Open Systems Dependability Core
– DS-Bench: Dependability System Benchmarks –

オープン・システム・ディペンダビリティ・コア
– ディペンダビリティ システム ベンチマーク –

DEOS Core Team

September 2009

科学技術振興機構
Japan Science and Technology Agency

Contents

List of Figures

List of Tables

Abstract

This document is an interim report about the dependability benchmark runtime environment being developed at the DEOS project.

本書は、DEOS プロジェクトで開発中のディペンダビリティベンチマーク実行環境についての中間報告書である。

DEOS Core Team:

Yutaka Ishikawa	University of Tokyo
Hajime Fujita	University of Tokyo
Shinpei Kato	University of Tokyo
Motohiko Matsuda	University of Tokyo
Midori Sugaya	JST
Toshihiro Hanawa	University of Tsukuba
Shinichi Miura	University of Tsukuba
Yuki Kinebuchi	Waseda University
Jin Nakazawa	Keio University
Yoichi Ishiwata	AIST
Yutaka Matsuno	AIST
Hiroki Takamura	AIST
Hiroshi Yamada	Keio University

DEOS コアチーム:

石川 裕	東京大学
藤田 肇	東京大学
加藤 真平	東京大学
松田 元彦	東京大学
菅谷 みどり	科学技術振興機構
塙 敏博	筑波大学
三浦 信一	筑波大学
杵渕 雄樹	早稲田大学
中澤 仁	慶應義塾大学
石綿 陽一	産業総合技術研究所
松野 裕	産業総合技術研究所
高村 博紀	産業総合技術研究所
山田 浩史	慶應義塾大学

Chapter 1

Introduction

Dependability has been recognized as an integrating concept that encompasses availability, reliability, safety, integrity, and maintainability [1]. A dependability benchmark is a technique used to evaluate the dependability of a system. It characterizes the dependability of a system component or a whole system, either qualitatively or quantitatively[2]. The dependability benchmark framework, a conceptual framework, was developed under the DBench project[3] from late 1990 to early 2000. In this framework, the benchmark environment consists of *Benchmark Target*, *Workload*, *Faultload*, and *Measurement*.

Dependability benchmarks are crucial for both designers and users of dependable systems. Users may use dependability benchmarks to define their dependability requirements, both qualitatively and quantitatively, and thus those benchmarks are used at procurement. System designers use dependability benchmarks to define the specifications and to test the parts of the system, as well as test the whole system. In the development phase, a dependability benchmark may reveal a weakness in the system.

ディペンダビリティは、可用性、信頼性、安全性、保全性、保守性を包含する統合コンセプトとして認識されてきた [1]。システムが持つディペンダビリティを評価する技術としてディペンダビリティ ベンチマークがある。ディペンダビリティ ベンチマークは、システム構成要素あるいはシステム全体で、故障時における性質を特徴づける [2]。1990 年代後半から 2000 年代前半に進められた DBench プロジェクトにおいて、ディペンダビリティベンチマークに対する概念レベルフレームワークが定義された [3]。本フレームワークでは、ベンチマーク環境は、ベンチマーク対象、ワークロード、故障ロード、計測ツールから構成される。

ディペンダビリティベンチマークは、ディペンダブルシステムの設計者、ユーザの双方に重要である。ユーザは、あるディペンダビリティを必要とするシステムを導入する時、ディペンダビリティベンチマークを用いて、システムに求められるディペンダビリティを定量的定性的に定義することが可能となり、調達仕様として利用できる。システム設計者は、仕様決め、部品テスト、製品テストにディペンダビリティベンチマークを使用する。開発フェーズでは、ディペンダビリティベンチマークは、システムの弱点を明らかにするであろう。

There are three issues we must concerned with the traditional dependability benchmark framework, DBench: 1) The framework does assume various hardware, software, and operator faults under a well defined closed environment, but does not assume an open system environment in which the system is connected to the Internet and any user may access the system. In such an environment, network attacks and unexpected operations such as user-caused overload (too many user requests), must be considered.

2) The framework does not consider performance under an anomaly situation with energy consumption as the dependability measurement. In mobile equipment powered by battery, the quality of service and performance must be controlled to reduce the energy consumption of the battery. Thus, a dependability benchmark should include evaluation of performance and energy consumption under both normal and abnormal situations.

3) There are no systematic dependability benchmark runtime environments currently available. DBench defines a conceptual dependability benchmark framework, but does not provide a software stack to support the framework. Thus, dependability benchmarks based on the DBench framework have been implemented independently. They develop fault loads and measurement tools that do not share any common evaluation environment.

This document defines a new dependability benchmark runtime environment, called DS-Bench for short, that overcomes the above issues. Unlike DBench, it defines a common benchmark runtime environment to achieve a benchmark software suite.

In this document, the concept of DS-Bench is firstly introduced in the following chapter. Then, the DS-Bench runtime environment is designed in Chapter 3.

