stream, it can be read by analysis tools. Data analysis tools can measure or identify a number of properties or metrics, for example, state (of processes, CPUs or devices at any given time); duration (of process execution or disk requests); or average (of CPUs or devices queue length). Analysis tools may also aid the detection of patterns of interest (disk input/output bottlenecks, frequent network retransmissions or timeouts).
- **Frameworks** (Section 2.5): framework is the entity within which most of the user interaction normally takes place. Accordingly, while the other sections focus more on the underlying mechanisms, the section on frameworks lists various tools that are currently available. The reader may also want to consult Annex B for an overview of the most commonly used tools in the market.

2.1 Acquired data – Their types and forms

The basic term for naming the process of information acquisition is **observation**. **Data** is the basic object resulting from observations. The analysis of software systems involves the close examination of this data as well as the program itself (both the source code and binary files).

Figure 2 shows that two main types of analysis can be done on software applications, each performed on a specific type of observation[4]:

- **Static analysis**: involves observations made on static (not running) source or binary code.
- **Dynamic analysis**: involves observations made on a running software system.

Types of data resulting from these observations are briefly described below.

[4] This text is presented as an introduction to the types of data that can be observed on software systems. More details on data analysis are given in Section 2.4.

- **Data independent of time**: the study of the source code of a software application provides observations that reveal its composition, structure and internal logic.
- **Data at one specific time "t"**: an instantaneous image of observations, showing the value of many variables of a software application at an instant "t", allows the study of its state at that specific moment.
- **Chronological and logical list or suite of data**: chronological observations of the same selected variables over time allows the study of the evolution of the software application's internal states and time-dependent paths of execution and scheduling.
- **Compiled statistical data**: statistics on a specific aspect of the software system may be studied if observations of the same variable(s) are recorded and compiled over time (at regularly or irregularly spaced time intervals). Examples are averages, durations, correlations, standard deviations and cross-correlations.

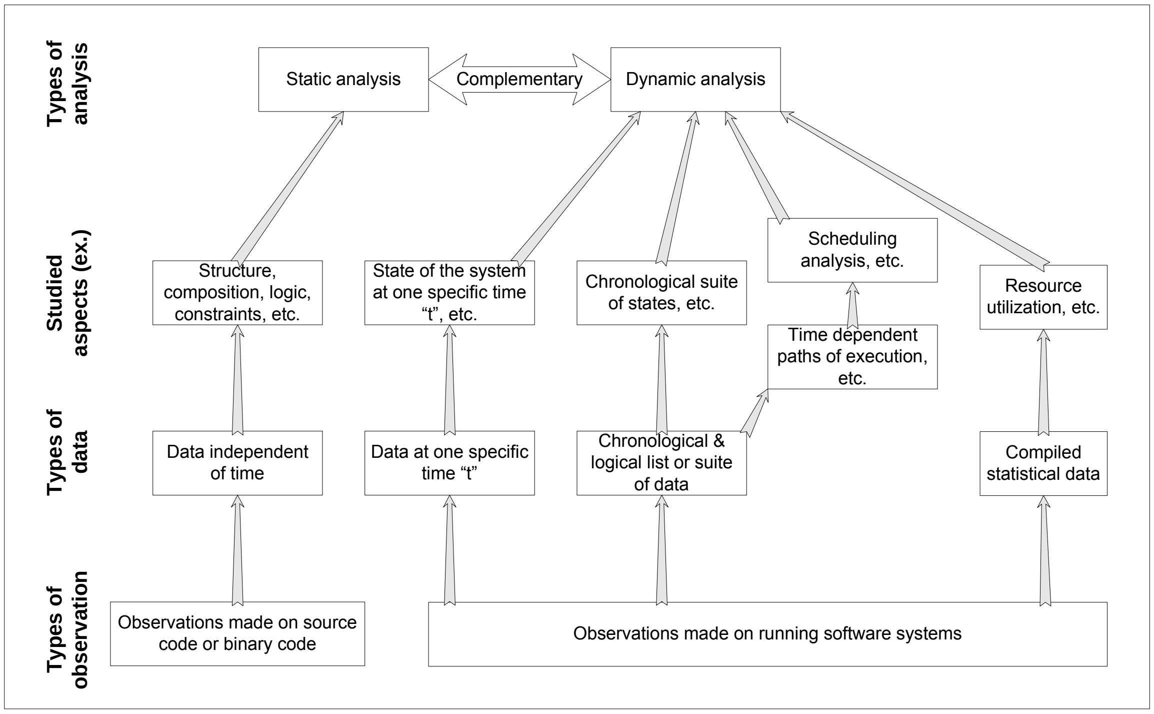
Types of analysis
Static analysis
Complementary
Dynamic analysis
Studied aspects (ex.)
Structure, composition, logic, constraints, etc.
State of the system at one specific time "t", etc.
Chronological suite of states, etc.
Scheduling analysis, etc.
Time dependent paths of execution, etc.
Resource utilization, etc.
Types of data
Data independent of time
Data at one specific time "t"
Chronological & logical list or suite of data
Compiled statistical data
Types of observation
Observations made on source code or binary code
Observations made on running software systems

Figure 2. Observations and corresponding types of analysis.

The following are two important terms that will be used throughout this document.

Sample: a sample is one record of a specific observation that was made on a running system at one specific instant "t". The record may be composed of one or more fields. Examples are: the degree of network load or the complete image of the system's memory at one given time.

Trace: a trace is a chronological suite of records resulting from observations that were made on a running system at different instants "t_n", during a specific period "delta-t". Traces are ordered lists of records specifying instructions that were executed on one or more local/distributed single-core/multi-core CPUs over time.

2.2 Data providers – The Methods

Three methods can be used to collect data out of a running software application. The first one is relatively simple: it is called **monitoring** (or **sampling**). It collects selected data through periodic external interventions during software execution. For example, as the time reaches a certain point, the execution of the application is stopped, the collection process starts, ends, and then the application’s execution is resumed. Time intervals for monitoring can be fixed or variable (even random). They can also be triggered by pre-identified events that occur during the execution of the application. Monitoring is achieved without having to add supplementary code to the software. Section 2.2.1 provides a brief description of sampling.

The second method that can be used to collect data out of a running software application is called **instrumentation**. It involves the addition of specific tasks or functions (in the form of code) at specific points in the software. As the application reaches one of these points, the code that was inserted is executed (the data is collected), and then the application continues its execution up to the next instrumentation point. There are two main types of instrumentation: 1) **automatic** or **manual static source-level instrumentation** and 2) **static** or **dynamic binary instrumentation** (described in sections 2.2.2 and 2.2.3 respectively).

A third method, which is quite different from the two defined above, can be used to collect data out of a virtually running software application: **emulation**. It is described in Section 2.2.4.

Strategies may be conceived for the concurrent use of one or more of these three methods.

2.2.1 Sampling

Sampling may be used to collect copious data about a running software application with relatively low disturbance. At regular intervals, the unmodified application is interrupted by an external source (e.g. programmable interrupt timer) and a number of variables of the system may be sampled. Examples are execution address and execution mode. The overhead associated with sampling can be controlled by changing the sampling frequency (and possibly the number of variables sampled). Tradeoffs between the sampling frequency and the tolerated overhead will determine the level of accuracy (level of detail) of the profile.

GProf (GNU, 2008), for example, uses the operating system virtual timer to interrupt the OS every 0.01 or 0.001 s of CPU time. At each interruption, the address of the currently executing instruction is taken and the counter associated with that region is incremented. The data collected are then used to further analyze the system. For example, GProf can instrument each function entry in order to produce a call graph. Some examples of performance are: running Gzip 1.2.4 on a 64 MB log file (Dagenais et al., 2005) takes 28.16 seconds. With sampling, the same task requires 28.30 seconds, and with sampling and function entries instrumentation, the task requires 29.88 seconds.

Most modern CPUs provide embedded performance counters which can be used for sampling. **Time sampling** is achieved by counting clock cycles and requesting an interrupt after, say, 1,000,000 clock cycles. Similarly, it is possible to count instructions, branch delays and many other metrics which may help define performance. By sampling the address of the currently executed instruction, one eventually gets histograms of relevant metrics (time, cache misses, branch delays, etc.) for the different program regions. It is then easy to identify the program sections that are consuming the most CPU time or causing the most cache misses. OProfile (OProfile, 2008) is an example of a performance analysis tool which runs on Linux and is based on performance counter sampling.

2.2.2 Automatic and manual static source-level instrumentation

Automatic static source-level instrumentation involves specialized tools that parse the source code, possibly generate a control flow graph, and then add the instrumentation code at specified points. Following are some available solutions for automatic static source-level instrumentation.

- Instrumentation code may be inserted automatically by compilers in the process of converting high-level language functions to a lower-level intermediate or assembly code. For example, the GNU Compiler Collection (GCC; GNU, 2008) provides a number of instrumentation compilation options for debugging and instrumentation purposes. Instrumentation typically provides information for profiling and code coverage tools. Examples of GCC compilation options are : *-finstrument-functions, -ftest-coverage* and *-fprofile-arcs*.
- Automatic instrumentation may also be done by using one of several source code parsing and transformation toolkits (available for various programming languages). Notable examples include TXL (TXL, 2008), and Javacc (Javacc, 2008), which is used by the Java Instrumentation Engine (JIE, 2008).

Manual static source-level instrumentation requires that the developers specify explicitly the instrumentation points in their target application source code. Trace points are thus manually inserted at pre-defined locations in the source code, where significant events occur. This kind of instrumentation may help us understand software system dynamics, achieve performance improvements, and generally troubleshoot. Static source-level instrumentation does not have the same constraints as dynamic binary instrumentation, and thus allows more flexibility for inserting low-overhead instrumentation. Following are some available solutions for manual static source-level instrumentation.

- **Libraries**: manually inserted instrumentation is present in most large systems for logging or tracing purposes. Libraries exist to provide simple single-line instrumentation statements in most programming languages. An example is the low-performance Linux kernel *printk*[5] statement, which is not optimized for modern multi-processor or real-time contexts.
- **High-level language API**: besides the usual print statement, **tracing** or **logging libraries** are available for most high-level programming languages. There are several frameworks for Java logging (JavaLoggin, 2008), .NET logging (NetLoggin, 2008), and C++ logging (C++loggin, 2008). JDK 1.4 was officially released in 2002 and included the Java logging API. This API provides a way to manually instrument Java source code for debugging purposes. The target application needs to create instances of the logger object, and for each one it may assign a different handler object (JavaHandler, 2008). The logger object can set

[5] Instruction *printk* is the kernel equivalent of *printf*, in standard C; standard C libraries are not available for the development of kernel modules. It may, for example, be used by developers of kernel modules to print debugging information on a non graphical console.

the level of any log message, and this can be used to control logging output. Handler objects also have their own level to filter out events at lower levels (SunWS, 2008; WikipediaWS, 2008).

- **Low-level language API**: as mentioned, Linux offered from the very beginning the low-efficiency but robust *printk* statement. Over the years, several **structured frameworks** have been proposed for logging and tracing. Examples are: Event Logger, or Evlog (Evlog, 2008); the Linux Kernel State Tracer, or LKST (LKST, 2008); the Linux Trace Toolkit, or LTT (LTT, 2008); the Driver Tracing Infrastructure, DTI (DTI, 2008); and the Linux Trace Toolkit next generation, or LTTng (LTTngWS, 2008). The buffering mechanism from LTT was called Relay (Relay, 2008). It was incorporated into the mainline Linux kernel in 2005. The source code instrumentation mechanism proposed by LTTng, Kernel Markers[6], was incorporated into the mainline Linux kernel in 2008. There seems to be general agreement that tracing is making its way into the mainline Linux kernel, but tradeoffs between simplicity, efficiency, low intrusiveness and low disturbance have to be made.

- **Driver Tracing Interface**: the device drivers constitute the largest portion of the Linux kernel source code. Many drivers or subsystems need to be traced for performance analysis, troubleshooting and debugging. Some of these subsystems already come with ad hoc tracers for events like wireless connections or SCSI disk commands. The Driver Tracing Interface (DTI) (DTIWS, 2008) was initially designed to enable driver developers to log events in a circular flight-recorder type buffer. The typical usage model consists in extracting and analyzing the buffer content after a system crash, trying to pinpoint the source of the problem. DTI allows debug data (per-CPU buffering) to be passed into user space using the Relay file system. The DTI API is architecture-independent, and is usable in both user and interrupt contexts. It can be used to register new traces, write a formatted string or a buffer to the trace, and set the trace level (e.g. from 0 to 6), which is used to select which level of records to keep. A tool for formatting traces is provided to extract, sort and format the events from each per-CPU data file. A major characteristic of DTI is that it generates a log in the kernel from the very beginning of the *start_kernel ()* init code. The DTI project has not been very active lately. However, with the inclusion of the Kernel Markers, there is now a mechanism in place to instrument the various device drivers.

2.2.3 Static and dynamic binary instrumentation

Binary instrumentation adds trace points to an application that is already compiled. This may be done **statically** (before the application is executed) or **dynamically** (while the application is

[6] The **Linux Kernel Markers** instrumentation mechanism provides static instrumentation of the kernel. These markers can be enabled dynamically. A marker placed in code provides a “hook”, which is used to call a function (or “probe”) that can be provided at runtime. A marker can be “on” (a probe is connected to it) or “off” (no probe is attached). When a marker is “off” it has no effect, except that it adds a tiny time penalty (for checking a condition for a branch) and a space penalty (for adding a few bytes for the function call at the end of the instrumented function and adding a data structure in a separate section). When a marker is “on”, the associated function is called every time the marker is executed. The execution returns to the caller when the function ends its execution. Markers are usually put at significant locations in the code, and they represent lightweight hooks that can pass an arbitrary number of parameters (described in a *printk*-like format string) to the attached probe function. They can for instance be used for tracing and performance accounting. Kernel markers are currently used by LTTng and SystemTAP.

running). The main challenge consists in adding new instructions (e.g. trace points) while not disrupting the rest of the program.

Static binary instrumentation. Several tools were developed over the years in order to read executable binary programs and rewrite equivalent programs with some instrumentation added. Examples are:

- Pixie, QPT (QPT, 2008), EEL (EEL, 2008) and ATOM (ATOM, 2008). They were mostly developed for architectures having fixed-length instructions (MIPS, Alpha and SPARC), and some were available at no cost for research purposes.
- The complexity of the variable-length instruction set of the popular I386 Intel architecture makes program rewriting more difficult on this platform and requires other techniques (see dynamic binary instrumentation).

Dynamic binary instrumentation techniques are used to execute "extra instructions" at specific locations in a program. Since instrumentation is added dynamically, a tracing overhead is incurred only when a program is dynamically instrumented to produce a trace. This is unlike source-level instrumentation, where the decision to instrument is taken at compile time, and some overhead is present when tracing is compiled even if it is not activated at compilation time.

Valgrind (Valgrind, 2008), DTrace (DTrace, 2008), SystemTap (SystemTapWiki, 2008) and GDB (GDB, 2008) are examples of dynamic binary instrumentation tools. DTrace, SystemTap and GDB are typically used to add a few trace points or breakpoints, while Valgrind is more commonly used for pervasive monitoring of the program, for example, tracing every access to memory.

DTrace, SystemTap and GDB change executed instructions using a trap mechanism. Instructions are overwritten by trap instructions where instrumentation code must be executed. Once trap instructions have been executed, the original instructions (overwritten by the trap) are restored, the processor's pointer is appropriately modified, and the original program's instructions are executed.

Valgrind, on the other hand, uses an approach similar to that of static binary instrumentation. Before it is executed, the binary code is de-compiled into a higher-level intermediate representation, which is then instrumented as needed, and then re-compiled.

2.2.4 Emulation

Emulation consists in virtually reproducing the execution of a program with the use of specialized software. Total control over the emulated system is possible in a virtual emulated environment. Every emulated instruction execution or memory access is visible and can be instrumented. For example, the boot-up code for new microprocessors is often run on a gate level simulation of the new processor. The flexibility of this technique may however lower performance. For example, a program may run 10 or 100 times slower on an emulated system.

If the simulation is **cycle accurate** (i.e. the emulated elapsed time is measured accurately), the performance of the emulated system can be measured precisely, even if it runs much slower than real time. This technique is often used to measure the performance of the processor pipeline or cache memory.

Limitations may be encountered if the system interacts with other real-time distributed systems. For example, it may not be possible to use emulation for the performance analysis of a server answering many Web requests and interacting with a high-performance disk subsystem and database servers.

Recently, new emulators have shown improved performance. They analyze code sequences before they are executed and replace them with equivalent binary instructions. Qemu (Qemu, 2008) and VMWare (VMWare, 2008) implement such mechanisms. Another emulation system, Valgrind (Valgrind, 2008), is used primarily to instrument programs and to aid validation (memory accesses, proper use of **malloc** and **free**) and performance analysis (cache and heap profiler).

2.3 Data acquisition – How to keep the acquired data

Trace points, once reached, may check whether some conditions are met (e.g. tracing is currently requested for the current process). Then, they may write data describing the event either directly through a communication channel or to a buffer. The collected data from the various data providers can be extracted from the traced kernel in different ways, each involving a different disturbance on the system.

