EBERHARD KARLS
UNIVERSITÄT
TÜBINGEN

# Analysis and Design of Java Compression Offload on the IBM Hybrid Platform

**Thesis**

Wilhelm-Schickard-Institute for Informatics

Eberhard-Karls-University Tübingen

Author:

Huiyan Roy

Gartenstr. 11

72213 Altensteig

Email: hroy@de.ibm.com

31.07.2008

Supervisor (University Tübingen)

**Prof. Dr. Wilhelm Spruth**

Wilhelm-Schickard-Institut for Computer Science

Technical Informatics

Sand 13

72076 Tübingen

Germany

Supervisor (IBM Deutschland Entwicklung GmbH)

**Roland Seiffert, Jochen Roth**

IBM Systems & Technology Group

Systems Software Development

Schönaicher Str. 220,

71032 Böblingen

Germany

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbständig und nur mit den angegebenen Hilfsmitteln verfasst habe und alle verwendeten Inhalte aus anderen Quellen als solche kenntlich gemacht habe.

I hereby declare that the work presented in this thesis has been conducted independently and without any inappropriate support and that all sources of information, be it experimental or intellectual, are aptly referenced.

Tübingen, 31st July, 2008 ____________________________

Signature

# Acknowledgment

First and foremost, I wish to express my deep appreciation and gratitude to Prof. Dr. Wolfgang Rosenstiel, Prof. Dr. Wilhelm G. Spruth and Dr. Peter Hans Roth for making it possible for me to work on this interesting and challenging thesis. Without their dedication, conviction, passion and steadfastness in bringing an extensive cooperation between the lab and University Tübingen into reality, this project wouldn't have come about. I wish this thesis to symbolize the fruit of their efforts.

I would like to express my sincere thanks to the IBM supervisors Roland Seiffert and Jochen Roth for mentoring, for giving guidance and sharing ideas that were so precious to me. Thanks Jochen for your time and so much practical help whenever I needed it.

Thanks to my dear husband Gaetan, for being supportive and for analyzing problems together with me at crucial times which was a source of inspiration. Thanks for the cooking when I had to come back so late.

Also thanks to all the others that have given me help during this thesis. It has been such a joy to get to know all of you.

And my thanks to my Heavenly Father which will be transfered to Him in another format.

# Executive Summary

The Java package java.util.zip provides classes for reading and writing the standard ZIP and GZIP file formats. In certain situations, this compression and decompression require a significant portion of the overall CPU capacity. This thesis analyzes a given Java workload on System z that heavily uses compression and proposes a design that leverages network-connected Cell/B.E. servers to perform the actual compression task, thus offloading these CPU cycles from the System z. To realize this, the server takes a Cell version GZIP compression utility as a library tool and optimizes its data access for better performance. This design is implemented prototypically by having client-side Java programs communicating with the Cell blade server. After evaluating their efficiency at different compression volumes, it demonstrates that this offload implementation is successful and well-profitable for System z.

# Table of Contents

# 1. Introduction

## 1.1. IBM System z in a Hybrid Platform

In today's world, Information Technology is woven into every aspect of our lives. The demand for robust, efficient, flexible and autonomic computer systems are greater than ever, driving us to develop more secure, scalable and stable computer systems with even higher performance. The mainframe computers of IBM stand firmly as the number one computer system solutions ever since their inception a half a century ago, taking a unique roll serving various needs around the world.

Mainframe computer systems are most often used by large and middle-sized organizations which impose high specifications for their critical applications, that is, applications with a need for high availability, scalability, security or applications that have been running faultlessly since centuries, that hold a high market value and that are not to be replaced. Failure to meet these requirements would essentially result in a loss of market as well as a loss of customers. These requirements are thus significant and usually decisive to the fate of an organization and therefore never to be underestimated.