DBenchに代表される従来のディペンダビリティベンチマークフレームワークには、以下の3つの問題点がある。第一に、フレームワークは、閉じられた系におけるハードウェア故障、ソフトウェア故障、オペレータの過ちだけを想定しているが、ネットワークに繋がり様々なユーザがアクセスするオープンな環境を想定していない。ネットワークにつながっている機器では、脅威やユーザ要求の過負荷などの耐性に関しても考慮する必要がある。

第二に、異常時のシステム計測として、エネルギー消費量と性能を考慮していない。バッテリ駆動型携帯機器では、バッテリ残量に応じてエネルギー消費量を抑えて、サービスの質や性能を制御する必要が生じる。また、システム導入前に、耐故障のためのエネルギー消費量増加を知ることは重要である。このように、正常時、異常時の双方で、エネルギー消費量と性能を考慮したベンチマークが必要である。

第三に、系統的ディペンダビリティベンチマーク実行環境が存在しない。DBenchでは、概念レベルでのフレームワークが定義されたが、フレームワークとしてのソフトウェアツールを提供していない。このため、DBenchに基づくディペンダビリティベンチマークは、個々のベンチマーク利用者が必要とする故障および計測ツールとして策定されている。故障、計測を系統だって作られてはいない。また、これらベンチマークは共通実行環境を有していない。

本ドキュメントは、このような従来の問題を解決するための新しいディペンダビリティベンチマーク実行環境(略称DS-Bench)を定義する。DBenchとは異なり、概念レベルのフレームワークだけでなく、ベンチマークソフトウェアを実現するための共通実行環境を提供する。

本ドキュメントでは、次章で、DS-Benchのコンセプトを紹介し、第3章においてDS-Benchフレームワークを紹介する。

Chapter 2

Basic Concept

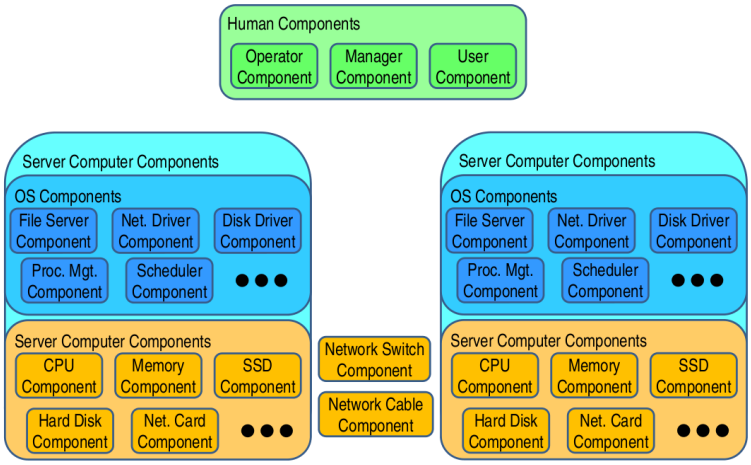

Figure 2.1: An Example of Components

2.1 System and Components

First, the terms used in this document are defined as follows: A part of the system is called a *component*. A component is either hardware, software, or a human who operates or uses the system. A system consists of many components, each of which may consist of other components. The system itself may be a component of another system. Thus, components are hierarchically defined.

まず、本稿で扱うシステムについて定義する。システムの構成要素をコンポーネントと呼ぶことにする。コンポーネントは、ハードウェア、ソフトウェアに加えて、システムを操作あるいは利用する人もコンポーネントとして扱う。コンポーネントは、複数のコンポーネントから構成され得る。システムは多数のコンポーネントから構成され、システム自身もコンポーネントとして扱われる。コンポーネントは階層構造化されて扱われる。

For example, as shown in Figure 2.1, a reliable file server may consist of the following components: A server machine comprising a hardware component which consists of a CPU, memory, a RAID disk, and network equipment components. A file server may consist of server machines. The file server software and the operating system running on those server machines are software components. The human components are the operators and users of the server.

例えば、図 2.1 に示す通り、ある信頼性のあるファイルサーバは次のようなコンポーネントから構成されるであろう。ハードウェアコンポーネントはサーバマシンであるが、これはさらに、CPU、メモリ、RAID ディスク、ネットワーク機器コンポーネントから構成される。ファイルサーバでは、複数のサーバマシンから構成されるだろう。各サーバマシン上で稼働しているオペレーティングシステム、ファイルサーバソフトウェアなどがソフトウェアコンポーネントである。そして、ファイルサーバを管理するオペレータや管理者、ユーザが人的コンポーネントとなる。

2.2 Components and Anomaly

A fault may occur in a component, and this fault may propagate another malfunction in another component. In the paper[1], the chain of fault, error, and failure is modeled. An example of such a chain in the file server example of Figure 2.1 is that a fault in a hard disk causes an error in the file system, and causes an application to fail. A fault does not always stop a service, but sometimes degrades the quality of the service. For example, a fault in a disk causes degradation of access performance due to retries of the disk access request.

故障は、コンポーネントの一つあるいは複数で発生し伝搬する。論文 [1] では、故障 (fault)、エラー (error)、障害 (failure) の連鎖としてモデル化している。例えば、図 2.1 のファイルサーバの例では、ディスクで生じた故障は、ファイルシステムエラーを起こし、ファイルを利用しているアプリケーションの障害へと連鎖する。故障は必ずしもサービスの停止とは限らない。例えば、ディスク故障を起こしていてもディスクの耐故障機能により（例えばリトライによってデータが読み書き可能な場合)、故障は性能低下を引き起こすかもれない。

There can be many sources of a fault in a software component. Software and some databases may be altered by attack from the Internet. Humans may make a mistake in a system operation, maliciously or non-maliciously. A user may also perform an unexpected operation. In this document, instead of using the term *fault*, *anomaly* is used to indicate a fault of in a hardware component, software bugs, and malicious/non-malicious operations.

ソフトウェアコンポーネントの故障は、従来のハードウェア故障と同じようには扱えない。インターネット脅威により、ソフトウェアやデータベースが改竄される。また、人コンポーネントは、不作為あるいは作為による誤操作あるいは想定外の操作を行うこともある。本稿では、このような状況を故障という用語の替わりに、Anomaly(日本語では異常と訳すことにする) という用語を用いる。

An anomaly can happen in a component or several components independently. An anomaly can cause an error in another component, and that error can lead to a failure. For example, when a human component makes a mistake in an operation involving a database and causes an inconsistency in the database, the database reports the inconsistency status to the application software, and the result may cause the system to shut down.