2.3.1 Writing to disk

Data collection may involve writing a huge amount of data (possibly gigabytes) to disk in order to get the full trace of the system. It allows a complete replay of the trace. It also provides complete information about the state of the system at any given time after the initial “state dump” is done. Under a medium to high event rate workload, a performance impact between 1% and 2% may be measured (Desnoyers and Dagenais, 2006a, 2006b). It will primarily impact the hard disk I/O performance.

2.3.2 Sending to the network

This is like writing to disk, but the data is sent to a remote computer through a network socket. Huge data may impact network performance.

2.3.3 Circular memory buffers in overwrite mode

This method consists in overwriting the oldest data in the buffers when they are filled. It is also called **flight recorder mode**. It is similar to a logic analyzer: when tracing is stopped, only the last few seconds of recording will be left in memory.

This kind of tracing can be very useful for diagnosing production systems because it provides meaningful information that can aid in resolving many problems. Writing to memory buffers is also a much cheaper way to extract data than sending it to network or disk. Under a medium to high event rate workload, a performance impact between 0.3% and 0.5% may be seen (Desnoyers and Dagenais, 2006a, 2006b).

2.4 Data analysis and visualization

Two types of analysis were briefly introduced in Section 2.1, namely static analysis and dynamic analysis. **Static analysis** is used to study the composition, structure and logic of software systems (in the form of static source code or binary code). **Dynamic analysis** is used to study the internal dynamics and evolution of software systems when they are executed. This section provides more information on the types of analysis that can be performed on acquired data.

2.4.1 On-line versus off-line analysis

Static analysis is usually done off-line (in the laboratory), whereas dynamic analysis can be done either off-line (in the laboratory) or on-line (during the execution of software systems).

- **Off-line dynamic analysis** involves the study of historical observed data (samples and traces) that was acquired from running software systems and saved for later use. It may also involve the study of observed data resulting from off-line and in-lab running (or replay) of the software applications, using specific scenarios of utilization. The results would take the form of a knowledge base that could eventually be used during operations as a support to proactive and reactive decision-making process. They could also serve as a key driver for on-the-fly (no-downtime) software maintenance, capability enhancement and adaptation.
- **On-line dynamic analysis** is performed during operations, at the same time as the software applications are executing. Observed data are analyzed in quasi-real time to identify potential problems and solutions. Traced data may be transferred to another machine (or CPU) for analysis in order to prevent overloading of the traced machine's CPU.

The main advantage of dynamic analysis is that, theoretically, it can provide access to almost any type of information describing the software system's behaviour. It could for instance involve the identification of all transactions within the software and their negative effects on the rest of the system. Following are examples of dynamic analysis.

- Performance analysis: profiling, elapsed time, delays, total number of events, causality links between events;
- dependency analysis: critical path;
- sensitivity analysis;
- state of the system: scheduling, processes, hardware, other logical resources;
- behaviour of the system: searching for specific patterns such as excessive swapping, spurious timeouts, overloaded disk subsystems;
- failure analysis, attack detection, differences between two running systems, robustness analysis, maintenance, evolution, and many others.

2.4.2 Visualization tools

Tools that perform analysis of observed data also produce views that help us understand software system behaviour. Thus, for practical visualization purposes, we add another criterion for classifying the types of analysis (and views) that can be produced on observed data: the **scope of the analysis** (Figure 3). This criterion is orthogonal to the first one (static/dynamic). It reveals whether or not an analysis addresses specific parts or modules of a software system (**local analysis**), or whether it studies the entire system (**global analysis**) (Yaghmour, 2001).

Local analysis studies both static and dynamic aspects of the “parts or components” of software systems. The semantics of the collected data and analysis are more significant for components than for the whole system. An example of local data would be a program counter used with a timer, and the resulting local analysis would be a time-based analysis of that specific program counter. GProf, Path Profiler, Quantify, Digital Continuous Profiling Infrastructure (DCPI) and Morph are examples of tools that can be used for local analysis.

Global analysis, in a broader perspective, studies both static and dynamic aspects of the whole software system (global states, emergent behaviours, etc.), and outputs a report on overall system health. System performance monitors such as UNIX *ps* and *top* are examples of tools that can be used for global analysis.

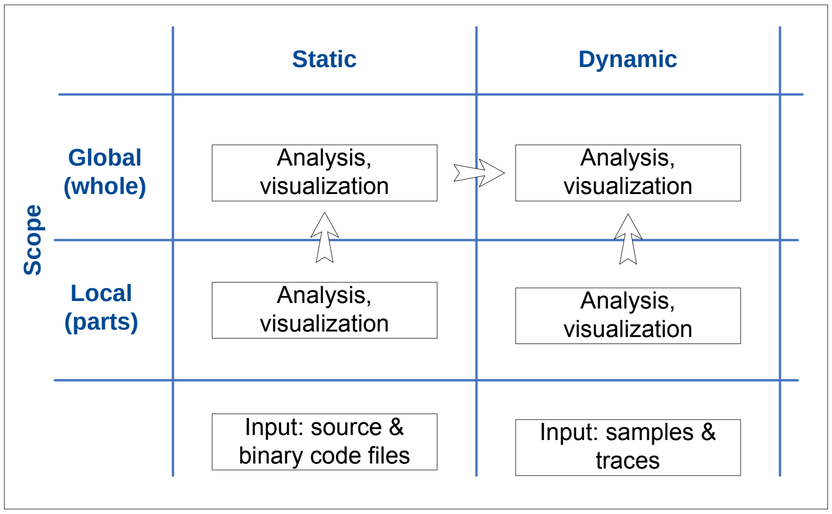

Figure 3. Four categories for grouping data analysis.

Examples of diagrams (or lists) are histograms showing the density of events with respect to time, graphical views showing statistics, graphical views showing structured data in the form of graphs, control views, lists of raw events (execution trace elements), etc. These diagrams offer functionalities such as bookmarking, pattern searching, condition checking, etc. A list of related tools can be found in Section 2.5.

2.4.3 Some potential problems with analysis of execution traces

Detailed dynamic analysis may involve the instrumentation of the operating system's kernel. Tracing instructions are added at strategic points in the kernel, to be called when specific events happen. The collected data is chronologically ordered in a buffer, allowing further reconstruction and understanding of the system dynamics.

Data analysis tools face the challenge of remaining interactive while handling huge amounts of data. Even in compact binary form, traces can be as large as 10 gigabytes or more, which generally exceeds the quantity of available RAM. Therefore, they cannot be completely loaded into memory. Moreover, when the user zooms in on a particular region of such huge traces, the I/O and computations necessary for extracting state information may take some time, and negatively impact the CPU.

2.5 Frameworks – Integrating the tools

Profilers and viewers can be integrated into frameworks that allow user interaction with data collectors, analyzers, viewers, etc. The framework may take the form of a global monitoring control console, which offers an interface that may provide a number of useful functions. For example, the user interface may be used to:

- selectively activate or deactivate static trace points;
- insert or remove dynamic trace points;
- start or stop the tracing of a given process, process group or the entire system;
- visualize the content of traces;
- act as programming environment, viewers;
- impose filters, assertions;
- trace bookmarks;
- show an event list to conveniently view a table of events, like in a database viewer;
- control flow view, show the state of different objects (e.g. process, disk, CPU) as a function of time, like Gantt charts;
- produce histograms of events (number of events of a certain type, value of a field for a certain type of event, etc.);
- produce graphs showing the number of disk read requests per second;
- derive statistics (total number of events by type, average duration, CPU usage, etc.).

Many such tools can be concurrently used within frameworks. They offer the flexibility of adding new analyses and viewing modules in the form of plug-ins. Some of them are briefly described in **Annex B**. Examples are:

- QNX (QNX, 2008);
- WindRiver (WindRiver, 2008);

- ZealCore (ZealCore, 2008);
- Intel VTune (IntelVTune, 2008);
- DTrace (DTrace, 2008);
- SystemTap (SystemTap, 2008);
- Frysk (Frysk, 2008); and
- LTTng & LTTV (LTTng, 2008).

3 January ’08 Tutorial/workshop – Overview of results

A technical tutorial/workshop involving many international experts in this domain (step 4 of the methodology presented in Section 1.3, page 2) was held in January 2008 at Ericsson Montreal (the “Monitoring Distributed Multi-core IT Systems for Optimization and Security” workshop; Mon/Trac Event, 2008). The goal of the January ’08 tutorial/workshop was to:

- ask experts[7] to present the edge of the knowledge relative to tracing and monitoring of computer systems, and
- ask advanced users to present the most challenging problems they are facing, and potential solutions.

The primary objective was to better target future R&D efforts in order to address problems faced by industrial and governmental organizations (for example, on-line diagnostics for security, reliability, accuracy, debugging, optimization, and resource utilization). This chapter presents the key findings of the January ’08 tutorial/workshop; the reader will find complete information on this event in Annex A.

3.1 Initial drivers of the event

The key points of the vision that was proposed to the workshop participants are:

- low-overhead instrumentation is critical to most real-world applications;
- equally important are in-lab debugging and in-field (online) monitoring;
- many software applications are distributed on many networked computers, which may be multi-core.

The technologies to be used in this study are:

- Linux will be considered as the most appropriate OS for this work[8];
- standard protocols, formats and integration frameworks will also be considered; and
- open-source reference implementations will be encouraged as well.

3.2 Identified long-term challenges

The following long-term challenges were identified by the workshop participants:

1) Full-spectrum trace visualization

[7] Ericsson practitioners (approximately 24 of them from Sweden, Finland, Italy and Canada) and domain experts (approximately 20 of them from the US and Canada).
[8] A wide variety of operating systems are encountered but Linux appears to be the most appropriate OS for R&D demonstration, since it is shared by a large community of users and expected to increase in importance in the future.

Full-spectrum trace visualization appears to be desirable in most analyses (i.e. from silicon, hypervisor, OS, VM, and simulator up to user application). Abstraction towards a modelling level was also expressed as a valuable visualization enabler in some system designs and analyses.

2) Multiple CPUs and multi-core systems

Understanding interactions between multiple CPUs and multi-core systems is also mandatory for complex system analysis, particularly to ensure scalability of performance.

3) Reusability and comparability of functions and traces

Tracing and monitoring functions should be designed to enable regressive testing and periodic (repetitive) maintenance.

4) Measurement of system health and interventions

Exploitation of low-level instrumentation to assess general system health of on-line components (at high level) and the activation of reactive measures when appropriate are also perceived as being critical to autonomous complex systems. Ultimately, a process of feedback-directed adaptation and optimization could emerge from such capabilities.

5) Forensics

Conditioning captured data for forensic exploitation could also be highly valuable for criminal investigations when malfunctions are suspected to originate from an attack on a key component of a critical infrastructure.

6) Ensure technology adoption in a broader community of users by prioritizing ease of use.

3.3 R&D threads and associated concepts and technologies to be investigated

Six key subjects or topics of R&D (here called **R&D threads**) were identified during the workshop. They are outlined below.

1) Adaptive fault probing

Static probes can be inserted at compilation time and remain dormant until activated at runtime, typically to generate tracing information as needed. Dynamic probes can be added at runtime to adapt the system behaviour, for example, to trace various parts of the OS for diagnosing a problem. DTrace is probably the best known recent implementation of static and dynamic probes. SystemTap, currently under development, offers a similar functionality for dynamic probes in Linux.

The group at Polytechnique has started working on static probes, providing an initial implementation for the mainline Linux kernel named Kernel Markers. The challenge is to simultaneously minimize the number of execution cycles required for the execution of a probe, including the effect of the added instructions on the memory cache, while not interfering with the real-time response of the system, and enabling the activation of probes even when the program may be simultaneously accessed by several processors on a multi-core system.

Some preliminary results have been promising. Further research and development to refine and optimize the underlying algorithms will be needed to provide a robust, low-disturbance and extremely efficient infrastructure for adaptive fault probing. While this infrastructure will be prototyped for Linux, the same algorithms will be reused in other operating systems in use at Ericsson, Defence R&D Canada and elsewhere to trace heterogeneous distributed systems.

2) Multi-trace handling

The probes installed at the different software layers (hypervisor, OS, virtual machine, system libraries, applications) may be used to provide monitoring and tracing data. Each processor, with its own local clock, then generates a steady flow of events. These events, from multiple cores on each system and from several distributed systems, must then be collected and stored efficiently. The events coming from the different cores must be synchronized.

The currently available system-level trace visualization tools are often used in small real-time embedded systems, or much less detailed system logs for larger systems. As a result, none of the systems evaluated for the January ’08 tutorial/workshop (Mon/Trac Event, 2008) were capable of handling traces of more than a few tens of megabytes.

The Linux Trace Toolkit Viewer, developed at École Polytechnique, is capable of handling huge traces of several gigabytes. However, further work is required to develop algorithms for the synchronization of events coming from multiple nodes, multiple cores and even multiple virtual machines. Also, a new architecture is required to handle huge traces while allowing traces to be collected from multiple systems and embedded devices, for both online and a posteriori off-line analysis and viewing.

3) Multi-level trace analysis and correlation

The detailed event lists gathered independently from several processors need to be processed to extract higher-level information suitable for analysis at higher levels of abstraction. A client-server request may for example be identified in a trace by finding specific sequences of “send” and “receive” messages. A file reading action is similarly represented by numerous, possibly out-of-order, disk block reads.

Trace reduction and abstraction are useful for reducing trace size; they also help the human viewer understand them. Moreover, they simplify the automated comparison of similar traces and identify equivalences in results, even if the systems’ implementation differs (an example would be two redundant servers having a different implementation).

Mechanisms developed in this thread will be useful in many contexts. Some of these are regression analysis of periodic run-time probing, monitoring of system performance, detection of performance degradations and their causes (security breaches, design flaws, programming errors, etc.).

4) Automated fault identification

By carefully examining execution traces of an information system, experts can detect problematic behaviours caused by software design flaws, inefficiencies or malicious activity. Examples of such faulty behaviours are excessive swapping, lock contention, undue latency, inefficient task scheduling, attempts to erase system logs and modification of system files.

Mechanisms for fault detection already exist in intrusion detection systems (IDS). Their functions include analyzing network packet traces and looking for patterns of attack. There are many specialized languages for representing fault conditions, scenarios, patterns, etc. These languages come in several varieties:

- domain-specific (such as Panoptis, Snort, NeVO);
- imperative languages (ASAX, BRO);
- finite state machines (STAT, IDIOT, BSML);
- expert systems (P-BEST, LAMBDA); and
- temporal logic (LogWeaver, Monid, Chronicles).

A similar approach is chosen in this project to provide a flexible, on-line automated fault identification mechanism for execution traces. The main goal of these mechanisms is to trigger alarms while the software is running, when specified problematic conditions, or when scenarios or patterns are detected in execution traces. Such a detection system will significantly improve the decision-making process and allow analysis while maximizing the time available for risk mitigation.

Problematic conditions detected in execution traces will first be examined closely to determine what semantics must be represented by the language. A SOTA will then be prepared to identify a number of complementary languages and potential solutions that could be used. Further R&D studies will then be needed to refine and integrate the solutions selected.

5) System health measurement

System irregularities and overall performance degradation will be identified by enabling the technologies described above (multi-trace analysis, automated fault identification, adaptive fault probing, etc.). The objective of this R&D thread is to define and validate quantitative measures that could be used to assess global system health. Indeed, system health can be evaluated using low-level (trace) metrics or from statistics computed directly in the probes.

These reliability measurements can be highly informative, allowing the system administrator to increase the level of surveillance or activate additional protective and reactive measures such as logging more information, modifying packet filtering rules, inserting and/or activating dynamic probes, modifying system behaviours, or simply shutting down some computers to protect them from hacking or malicious exploitation.

6) Trace-directed modelling

Many benefits can be achieved by processing trace events so as to reconstruct a higher-level model, such as a state machine or an interaction model. Operations that are necessary to build such a model include filtering out less relevant items (e.g. utilities) from the trace and detecting patterns of trace events that should be treated as more abstract entities. Key challenges are scope management and determining the appropriate abstractions; this can be facilitated by the use of the system's UML models (if available). An example would be the abstraction of a series of transmissions into a single transaction.

The reconstructed model can then be applied in several ways.

1. It can be used for anomaly detection and analysis by comparing it to either the original UML model or other models generated from previous correct or anomalous runs.

2. Delays, timing and resource usage can be analyzed at the level of the abstract entities in the model.

3. Various visualization techniques can be applied to the model.

4. It is possible to change the model (e.g. change the amount or distribution of resources or change a condition) and estimate the effect on system performance or behaviours.

Our industry partner Ericsson makes extensive use of the UML modelling tool Rational Software Developer - Real Time. They would benefit from the above approach, since currently their ability to correlate run-time data to their design models is insufficient. Zeligsoft has indicated an interest in the approach, and it is willing to contribute development resources to incorporate the results of the research into their tools.

The feasibility studies of selected concepts and technologies associated with the six R&D threads should be studied within the proposed R&D partnership in order to realize the long-term vision summarized above.