A good example is the applications of a bank. It requires a technology-based system that is one hundred percent secure, that remains stable at all times that never makes a mistake, thereby avoiding  those situations which a normal computer system often fails at. These requirements have their reasons: A bank has to handle a high amount of data transactions on a daily basis like the on-line queries of their many clients, drawing money at a bank machine, doing a wire transfer, paying at the supermarket or banking via Internet. These transactions and queries must be executed one hundred percent correctly and often need to be accomplished in real-time. The system should be able to foresee, recognize and also handle these tasks during peak times, for example before Christmas or during holidays, taking corrective measures like distributing the workload to other machines horizontally or vertically to ensure quality of service. Since some system tasks are more critical than others, the system should be able to handle these based on task priorities. Some work is to be done within milliseconds, some within seconds, others within minutes and some other non-critical jobs within a day or two.

As globalization is bringing the world closer together, the machine can not anymore take its official “night break” as it did years before and has to practically run all 24 hours of the day, 7 days a week. A machine like this needs to manage many maintenance tasks without having to be shut down. The work of software installation, system updates and backups should be accomplished when the system is “live” at work. Meeting these requirements demands much know-how and a high sophistication of the built-in mechanism.

The IBM System z computers fulfill and offer thought-through solutions for these requirements. These machines are highly reliable and practically never need to be shut down, not even during an operating system installation. Once correctly configured, the system stays extremely secure under the safeguard of its unique hardware architecture and the protection of its comprehensive software mechanisms. The built-in workload management provides constant monitoring in order to achieve performance goals. Through defining system policies, the decision maker of an organization can configure the system to reflect the requirements of the organization. System z supports strict backward compatibility to the System/360 generation. Old applications that ran on a 360 machine decades ago can flawlessly run on the newest generation of System z, which is, at the moment this article is written, z10 (also called eClipz). A System z can run a diversity of operating systems and share data amongst each other as members of the same body. A single System z machine is therefore able to replace many smaller servers providing higher scalability and reliability and saving operational costs. At energy prices soaring, System z also stands out as a green solution by using energy more efficiently. According to a Robert Frances Group study, a company analyzed the consolidation of hundreds of UNIX servers to one System z mainframe. The calculations showed monthly power costs of $30,165 for the UNIX servers versus $905 for System z. That company calculated they would save over $350,000 in power costs annually [z01].

The only disadvantage of a System z is that its CPU cycles are relatively expensive. Due to the sophistication of hardware and software behind it, computation on a System z can never be priced as low as the less well-equipped server systems. Through introducing simplified CPU models of ZAAPs, ZIIPs, IFLs, etc., hosting on System z is not as expensive as before. It is however still too costly for smaller sized companies, although some would achieve higher efficiency by utilizing IBM technology. Dedicated to solve this problem, the hybrid system has come into being which allows some less expensive co-systems to be combined with z, taking over part of its off-loadable computation. Cell, AMD and Intel processors are all candidates for an accelerator system. Through moving some compute-intensive workloads onto the Cell blade systems, the IBM hybrid system can achieve a higher scalable performance and offer a more affordable price.

## 1.2. Purpose of Thesis

The purpose of this thesis is to analyze the feasibility and effectiveness of such an offload system using the Java compression mechanism. Java compression and decompression can in certain situations cause cause significant CPU load. A co-system that offloads the compression cycles of System z should help to achieve a better general performance. This thesis analyzes the feasibility of an offloading concept, proposes a design and then evaluates the efficiency of the design.

## 1.3. Structure of Thesis

Chapter two offers a short introduction of the Cell processor, the Cell/B.E. structure and its programming concepts. It also takes a glance at the IBM Cell blade and Linux on the Cell operating system that build up the test environment of the offload design.  It then looks into the requirements of on-line gaming and proposes the hybrid system solution by illustrating the necessity of the compression offloading concept. It furthermore explains the reason, why hardware based compression offloading is not appropriate.

Chapter three introduces the widely used open source zlib library, its algorithms, data format and efficiency. It also introduces the gzip utility implementing a zlib version that is optimized for the Cell processor.