コンポーネント単位で異常は発生する。同時に複数のコンポーネントが異常を発生する可能性もある。異常は他のコンポーネントのエラーを誘因し、エラーは障害を誘因する。例えば、人コンポーネントの誤操作によりデータベースの一貫性が失われ、データベースソフトウェアがアプリケーションソフトウェアにエラーを報告し、アプリケーションは動作しなくなる、ということが生じる。

2.3 Components and Design Requirements

A dependable system must maintain the behavior required at the design phase, even though an anomaly occurs in some component of the system. The quantitative requirements of a dependable system are measured by five metrics: i) the ratio of anomalies/failures, ii) the detection time of an anomaly, iii) the failure recovery time, iv) performance metrics, and v) energy consumption. The requirements of the anomaly/failure ratio can be defined for hardware components, each of which has such a ratio.

ディペンダブルシステムにおいては、コンポーネントに異常が生じても、設計時の要求どおりにシステムが振る舞われなければならない。システムのディペンダビリティにおける定量的要求には、異常・障害率、異常検知時間、障害回復、性能、消費電力、の5つの側面があり、コンポーネント毎に定義される。異常・障害率は、ハードウェアコンポーネントの故障のように、あらかじめ、故障率が定義できるようなコンポーネントにおいて使用される要件である。

The detection time of an anomaly is the time between the occurrence of an anomaly and the report of that anomaly. The detection time requirement may be defined per component. The failure recovery time is defined by the sum of the detection time of a particular anomaly and the recovery time of the failure caused by that anomaly. For example, a fault (anomaly) in a disk is detected by the operating system, and the fault is reported to the file server software. The recovery time against the disk fault is the sum of the report time from the operating system and the time to it takes to carry out the recovery operation.

異常検知時間は、異常が生じてから、その異常が認識されるまでの時間であり、コンポーネント毎に要求されるものである。障害回復時間は、異常検知時間と障害回復にかかる時間の和で求まる。例えば、ハードディスクの故障(異常)が、オペレーティングシステムコンポーネントで検知され、ファイルサーバソフトウェアに故障が報告される。ファイルサーバソフトウェアの障害回復時間は、見積もるためには、オペレーティングシステムの故障検知時間が必要とされる。

Performance and energy consumption are important requirements under both normal and abnormal conditions. Performance and energy consumption are sometimes conflicting requirements because more performance requires more energy consumption. A fault tolerant server is usually obtained by introducing redundancy of hardware that consumes more energy.

性能、消費電力は、正常下においても重要な要件である。性能と消費電力は相反する要求であることが多い。なぜなら、性能を上げるためには消費電力が多くなるからである。耐故障サーバは冗長ハードウェアによって実現される。冗長ハードウェアは、より多くの電力を消費する。

Table 2.1: Dependability Requirements against Anomaly Source

Requirements	Anomaly Source								
	Hardware			Software			Human		
	CPU	Memory	···	Disk driver	Server	···	Operator	User	···
Detection Time									
Recovery Time									
Performance									
Energy									

2.4 DS Benchmark

A dependable system must meet the dependability requirements under anomaly conditions that were anticipated at the system development phase. A runtime environment can be developed to test whether or not the dependability requirements are satisfied under the anomaly conditions. We call this *dependable system benchmark runtime environment*, DS-Bench for short.

システム開発時には、想定可能な異常状態を列挙し、異常状態下においてもシステムの要求事項が満たされているか確認しなければならない。ディペンダビリティを阻害する列挙可能な異常状態に対して、システムが設計時要求通りの振る舞いをしているかを計測する実行環境をディペンダブルシステムベンチマーク実行環境 (Dependable System Benchmark Runtime Environment)、略称、DS-Bench、と呼ぶことにする。

The test coverage of DS-Bench is summarized by two dimensional matrices, i.e., anomaly sources and requirements as shown in Table 2.1. Anomaly sources include hardware faults, anomalies in software components, and human-caused faults. The requirements include the detection time of an anomaly, the recovery time of an anomaly, performance under the anomaly, and energy consumption under the anomaly. The requirements are not only for the whole system, but also for each component. For example, a requirement for the OS component is the required detection time for detection of a disk fault.

表 2.1 に示す通り、DS-Bench が扱う内容は、異常発生源と異常下における要求事項の 2 次元マトリクスとしてまとめることができる。異常には、ハードウェア故障、ソフトウェアコンポーネントの異常、人的操作異常などがある。要求事項には、異常検知時間、異常回復時間、異常下における性能、異常下における消費電力がある。要求事項は、システム全体だけでなく、コンポーネントに対する要求事項であっても良い。例えば、ハードディスクが故障した時にオペレーティングシステムが故障を検知する時間など。

2.4.1 Advantages

DS-Bench has the following advantages:

Developer:

The developer may use DS-Bench during the product life cycle. In the specification and design phases, the dependability requirements of components are defined by the anticipated results under certain conditions obtained by the execution of DS-Bench for the system. In the implementation and test phases, DS-Bench checks whether or not the requirements are satisfied. In other words, DS-Bench provided evidence of the existence of a dependability characteristic for the system. It also reveals the weakness of components and the whole system. The results of DS-Bench execution are stored in a database so that the database can be used during the operation phase.

User:

The user may define the dependability requirements of the system using DS-Bench at system procurement. The results obtained from DS-Bench are used as a part of the specifications. DS-Bench is also used to test candidate systems and provide useful comparisons for making procurement decisions.

Operation Phase:

When an anomaly condition occurs during the operation phase, the database produced by DS-Bench is queried, especially if the same anomaly condition has already been encountered or tested for. For example, consider a situation where the performance of a stream server is degraded. The same case is extracted from the result database of DS-Bench, and the operator can confirm whether or not the same anomaly has happened.