4 Concluding remarks and recommendations

Key R&D issues and potential solutions were presented in this document. They are based on the technical challenges described by the user community at the January ’08 tutorial/workshop (Mon/Trac Event, 2008).

Recommendations and guidelines for executing the next steps of the methodology (described in Section 1.3) are presented in this chapter.

4.1 Recommendations and guidelines for the next steps

4.1.1 Six R&D threads

The objective of the work done by international experts during the January ’08 tutorial/workshop was to study:

1. high-precision, low-overhead, low-disturbance mechanisms for inserting or activating software probes in a live system for the purpose of tracing or monitoring the system or adapting it to the changing environment;

2. combining and analyzing execution traces coming from distributed networked multi-core systems;

3. measurement of system health and identification of a wide range of faults and problems; and

4. correlation of the events in execution traces with the system models in order to determine critical paths and resource bottlenecks, and compute the performance to be expected if the system is upgraded (e.g. adding disks or memory).

Six main areas of R&D (**R&D threads**) that could lead to the implementation of such capabilities and achieve these objectives were identified. They are:

1. **Adaptive fault probing**: inserting or activating low-overhead probes in a live real-time system.

2. **Multi-trace handling**: synchronizing events from numerous – possibly huge – traces collected on distributed multi-core systems.

3. **Multi-level trace analysis and correlation**: abstracting low-level events, correlating events from multiple related traces and quantifying differences.

4. **Automated fault identification**: defining languages and structures for building symptom catalogues.

5. **System health measurement**: defining and testing system health metrics, and studying the possible activation of protection, adaptation and optimization services.

6. **Trace-directed modelling**: relating the events to a model of the traced system in order to provide higher-level answers such as decomposing the time taken for a request or estimating the benefits of adding resources (e.g. disk or memory).

4.1.2 Feasibility studies

Literature reviews relative to each R&D thread (and associated technologies) should first be completed. A limited number of technological options should then be selected and studied more in depth in order to identify the critical information that will aid the definition of an R&D project. Costs, risks and probable duration of the R&D efforts should be addressed (among other topics).

4.1.3 R&D project to be proposed for funding

The solutions identified in the feasibility studies should be the subject of further, more in-depth R&D efforts in order to provide appropriate solutions to address described problems and needs. These efforts should take the form of a 3-year R&D project having six main work breakdown elements (WBEs), one for each of the six R&D threads.

Collaboration opportunities between governmental, academic and industrial organizations should first be defined. The aim is to form a robust and integrated team of professors, students (MSc and PhD) and other key experts having complementary areas of expertise. Based on all this theoretical and technical information, the team should then complete the project proposal and submit it for funding.

References

4.2 Referenced papers and thesis

Bligh, M., M. Desnoyers, and R. Schultz, 2007. Linux kernel debugging on Google-sized clusters. In Proceedings of the 2007 Linux Symposium, (Ottawa, Ontario, Canada). pp. 29-40.

Dagenais, M. R., K. Yaghmour, C. Levert, M. Pourzandi, M., 2005. Software Performance Analysis. Computer Science.

Desnoyers M. and M. Dagenais, 2006a. Low disturbance embedded system tracing with Linux Trace Toolkit next generation. In Proceedings of the 2006 Consumer Electronics Linux Forum, (San Jose, California, USA).

Desnoyers M. and M. Dagenais, 2006b. The LTTng tracer: A low impact performance and behavior monitor for GNU/Linux. In Proceedings of the 2006 Linux Symposium, (Ottawa, Ontario, Canada), pp. 209-224.

Isci, C., A. Buyuktosunoglu, C. Cher, P. Bose, and M. Martonosi, 2006. An Analysis of Efficient Multi-Core Global Power Management Policies: Maximizing Performance for a Given Power Budget. In Proceedings of the 39th Annual IEEE/ACM International Symposium on Micro architecture, pp. 347-358.

Yaghmour, K. (2001). Analyse de performance et caractérisation de comportement à l'aide d'enregistrement d'événements noyau. Master thesis. École Polytechnique de Montréal, number of pages : 105.

Yaghmour, K. and M. R. Dagenais, 2000. Measuring and characterizing system behavior using kernel-level event logging. In Proceedings of the USENIX Annual 2000 Technical Conference, (San Diego, California, USA), pp. 13-26.

Zanussi, T., K. Yaghmour, R. Wisniewski, R. Moore, and M. Dagenais, 2003. Relayfs: An efficient unified approach for transmitting data from kernel to user space. In Proceedings of the Ottawa Linux Symposium, (Ottawa, Canada), pp. 519-531.

4.3 Referenced Web sites

ATOM, 2008. http://www.tru64unix.compaq.com/atom/

C++loggin, 2008. http://logging.apache.org/log4cxx

EEL, 2008. http://www.cs.wisc.edu/~larus/eel.html

Evlog, 2008. http://evlog.sourceforge.net/

DTI, 2008. http://ols.108.redhat.com/2007/Reprints/wilder-Reprint.pdf

DTIWS, 2008. http://sourceforge.net/projects/dti

DTrace, 2008. http://en.wikipedia.org/wiki/Dtrace, http://www.sun.com/bigadmin/content/dtrace/

Foldoc, 2008: http://foldoc.org/

Frysk, 2008. http://sourceware.org/frysk/

GDB, 2008. http://en.wikipedia.org/wiki/Gdb, http://www.gnu.org/software/gdb/

GNU, 2008. http://en.wikipedia.org/wiki/GNU_Compiler_Collection

IntelVTune, 2008. http://www.intel.com/cd/software/products/asmo-na/eng/239144.htm.

Javacc, 2008. http://en.wikipedia.org/wiki/Javacc

JavaHandler, 2008. http://java.sun.com/j2se/1.4.2/docs/api/java/util/logging/Handler.html

JavaLoggin, 2008. http://en.wikipedia.org/wiki/Java_Logging_Frameworks

JIE, 2008. http://people.csail.mit.edu/tromer/jie/

LKST, 2008. http://lkst.sourceforge.net/

LTT, 2008. http://www.opersys.com/LTT/

LTTng, 2008. http://ltt.polymtl.ca

Mon/Trac Event, 2008. http://ltt.polymtl.ca/tracingwiki/index.php/TracingSummit2008

NetLogin, 2008. http://en.wikipedia.org/wiki/Log4net

OProfile, 2008. http://oprofile.sourceforge.net/

Qemu, 2008. http://en.wikipedia.org/wiki/Qemu, http://fabrice.bellard.free.fr/qemu/

QPT, 2008. http://www.cs.wisc.edu/~larus/qpt.html

QNX, 2008. http://www.qnx.com/

Redhat, 2008. http://www.redhat.com/docs/manuals/linux/RHL-6.2-Manual/getting-started-guide/ch-glossary.html

Relay, 2008. http://relayfs.sourceforge.net/

Snort, 2008. http://www.snort.org/

SystemTap, 2008. https://ltt.polymtl.ca/tracingwiki/index.php/SystemTap, http://sourceware.org/systemtap/, http://en.wikipedia.org/wiki/SystemTap

SunWS, 2008. http://java.sun.com/j2se/1.4.2/docs/guide/util/logging/overview.html

TXL, 2008. http://en.wikipedia.org/wiki/TXL_%28programming_language%29

Valgrind, 2008. http://www.valgrind.org/

VMWare, 2008. http://en.wikipedia.org/wiki/Vmware

WikipediaWS, 2008. http://en.wikipedia.org

WindRiver, 2008. http://www.windriver.com/

ZealCore, 2008. http://www.zealcore.com/

This page intentionally left blank.

Annex A The "Monitoring Distributed Multi-core IT Systems for Optimization and Security" workshop

The text in this annex (*italic*) was captured from the official Web site of the January '08 tutorial/workshop (Mon/Trac Event, 2008).

A.1 Scientific steering committee

***Michel Dagenais** has been active in the domain of system analysis tools for the past 15 years. The Linux Trace Toolkit, developed under his supervision at Ecole Polytechnique, is used throughout the world and gained the cooperation of a large number of industrial contributors over the years from Autodesk Media and Entertainment, Ericsson, Google, IBM, Monte Vista, Sony and others.*

- *Phone: +1 514 340 4711 x4029*
- *E-mail: michel.dagenais@polymtl.ca*

***Dominique Toupin** has been working at Ericsson on software engineering improvements with the open source community, researchers and commercial companies. He developed systems to manage wireless networks.*

- *Phone: +1 514 345 6123*
- *E-mail: dominique.toupin@ericson.com*

***Robert Charpentier**, after working at CAE Electronics on flight simulators, joined Canada Defense Research at Valcartier where he specialized in infrared imagery and space-based surveillance. His current research interests are in secure interoperability and software robustness.*

- *Phone: +1 418 844 4000 x4371*
- *E-mail: Robert.Charpentier@drdc-rddc.gc.ca*

A.2 List of attendees

Table 1. List of attendees (January '08 tutorial/workshop).

Name	**e-mail**
Onofrio Amendola	onofrio.amendola@ericsson.com
Stephan Bill	stephan.bill@ericsson.com

Pierre Boucher	pierre.boucher@ericsson.com
Jacques Bouthillier	jacques.bouthillier@ericsson.com
Felix Burton	Felix.Burton@windriver.com
Andrew Cagney	cagney@redhat.com
Anders Caspár	anders.caspar@ericsson.com
Eugene Chan	ewchan@ca.ibm.com
Robert Charpentier	Robert.Charpentier@drdc-rddc.gc.ca
Nawel Chefai	Nawel.Chefai@drdc-rddc.gc.ca
François Chouinard	francois.chouinard@ericsson.com
Mario Couture	Mario.Couture@drdc-rddc.gc.ca
Michel Dagenais	michel.dagenais@polymtl.ca
Roch Decoste	ROCH.DECOSTE@RCMP-GRC.gc.ca
Paola Delsante	paola.delsante@ericsson.com
Mathieu Desnoyers	mathieu.desnoyers@polymtl.ca
Peter Dibble	peter.dibble@timesys.com
Samir Douik	samir.douik@ericsson.com
Frank Ch. Eigler	fche@redhat.com
Tamás Eiler	Tamas.Eiler@ericsson.com
Pierre-Marc Fournier	pierre-marc.fournier@polymtl.ca

Stuart Fullmer	sfullmer@mvista.com
Steve Furr	Steve.Furr@freescale.com
Mario Godin	GodinM@smtp.gc.ca
Niclas Gustafsson	niclas.gustafsson@ericsson.com
Joel Huselius	joel.xj.huselius@ericsson.com
Marc Khouzam	marc.khouzam@ericsson.com
Béchir Ktari	Bechir.Ktari@ift.ulaval.ca
Paul Lamoureux	Lamoureux.JRP@forces.gc.ca
Alf Larsson	alf.larsson@ericsson.com
Sven Lundblad	Sven.Lundblad@enea.se
Marco Masse	marco.masse@ericsson.com
Gabriel Matni	gabriel.matni@polymtl.ca
Andrew McDermott	andrew.mcdermott@windriver.com
Bo Nilsson	bo.nilsson@ericsson.com
Benjamin Poirier	benjamin.poirier@polymtl.ca
Robert Roy	robert.roy@polymtl.ca
Alvaro Sanchez-Leon	alvaro.sanchez-leon@ericsson.com
Mats Sandberg	mats.sandberg@ericsson.com
Steven Shaw	steveshaw@ca.ibm.com

Nathan Sidwell	nathan@codesourcery.com
Peter Smith	psmith@redback.com
Jonas Strömgren	jonas.stromgren@ericsson.com
Sofiene Tahar	tahar@encs.concordia.ca
Eero Tamminen	eero.tamminen@nokia.com
David Taylor	dtaylor@shoshin.uwaterloo.ca
Henrik Thane	henrik.thane@zealcore.com
Beth Tibbitts	tibbitts@us.ibm.com
Maria Toeroe	maria.toeroe@ericsson.com
Dominique Toupin	dominique.toupin@ericsson.com
Guy Turcotte	Guy.Turcotte@drdc-rddc.gc.ca
Carlo Vitucci	carlo.vitucci@ericsson.com
Robert Wisniewski	bob@watson.ibm.com
Elena Zannoni	elena.zannoni@oracle.com

A.3 List of presentations – Day one

9h00-10h20 Michel Dagenais, Gabriel Matni, Pierre-Marc Fournier, Robert Roy. Survey of the best available tools and techniques for system performance tracing and analysis (part1, part2, part3, part4)

The talk will describe the different types of tools and techniques used to extract, analyze and visualize system execution and performance data. The survey will cover source-level and binary instrumentation (e.g. Dtrace, SystemTap), as well as sampling (e.g. gprof, OProfile) for extracting data. Different visualization and analysis tools and frameworks will be described, including tools from the Eclipse framework, QNX and Wind River. As will be described by Robert

Roy, for large parallel computing applications, the ultimate challenge of parallel performance analysis tools is to help in finding bottlenecks and synchronization problems and further optimizing the code. He will focus on the use of performance analysis tools for developing parallel solvers (inputs, ease of use, scalability).

10h30 – 10h50 Frank C. Eigler, RedHat. <u>SystemTap</u>

Description of the SystemTap free software infrastructure to simplify the gathering of information about running Linux systems. SystemTap eliminates the need for the developer to go through the tedious and disruptive instrument, recompile, install, and reboot sequence that may be otherwise required to collect data.

11h00-11h20 Andrew Cagney, RedHat. <u>Frysk; and tracing from userland</u>

The Frysk group is achieving a framework of tools for, tracing, monitoring and debugging large distributed applications. After a brief overview of Frysk and its architecture; this presentation will examine some of the challenges with us taking a largely user-land approach. Ptrace and utrace, and DWARF will each be discussed.

11h30 – 12h20 Robert Wisniewski, IBM. <u>Performance analysis and debugging at IBM</u>

Discussion of the performance analysis and debugging he has done at IBM. He will describe the challenges presented by both external and internal IBM clients, typically in the domain of HPC and multi-core servers. He will then describe the tools he has worked on to address those challenges.

13h30-14h20 Eugene Chan. IBM. <u>Eclipse Test & Performance Tools Platform Extending Eclipse Test and Performance Tools Platform (TPTP)</u>

Overview of the TPTP architecture following by a quick look at the profiling features of TPTP. Finally, we will focus on UI extensions that are available to support non-Java context.

14h30- 15h20 Steven Shaw, IBM. <u>Rational IBM Rational Systems Developer Extensibility with consideration of UML Debug and Trace integrations</u>

This presentation will provide a brief overview of the Rational Systems Developer tooling and then drive into the different extensibility features that it offers. The architecture of the product with respect to the open source components that it leverages will be explored as well as the relevant extensibility of those components. For example, we will briefly look at the extensibility mechanisms for Eclipse platform, EMF (Meta-model framework), UML2 (concrete meta-model built using EMF), GEF (Graphical Editing Framework), GMF (Graphical / Generative Modeling Framework), RSD (Rational Systems Developer). Finally we will create / examine some example plug-ins / plug-lets that use the extensibility with consideration of typical integrations that would be important for debugging a UML system.

15h30-15h50 Elena Zannoni, Oracle. <u>Tracing at Oracle</u>

Elena will talk about the experience of the Oracle DataBase team with tracing and the problems they face.

16h00-16h20 Mathieu Desnoyers, Ecole Polytechnique. Tracing the Google platform

Mathieu will describe his experience in tracing while working for the platform team at Google. He will describe a number of typical hard to diagnose problems encountered on extremely large clusters and how they were solved using tracing tools. The specific challenges and expectations for very large high performance online server clusters will be exposed.

16h30-16h50 Steve Furr, Freescale. Multi-core Enablement Challenges for Semiconductor Manufacturers

This presentation will highlight some of the challenges that face vendors as multi-core processors become more prevalent. Multi-core solutions increase the complexity of static and real-time debugging tools. The difficulties, motivations and aims behind enabling multiple vendors of dynamic debug and profiling tools are examined from the perspective of a semiconductor manufacturer's perspective.

17h00-17h20 Stuart Fullmer, MontaVista. Tracetools with MontaVista Linux

Short presentation regarding LTTng and TimeDoctor. The presentation will give a quick overview of the tools and our level of integration of the tools and where MontaVista sees itself with future offerings.

19h00 – 20h00 BOF (Birds Of a Feather) sessions

- *System Level Target Data Collection*
- *Eclipse Tracing Monitoring Framework*

A.4 Eclipse Tracing Monitoring Framework BOF – Day one

A.4.1 Participants

Approximately 25 people attended the Eclipse BOF, including Eugene Chan from IBM (Eclipse TPTP Test and Performance Tools Platform), Beth Tibbitts from IBM (Eclipse PTP Parallel Tools Platform), Dominique Toupin, François Chouinard, Marc Khouzam from Ericsson.