Chapter four analyzes the necessity and ways of further optimization of the gzip utility for the offload purpose and describes its workflow. It implements the hybrid design prototypically, the client and server communicating through the network socket pairs. Furthermore it illustrates the key points of the programming code on both sides of the system.

Chapter five evaluates the efficiency of the socket design at different compression volumes and compares them with Java's original compression performance. It concludes that this is a well-profitable hybrid design for offloading purpose.

Chapter six gives an overview how the results of this thesis could be used for further research endeavors.

# 2. Java Compression Offload Background

## 2.1. Cell Processor

### 2.1.1. Cell Processor Isn't Just for Games

The Cell processor was jointly developed by the “STI” design center, an alliance between Sony Computer Entertainment, Toshiba and IBM. The project was originally designed to bridge the gap between the desktop processors (like AMD's Athlon64, Intel's Core 2, etc) and the high-performance processors that are most widely used in scientific applications. The Cell processor has taken even more of the world programmers' attention ever since the PlayStation 3 (commonly abbreviated PS3) was launched in November 2006, the third generation of the home video game console from Sony. The Cell processor is however not only good for games. Its special architecture makes it an outstanding processor for running compute-intensive tasks. MPR Analysts’ Choice Award rated the Cell processor in its Microprocessor Report[1] [cell01] as the “Best High-Performance Embedded Processor of 2005” :

> *“We chose the Cell BE as the best high-performance embedded processor of 2005 because of its innovative design and future potential.... ”*

One of Forbes' articles in January 2006 titled “Holy Chip” wrote [cell02]:

> *“IBM's radical Cell processor, to debut in Sony's PlayStation 3, could reshape entertainment and spark the next high-tech boom... Cell could power hundreds of new apps, create a new video- processing industry and fuel a multi-billion-dollar build out of tech hardware over ten years.”*

IEEE Spectrum's special issue “Winners and Losers 2006” wrote [cell03]:

1 Magazine Microprocessor Report is a premiere reference material for detailed explanations and in-depth knowledge of new high-performance microprocessors.

*“It was originally conceived as the microprocessor to power Sony's PS3, but it is expected to find a home in lots of other broadband-connected consumer items and in servers too.”*

The Cell processor has ever since been spreading rapidly in a growing range of computing areas. Industry foresees the Cell processor playing a role in mobile phones, high-definition digital televisions (HDTV), hand-held video players and more. Stanford University is building a Cell-based supercomputer. Cell is also utilized in military missile systems and in medical imaging machines. The IBM's QS series Blade Server utilizes two Cell processors jointly working with each other - the newest market version QS22 has just been released in May 2008. The most recent news of Cell’s application is in IBM's supercomputer Roadrunner, a hybrid design with 12,960 Cell processors and 6,480 AMD Opteron dual-core processors. It reaches a peak performance of 1.7 petaflops and stays at the top of the TOP500 list [cell08].

### 2.1.2. Cell and the “Three Challenges”

Stanford's Professor Bill Dally has a nice analogy that explains the memory wall problem that general purpose processors have run into. He lets you imagine that you are doing a plumbing project. As you start the work, you notice that you need a pipe. So you drive to the store and buy the pipe. Once back with the pipe, you discover you need a fitting. So you drive to the store again and purchase a fitting. Then you discover that you also need to solder both together... This wasn't such a problem 30 years ago as reading the memory only cost several processor cycles. But nowadays reading main memory costs much more as it can take up to hundreds of processor cycles, so reading memory is becoming more like driving hours to buy one thing at a plumbing store.

Although the conventional processors try to solve the above problem with cache, they cannot avoid the situation that a cache miss happens. In this case the CPU stalls and has to wait for the data to be fetched. At each stall the processor waste hundreds of cycles. The result of this architecture is that application performance is in most cases limited by memory latency rather than by peak compute capability. Statistics show that the processor can spend up to 80% of its time waiting for memory. The problem gets worse with the dual-core CPUs. When the cores try to access the same memory address, the data in the cache gets out of date and needs to be updated. To do this involves a lot of logic, takes a lot of time and the more cores the system has, the more complicated the problem will get.