DS-Bench は、次のような利点がある。

開発者:

開発者は、製品のライフサイクルに応じて DS-Bench を使用することが可能である。システムの仕様・設計フェーズでは、システム構成要素のディペンダビリティ要求事項をベンチマークの予測結果を用いて定義できる。そして、実装・テスト時には、ベンチマークによって、設計時要求が満たされているかどうかを確認できる。言い換えれば、DS-Bench によってシステムのディペンダビリティ性質の証拠を示すことになる。ベンチマークの実行結果をデータベース化することにより、運用時の障害に備える。

ユーザ：

ユーザは、あるディペンダビリティを必要とするシステムを導入する時、ディペンダビリティベンチマークを用いて、システムに求められるディペンダビリティを明確化するとともに、DS-Bench の実行結果を調達仕様として利用することが可能となる。システム検討時、複数の候補から一つのシステムを選択するとき、必要とされるディペンダビリティが実現されているかどうかを検査するために DS-Bench が利用される。

運用時

運用時に障害が発生した場合、DS-Bench の実行結果データベースに同じ障害が発生していれば、障害の原因を突きとめやすくなる。例えば、ストリーム配信サーバの例において、応答性能が低下するということが生じたとする。DS-Bench の実行結果から、応答性能が低下する事例を取り出し、それら異常原因が実機上で発生していないかを確認することが可能となる。

2.4.2 Limitations

DS-Bench is only effective for anticipated anomaly conditions. It does not reveal weakness of the system for unexpected anomaly conditions, and it does not find out the source of such conditions. For example, again consider the condition that the performance of a stream server is degraded. If the anomaly is unknown, it is difficult to find out what really happened.

DS-Benchは、想定される異常に対して有効であるが、想定されない、あるいは未知の異常に対しては無力である。先のストリーム配信サーバ例において、原因不明の応答性能低下が生じても、DS-Benchで想定した異常以外の未知の異常原因の場合には、原因を特定するのに多くの時間を要する。

In such a case, after finding the source of the anomaly, a program to generate the anomaly, called anomaly load, must be developed if no such program exists. The developed program is registered to DS-Bench so that the capability of dependability checks is increased in the system. This incremental benchmark improvement leads to the improvement of overall system dependability.

このような場合、原因が特定された後、その原因を人工的に生成するプログラムを開発し、DS-Benchに登録する。これによりディペンダビリティ検査項目が増加する。このインクリメンタルなベンチマーク改良はシステムのディペンダビリティ向上につながる。

It should be noted that support for finding out unexpected/unknown anomalies is being discussed in the DEOS project.

なお、不明な原因を特定するための支援については、DEOSプロジェクトにおいて研究開発を行っているところである。

Chapter 3

DS-Bench Runtime Environment

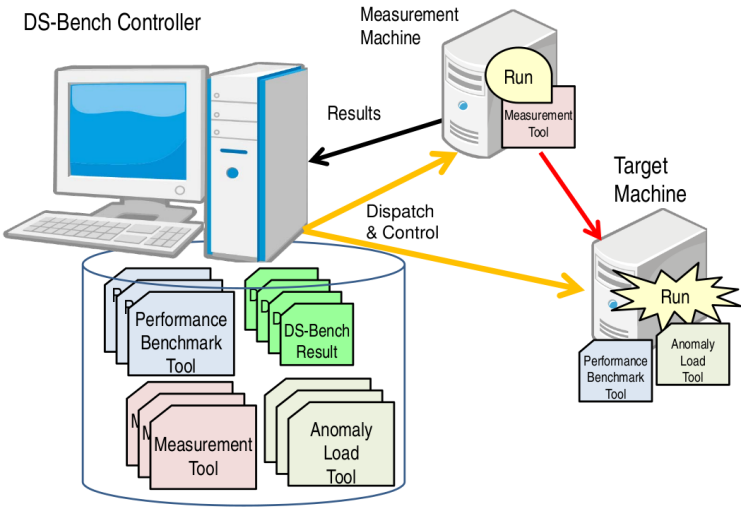

Figure 3.1: An Example of Components

3.1 Overview

The system dependability requirements depend on the application domain. In the factory automation and robot domains, realtime operation and safety of the system are extremely important, while in the computer server domain, availability, responsibility, and low power consumption are the main concerns. Thus, dependability benchmarks are provided based on the relevant application domain. On the other hand, some benchmarks, such as CPU performance and power consumption, are common to all application domains.

システムソフトウェアのディペンダビリティに関する要件はアプリケーション分野毎に異なる。FA・ロボット分野では、実時間性、制御対象の安全性が重要であり、サーバ分野では、耐故障性、応答性能、省電力が重要となる。したがって、ディペンダビリティベンチマークは、アプリケーション分野毎に整備されるべきである。一方で、CPU 性能と消費電力のような、どの分野にも共通するベンチマークも存在する。

The DS-Bench is a runtime environment designed to handle all dependability benchmarks. As shown in Figure 3.1, the DS-Bench environment consists of three computers: the controller, the measurement machine, and the target machine.

DS-Bench は、ディペンダビリティベンチマーク群を統一的に扱うための実行環境である。図 3.1 に示す通り、DS-Bench 実行環境は、コントローラ、計測マシン、ターゲットマシンの 3 種類のコンピュータから構成される。

The controller contains the performance benchmark tools, measurement tools, anomaly load tools, and the database used to archive the results of the benchmarks. Examples of performance benchmark tools are SPEC CPU, SPEC WEB, and other existing performance benchmarks. The measurement tools comprise, for example, a power consumption instrument, and logging tools for memory usage and CPU usage, respectively. The anomaly load tools include tools for injecting hardware error and injecting overloaded situations.