A.4.2 Discussions (Chairman's personal notes)

Eclipse Test & Performance Tools Platform:

Questions were asked about support for non Java languages in TPTP. Eugene Chan answers that the language model was extended to support other languages such as C++. They do not have the resources to implement the C++ plug-ins now and hope that outside contributions will fill the void. With C++ plug-ins, all the nice TPTP views will become available to C++ programs. Discussions were held to determine within which framework tracing and monitoring tools should *be integrated, base Eclipse (RCP), Eclipse IDE or Eclipse TPTP. The dependency of TPTP on Java and JDT is costly. The IDE may be the right place since developers often need both tools together (IDE, debugger and tracing views). This is typically not the case for system administrators who use only tracing/monitoring. Beth Tibbits mentions that PTP is based on the IDE and not on TPTP. Other tools, such as TAU or IBM HPC tools are simply called from Eclipse and not tightly integrated.*

Storage:

The following topic was how to interface tracing views to different data providers, such as WindRiver SystemView to VxWorks and LTTng traces. The native traces are converted to an event database (using SQLite). Additional synthetic events (e.g. states) are added to the database. The database thus consists in contexts (process, CPU, state), event descriptors and events. The various views then interact directly with the database.

Event:

The OASIS Common Base Event (CBE) could be a way to interface views to event sources but it is too heavy. Creating a unifying event structure for Java, C, C++, system level events, etc. is a challenge.

Visualisation:

Visual representation is essential to identifying problems. One view with events at all levels (application, network, system, etc.). How to filter the trace data. Beside LTTV (not Eclipse based) no other visualization tool seem to handle big traces (seval Gb), tools are either crashing, not loading the trace or cannot be used because not responsive.

A.5 System Level Target Data Collection BOF – Day one

A.5.1 Participants

- Robert Wisniewski, IBM Research (Blue Gene project)
- Paola Delsante, Ericsson
- Elena Zannoni, Oracle (Linux Team)
- Carlo Vitucci, Ericsson
- Mario Couture, Defence R&D Canada
- Peter Dibble, Timesys
- Pierre-Marc Fournier, Ecole Polytechnique de Montreal, LTTng/LTTV
- Mathieu Desnoyers, Ecole Polytechnique de Montreal, LTTng/LTTV maintainer
- Felix Burton, Windriver

A.5.2 Discussions (Chairman's personal notes)

Accurate time base on different cores:

The problem is timestamp counters are not always synchronized between cores.

Mathieu: On a system with shared memory, we would like to have almost cycle-accurate timestamping.

Peter: On powerpc, there is a time register that gets its value from the bus. It is a synchronized time source available to all chips.

Bob: As part of a tracing system, we need an infrastructure to synchronize between cores, chips, and distinct systems.

Bob: The problem is complex if cores are changing their speeds without our knowing.

Bob: Experiments have showed that on the long run, there is a drift of as much as 0.1% between TSCs.

Bob and Mathieu: idea of adding synchronization points regularly to the trace, or the possibility to add them when their is a scaling event

Mathieu: Intel is working on fixing the time counter, but that counter will have a frequency that is slower than current TSCs.

Carlo: Post-mortem debugging is important for Ericsson

Mathieu: It's implemented in LTTng, although it has a small performance impact when enabled.

Carlo: On an Ericsson real-time OS, they have a very good post-mortem analysis tool. For Ericsson, it is imperative, if they want to use Linux, to have post-mortem debugging. If nodes crash, they need to know what happened.

On a crashed system, there are the last events that occurred before the trace, as well as extra information that is not usually available on a live system, like the stacks and registers of all processes and the state of all kernel structures.

Carlo: wants a way for a process to dump some data only when the system crashes

Mathieu and Bob: This is what the flight recorder is used for.

Pierre-Marc: It is not practical to dump regularly a huge number of variables or a large quantity of data in the flight recorder.

Bob: You have that data in the crash dump, but to extract it, you need to find out what processes were running and where, and it's not simple. What we are talking about is a smart dumper that can dump specific variables of specific processes.

Carlo: for him, it is very important to have a hook to tell the kernel or some other mechanism what to print when the system crashes

Bob and Felix: this is an optimization of the crash dump extraction problem and the people present here are not experts on this.

Conclusion: from a tracing point of view, the way to save data for crash dump extraction is to use flight recorder and to fire events that saves this data regularly.

Impact of virtualization on tracing:

Problem of the synchronization of TSCs on virtual machines vs physical machines.

Bob: Recommends that there be one synchronized timestamp on a single chip (all cores, all virtual machines).

Felix: possibility of having a chip-wide read-only register that increments by one each time you read it, for having a monotonic time source.

Discussion about the effect of heat on clock drift, and the fact that temperature can change very quickly. Sub-second temperature peaks are a reality. Clock drift due to temperature is a fact.

Should we or should we not virtualize the TSC? Is it more important to hide to the virtualized os the fact that it is virtualized, or to permit tracing by using an absolute time source? Having a synchronized TSC across virtualization boundaries definitely makes tracing easier.

Bob: There are a few cases where a virtualized TSC is needed. One example is that to keep track accurately of the usage of resources on the system through time, you need a virtualized time source.

Standardization of trace format:

Mathieu and Bob: Need to have a standard trace format, minimalistic, need to be able to skip the event even if you don't know what it is when reading the trace.

There is a discussion about the need to have a major/minor part where the major indicates the category of the event. This is useful for quick filtering by category and was used by Bob in a previous project. As Mathieu observes, however, this is not necessary with the markers, since they can be enabled on an individual basis. Bob agrees.

A.6 List of presentations – Day two

9h00-9h50 Sven Lundblad, ENEA. <u>Advanced Event - Action System and Flexible Profiling</u>

Enea's advanced Event Action system. Use cases (system tracing and debugging), functionality (event types, filtering, different actions, event action states), intrusiveness, and controlling tools. Possibilities and challenges in multi core systems. The Event Action implementation in OSE and possible Linux support. Optima Profiling capabilities Advanced custom profiling, memory profiling and CPU usage profiling (in single core and SMP systems).

10h00 – 10h20 Henrik Thane, ZealCore. <u>Monitoring, Information Fusion, Reverse Engineering and Replay Debugging</u>

Discussion on different trace recording approaches including handling of multi-core systems, and most importantly what do you do with the recorded information? How can you fuse different trace logs, do reverse engineering and raise the level of Debugging? We will especially look into the problem on how to achieve forward and backward replay debugging on the UML model level (reanimated state-machines and sequence diagrams) based on reverse engineering of minimal target run-time recordings.

10h30-11h20 Beth Tibbitts, IBM. Research Eclipse Parallel Tools Platform & Performance Tools Framework

The Eclipse Parallel Tools Platform (http://eclipse.org/ptp) aims to produce an open-source industry-strength platform that provides a highly integrated environment specifically designed for parallel application development. This includes a standard, portable parallel IDE that supports a wide range of parallel architectures and runtime systems, a scalable parallel debugger, support for the integration of a wide range of parallel tools, an environment that simplifies the end-user interaction with parallel systems. A performance tools framework is being designed to ease the integration of performance tools into Eclipse and PTP. The University of Oregon has developed Eclipse plug-ins for PTP for its TAU (Tuning and Analysis Utilities) and we are working to generalize and make these features and others available to ease integration of other tools, including performance tools with PTP and EclipseParallel computing tracing with Eclipse Parallel Tools Platform Performance Analysis Framework / Tuning and Analysis Utilities

12h30-13h20 Andrew McDermott, WindRiver. Developing OS-agnostic visualization tools

Today it is desirable to use a common [visualization] tool for diagnosing system behaviour. However, it is more common to find different tools, different user interfaces, different data formats and different levels of overall tool capability on a [operating] system-by-system basis. Additionally, the mechanics of querying the data in an ad-hoc manner outside the boundaries of the tool are typically programming language specific or non-existent. Wind River's System Viewer, previously known as WindView, is a sophisticated visualization tool that enables developers to view the dynamic operation of their system. System Viewer has evolved to address the deficiencies listed above to be an OS-agnostic visualization tool (e.g., VxWorks and LTT), providing a separate and SQL-query able data engine that can accommodate new systems.

13h30-14h20 Felix Burton, WindRiver. Target Communication Framework

Today almost every device software development tool on the market has its own method of communicating with the target system. Communication methods often conflict with each other, requiring individual setup, configuration, maintenance, and imposes different limitations for different tools. TCF is designed to establish a common communication mechanism between development tools and embedded devices for the purposes of debugging, performance monitoring, file system access, etc.

Wind River Sensorpoint Technology

Increasingly complex systems are hard to debug using traditional techniques such as breakpoints and stepping or printf. Sensorpoints allow dynamic instrumentation of software in the lab as well as in the field with minimal impact on the system performance and system resources. This makes *it ideal for diagnosing complex problems as well as creating patches to fix such issues in deployed devices. Sensorpoints are code fragments that can be dynamically "injected" into a binary image, without requiring a restart of the application or the device.*

14h45-16h15 Michel Dagenais, Robert Charpentier, Dominique Toupin Panel Concluding Panel (see annex A.5)

Michel Dagenais, Robert Charpentier and Dominique Toupin will animate a discussion to identify the most important current and upcoming challenges for system analysis shared by the participants.

A.7 Concluding panel (attendee's personal notes) – Day two

System level tracing has long been an essential element in the toolbox of real-time embedded systems developers. Widely used embedded operating systems such as WindRiver VxWorks, QNX and MontaVista Linux contain many static trace points to monitor all the key events. Flexible dynamic trace points have been added recently (e.g. WindRiver SensorPoints) to complement static trace points. High performance computer clusters also have a long history of using tracing tools, most often user level instrumentation of parallel processing libraries such as MPI.

General purpose computer systems were in many cases left with CPU profiling tools and simple logs (e.g. syslog), unsuitable for fine grained system performance analysis. However, many of these general purpose systems, for example telecommunication and online servers as well as multimedia workstations and gadgets, have become complex soft real-time performance critical systems. A low overhead system level tracing thus becomes an important tool to tune these systems. While DTrace has been very well received by Solaris users, Kernel Markers, SystemTap and LTTng should offer an excellent tool set for Linux systems.

During the final panel, the requirements which came out repeatedly were outlined to insure that all the important points had been covered.

- *Low-overhead instrumentation is essential to monitor real-time production systems.*
- *Static probes (e.g. Linux Kernel Markers) must be placed at key points in the system, dynamic probes (e.g. SystemTap) should be insertable pretty much anywhere. Dynamic probes have a lower cost (zero) when not inserted but a higher cost when activated, in comparison to static probes.*
- *Probes should be activated on demand through a nice GUI interface or by scripting, with an associated action (log event, user defined hook, increment statistic counter).*
- *Trace synchronization is required to view and analyze multi-computer, multi-core traces. It can be achieved either at collection time (common time source) or at analysis time (align time bases using matching events like send/receive packet).*
- *Huge traces (several GB, larger than available RAM) are needed in some cases and must be handled by the tool chain (viewers).*

- *The same framework and views should serve for events coming from several sources. There should be a common protocol or format for mapping equivalent events. Examples include the OASIS Common Base Event.*
- *A Target Communication Framework, preferably with open-source reference implementation, is needed to interface multiple heterogeneous sources of tracing data. Tracing data may come from different systems or from different versions (and log formats) of the same systems.*
- *Tracing events will come from all layers, silicon, hypervisor, operating system, virtual machine, libraries, applications. They may map to resources, source code lines or even modeling level (e.g. UML).*
- *Traces will come from distributed nodes in the system. Views must help to analyze simultaneously the union of these traces.*
- *Filter events at collection time (activating only the relevant probes, checking for the context such as process or user ID) or at analysis time.*
- *Automate at least partly the trace analysis (comparison to reference traces, catalog of good patterns and of faulty behavior patterns). Automate fault identification and triggering. Have a standard way to express patterns.*
- *Tracing and monitoring is equally useful in the lab and in the field.*
- *Dynamic probes may serve for corrective measures (code patching).*
- *There may be access control and privacy issues for activating the probes and viewing the resulting trace.*
- *Multi-CPU, multi-core systems are here, insure that they are well supported with the needed scalability and tools to help understanding their interactions.*
- *A wide range of different trace analysis and views may be relevant. Thus, a plug-in system is required to allow the easy addition of custom modules.*
- *The different activation, viewing and analysis tools should be available through a common framework, presumably Eclipse already used by most participants.*

A.8 List of presentations – Day three

9h00-9h50 Robert Charpentier, Defence R&D Canada. Monitoring Critical Infrastructures for Security and Optimization.

This talk will be divided in 2 parts. The first one will be a general introduction to the methodology needed to develop collaborative R&D efforts, with a focus on the way to elucidate mid-term technical requirements. This will be followed by a review of the main conclusions of the tutorial (29-30 Jan) and their relevance for governmental organizations, including the Canadian Forces.

10h00-10h50 Alf Larsson, Ericsson. Cello Packet Platform (CPP); CPP from a trace & debug perspective, a real time and distributed challenge.

Description of Ericsson 3G concept as a platform and the scenarios facing developers and users of the system. The focus will be on the current status of trace & debug and a wanted position as well as the future functionality in this area.

11h00-11h20 Carlo Vitucci, Ericsson. <u>AXE-RP; AXE RP Debugging and Tracing.</u>

Brief description of the AXE RP environment and what types of debugging is present, by means of both AXE RP platform designers and platform user. Eventually, the multi-core challenge is shown, outlining what is the impacts for debugging matter.

11h30-11h50 Onofrio Amendola, Ericsson. <u>AXE-IO; Tracing, debugging and monitoring in AXE-IO.</u>

Description of tracing, debugging and monitoring tools currently used in the AXE-IO area.

13h00-13h20 Bo Nilsson, Ericsson. <u>Mobile Platform; Main challenges faced by the Mobile Platform.</u>

Short description of the platform and main difficulties encountered with the currently available tools and techniques for the performance analysis and system understanding and diagnosis.

13h30-13h50 Tamás Eiler, Marc Khouzam, Ericsson. <u>TSP; Tracing facilities and challenges in TSP.</u>

Short description of TSP, of existing tracing facilities and other third parties trace programs / debuggers / profilers used in TSP. Current challenges (64bit development, SMP), some words about debugging facilities available in TSP and our plans. Overview of the user-experiences and their challenges.

14h00-14h20 Simon Kågström, Francois Chouinard, Ericsson. <u>IS; Development environment and tracing at IS/MXB.</u>

The presentation will be an overview of tools used at IS/MXB and some experiences we've had with tracing. IS/MXB uses WindRiver workbench and its set of support tools, as well as some third-party tools.

14h30-14h50 Peter Smith, Ericsson (Redback). <u>Current methodologies, tracing/monitoring needs and challenges.</u>

Redback Networks was acquired by Ericsson approximately one year ago, and some joint development projects are underway. From a tools perspective, there has been very little integration with Ericsson, and we are looking to adopt some of the best practices that are already available within the company. This presentation will summarize our development environment, our current debug/tracing tools, and will list the tools that we'd like to adopt in the near future.

15h00-16h30 Michel Dagenais, Robert Charpentier, Dominique Toupin. <u>Panel</u>.

Michel Dagenais, Robert Charpentier and Dominique Toupin will animate a discussion to identify the most important current and upcoming challenges for system analysis of interest to Ericsson and Canada Defence R&D.

A.9 Some pictures of the event

Figure 4. From left to right: D. Toupin, M. Dagenais and R. Charpentier.

Figure 5. Attendees of the workshop.

Annex B Overview of selected tools

This annex provides an overview of selected tools in the domains of monitoring and tracing. The text in this annex (*italic*) was captured from the official Web site of the January '08 tutorial/workshop (Mon/Trac Event, 2008).

B.1 Data providers

B.1.1 DTrace

DTrace (Dtrace, 2008) is a dynamic instrumentation framework built for and currently integrated into the Solaris 10 and Open Solaris operating systems. Apple has already integrated DTrace in Mac OS X 10.5. The main components of the system are the DTrace framework and the instrumentation providers. The main idea behind separating the two sub-components resides in the fact that different providers can have different instrumentation techniques. Instrumentation providers are loadable kernel modules that define a set of probes they can activate on demand. These probes are advertised to consumers and can be identified by the following 4-tuple: <provider name, module, function, probe name>.

Users can specify in a script file the 4 elements to activate a specific probe, or any subset to activate all probes that match with the specified subset. When a probe fires, the corresponding provider transfers control to the DTrace framework which disables interrupts on the current CPU, and executes the corresponding actions previously specified in the script file. The DTrace framework handles the case where multiple consumers are assigned to the same probe. When done, interrupts are enabled, and control is transferred back to the provider.

The DTrace framework allocates a kernel per-CPU buffer for every consumer. Data is read out of the kernel by having two per-CPU buffers: an active and an inactive buffer. When the consumer issues a call to read a buffer on a particular CPU, the active and inactive buffers are switched after disabling the interrupts on the CPU. Then, interrupts are re-enabled, and the inactive buffer is copied out to the consumer. When a consumer buffer lacks sufficient space for data recording, a drop count is incremented and data is lost for this consumer.

DTrace providers use different instrumentation methodologies. For example, the Function Boundary Tracing provider is purely dynamic, and uses instrumentation techniques that are highly dependent on the CPU instruction set architecture. On SPARC, it uses the unconditional annulled branch-always instruction. On x86, FBT uses a trap-based mechanism to transfer control to the IDT handler which in turn transfers control to DTrace. This provider offers more than 25,000 probes for the entry and exit from almost all kernel functions.