コントローラは、Performance Benchmark ツール、Measurement ツール、Anomaly Load ツール、ベンチマーク結果を保存するデータベースから構成される。Performance Benchmark ツールは、SPEC CPU、SPEC WEB など従来のベンチマーク群である。Measurement ツールは、電力測定やメモリ利用率、CPU 利用率などをログするツール群である。Anomaly Load ツールは、ハードウェア故障や過負荷状態などを人工的に生成するツール群である。

The controller manages the measurement and target machines so that it distributes the tools listed above to the target machines, if needed, and runs the benchmark. In some benchmarks, no measurement machine is required, while one measurement machine may be required in other benchmarks, and several such machines are required in yet other benchmarks. Results of the benchmarks are sent to the controller and they are archived in the database.

コントローラは、計測マシン、ターゲットマシンを管理し、必要に応じて上述したツール群を計測マシン、ターゲットマシンに配布し実行する。ベンチマークによって、計測マシンが必要ない場合があるし、複数台必要になる場合もある。また、ターゲットマシンは一台ではなく複数台の場合もある。ベンチマーク結果は、コントローラに送られ、データベース化される。

On the benchmark target machine, certain aspects of the targeted component are measured. An anomaly load creates a certain anomaly situation for the target component. For example, in order to test the component behavior under a disk access overload situation, the bonie++, open source benchmark, is run. The bonie++ program is a benchmark program, but it is also used to yield a disk access load.

ベンチマークターゲットマシン上で、ベンチマークターゲットコンポーネントの振る舞いを計測する。Anomaly Load は、ターゲットコンポーネントに対する Anomaly を生成する。例えば、ディスクアクセス過負荷状態において、アプリケーションが正常に動作するかどうかを検査するために、オープンソースである bonie++ ベンチマークを実行させる。ベンチマークプログラムである bonie++を動かすことによってディスクアクセス負荷を生成する。

The specifications of these tools are defined in this Chapter.

本章では、これらツール群の仕様を定義する。

Table 3.1: Anomaly Sources

Component	Anomaly	Tool
CPU, Memory Disk, NIC, ...	fault	VM, virtual BMC
Network cable and switch	disconnect, packet loss	switch capability, NIST net
Power supply	power off	special tap
Computer	down	VM or a special equipment
File System	create/delete/write/read error	*Pbus-al-fs*
Process mgmt	no process creation	*Pbus-al-procs*
Memory mgmt	no memory allocation	*Pbus-al-memalloc*
Application	overloaded file access	bonie++
Application	overloaded process execution	hackbench
Application	overloaded process creation	hackbench
Application	huge memory allocation	*DS-memalloc*
Application	overloaded connection requests overloaded send/receive requests	*DS-tcpip*

3.2 Anomaly Load

Table 3.1 summarizes the anomaly load injection methods for target components provided by DS-Bench. The injection of an anomaly into hardware components, such as CPUs, memory modules, disks, and network interface cards, is accomplished by the VM (Virtual Machine) monitor. The injection of an anomaly into a network, such as the disconnection of a network cable or a network switch fault, is simulated by disabling a network port of the switch. Since a power tap with a remote controller can be switched on/off remotely, an anomaly of involving the shutting down of a computer is accomplished by utilizing such an equipment, as well as by using the VM monitor.

If the benchmark target is a PC server with BMC (Baseboard Management Control) equipment and its operating system provides dependability using the BMC equipment, such anomaly loads in the CPU board can be simulated by a virtual BMC tool. Note that the BMC equipment monitors memory fault, CPU FAN fault, and so on, and alerts are sent to the operating system when a fault occurs[4].

DS-Benchが提供するコンポーネントの異常の発生方法を表3.1に示す。計算機のハードウェアコンポーネントであるCPU、メモリ、ディスク、ネットワークインターフェイスカードの異常発生は、VM (Virtual Monitor) モニタ上で提供する。ネットワークケーブルの断線やネットワークスイッチの故障は、ネットワークスイッチのポートの無効化で模擬する。電源断は、リモート電源ON/OFF機能のある電源タップにより実現可能である。計算機の停止は、VMあるいは、電源断で対応する。

ターゲット環境がPCサーバで、BMC (Baseboard Management Control) を利用したOSのディペンダビリティ機能を検査する場合、仮想BMCツールにより故障を模擬することができる。なお、BMCは、CPUボード上のMemory、温度、ファンなどの故障をロギングするとともにOSに通知する[4]。

Anomaly loads in the operating system, such as a file system, process or memory management anomaly, are implemented using the P-Bus infrastructure[5] that supports OS extensions developed by the DEOS project.

ファイルシステム、プロセス管理、メモリ管理などの OS の異常は、それぞれ、DEOS で開発される P-Bus OS 拡張基盤 [5] を用いて実装される。

Application-level anomaly loads are produced as follows: Some performance benchmark programs are used to generate overload situations, such as the bonie++ benchmark program for file systems and the hackbench program for processes. The DS-memalloc tool is provided to reveal the target's behavior at the time requests for a huge memory allocation. In order to discover the behavior of the target at an overload-level request involving network connections and network traffic, the DS-tcpip tool is provided.

アプリケーションレベルの異常生成は以下のように実現される。性能パフォーマンスベンチマークプログラムは過負荷状態を生成するために使用する。ファイルシステムに対する過負荷アクセスのために bonie++を使用する。プロセスを大量に生成する目的で hackbench を使用する。大量メモリを割り当てた時の振る舞いを確認するために、DS-memalloc ツールを提供する。TCP/IP 処理を確認するために、過負荷コネクション要求および大量データの送受信を行うツール DS-tcpip を提供する。

3.3 Measurements

The following measurement tools are provided to discover system behavior vis-à-vis the anomaly loads. Note that the measurement tools for performance will be given in the next section.