The Statically defined tracing provider has a different mechanism. The code is statically instrumented at first through a C macro that expands to a call to a non-existent function having a well-defined prefix (__dtrace_probe_). When the kernel linker detects this, it automatically replaces this call with a no-operation, and saves the tuple <function name, address of the call site>. When the corresponding probe is enabled, the provider replaces dynamically the no-

operation by a call to transfer control into the DTrace framework. When the probe is disabled, the added no-op instruction has a negligible effect. Other providers can be used to gather information regarding disk input and output, lock contention, CPU scheduling and process creation and termination.

The D language is derived from a large subset of C, and allows access to the kernel's native types and global variables. Users can specify inside the D script file the providers names, the probes names, and the actions to take whenever each probe is hit. This file can then be compiled by the D compiler implemented in the DTrace library and invoked by the DTrace command. Upon execution, the DTrace framework enables the probes that were specified in the file by making appropriate calls to the probe's providers.

User space tracing is also possible using DTrace. The first step is to write a data provider in the D programming language and specify probes in it. Then in the user's C code, the macro DTRACE_PROBE can be called where tracing is desired, with the two essential parameters: the provider name and probe name. Other pertinent arguments can also be passed. The dtrace -G option can be used to link the provider and probe definitions with the application object file.

B.1.2 SystemTap

Systemtap (SystemTap, 2008) is a debugging and kernel troubleshooting tool. It is an open source project with contributors from IBM, Red Hat, Intel, Hitachi, Oracle and others. It uses mainly the kprobes API to dynamically instrument the Linux Kernel. Other data providers can be used with Systemtap such as kernel markers and /proc file system. Attempts are currently being made to provide user space tracing. Systemtap also allows the user to write script files where he can specify probe points and associate handlers to them. Probe points currently supported include function entry, exit and down to almost any machine instruction. Probing can also be enabled asynchronously, at regular time interval [1].

Here are some probing points declarations that can be specified in a script file:

- *Probing a kernel function entry:*
  - `kernel.function("sys_read"){ ... }`
- *Probing a kernel function exit:*
  - `kernel.function("sys_read").return{ ... }`
- *Probing all function entries in a module:*
  - `module("ext3").function("*@fs/ext3/inode.c"){ ... }`
- *Asynchronously probing using a timer:*
  - `probe timer.ms(10000){ ... }`
- *Attaching to Linux Kernel Markers:*
  - `probe kernel.mark(“context_switch”) {residency[$arg2,cpu()]++}` [2]
- *Probing a user space program (available in near future):*

- *probe program("/lib/glibc.so").function("malloc") {if($size > 1024*1024) log("hog!")}*

The default contextual variables at a probe point that can be retrieved include: cpu number, egid, euid, execname, gid, pexecname, pid, ppid, tid, uid, stack_size, target pid. One limitation here is that these functions may not return correct values when the probe is hit in an interrupt context. It is possible however to access (read or even modify) variables from within the context of the traced program. These target variables can be referenced on demand in the handlers of the script file and their names should be prefixed with a $ sign. They will be resolved to memory addresses during the script elaboration phase. It is worth noting here that such information may not always be available whenever optimization options are used during kernel build.

The probe handler scripts initially pass through the elaboration phase where kernel references including function parameters, local and global variable, etc. get resolved to run-time real addresses. This is achieved by looking into the DWARF debugging information generated by the compiler during the kernel build. The next phase consists in translating the processed script into C code and compiling it as a kernel module. The module first inserts the probes then waits for a probe to be hit [3] (…). Since the generated code will run in kernel mode and for safety issues, it is made open for inspection.

B.1.3 Valgrind

Valgrind (Valgrin, 2008) is a framework for creating dynamic analysis tools. It was released in late February 2002. It consists of a main core and a suite of tools for memory debugging, memory leak detection and cache profiling. It can be used on programs running on the Linux operating system for the x86, amd64 and ppc32 architectures.

Valgrind currently comes with six different tools. Cachegrind is a cache simulator which computes the number of instructions executed per line of code in a program and determines the number of cache misses. Callgrind adds some functionalities on top of Cachegrind like annotating threads separately. Helgrind is used to check for potential race conditions. Lackey is a sample tool that helps create other tools. Massif is a heap profiler that determines how much heap memory is allocated by a program. Memcheck is a memory checker tool that is able to detect any access to an uninitialized or freed memory. It can also find memory leaks where pointers to allocated memory areas are lost, or where the amount of freed memory doesn't match with the allocated one.

The Valgrind core disassembles the client code into an intermediate representation (IR), passes the output to one of the tools for instrumentation, and finally converts the IR code back into machine code. The amount of added instrumentation depends greatly on the tool. For example, Memcheck multiplies the program size by a factor of 12, and slows down the execution by a factor 25 to 50. It is worth mentioning that Valgrind profiles not only the program code but also all dynamically-linked libraries it uses. The core is incapable of tracing into the kernel; in order for the tools to know which registers and memory addresses were accessed, they can register a callback function to be called each time a system call occurs. If the program is compiled with the debugging information (-g option), Valgrind will be able to pinpoint relevant source code lines. The output of Valgrind can be redirected to three different places. A file descriptor, (the default is stderr), a logfile or a network socket.

The Memcheck tool is Valgrind's most powerful tool. It uses shadow values to store information about every register and every byte of memory used by the program. One shadow byte per byte of live original memory is needed. These 8 shadow bits indicate if the 8 corresponding bits in the real live byte are defined. An additional shadow bit per memory byte is also saved; It indicates if the represented byte is addressable or not. Any operation that accesses memory should be instrumented in a way that keeps the shadow memory state up-to-date. Memcheck can detect any use of uninitialized memory, any read or write to a memory that was already freed, and memory leaks.

Programs executed under Valgrind usually run 25 to 50 times slower, and consume more memory than the regular program execution. Valgrind can also produce false positives if the code is optimized. Valgrind doesn't support 3DNow! instructions and atomic instruction sequences are not preserved.

- *Source-level, auto-generated (gcc: instrumentation, profiling, test)*
- *Source-level, transformation (analysis and source transformation: javacc, TXL)*
- *Source-level, manual injection (Logging API, Java, log4cpp, printk, evlog, driver tracing infrastructure, kernel markers)*
- *Static binary instrumentation (ATOM, EEL)*
- *Dynamic instrumentation (DTrace, SystemTap, DynInst, GDB)*
- *Sampling (GProf, OProfile)*
- *Simulator (Valgrind, QEMU)*

B.2 Frameworks

B.2.1 Eclipse Framework

Eclipse is a Java-based, plug-in extensible, open source development platform. It started as an IBM Canada project in November 2001. In January 2004, the Eclipse Foundation was created as an independent not-for-profit, member supported corporation. The founding Strategic Developers and Strategic Consumers were Ericsson, HP, IBM, Intel, MontaVista Software, QNX, SAP and Serena Software. More background information is available at: http://www.eclipse.org/org/.

In its simplest form, Eclipse is a framework providing a set of services for building plug-in components and therefore, all plug-ins are created equal. One of the indispensable components is the Workbench. It consists of perspectives, views and editors. A perspective is a collection of one or more views and editors (e.g. Navigator view, Outline view, Tasks view). It also comes with CVS support built-in. Workbench also provides extension points allowing any plug-in to extend the Eclipse user interface by adding new interface functionalities, creating new views, etc... Every plug-in declares some descriptive information in a manifest XML file called plugin.xml, such as the plug-in name, version number, .jar file name, dependencies on other plug-ins, extension points, etc... An extension point may declare additional specialized XML element types for use in the extensions. This allows the plug-in supplying the extension to communicate arbitrary information to the plug-in declaring the corresponding extension point. The manifest *information is used by Eclipse to integrate this plug-in into the framework. The new plug-in can be attached, tested and debugged without having to restart Eclipse. Various plug-ins with different functionalities already exist or are currently being developed. As an example, the plug-in CDT provide support for other programming languages like C/C++. Other plug-ins such as TPTP address software testing, profiling and trace analysis. Due to the flexibility of the robust design of Eclipse, different projects based on Eclipse are currently being developed and are targeting the simultaneous release of the Ganymede version on June 25, 2008. (Ganymede Simultaneous Release Projects;*

http://wiki.eclipse.org/index.php/Ganymede_Simultaneous_Release#Projects). A more detailed list of eclipse based projects is available at: http://en.wikipedia.org/wiki/List_of_Eclipse_projects.

B.2.2 Intel VTune

VTune Performance Analyzer is an application profiler for systems based on Intel processors. It supports all language compilers (C, C++, Fortran) that follow industry standards (ELF, STABS and DWARF). Java and mixed Java are also supported. The Graphical User Interface is based on Eclipse. The VTune analyzer searches for symbol information in the binary file it launches. If it cannot find this information, Vtune disassembles the code and provides static analysis of the basic blocks in the code. VTune can collect information through two methods: sampling and profiling.

Sampling

VTune conducts non-intrusive sampling and doesn't instrument the binary code. Instead, at regular intervals, it interrupts the processor and collects samples of instruction addresses, one per interrupt. This procedure stops when the application ends or when the specified duration is over or when sampling is manually stopped using the ActivityController. VTune stores 32 bytes per sample in a user configurable sampling buffer. When the buffer is full, VTune suspends sampling temporarily to write the data to the disk. By default, the collected sampling information is stored in $HOME/.VTune directory. The sampling collector has different options that can be configured by the user such as the sampling interval, the sampling buffer size, the delay before sampling starts, the conditions to stop sampling (i.e. sampling time, maximum samples to collect, etc.) Event based sampling can be specified to identify low level performance problems such as cache misses and misdirected branches; By default, the Clockticks and Instructions Retired events are pre-selected. Different processors support different events. Vtune can determine the ones that are supported by the used processor (vtl -help -c sampling). Data collected can be filtered and visualized in three different views: process view, module view and hotspot view. Hotspots indicate sections of code within a module that took a long time to execute. The hotspot view can display hotspots of active functions or active source files in the module. The information can be filtered by process id, CPU number and module name.

Profiling

Call Graph profiling is used to determine the program flow and the critical path in the program. It counts how many times a function calls another function and determines the time spent in each of them. The Call graph uses the binary instrumentation technique to record timing and call *sequence information. By default it instruments all Ring 3 modules used by the application slowing down its execution. For Java and .NET applications, call graph uses Java Virtual Machine Profiling Interface (JVMPI) and .NET Profiling API to collect performance data. Vtune stores the collected data in a user configurable Call Graph buffer having a default size of 16384 KB. When the buffer is full, VTune suspends the profiling temporarily to write the data to the disk. There are 3 instrumentation levels for the modules that can be configured by the user:*

- *"**all-functions**" instrumentation which is only possible if the selected module has debug information,*
- *"**exports**" level which is used to instrument functions in the module's export table,*
- *"**minimal**" level which instruments the module but does not collect data. It only reports calls from this "unimportant" module to other "important" modules.*

For user executables, only all-function and minimal instrumentation levels are possible. Again data can be visualized in different views such as the process view, thread view, module view, function view, view by call site and the critical path view. The critical path is the most time consuming call sequence. Some of the VTune Timing Information that can computed are the total time spent in a function, the self time which excludes the time spent in calls to other instrumented function, the time spent in a function and (or without) its children while the thread was suspended, etc.

B.2.3 LTTng

The Linux Trace Toolkit Next Generation Viewer (LTTV) is a graphical application for viewing and analyzing traces recorded with LTTng. It is being developed mainly at Ecole Polytechnique de Montreal and released under the GNU GPL.

General characteristics

LTTV is designed to be very modular. Its powerful plug-in system makes it easy to add features.

LTTV is optimized for handling very large traces and has been tested successfully with 15 GB traces.

LTTV is capable of opening several traces at once and merging their data. This makes it possible to view traces of virtual machines and their host simultaneously in order to study problems that involve the interaction between them. In order for this to work correctly, timestamping in all the traces must be consistent.

Resource view showing a virtual machine (Trace 1) and its host (Trace 0)

LTTV is capable of opening several traces at once and merging their data. This makes it possible to view traces of virtual machines and their host simultaneously in order to study problems that involve the interaction between them. In order for this to work correctly, timestamping in all the traces must be consistent.

Histogram

This is a graph of the density of events through time. It makes it easy to find periods of high system activity. Filters can be applied so the graph applies only to specific event types.

Control flow view

The Control flow view displays the state of every process in the system through time.

B.2.4 QNX Momentix

Company

QNX Software Systems provide different products targeting primarily the embedded systems market. Their proprietary operating system, called QNX Neutrino, is a real-time operating system. As of 12 September 2007, a free license was released for non-commercial use together with the source code of QNX Neutrino. The company also provides a development suite called QNX Momentics which is described briefly later in this document. Other products include middleware for multimedia applications, acoustic processing as well as 2D and 3D graphics rendering.

Kernel Tracing

The instrumented QNX microkernel is equipped with an event-gathering module and runs at 98% of the speed of the regular microkernel. When tracing, system activity is intercepted by generating time-stamped and CPU stamped events to be stored in a circular linked list of small buffers. This allows the data capture program to store the filled buffers to the disk while other available buffers can still be used to log more events. Kernel activities that generate events are system calls, scheduling activities, interrupt handling, thread/process creation, destruction and state changes. The instrumented kernel can't flush a buffer or switch to another one within an interrupt. To solve this problem, the instrumented kernel requests a buffer flush when it becomes 70% full. The buffer is composed of 1024 event slots, of 16 bytes each and most interrupt routines require less than 300 event buffer slots (approximately 30% of 1024 event buffer slots). Most events (simple events) usually fit in a single event buffer slot, but this is not the rule. Events holding too much information (combine event) can consume two or more event buffer slots. In fast mode, only one buffer slot is allocated to all events (combine events will be incomplete). The traceevent_t structure is 16 bytes long. To represent a combine event, multiple traceevent_t elements are required. The traceevent_t structure includes a 2-bit flag indicating whether the event is a single or combine one. Data is written in raw binary format to a device or file for offline processing. The library libtraceparser provides a set of API functions that allow the user to set up a series of callback functions associated with each event, and called when complete buffer slots of event data have been read and assembled from the binary event stream. The libtraceparser API transparently assemble combine events and sorts the events with respect to their timestamp. The timestamp uses the 32 LSB only of the 64-bit clock to reduce the amount of data in the trace. When this portion rolls over, a control event is issued. A provided tool called traceprinter uses this library and outputs all of the trace events in a human readable format and ordered linearly by their timestamp.

Measurements

The aim of this section is to determine the cost in time per traced event. The benchmark consists of compiling the gcc compiler on a Pentium 4 3.0GHz processor with 2GB of RAM. Only kernel events were traced to avoid loosing some events when writing to the tracefile. The compilation generated around 13 million events on average, with an overhead of around 24 additional seconds to finish the test execution. This gives an approximate cost of 1.85 x 10-6 seconds per event.

QNX Momentics Development Suite

QNX Momentics Development Suite is an Eclipse based IDE for C/C++ and embedded C++ development. It supports three host platforms which are Linux, QNX Neutrino and Windows. It has a built-in support for the CVS source-control protocol with support for both remote pserver and secure SSH repository access. It provides many tools for trace analysis and application debugging.

Trace Analysis

The System Profiler Perspective in the QNX Momentics IDE groups different views for trace analysis. Some of these views are described below:

*The **summary view** shows the time spent when the system was idle, was processing interrupts, was executing kernel code and user code. The **CPU activity** view shows the total amount of CPU time a thread or a process took throughout the period of the trace file or during a selected interval. The **Timeline** view shows the succession of events relative to every process, where the different thread states are shown in different colors. The **CPU Migration pane** displays a chart showing the number of CPU scheduling migrations over time. The count is incremented every time a thread migrates from one CPU to another. Another chart shows the number messages sent between a client and a server running on two different processors. The **Bookmarks** view can be used to bookmark events of particular interest. In the **Client/Server CPU Statistics** view, the thread running time is divided into self time and imposed time. For example, if a client X requests a service from server Y then the time Y consumes is considered an "imposed on" time. It becomes easier to find which clients are the most demanding and are causing the bottleneck. The **Overview** view has two charts. The first one displays the CPU usage over the period of the trace file, and the second shows the event distribution over time. The **Trace Event Log** view shows all the fields of an event including its number, timestamp, class, type and others. The **why running tool** enables the user to backtrace relevant events leading up to the execution of a selected thread.*

Application Analysis Tools

- *Memory analysis tools*
- *Application profiler*
- *Code coverage*

Memory Analysis tools

When a program is launched with the Memory Analysis tools, it uses the debug version of the malloc library (libmalloc_g.so). This library tracks the history of every allocation and deallocation the program does and provides cover functions that validate the corresponding *function's parameters before using them. This is used to detect memory leaks and memory errors such as overruns, underruns and freeing the same memory twice. Whenever a memory error is encountered at runtime, the developer can either request a process core dump file, or switch to the debugger view where he can identify the erroneous lines of code. Offline analysis is also possible. Once the information is available, a graph showing the requested, allocated and freed memory can be displayed in the Memory Analysis perspective. A list of memory errors is also provided showing the type, the pointer, the timestamp, the pid, the tid and the memory operation requested.*

Application profiler

The main target of the application profiler is to identify highly executed sections of code which are the most eligible candidates for optimization. The Application Profiler provides two types of profiling: Statistical and Instrumented profiling. In statistical profiling, the tool relies on sampling the running code every millisecond and recording the address being executed. For this type of profiling to be accurate, the program should run for a sufficiently long time. The advantage of this method is that no instrumentation and code recompilation is required, and the profiler can be attached dynamically to a running process on the target system. In instrumented profiling. The compiler inserts snippets of code into the different functions to report the addresses of the called and calling functions. If the program is to be run on an embedded system, the profiler update interval can be set accordingly. For example, a low interval would result in continuous, but low network traffic whereas a long interval may result in forcing the embedded system to buffer a relatively large amount of profiled data.