Anomaly Load に対して、システムがどのような振る舞いをするかを計測するためのツール群を提供する。なお、性能については、次節で説明する。

Energy Consumption:
The DS-Bench assumes that some equipment is available to measure energy consumption.

エネルギー消費量:
エネルギー消費量を計測できる機材を想定する。

System down ratio and time:
The DS-Bench assumes that the benchmark target machine is connected to a network so that the system availability is monitored remotely by the `ping` command. The `ping` command uses the ECHO packet of the IP protocol. Thus, the accuracy of the system down detection time depends on the round trip time of the ECHO packet in the network. Such round trip times involve measurement in the milli second order. If more accuracy is required, some special purpose hardware must be installed.

システムダウン率、システムダウンタイム:
DS-Bench は、ネットワーク接続を想定している。OS の動作確認は、ping (IP の ECHO パケット) に反応するかで確認する。精度は、ECHO パケットの往復遅延となる。往復遅延時間は、CPU およびネットワーク性能に依存するが、概ねミリ秒である。それ以下の精度を必要とする場合は、特別なハードウェアを実装すべきである。

Delay time and ratio for error reporting:
This delay time is recorded in the logging data. Each component must record the reporting time in the logging data. If no information is available, the ratio of error reporting is 0 %.

エラー報告遅延時間、エラー報告率:
コンポーネント毎に定義されているエラー報告のログギングデータから求められる。エラー報告のロギング機能を有しないコンポーネントは、エラー報告率は 0 %となる。

Table 3.2: Performance Benchmarks for OS

Component	Performance Benchmark	Component	Performance Benchmark
CPU	SPEC CPU	CPU scheduler	*DS-realtime-bench*
OS	LMBench	TCP/IP	iperf
File System	bonie++	Distributed File System	—

Table 3.3: Example of Performance Benchmarks for Applications

Application	Performance Benchmark	Application	Performance Benchmark
WEB server	SPEC Web	Streaming	—
Data base	OLTP	HPC	IMB

Delay time and ratio of error detection:
In order to obtain this delay time, the anomaly load tools must record the time when an anomaly load starts. The system also records the time when the source of the anomaly is detected. The delay time is given by the period between those events. If the system does not record the detection event within a certain period, the DS-Bench concludes that the system has not detected the error.

故障原因検出遅延時間、故障原因検出率:
Anomaly Load により異常を人工的に生成した時間をロギングするとともに、システムが故障原因を特定した時間もロギングする。2 つのイベントの発生時間の差が原因検出時間となる。故障をある一定時間検出出来なかったときは、その故障は検出されなかったとする。

3.4 Performance Benchmarks

As shown in Tables 3.2 and 3.3, the performance benchmarks are categorized with two benchmarks, i.e., one for OS function performance and one for application performance. It is not intended that all performance benchmarks will be developed for DS-Bench, but the existing performance benchmarks such as SPEC and OLTP are integrated.

表 3.2 および表 3.3 に示すとおり、性能ベンチマークは、OS 機能の性能およびアプリケーションの性能を計測する 2 つに分類される。DS-Benchmark では、性能ベンチマークを新たに開発するのではなく、SPEC や OLTP などの既存ベンチマークを統合して使用する。

```
 L M B E N C H  3 . 0   S U M M A R Y
                -----------------------------------
                (Alpha software, do not distribute)
Basic system parameters
----------------------------------------------------------------------------
Host                 OS Description              Mhz  tlb  cache  mem   scal
                                                     pages line   par   load
                                                           bytes
--------- ------------- ----------------------- ---- ----- ----- ------ ----
rokko     Linux 2.6.29-        x86_64-linux-gnu 2639     8   128 6.5800    1

Processor, Processes - times in microseconds - smaller is better
----------------------------------------------------------------------------
Host                 OS  Mhz null null      open slct sig  sig  fork exec sh
                             call  I/O stat clos TCP  inst hndl proc proc proc
--------- ------------- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ----
rokko     Linux 2.6.29- 2639 0.10 0.16 5.98 7.10      0.21 1.97 880. 2416 6331

Basic integer operations - times in nanoseconds - smaller is better
-------------------------------------------------------------------
Host                 OS  intgr intgr  intgr  intgr  intgr
                          bit   add    mul    div    mod
--------- ------------- ------ ------ ------ ------ ------
rokko     Linux 2.6.29- 0.4000 0.3600 0.1400 7.8700 7.7400
...
```

Figure 3.2: An example of the LMBench benchmark result

Since existing benchmarks have been independently developed, their execution methods and result formats differ. For example, the LMBench benchmark, which measures the performance of OS functions, generates the result shown in Figure 3.2. This result format is readable for humans, but it is not easy readable for software. Thus, we need to standardize a readable result format for software, and then design a translator to translate the results from each benchmark into the standardized format.

既存ベンチマーク群は、それぞれ独立して開発されているため、実行方法、実行結果のデータフォーマットは、まちまちである。例えば、OS 機能の性能を計測する LMBench の実行結果を図 3.2 に示す。人には見やすい形式で表示されるが、ソフトウェアが読みやすい形式ではない。それゆえ、ソフトウェアによって読みやすい形式を標準化し、各ベンチマークの実行結果を標準形式に変換する変換器を設計する。

DS-Bench provides a runtime environment in which several existing benchmarks are integrated. The result of each benchmark is converted to the standard format using the XML language. Since existing benchmarks have their own result formats, the translation instruction, transforming the results of each benchmark into the XML format, is also defined in the XML language. The DS-Bench provides the translator that takes the result of a benchmark and the translation instruction, and registers the translated results in the XML format in the database.