Code coverage

The code coverage tool in the Momentics IDE is a visual front end to the gcov metrics generated by the gcc compiler. A list of coverage information is generated showing, lines fully covered, not covered and partially covered. The coverage tool also highlights the lines of code that were not executed during the testing phase. Branch coverage is not yet provided by the IDE although this metric is already produced by gcc. Finally a report can be generated and saved for later analysis or comparison with other reports. In the report we get for every function, the number of lines that were not covered, that were partially covered and that were fully covered.

B.2.5 WindRiver Workbench

Company

Wind River Systems, Inc. is a Software company providing mainly solutions for embedded systems including operating systems such as vxWorks and Wind River Linux, as well as development tools.

Kernel Tracing

Wind River Linux uses the LTTng framework as a data provider. Tracing can be activated in three different modes: normal mode, flight recorder mode and hybrid mode. In normal mode, all events are traced and written completely to disk during the trace period. In flight recorder mode, when the memory buffers are filled, the oldest data is overwritten. In hybrid mode, only critical, *low rate information is logged to disk such as process create/exit, while frequently occurring events are logged in a flight recorder buffer. The number and size of memory sub-buffers is user configurable. Once the LTTng trace file is generated, it is converted to System Viewer format (.wvr) so that it can be analyzed inside the Wind River Workbench.*

Wind River Workbench

Wind River Workbench is an Eclipse based development suite providing a rich set of tools for software development and debugging. This document describes a few but very important features.

Target Debugging

The target can be connected with the workbench host for debugging via one of the three possible ways: A direct connection to the target with a 8250 serial connection using a null-modem cable, or via a terminal server with TCP, or by Ethernet with UDP. Once the connection is established, workbench shifts to the Device Debug perspective. Here it is possible to add breakpoints in the kernel code. Once that breakpoints is hit, the Editor inside the perspective will open the corresponding source file. On specific architectures like the IA-32, hardware breakpoints can be set. Therefore, the normal execution of the program can be interrupted when a specific variable is read or written, or when a specific instruction is read for execution. Breakpoint properties can be saved to a file and imported from it later on.

System Viewer

The Wind River Workbench System Viewer is used to capture the dynamic interactions of the operating system, device software application and target hardware. It displays graphically the trace data, allowing device software developers to detect anomalous behavior relatively quickly. The tool is architecture independent and can be used with VxWorks simulator (for VxWorks 6.x) before hardware is available and with Linux platforms using the Linux Trace Toolkit (LTTng) instrumentation. Logged events correspond to one of the following kernel activities: Semaphore gives and takes, task spawns and deletions, timer expirations, interrupts, message queue sends and receives, watchdog time activity, exceptions, signal activity, system calls, I/O activity, networking activity, memory allocation, freeing and partitioning, protection domain activity (for VxWorks 653 only).

Event Log Analysis

The System Viewer provides various graphical ways to make data analysis most effective:

The Event Graph: The Event Graph displays the succession of events relative to each thread. Holding the mouse over any event will make the event name, time stamp and argument values appear. Icons can help identify events in the graph. Event type filters can be used to considerably reduce the amount of data to analyze.

The Event Table: The Event Table displays events as rows of information ordered by their time stamp. The columns show the event name, the timestamp, the thread state, the event id, the time difference between current and previous event, and other different event specific arguments.

The Memory usage graph displays memory allocation and deallocation resulting from memLib function calls. This graph shows the memory usage versus time. It also shows the addresses of memory blocks that were allocated and then freed.

Upload Modes

An upload mode consists of sending trace data from the target to the host. The choice of the upload mode depends on several factors including the amount of trace data being generated, the kind of problem being tracked down, the target buffering capacity, etc... The three types of uploads that can be selected are the Deferred Upload, the Continuous Upload and the Post-Mortem Upload.

- *Deferred Upload*

In the Deferred Upload (default mode), data is uploaded when the Upload command is issued from the System Viewer user interface. When using circular buffers on some systems (not VxWorks 653), the oldest data is overwritten with the newest data when the ring is full. With non-circular buffers, logging is stopped when the buffer becomes full. The limitation of this mode is that the volume of collected data depends on whatever free memory is available on the target.

- *Continuous upload*

Here, data is uploaded periodically from the target as buffers are filled. Tracing continues until stopped by a trigger, an API call or on demand. This method is more intrusive than the preceding one and has more impact on target performance but can provide large amounts of trace data with no interruption.

- *Post-Mortem Upload*

This mode, also known as flight-recorder mode, is used to collect events leading up to a system failure. Events are stored in a circular ring of buffers where the oldest data is overwritten by the newest data when the ring is full. After a crash, the buffer contents can be uploaded from the System viewer user interface. This method requires that the buffer is stored in memory that is not overwritten on rebooting. This method can still be used even if the target system is not expected to crash.

Sensorpoints

Wind River Workbench diagnostics is a plug in to Wind River Workbench and provides Sensorpoints (http://www.windriver.com/products/device_management/sensorpoint/) technology which can help test software relatively quickly. Sensorpoints are lines of code that can be used to instrument dynamically a running application without having to rebuild the application or reboot the device. Sensorpoints give access to global and local variables within the scope of the function to patch. They can be used to monitor, or to interfere with the execution of the application by injecting specific values in variables. Sensorpoints can also be used to create a trace of global and local variables, function calls, and to timestamp any point in the system to reveal timing problems. They can be disabled when the test is finished.

B.2.6 Zealcore System debuger

Company

Zealcore is a Swedish software development company whose main interest is to provide developers with debugging tools to help them "find the last bug". Its products are targeted to accelerate development and troubleshooting of embedded systems software through instrumentation, trace analysis, profiling and visualization tools.

Products

The three main products of Zealcore are the System Debugger, the Model Replay Debugger and the System Recorder. The System Recorder provides a general framework for recording system data. The currently supported operating systems are VxWorks, Enea OSE and Linux. The System Debugger is a post analyzer and graphical viewer of logged information. It will be described briefly in the next section. The Model Replay Debugger records and displays the behavior of UML models. It can also check for assertions to help identify model faults.

System Debugger

The System Debugger provides multiple browsers to visualize the logged information in different ways. They are the Timeline Browser, Gantt chart, Text browser, Sequence diagram, state diagram and the plot. The Timeline Browser displays the relative logarithmic time distance, between events. Different events can be shown with different colors on the timeline. A dynamic legend displays the events colors just for the ones appearing on the timeline. When an event is selected, its occurrence time is shown. The common timebase for the system debugger is time in nanoseconds since the epoch. The Gantt chart displays the scheduling of threads. Every thread execution is drawn horizontally taking into consideration the time it was scheduled in, how long it executed, and the time it was scheduled out. The text browser displays the events with all of their fields in a human readable way, one event per line. The Sequence diagram shows processes or objects (actors) as parallel vertical lines, and messages passed between them as horizontal arrows. The State Diagram shows for a given actor the model of its recorded behavior. The current state is marked by a thicker frame and the next state is marked by a dashed arrow together with the transition condition. All of these different views discussed so far are synchronized in time. The plot view draws charts derived from values of a basic search. For example, if there is an event collecting the actual temperature, then it is possible to plot the temperature versus time. System Debugger has also a logmark view that can be used to bookmark events of special interest. Another feature is the ability to define and run assertions. An assertion is created by selecting first an event type, and then adding a constraint on one or more of its fields. As an example, an assertion can state that the temperature variable, which is collected from a specific type of events, is always less than 50 degrees. After running the created assertion, a view of all the events that violate the specified constraints will be displayed.

Glossary

This glossary provides minimal definitions for most frequently used terms in the domain of monitoring, tracing and analysis of critical software systems. It does not pretend to: 1) be complete or 2) reflect the work of all researchers in this domain. It rather aims at giving the reader a “first-order definition” for each of these terms.

Application, application suite, system software

WikipediaWS (2008): ***Application software*** *is a subclass of computer software that employs the capabilities of a computer directly and thoroughly to a task that the user wishes to perform. This should be contrasted with* ***system software*** *which is involved in integrating a computer's various capabilities, but typically does not directly apply them in the performance of tasks that benefit the user. In this context the term application refers to both the application software and its implementation.* (…) *Typical examples of software applications are word processors, spreadsheets, and media players. Multiple applications bundled together as a package are sometimes referred to as an* ***application suite****. Microsoft Office and OpenOffice.org, which bundle together a word processor, a spreadsheet, and several other discrete applications, are typical examples. The separate applications in a suite usually have a user interface that has some commonality making it easier for the user to learn and use each application. And often they may have some capability to interact with each other in ways beneficial to the user. For example, a spreadsheet might be able to be embedded in a word processor document even though it had been created in the separate spreadsheet application.*

Foldoc (2008): *A complete, self-contained program that performs a specific function directly for the user. This is in contrast to system software such as the operating system kernel, server processes, libraries which exists to support application programs and utility programs.*

Application Programming Interface (API)

WikipediaWS (2008): *An application programming interface (API) is a source code interface that an operating system or library provides to support requests for services to be made of it by computer programs.[1]*

Foldoc (2008): *The interface (calling conventions) by which an application program accesses operating system and other services. An API is defined at source code level and provides a level of abstraction between the application and the kernel (or other privileged utilities) to ensure the portability of the code. An API can also provide an interface between a high level language and lower level utilities and services which were written without consideration for the calling conventions supported by compiled languages. In this case, the API's main task may be the translation of parameter lists from one format to another and the interpretation of call-by-value and call-by-reference arguments in one or both directions.*

Cache

WikipediaWS (2008): *In computer science, a cache is a collection of data duplicating original values stored elsewhere or computed earlier, where the original data is expensive to* *fetch (owing to longer access time) or to compute, compared to the cost of reading the cache. In other words, a cache is a temporary storage area where frequently accessed data can be stored for rapid access. Once the data is stored in the cache, future use can be made by accessing the cached copy rather than re-fetching or recomputing the original data, so that the average access time is shorter. Cache, therefore, helps expedite data access that the CPU would otherwise need to fetch from main memory. Cache have proven to be extremely effective in many areas of computing because access patterns in typical computer applications have locality of reference.*

Foldoc (2008): *A small fast memory holding recently accessed data, designed to speed up subsequent access to the same data. Most often applied to processor-memory access but also used for a local copy of data accessible over a network etc. When data is read from, or written to, main memory a copy is also saved in the cache, along with the associated main memory address. The cache monitors addresses of subsequent reads to see if the required data is already in the cache. If it is (a cache hit) then it is returned immediately and the main memory read is aborted (or not started). If the data is not cached (a cache miss) then it is fetched from main memory and also saved in the cache. The cache is built from faster memory chips than main memory so a cache hit takes much less time to complete than a normal memory access. The cache may be located on the same integrated circuit as the CPU, in order to further reduce the access time. In this case it is often known as primary cache since there may be a larger, slower secondary cache outside the CPU chip.*

Central Processing Unit (CPU)

WikipediaWS (2008): *A central processing unit (CPU), or sometimes just processor, is a description of a class of logic machines that can execute computer programs. This broad definition can easily be applied to many early computers that existed long before the term "CPU" ever came into widespread usage. However, the term itself and its initialism have been in use in the computer industry at least since the early 1960s (Weik 1961). The form, design and implementation of CPUs have changed dramatically since the earliest examples, but their fundamental operation has remained much the same. Early CPUs were custom-designed as a part of a larger, usually one-of-a-kind, computer. However, this costly method of designing custom CPUs for a particular application has largely given way to the development of mass-produced processors that are suited for one or many purposes. This standardization trend generally began in the era of discrete transistor mainframes and minicomputers and has rapidly accelerated with the popularization of the integrated circuit (IC). The IC has allowed increasingly complex CPUs to be designed and manufactured in very small spaces (on the order of millimeters). Both the miniaturization and standardization of CPUs have increased the presence of these digital devices in modern life far beyond the limited application of dedicated computing machines.*

Foldoc (2008): *The part of a computer which controls all the other parts. Designs vary widely but the CPU generally consists of the control unit, the arithmetic and logic unit (ALU), registers, temporary buffers and various other logic. The control unit fetches instructions from memory and decodes them to produce signals which control the other parts of the computer. These signals cause it to transfer data between memory and ALU or to activate peripherals to perform input or output. Various types of memory, including cache, RAM and ROM, are often considered to be part of the CPU, particularly in modern microprocessors where a single integrated circuit may contain one or more processors as* *well as any or all of the above types of memory. The CPU, and any of these components that are in separate chips, are usually all located on the same printed circuit board, known as the motherboard. This in turn is located in the system unit (sometimes incorrectly referred to as the "CPU"). A parallel computer has several CPUs which may share other resources such as memory and peripherals. The term "processor" has to some extent replaced "CPU", though RAM and ROM are not logically part of the processor.*

Clock cycle

WikipediaWS (2008): *In electronics and especially synchronous digital circuits, a clock signal is a signal used to coordinate the actions of two or more circuits.*

Code coverage

WikipediaWS (2008): *Code coverage is a measure used in software testing. It describes the degree to which the source code of a program has been tested. It is a form of testing that looks at the code directly and as such comes under the heading of white box testing. Currently, the use of code coverage is extended to the field of digital hardware, the contemporary design methodology of which relies on Hardware description languages (HDL's). Code coverage techniques were amongst the first techniques invented for systematic software testing. The first published reference was by Miller and Maloney in Communications of the ACM in 1963.*

Context switch

WikipediaWS (2008): *A context switch is the computing process of storing and restoring the state (context) of a CPU such that multiple processes can share a single CPU resource. The context switch is an essential feature of a multitasking operating system. Context switches are usually computationally intensive and much of the design of operating systems is to optimize the use of context switches. A context switch can mean a register context switch, a task context switch, a thread context switch, or a process context switch. What constitutes the context is determined by the processor and the operating system.*

Foldoc (2008): *When a multitasking operating system stops running one process and starts running another. Many operating systems implement concurrency by maintaining separate environments or "contexts" for each process. The amount of separation between processes, and the amount of information in a context, depends on the operating system but generally the OS should prevent processes interfering with each other, e.g. by modifying each other's memory. A context switch can be as simple as changing the value of the program counter and stack pointer or it might involve resetting the MMU to make a different set of memory pages available. In order to present the user with an impression of parallism, and to allow processes to respond quickly to external events, many systems will context switch tens or hundreds of times per second.*

CPU modes, privilege mode, kernel mode, protected mode, user modes

WikipediaWS (2008): *CPU modes (also called processor modes or CPU privilege levels, and by other names) are operating modes for the central processing unit of some computer architectures that place restrictions on the operations that can be performed by the process currently running in the CPU. This design allows the operating system to run at different* *privilege levels. These different privilege levels are called rings when referring to their implementation at the OS abstraction level, while CPU modes when referring to their implementation at the cpu firmware abstraction level.* (…) *At a minimum, any CPU with this type of architecture will support at least two distinct operating modes, and at least one of the modes will provide completely unrestricted operation of the CPU. The unrestricted mode is usually called kernel mode, but many other designations exist (master mode, supervisor mode, privileged mode etc.). Other modes are usually called user modes, but are occasionally known by other names (slave mode etc.). In kernel mode, the CPU may perform any operation provided for by its architecture. Any instruction may be executed, any I/O operation may be initiated, any area of memory may be accessed, and so on. In the other CPU modes, certain restrictions on CPU operations are enforced by the hardware. Typically certain instructions are not permitted, I/O operations may not be initiated, some areas of memory cannot be accessed etc. Usually the user-mode capabilities of the CPU are a subset of the kernel mode capabilities, but in some cases (such as hardware emulation of non-native architectures), they may be significantly different from kernel capabilities, and not just a subset of them. At least one user mode is always defined, but some CPU architectures support multiple user modes, often with a hierarchy of privileges. These architectures are often said to have ring-based security, wherein the hierarchy of privileges resembles a set of concentric rings, with the kernel mode in the central, innermost ring. Multics hardware was the first significant implementation of ring security, but many other hardware platforms have been designed along similar lines, including the Intel 80286 protected mode, and the IA-64 as well, though it is referred to by a different name in these cases. Mode protection may extend to resources beyond the CPU processing hardware itself. Hardware registers track the current operating mode of the CPU, but additional virtual-memory registers, page-table entries, and other data may track mode identifiers for other resources. For example, a CPU may be operating in Ring 0 as indicated by a status word in the CPU itself, but every access to memory may additionally be validated against a separate ring number for the virtual-memory segment targeted by the access, and/or against a ring number for the physical page (if any) being targeted. This has been demonstrated with the PSP handheld system.*

Device driver

WikipediaWS (2008): *A device driver, or software driver is a computer program allowing higher-level computer programs to interact with a computer hardware device. A driver typically communicates with the device through the computer bus or communications subsystem to which the hardware is connected. When a calling program invokes a routine in the driver, the driver issues commands to the device. Once the device sends data back to the driver, the driver may invoke routines in the original calling program. Drivers are hardware-dependent and operating-system-specific. They usually provide the interrupt handling required for any necessary asynchronous time-dependent hardware interface.*

Foldoc (2008): *Software to control a hardware component or peripheral device of a computer such as a magnetic disk, magnetic tape or printer. A device driver is responsible for accessing the hardware registers of the device and often includes an interrupt handler to service interrupts generated by the device. Device drivers often form part of the lowest level of the operating system kernel, with which they are linked when the kernel is built. Some more recent systems have loadable device drivers which can be installed from files after the operating system is running.*

Dynamic program analysis

WikipediaWS (2008): *Dynamic program analysis is the analysis of computer software that is performed with executing programs built from that software on a real or virtual processor (analysis performed without executing programs is known as static code analysis). Dynamic program analysis tools may require loading of special libraries or even recompilation of program code. For dynamic program analysis to be effective, the target program must be executed with sufficient test inputs to produce interesting behavior. Use of software testing techniques such as code coverage helps to ensure that an adequate slice of the program's set of possible behaviors has been observed. Care must be taken to minimize the effect that instrumentation or probing has on the execution (including temporal properties) of the target program, to minimize the appearance of Heisenbugs.*

Foldoc (2008): *Evaluation of a program based on its execution. Dynamic analysis relies on executing a piece of software with selected test data.*

Formal methods

WikipediaWS (2008): *In computer science and software engineering, formal methods are mathematically-based techniques for the specification, development and verification of software and hardware systems.[1] The use of formal methods for software and hardware design is motivated by the expectation that, as in other engineering disciplines, performing appropriate mathematical analyses can contribute to the reliability and robustness of a design.[2] However, the high cost of using formal methods means that they are usually only used in the development of high-integrity systems, where safety or security is important.*

Foldoc (2008): *Mathematically based techniques for the specification, development and verification of software and hardware systems. Referentially transparent languages are amenable to symbolic manipulation allowing program transformation (e.g. changing a clear inefficient specification into an obscure but efficient program) and proof of correctness.*

Framework

WikipediaWS (2008): *A framework is a basic conceptual structure used to solve or address complex issues. This very broad definition has allowed the term to be used as a buzzword, especially in a software context.*

Hardware performance counter or hardware counter

WikipediaWS (2008): *In computers, hardware performance counters, or hardware counters are a set of special-purpose registers built in modern microprocessors to store the counts of hardware-related activities within computer systems. Advanced users often rely on those counters to conduct low-level performance analysis or tuning.*

Hooking, hooks

WikipediaWS (2008): *Hooking in programming is a technique employing so called hooks to make a chain of procedures as an event handler. Thus, after the handled event occurs, control flow follows the chain in specific order. The new hook registers its own address as handler for the event and is expected to call the original handler at some point, usually at the end. Each hook is required to pass execution to the previous handler, eventually arriving to* *the default one, otherwise the chain is broken. Unregistering the hook means setting the original procedure as the event handler. Hooking can be used for many purposes, including debugging and extending original functionality. It can also be misused to inject (potentially malicious) code to the event handler - for example, rootkits try to make themselves invisible by faking the output of API calls that would otherwise reveal their existence. A special form of hooking employs intercepting the library functions calls made by a process. Function hooking is implemented by changing the very first few code instructions of the target function to jump to an injected code.*

Instruction

WikipediaWS (2008): *In computer science, an instruction is a single operation of a processor defined by an instruction set architecture. In a broader sense, an "instruction" may be any representation of an element of an executable program, such as a bytecode.*

Instrumentation

The insertion of additional code or breakpoints into software in order to collect runtime information about program execution which would otherwise not be obtainable.

Interrupt

WikipediaWS (2008): *In computing, an interrupt is an asynchronous signal from hardware indicating the need for attention or a synchronous event in software indicating the need for a change in execution. A hardware interrupt causes the processor to save its state of execution via a context switch, and begin execution of an interrupt handler. Software interrupts are usually implemented as instructions in the instruction set, which cause a context switch to an interrupt handler similar to a hardware interrupt. Interrupts are a commonly used technique for computer multitasking, especially in real-time computing. Such a system is said to be interrupt-driven. An act of interrupting is referred to as an interrupt request ("IRQ").*

Kernel

WikipediaWS (2008): *In computer science, the kernel is the central component of most computer operating systems (OS). Its responsibilities include managing the system's resources (the communication between hardware and software components).[1] As a basic component of an operating system, a kernel provides the lowest-level abstraction layer for the resources (especially memory, processors and I/O devices) that application software must control to perform its function. It typically makes these facilities available to application processes through inter-process communication mechanisms and system calls. These tasks are done differently by different kernels, depending on their design and implementation. While monolithic kernels will try to achieve these goals by executing all the code in the same address space to increase the performance of the system, microkernels run most of their services in user space, aiming to improve maintainability and modularity of the codebase.[2] A range of possibilities exists between these two extremes.*

Kernel space and user space

WikipediaWS (2008): *A conventional operating system usually segregates virtual memory into kernel space and user space. Kernel space is strictly reserved for running the kernel, kernel extensions, and some device drivers. In most operating systems, kernel memory is* *never swapped out to disk. In contrast, user space is the memory area where all user mode applications work and this memory can be swapped out when necessary. The term userland is often used for referring to operating system software that runs in user space. Each user space process normally runs in its own virtual memory space, and, unless explicitly requested, cannot access the memory of other processes. This is the basis for memory protection in today's mainstream operating systems, and a building block for privilege separation. Depending on the privileges, processes can request the kernel to map part of another process's memory space to its own, as is the case for debuggers. Programs can also request shared memory regions with other processes. Another approach taken in experimental operating systems, is to have a single address space for all software, and rely on the programming language's virtual machine to make sure that arbitrary memory cannot be accessed — applications simply cannot acquire any references to the objects that they are not allowed to access.[1] This approach has been implemented in JXOS, Unununium as well as Microsoft's Singularity research project.*

Library

Redhat (2008). *When speaking of computers, refers to a collection of routines that perform operations which are commonly required by programs. Libraries may be shared, meaning that the library routines reside in a file separate from the programs that use them.*

Foldoc (2008): *A collection of subroutines and functions stored in one or more files, usually in compiled form, for linking with other programs. Libraries are one of the earliest forms of organized code reuse. They are often supplied by the operating system or software development environment developer to be used in many different programs. The routines in a library may be general purpose or designed for some specific function such as three dimensional animated graphics. Libraries are linked with the user's program to form a complete executable. The linking may be static linking or, in some systems, dynamic linking.*

Logging

WikipediaWS (2008): *Data logging is the practice of recording sequential data, often chronologically.* (…) *In computerized data logging, a computer program may automatically record events in a certain scope in order to provide an audit trail that can be used to diagnose problems. Examples of physical systems which have logging subsystems include process control systems, and the black box recorders installed in aircraft. Many operating systems and multitudinous computer programs include some form of logging subsystem. Some operating systems provide a syslog service (described in RFC 3164), which allows the filtering and recording of log messages to be performed by a separate dedicated subsystem, rather than placing the onus on each application to provide its own ad hoc logging system. In many cases, the logs are not esoteric and hard to understand; they need to be subjected to log analysis in order to make sense of them. It can be useful to combine log file entries from multiple sources. This approach, in combination with statistical analysis, may yield correlations between seemingly-unrelated events on different servers. Other solutions employ network-wide querying and reporting.*

Multi-core CPU

WikipediaWS (2008): *A multi-core CPU (or chip-level multiprocessor, CMP) combines two or more independent cores into a single package composed of a single integrated circuit (IC),* *called a die, or more dies packaged together. A dual-core processor contains two cores and a quad-core processor contains four cores. A multi-core microprocessor implements multiprocessing in a single physical package. A processor with all cores on a single die is called a monolithic processor. Cores in a multicore device may share a single coherent cache at the highest on-device cache level (e.g. L2 for the Intel Core 2) or may have separate caches (e.g. current AMD dual-core processors). The processors also share the same interconnect to the rest of the system. Each "core" independently implements optimizations such as superscalar execution, pipelining, and multithreading. A system with N cores is effective when it is presented with N or more threads concurrently. The most commercially significant (or at least the most 'obvious') multi-core processors are those used in computers (primarily from Intel & AMD) and game consoles (e.g., the Cell processor in the PS3). In this context, "multi" typically means a relatively small number of cores. However, the technology is widely used in other technology areas, especially those of embedded processors, such as network processors and digital signal processors, and in GPUs.*

Multi-processing, multi-processors

WikipediaWS (2008): *Multiprocessing is the use of two or more central processing units (CPUs) within a single computer system. The term also refers to the ability of a system to support more than one processor and/or the ability to allocate tasks between them.[1] There are many variations on this basic theme, and the definition of multiprocessing can vary with context, mostly as a function of how CPUs are defined (multiple cores on one die, multiple chips in one package, multiple packages in one system unit, etc.). Multiprocessing sometimes refers to the execution of multiple concurrent software processes in a system as opposed to a single process at any one instant. However, the term multiprogramming is more appropriate to describe this concept, which is implemented mostly in software, whereas multiprocessing is more appropriate to describe the use of multiple hardware CPUs. A system can be both multiprocessing and multiprogramming, only one of the two, or neither of the two.*

Multi-tasking

WikipediaWS (2008): *In computing, multitasking is a method by which multiple tasks, also known as processes, share common processing resources such as a CPU. In the case of a computer with a single CPU, only one task is said to be running at any point in time, meaning that the CPU is actively executing instructions for that task. Multitasking solves the problem by scheduling which task may be the one running at any given time, and when another waiting task gets a turn. The act of reassigning a CPU from one task to another one is called a context switch. When context switches occur frequently enough the illusion of parallelism is achieved. Even on computers with more than one CPU (called multiprocessor machines), multitasking allows many more tasks to be run than there are CPUs.*

Non-uniform memory access (NUMA)

WikipediaWS (2008): *Non-Uniform Memory Access or Non-Uniform Memory Architecture (NUMA) is a computer memory design used in multiprocessors, where the memory access time depends on the memory location relative to a processor. Under NUMA, a processor can access its own local memory faster than non-local memory, that is, memory local to another processor or memory shared between processors. NUMA architectures logically follow in scaling from symmetric multiprocessing (SMP) architectures. Their commercial development came in work by Burroughs, Convex Computer (later HP), SGI, Sequent and Data General* *during the 1990s. Techniques developed by these companies later featured in a variety of Unix-like operating systems, as well as to some degree in Windows NT.*

Open source

WikipediaWS (2008): *Open source is a set of principles and practices on how to write software, the most important of which is that the source code is openly available. The Open Source Definition, which was created by Bruce Perens[1] and Eric Raymond and is currently maintained by the Open Source Initiative, adds additional meaning to the term: one should not only get the source code but also have the right to use it. If the latter is denied the license is categorized as a shared source license.*

Foldoc (2008): *A method and philosophy for software licensing and distribution designed to encourage use and improvement of software written by volunteers by ensuring that anyone can copy the source code and modify it freely. The term "open source" is now more widely used than the earlier term "free software" (promoted by the Free Software Foundation) but has broadly the same meaning - free of distribution restrictions, not necessarily free of charge. There are various open source licenses available. Programmers can choose an appropriate license to use when distributing their programs. The Open Source Initiative promotes the Open Source Definition.*

Page table

WikipediaWS (2008): *A page table is the data structure used by a virtual memory system in a computer operating system to store the mapping between virtual addresses and physical addresses. Virtual addresses are those unique to the accessing process. Physical addresses are those unique to the CPU, i.e., RAM.*

Parallel computing, parallel processing

WikipediaWS (2008): *Parallel computing is a form of computing in which many instructions are carried out simultaneously.[1] Parallel computing operates on the principle that large problems can almost always be divided into smaller ones, which may be carried out concurrently ("in parallel"). Parallel computing exists in several different forms: bit-level parallelism, instruction level parallelism, data parallelism, and task parallelism. It has been used for many years, mainly in high performance computing, but interest in it has become greater in recent years due to physical constraints preventing frequency scaling. Parallel computing has recently become the dominant paradigm in computer architecture, mainly in the form of multicore processors.[2] Parallel computer programs are harder to write than sequential ones,[3] because concurrency introduces several new classes of potential software bugs, of which race conditions are the most common. Communication and synchronization between the different subtasks is typically one of the greatest barriers to getting good parallel program performance. In recent years, power consumption in parallel computers has also become a great concern.[4] The speed up of a program as a result of parallelization is given by Amdahl's law.*

Foldoc (2008): *The simultaneous use of more than one computer to solve a problem. There are many different kinds of parallel computer (or "parallel processor"). They are distinguished by the kind of interconnection between processors (known as "processing elements" or PEs) and between processors and memory. Flynn's taxonomy also classifies* *parallel (and serial) computers according to whether all processors execute the same instructions at the same time ("single instruction/multiple data" - SIMD) or each processor executes different instructions ("multiple instruction/multiple data" - MIMD). The processors may either communicate in order to be able to cooperate in solving a problem or they may run completely independently, possibly under the control of another processor which distributes work to the others and collects results from them (a "processor farm").* (…) *Processors communicate via some kind of network or bus or a combination of both. Memory may be either shared memory (all processors have equal access to all memory) or private (each processor has its own memory - "distributed memory") or a combination of both. Many different software systems have been designed for programming parallel computers, both at the operating system and programming language level. These systems must provide mechanisms for partitioning the overall problem into separate tasks and allocating tasks to processors. Such mechanisms may provide either implicit parallelism - the system (the compiler or some other program) partitions the problem and allocates tasks to processors automatically or explicit parallelism where the programmer must annotate his program to show how it is to be partitioned. It is also usual to provide synchronization primitives such as semaphores and monitors to allow processes to share resources without conflict. Load balancing attempts to keep all processors busy by allocating new tasks, or by moving existing tasks between processors, according to some algorithm. Communication between tasks may be either via shared memory or message passing. Either may be implemented in terms of the other and in fact, at the lowest level, shared memory uses message passing since the address and data signals which flow between processor and memory may be considered as messages. The terms "parallel processing" and "multiprocessing" imply multiple processors working on one task whereas "concurrent processing" and "multitasking" imply a single processor sharing its time between several tasks.*

Performance counter

WikipediaWS (2008): *In computers, hardware performance counters, or hardware counters are a set of special-purpose registers built in modern microprocessors to store the counts of hardware-related activities within computer systems. Advanced users often rely on those counters to conduct low-level performance analysis or tuning.* (…) *Compared to software profilers, hardware counters provide low-overhead access to a wealth of detailed performance information related to CPU's function units, caches and main memory etc. Another benefit of using them is that no source code modifications are needed in general. However, the types and meanings of hardware counters vary from one kind of architecture to another due to the variation in hardware organizations. Also, it is difficult to correlate the low level performance metrics back to source code. The limited number of registers to store the counters often force users to conduct multiple measurements to collect all desired performance metrics.*

Performance engineering

WikipediaWS (2008): *Within Systems engineering, Performance engineering encompasses the set of roles, skills, activities, practices, tools, and deliverables applied at every phase of the Systems Development Lifecycle which ensures that a solution will be designed, implemented, and operationally supported to meet the non-functional requirements defined for the solution. It may be alternatively referred to as software performance engineering within software engineering; however since performance engineering encompasses more* *than just the software, the term performance engineering is preferable. Adherence to the non-functional requirements is validated by monitoring the production systems. This is part of IT Service Management (see also ITIL). Performance engineering has become a separate discipline at a number of large corporations, and may be affiliated with the enterprise architecture group. It is pervasive, involving people from multiple organizational units; but predominantly within the information technology organization.*

Process

WikipediaWS (2008): *In computing, a process is an instance of a computer program that is being sequentially executed.[1] A computer program itself is just a passive collection of instructions, while a process is the actual execution of those instructions. Several processes may be associated with the same program, for example opening up two windows of the same program typically means two processes are being executed. A single computer processor executes only one instruction at a time. To allow users to run several programs at once, single-processor computer systems can perform time-sharing - processes switch between being executed and waiting to be executed. In most cases this is done at a very fast rate, giving an illusion that several processes are executing at once. Using multiple processors achieves actual simultaneous execution of multiple instructions from different processes, but time-sharing is still typically used to allow more processes to run at once. For security reasons most modern operating systems prevent direct inter-process communication, providing mediated and limited functionality. However, a process may split itself into multiple threads that execute in parallel, running different instructions on much of the same resources and data. This is useful when, for example, it is necessary to make it seem that multiple things within the same process are happening at once (such as a spell check being performed in a word processor while the user is typing), or if part of the process needs to wait for something else to happen (such as a web browser waiting for a web page to be retrieved).*