A part of the translation instruction is shown in Figure 3.3. It is assumed that the results of a benchmark contains several tables that include some headers, results, and/or separators between tables. Figure 3.3 shows an overview of the instruction extracting one table where the $< table >$ node defines one table in the result.

The DS-Bench runtime environment assumes that a table of the results starts the string matched by a regular expression defined in the $< begin >$ node and ends the string matched by a regular expression defined by the $< end >$ node. That is, the DS-Bench takes the result of the benchmark and searches the string matched by the regular expression defined in the $< begin >$ node, and then it searches the string matched by the regular expression defined in $< end >$ node. The strings between the two strings matched with the $< begin >$ and $< end >$ nodes are considered as the result data. The $< data >$ and $< valid >$ nodes define the rule to extract result values in the result data. An example of the definition will be given later.

If the $< begin >$ and $< end >$ nodes are omitted in the definition, the DS-Bench runtime environment searches for string matching one of the regular expressions defined by the $< caption >$, $< header >$, and $< data >$ nodes. After that, the strings appearing in the results of the benchmark are considered as the result data until the DS-Bench finds string matching one of the regular expressions defined by the $< caption >$, $< header >$, and $< data >$ nodes.

DS-Benchは、様々なベンチマークを容易に統合できるための環境を提供する。ベンチマークの実行結果をXMLフォーマットに変換する。既存ベンチマークは独自のフォーマットで実行結果を表示しているため、ベンチマーク毎にXMLフォーマットに変換する必要がある。そのための定義もXMLで記述する。DS-Benchは、あるベンチマーク結果とそれに対応する変換定義を受け取り、結果をXMLフォーマットに変換してデータベースに格納する。

図3.3に変換定義の一部を示す。ベンチマーク結果にはいくつもの表が含まれていて、それぞれの表は、見出し(header)、結果(data)、表間の区切りから構成されていると仮定している。図3.3は、一つのテーブルの定義の概要を示している。$< table >$ノードは、表形式の出力を処理するための定義である。

DS-Bench実効環境では、表は$< begin >$ノードで定義される正規表現と合致する文字列で始まり、$< end >$ノードで定義される正規表現と合致する文字列で終わると仮定している。すなわち、DS-Benchは、ベンチマーク結果を受け取り$< begin >$ノードで定義される正規表現と合致する文字列を探し、次に$< end >$ノードで定義される正規表現と合致する文字列を探す。$< begin >$ノードと$< end >$ノードで定義される正規表現と合致する文字列の間に現れる文字列が結果データとして処理される。結果データの文字列から値を取り出すための規則が$< data >$ノードおよび$< valid >$ノードで定義される。これらの定義例は後述する。

$< begin >$ノードと$< end >$ノードが省略された場合は、$< caption >$ノード、$< header >$ノード、$< data >$ノードで定義される正規表現と合致する文字列を探す。次にこれら正規表現と合致する文字列が見つかるまでの間の文字列が、結果データとして処理される。

```
<dsxml>
<benchmark>Performance Benchmark Name</benchmark>
<table>
  <caption>表題の形式（省略可、正規表現もしくは文字列指定）</caption>
    <header>見出しの形式（省略可、正規表現もしくは文字列指定）</header>
    <header2>DS-Bench 内での新しい見出し（省略可、文字列指定）</header2>
    <data>値の形式（省略不可、正規表現指定）</data>
    <begin>表の始まりの形式（省略可、正規表現もしくは文字列指定）</begin>
    <end>表の終わりの形式（省略可、正規表現もしくは文字列指定）</end>
    <valid>値の有効・無効（列ごとに 0 と 1 で指定、デフォルト 1）</valid>
</table>
<table>
      .
      .<table>ノードは複数記入可能
      .
</table>
</dsxml>
```

Figure 3.3: An Overview of DSXML Definition

Figure 3.4 is a sample translation instruction for the results generated by the LMBench benchmark program. Two tables are defined in this definition, i.e., one is for the table whose caption starts the string "Processor, Process" and the other is for the table whose caption starts "Basic integer operations" as shown in Figure 3.2. The first $< table >$ node in Figure 3.4 defines the table whose caption matches the string "Processor, Process.*" where ".*" is a regular expression. Three lines are headed before the result data appears in the result shown in Figure 3.2. The first line of the header is expressed by the $< header >$ node. This line defines the columns of the table, and thus we do not need to express the next two lines. The following regular expression is the first line of the header.

図 3.4 は、LMBench ベンチマークが生成する結果を変換する定義例であり、2 つの表を定義している。図 3.2 に示した LMBench の実行結果には、「Processor, Process」および「Basic integer operations」の 2 つの表が含まれている。図 3.4 において、1 つ目の $< table >$ ノードは「Processor, Processes」用で、$< caption >$ は正規表現で「Processor, Processes.*」としている。見出しは 2 行になっているが、1 行目を用いて検索すればよいので、$< header >$ は、以下の正規表現としている。

```
Host\s*OS\s*Mhz\s*null\s*null\s*open\s*slct\s*sig\s*sig\s*fork\s*exec\s*sh
```

$< header2 >$ defines the header format used in the translated result. DS-Bench uses this header for the translated result. The $< data >$ node defines the format of the result values. The $< valid >$ node selects the result values. In Figure 3.4, the $< valid >$ node is "0011111111111" which means the first two columns are not selected, i.e., the Host and OS columns are not selected.

$< header2 >$ に変換結果に使用する見出しを指定する。DS-Bench が、データベース化するとき、$< header2 >$ の見出しが利用される。$< data >$ に、取り出すべき値のフォーマットが記述される。取り出すべき値の取捨選択は、$< valid >$ で指定可能である。ここでは、表の 1 列目「Host」と 2 列目「OS」を削除するために、$< valid >$ に「0011111111111」を指定している。

```
<dsxml>
<benchmark>LMBench</benchmark>
<table>
    <caption> Processor, Processes.*</caption>
        <header> Host \s* OS \s* Mhz \s* null \s* null \s* \s\s\s\s \s* open \s* slct \s*
        <header2> Host,OS,Mhz,null call, null I/O, stat, open close, select TCP, signal in
<data>.{9} \s .{13} \s .{4} \s .{4} \s .{4} \s .{4} \s .{4} \s .{4} \s .{4} \s .{4} \s .{4
        <begin> Processor, Processes.*</begin>
        <end>^$</end>
        <valid>0011111111111</valid>
</table>
<table>
    <caption> Basic integer operations.*</caption>
        <header>Host\s*OS \s* intgr \s* intgr \s* intgr \s* intgr \s* intgr </header>
        <header2>Host, OS, integer bit, integer add, integer mul, integer div, integer mod
<data>.{9}\s.{13}\s.{6}\s.{6}\s.{6}\s.{6}\s.{6}</data>
        <begin> Basic integer operations.*</begin>
        <end>^$</end>
        <valid>0011111</valid>
</table>
</dsxml>
```

Figure 3.4: Translation Instruction of LMBench Performance Benchmark

Figure 3.5 shows an example of a database generated by the results of the `LMBench` execution under the DS-Benchmark environment.

DS-Benchmark 環境が `LMBench` を実行後、図 3.4 の定義にしたがって生成されるデータベースを図 3.5 に示す。

```
<dsxml>
    <benchmark>LMBench</benchmark>
        <config>…</config>
        <type name= ” Processor, Processes ”>
                <data name= ”Mhz ”>2639</data>
                <data name= ”null call ”>0.10</data>
                <data name= ”null I/O ”>0.10</data>
                                .
                                .
                                .
        </type>
        <type name= ”Basic integer operations ”>
                <data name= ”integer bit ”>0.4000</data>
                <data name= ”integer add ”>0.3600</data>
                <data name= ”integer mul ”>0.1400</data>
                                .
                                .
                                .
        </type>
</dsxml>
```

Figure 3.5: An XML-formatted Result of LMBench Performance Benchmark

Chapter 4

Summary and Future Work

In this document, a dependability benchmark runtime environment called DS-Bench, being developed by the DEOS project, has been introduced. DS-Bench is a runtime environment used to quantitatively measure system behavior under an anomaly situation. It consists of three computers: the controller, the measurement machine, and the target machine. The controller is comprised of performance benchmark tools, measurement tools, anomaly load tools, and a database to archive the results of various benchmarks.

The uniqueness of DS-Bench is summarized as follows:
1) DS-Bench provides a runtime environment that measures five quantitative metrics: i) the ratio of anomalies (or failures), ii) the detection and reporting time of anomaly, iii) the failure recovery time, iv) performance under an anomaly, and v) energy consumption under an anomaly.

2) DS-Bench measures the behavior of the system under an anomaly situation and then registers the result of the behavior formatted in XML into a database. That is, DS-Bench gives quantitative evidence of the dependability characteristics of the system.

本稿では、DEOS プロジェクトで開発中のディペンダビリティベンチマーク実行環境 DS-Bench について報告した。DS-Bench は、システムの異常時の振る舞いを定量的に計測するためのベンチマーク実行環境であり、コントローラ、計測マシン、ターゲットマシンの 3 種類のコンピュータから構成される。コントローラは、Performance Benchmark ツール、Anomaly Load ツール、Measurement ツール、ベンチマーク結果を保存するデータベースから構成される。

DS-Bench の特徴をまとめると以下の通りになる。
1) 異常・障害率、異常検知報告時間、障害回復、異常時性能、異常時消費電力の 5 つのディペンダビリティに関する定量的指標を計測する環境を提供している。これは、ディペンダビリティ性質の証拠を示すことになる。

2) システムテスト時にシステムの異常時の振る舞いを計測し、その結果を XML でデータベース化している。運用時に生じる異常時の振る舞いからデータベース上に登録されている振る舞いと同じかどうかを確認でき、既知の振る舞いであれば、早期に異常元を発見できる可能性がある。

3) DS-Bench is expandable for unknown anomalies and new requirements in the sense that it is easy to integrate new performance benchmarks, anomaly loads, and measurement tools in the future. Thus, by using these tools, system dependability testing can be performed systematically, leading to improvement of the system.

A prototype implementation of the DS-Bench runtime environment is being developed. After the evaluation of this prototype system, DS-Bench will be redesigned to incorporate what we have learned from the evaluation.

3) 未知の異常、新たなる要求に対して、Performance Benchmark ツールや Anomaly Load ツール、Measurement ツールを将来追加できる拡張性に富んだ設計となっている。これらツール群の組み合わせにより、システム改良時にもシステムのディペンダビリティ確認がシステマティックに可能となる。

現在、DS-Bench 実行環境のプロトタイプ実装を進めている。本プロトタイプ実装の評価を通して、DS-Bench を再設計する予定である。

Bibliography

[1] Algridas Avizienis, Jean-Claude Laprie, Brian Randell, and Carl Landwehr. Basic concepts and taxonomy of dependable and secure computing. *IEEE Transactions on Dependable and Secure Computing*, 1(1):11–33, 2004.

[2] Karama Kanoun and Lisa Spainhower, editors. *Dependability Benchmarking for Computer Systems*. Wiley, 2008.

[3] Dependability benchmarking project. http://www.laas.fr/DBench/.

[4] Intel, Hwelett-Packard, NEC, and Dell, editors. - *IPMI - Intelligent Platform Management Interface Specification Second Generation v2.0*. Intel, 2009.

[5] P-Bus interface manual. http://dependable.os.net/.