Foldoc (2008): *The sequence of states of an executing program. A process consists of the program code (which may be shared with other processes which are executing the same program), private data, and the state of the processor, particularly the values in its registers. It may have other associated resources such as a process identifier, open files, CPU time limits, shared memory, child processes, and signal handlers. One process may, on some platforms, consist of many threads. A multitasking operating system can run multiple processes concurrently or in parallel, and allows a process to spawn "child" processes.*

Process control

WikipediaWS (2008): *Process control is a statistics and engineering discipline that deals with architectures, mechanisms, and algorithms for controlling the output of a specific process.*

Profiling, performance analysis, profilers

WikipediaWS (2008): *In software engineering, performance analysis, more commonly **profiling**, is the investigation of a program's behavior using information gathered as the program runs (i.e. it is a form of dynamic program analysis, as opposed to static code analysis). The usual goal of performance analysis is to determine which parts of a program to optimize for speed or memory usage.* (…) *In software engineering, performance analysis,* *more commonly profiling, is the investigation of a program's behavior using information gathered as the program runs (i.e. it is a form of dynamic program analysis, as opposed to static code analysis). The usual goal of performance analysis is to determine which parts of a program to optimize for speed or memory usage.* (…) *A **profiler** is a performance analysis tool that measures the behavior of a program as it runs, particularly the frequency and duration of function calls. The output is a stream of recorded events (a trace) or a statistical summary of the events observed (a profile). Profilers use a wide variety of techniques to collect data, including hardware interrupts, code instrumentation, operating system hooks, and performance counters. The usage of profilers is called out in the performance engineering process. As the summation in a profile often is done related to the source code positions where the events happen, the size of measurement data is linear to the code size of the program. In contrast, the size of a trace is linear to the program's execution time, making it somewhat impractical. For sequential programs, a profile is usually enough, but performance problems in parallel programs (waiting for messages or synchronization issues) often depend on the time relationship of events, thus requiring the full trace to get an understanding of the problem.*

Program counter

WikipediaWS (2008): *The program counter, or shorter PC (also called the instruction pointer, part of the instruction sequencer in some computers) is a register in a computer processor which indicates where the computer is in its instruction sequence. Depending on the details of the particular machine, it holds either the address of the instruction being executed, or the address of the next instruction to be executed. The program counter is automatically incremented for each instruction cycle so that instructions are normally retrieved sequentially from memory. Certain instructions, such as branches and subroutine calls and returns, interrupt the sequence by placing a new value in the program counter. In most processors, the instruction pointer is incremented immediately after fetching a program instruction; this means that the target address of a branch instruction is obtained by adding the branch instruction's operand to the address of the next instruction (byte or word, depending on the computer type) after the branch instruction. The address of the next instruction to be executed is always found in the instruction pointer. The basic model (non von Neumann) of Reconfigurable Computing systems, however, uses data counters instead of a program counter.*

Real time

WikipediaWS (2008): *In computer science, real-time computing (RTC) is the study of hardware and software systems which are subject to a "real-time constraint"—i.e., operational deadlines from event to system response. By contrast, a non-real-time system is one for which there is no deadline, even if fast response or high performance is desired or even preferred. The needs of real-time software are often addressed in the context of real-time operating systems, and synchronous programming languages, which provide frameworks on which to build real-time application software. A real time system may be one where its application can be considered (within context) to be mission critical.* (…) *Real-time computations can be said to have failed if they are not completed before their deadline, where their deadline is relative to an event. A real-time deadline must be met, regardless of system load.*

Sample

(From this document): a sample is one record of a specific observation that was made on a running system at one specific instant “t”. The record may be composed of one or more different fields. Examples are: the degree of network load or the complete image of the system’s memory at one given time.

Static code analysis

WikipediaWS (2008): *Static code analysis is the analysis of computer software that is performed without actually executing programs built from that software (analysis performed on executing programs is known as dynamic analysis). In most cases the analysis is performed on some version of the source code and in the other cases some form of the object code. The term is usually applied to the analysis performed by an automated tool, with human analysis being called program understanding or program comprehension. The sophistication of the analysis performed by tools varies from those that only consider the behavior of individual statements and declarations, to those that include the complete source code of a program in their analysis. Uses of the information obtained from the analysis vary from highlighting possible coding errors (e.g., the lint tool) to formal methods that mathematically prove properties about a given program (e.g., its behavior matches that of its specification). Some people* (…) *consider software metrics and reverse engineering to be forms of static analysis. A growing commercial use of static analysis is in the verification of properties of software used in safety-critical computer systems and locating potentially vulnerable code.*

Swap, swap space

Redhat (2008): *Also known as "swap space." When a program requires more memory than is physically available in the computer, currently-unused information can be written to a temporary buffer on the hard disk, called swap, thereby freeing memory. Some operating systems support swapping to a specific file, but Linux normally swaps to a dedicated swap partition. A misnomer, the term swap in Linux is used to define demand paging.*

Symmetric multi-processing (SMP), symmetric multi-processors

WikipediaWS (2008): *Symmetric multiprocessing, or SMP, is a multiprocessor computer architecture where two or more identical processors are connected to a single shared main memory. Most common multiprocessor systems today use an SMP architecture. In case of multi-core processors, the SMP architecture applies to the cores, treating them as separate processors. SMP systems allow any processor to work on any task no matter where the data for that task are located in memory; with proper operating system support, SMP systems can easily move tasks between processors to balance the workload efficiently*.

Task

WikipediaWS (2008): *A task is "an execution path through address space". In other words, a set of program instructions that is loaded in memory. The address registers have been loaded with the initial address of the program. At the next clock cycle, the CPU will start execution, in accord with the program. The sense is that some part of 'a plan is being accomplished'. As long as the program remains in this part of the address space, the task can continue, in principle, indefinitely, unless the program instructions contain a halt, exit, or return. In the* *computer field, "task" has the sense of a real-time application, as distinguished from process, which takes up space (memory), and execution time.* (…) *Both "task" and "process" should be distinguished from event, which takes place at a specific time and place, and which can be planned for in a computer program. In a computer graphical user interface (GUI), an event can be as simple as a mouse click or keystroke.*

Thread

WikipediaWS (2008): *A thread in computer science is short for a thread of execution. Threads are a way for a program to fork (or split) itself into two or more simultaneously (or pseudo-simultaneously) running tasks. Threads and processes differ from one operating system to another but, in general, a thread is contained inside a process and different threads of the same process share some resources while different processes do not. Multiple threads can be executed in parallel on many computer systems. This multithreading generally occurs by time-division multiplexing ("time slicing") in very much the same way as the parallel execution of multiple tasks (computer multitasking): The processor switches between different threads. This context switching can happen so fast as to give the illusion of simultaneity to an end user. On a multiprocessor or multi-core system, threading can be achieved via multiprocessing, wherein different threads and processes can run literally simultaneously on different processors or cores. Many modern operating systems directly support both time-sliced and multiprocessor threading with a process scheduler. The operating system kernel allows programmers to manipulate threads via the system call interface. Some implementations are called a kernel thread, whereas a lightweight process is a specific type of kernel thread that shares the same state and information. Absent that, programs can still implement threading by using timers, signals, or other methods to interrupt their own execution and hence perform a sort of ad hoc time-slicing. These are sometimes called user-space threads.*

Tracing, trace

WikipediaWS (2008): *In software engineering, tracing is a specialized use of logging to record information about a program's execution. This information is typically used by programmers for debugging purposes, and additionally, depending on the type and detail of information contained in a trace log, by experienced system administrators or technical support personnel to diagnose common problems with software. Tracing is a cross-cutting concern. There is not always a clear distinction between tracing and other forms of logging, except that the term tracing is almost never applied to logging that is a functional requirement of a program.*

(From this document): a trace is a chronological suite of records resulting from observations that were made on a running system at different instants “t_n”, during a specific period “delta-t”. Traces are ordered lists of records specifying instructions that were executed on one or many local/distributed single-core/multi-core CPUs with respect to time.

Virtual Machine (VM), system virtual machine, process virtual machine

WikipediaWS (2008): *In computer science, a virtual machine (VM) is a software implementation of a machine (computer) that executes programs like a real machine. A virtual machine was originally defined by Popek and Goldberg as an efficient, isolated duplicate of a real machine. Current use includes virtual machines which have no direct* *correspondence to any real hardware.[1] Example: A program written in Java would receive services from the Java Runtime Environment software by issuing commands from which the expected result is returned by the Java software. By providing these services to the program, the Java software is acting as a "virtual machine," taking the place of the operating system or hardware for which the program would ordinarily have had to have been specifically written. Virtual machines are separated in two major categories, based on their use and degree of correspondence to any real machine. A system virtual machine provides a complete system platform which supports the execution of a complete operating system (OS). In contrast, a process virtual machine is designed to run a single program, which means that it supports a single process. An essential characteristic of a virtual machine is that the software running inside is limited to the resources and abstractions provided by the virtual machine -- it cannot break out of its virtual world.*

Virtual memory

WikipediaWS (2008): *Virtual memory is a computer system technique which gives an application program the impression that it has contiguous working memory, while in fact it is physically fragmented and may even overflow on to disk storage. Systems which use this technique make programming of large applications easier and use real physical memory (e.g. RAM) more efficiently than those without virtual memory. Note that "virtual memory" is not just "using disk space to extend physical memory size". Extending memory is a normal consequence of using virtual memory techniques, but can be done by other means such as overlays or swapping programs and their data completely out to disk while they are inactive. The definition of "virtual memory" is based on tricking programs into thinking they are using large blocks of contiguous addresses. All modern general-purpose computer operating systems use virtual memory techniques for ordinary applications, such as word processors, spreadsheets, multimedia players, accounting, etc.*

This page intentionally left blank.

Distribution list

Document No.: DRDC Valcartier TM 2008-144

	LIST PART 1: Internal Distribution by Centre
3	Document library
1	Guy Turcotte
1	Jean-Claude St-Jacques
1	Robert Charpentier
1	Mario Couture
1	Frédéric Michaud
1	Frédéric Painchaud
1	Nawel Chefai
1	Philippe Charland
1	Martin Salois
12	TOTAL LIST PART 1
	LIST PART 2: External Distribution by DRDKIM
1	DRDKIM
1	Library and Archives Canada
1	Dr Julie Lefebvre, DRDC Ottawa, 3701 Carling Avenue, Ottawa, Ontario, K1A 0Z4
1	Chris McMillan, DRDC Ottawa, 305 Rideau Street, Ottawa, Ontario, K1A 0K2
1	Paul Béland, DRDC Ottawa, 305 Rideau Street, Ottawa, Ontario, K1A 0K2
1	LCol. J.M. Drapeau, CFNOC, NDHQ 101 Col By dr. Ottawa K1A 0K2
1	LCol. Peter Scott, DRDC, 305 Rideau Ottawa K1A 0K2
1	Paul Lamoureux, DRDC Ottawa, NDHQ 101 Col By dr. Ottawa K1A 0K2
	Dominique Toupin, Ericsson (*)
	Dr. Michel Dagenais, Polytechnique de Mtl (*)
	Dr. Béchir Ktari, Université Laval (*)
	François Chouinard, Ericsson (*)
	Mathieu Desnoyers, Polytechnique de Mtl (*)
	Gabriel Matni, Polytechnique de Mtl (*)
	François Prenoveau, Polytechnique de Mtl (*)
	Dr. Timothy C. Lethbridge, University of Ottawa (*)
	Dr. Alex Navarre, NSERC (*)
	Dr. Robert Roy, Polytechnique de Mtl (*)
	Dr. Marc Khouzam, Ericsson (*)
	Pierre-Marc Fournier, Polytechnique de Mtl (*)
	(*) These copies were already distributed.
8	TOTAL LIST PART 2
20	**TOTAL COPIES REQUIRED**

This page intentionally left blank.

DOCUMENT CONTROL DATA

(Security classification of title, body of abstract and indexing annotation must be entered when the overall document is classified)

1. ORIGINATOR (The name and address of the organization preparing the document. Organizations for whom the document was prepared, e.g. Centre sponsoring a contractor's report, or tasking agency, are entered in section 8.) Defence R&D Canada – Valcartier 2459 Pie-XI Blvd North Quebec (Quebec) G3J 1X5 Canada	2. SECURITY CLASSIFICATION (Overall security classification of the document including special warning terms if applicable.) UNCLASSIFIED

3. TITLE (The complete document title as indicated on the title page. Its classification should be indicated by the appropriate abbreviation (S, C or U) in parentheses after the title.)

Monitoring and tracing of critical software systems: State of the work and project definition

4. AUTHORS (last name, followed by initials – ranks, titles, etc. not to be used)

M. Couture, M. Dagenais, D. Toupin, R. Charpentier, G. Matni, M. desnoyers, P.M. Fournier

5. DATE OF PUBLICATION (Month and year of publication of document.) December 2008	6a. NO. OF PAGES (Total containing information, including Annexes, Appendices, etc.) 86	6b. NO. OF REFS (Total cited in document.) 43

7. DESCRIPTIVE NOTES (The category of the document, e.g. technical report, technical note or memorandum. If appropriate, enter the type of report, e.g. interim, progress, summary, annual or final. Give the inclusive dates when a specific reporting period is covered.)

Technical Memorandum

8. SPONSORING ACTIVITY (The name of the department project office or laboratory sponsoring the research and development – include address.)

Defence R&D Canada – Valcartier
2459 Pie-XI Blvd North
Quebec (Quebec)
G3J 1X5 Canada

9a. PROJECT OR GRANT NO. (If appropriate, the applicable research and development project or grant number under which the document was written. Please specify whether project or grant.) 15bz10	9b. CONTRACT NO. (If appropriate, the applicable number under which the document was written.)
10a. ORIGINATOR'S DOCUMENT NUMBER (The official document number by which the document is identified by the originating activity. This number must be unique to this document.) DRDC Valcartier TM 2008-144	10b. OTHER DOCUMENT NO(s). (Any other numbers which may be assigned this document either by the originator or by the sponsor.)

11. DOCUMENT AVAILABILITY (Any limitations on further dissemination of the document, other than those imposed by security classification.)

Unlimited

12. DOCUMENT ANNOUNCEMENT (Any limitation to the bibliographic announcement of this document. This will normally correspond to the Document Availability (11). However, where further distribution (beyond the audience specified in (11) is possible, a wider announcement audience may be selected.))

Unlimited

13. ABSTRACT (A brief and factual summary of the document. It may also appear elsewhere in the body of the document itself. It is highly desirable that the abstract of classified documents be unclassified. Each paragraph of the abstract shall begin with an indication of the security classification of the information in the paragraph (unless the document itself is unclassified) represented as (S), (C), (R), or (U). It is not necessary to include here abstracts in both official languages unless the text is bilingual.)

This document presents a summary of the results of the tutorial/workshop “Monitoring Distributed Multi-core IT Systems for Optimization and Security”, which was held in January ’08 at Ericsson’s Montreal office. The aim of this event was to identify important problems and challenges that are experienced by researchers and industry in this domain, as well as potential avenues of research and development that may lead to concrete solutions for critical operations.

A brief review of the state of the art in this domain is presented as an introduction. The main results of the January ’08 tutorial/workshop are then presented. These results will eventually be used in the definition and set-up of R&D projects designed to put forward potential solutions for critical operations. The annexes contain complete information relative to the January ’08 event, a brief description of tools that are currently used in this domain, as well as a glossary.

Ce document présente un résumé des résultats qui ont été obtenus lors du tutoriel/atelier « Monitoring Distributed Multi-core IT Systems for Optimization and Security », lequel a eu lieu dans les bureaux d’Ericsson à Montréal en janvier 2008. L’activité visait à identifier les problèmes et défis importants que les chercheurs et l’industrie rencontrent dans ce domaine, ainsi que des pistes de recherche et de développement potentielles pouvant mener à des solutions concrètes pour les opérations critiques.

Une brève revue de l’état de la technologie de pointe dans ce domaine est d’abord présentée en introduction. Les résultats principaux de l’activité de janvier 2008 sont ensuite présentés. Ces résultats seront éventuellement utilisés lors de la définition et de la mise en œuvre de projets de R et D visant à pousser plus loin des solutions potentielles pour les opérations critiques. Le lecteur trouvera dans les annexes l’information complète relative à l’activité de janvier 2008, une brève description des outils utilisés dans ce domaine ainsi qu’un glossaire des termes utilisés.

14. KEYWORDS, DESCRIPTORS or IDENTIFIERS (Technically meaningful terms or short phrases that characterize a document and could be helpful in cataloguing the document. They should be selected so that no security classification is required. Identifiers, such as equipment model designation, trade name, military project code name, geographic location may also be included. If possible keywords should be selected from a published thesaurus, e.g. Thesaurus of Engineering and Scientific Terms (TEST) and that thesaurus identified. If it is not possible to select indexing terms which are Unclassified, the classification of each should be indicated as with the title.)

Monitoring, tracing, multi-cores systems, distributed systems, information systems, project definition.