The Cell processor solves this problem by making something like a shopping list. Each SPE (Synergistic Processing Element) is equipped with a series of cache-like Local Stores. Instead of getting data from main memory every time it is needed, the SPUs (Synergistic Processing Unit) construct a list of the needed information and go get the data from the main memory through DMA transfer all at once. The processors can then be kept working as much as possible. The SPE processor cannot access the main memory directly. This avoids applying the complexity of the caching mechanism and delivers data to the SPE registers at an extremely high speed, making the data to be processed at a cache speed inside a SPE. The three-level data transfer of register file, local store and the main storage with its asynchronous DMA transfers between the local store and the main storage is a major breakthrough in processor architecture, resulting in the extreme parallelization of computation and data transfer. The challenge left to the programmer is to feed the SPEs with enough data to achieve peak performance.

The above was one of the “Three Challenges” that confronted the developers of the Cell processor. The other two main challenges of the chip development were the power wall and frequency limitation. To achieve high processing power, the engineers could have typically put more transistors on the chip, resulting in a heat increase. The Cell engineers solved the problem by allocating different functions to the processors. Power efficiency was improved instead of increasing the complexity of the hardware. A general-purpose PPE (Power Processor Element) runs the operating system while eight special SPEs run the compute-intensive tasks. They developed specially adapted software tools like more intelligent compilers (gcc, XLC), so that the burden to the chip hardware could be reduced.

To conquer the frequency wall problem, a combination of hardware and software optimization was applied. The Local Store of a SPU allows large shared register files which increase the processing speed significantly. More software controlled branching was built in which made deeper pipelines possible.

The efforts bring out good results. The Cell processor achieves approximately ten times the peak performance of a conventional processor that uses up the same amount of power. As to the actual application performance, some applications benefit less from the SPEs while some others show a performance increase of more than 10 times. In general, compute-intensive applications that use 32-bit or smaller data formats are the most suitable candidates for the Cell BE and can expect higher performance on the Cell.

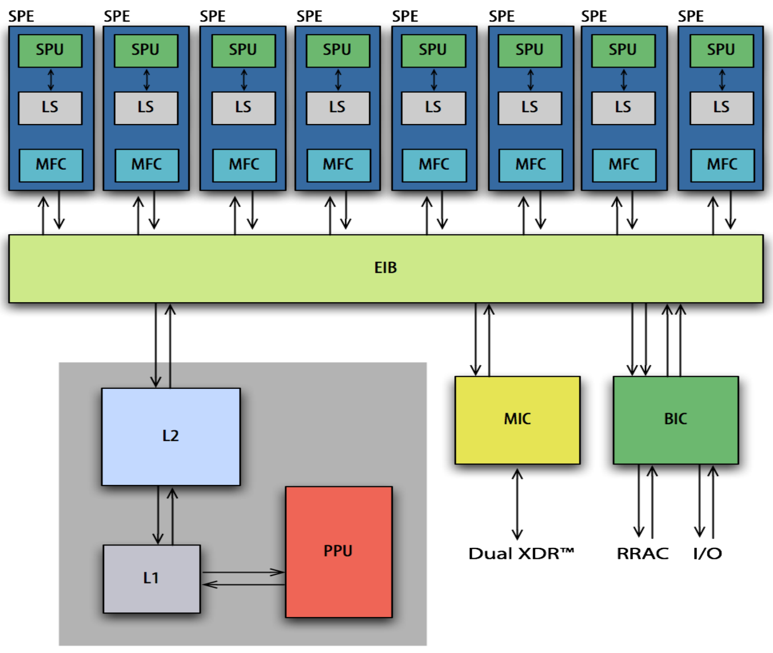
### 2.1.3. Cell/B.E. Architecture

The Cell processor's unique structure is the contributive factor that enables the PPE's intensive relationship with the SPEs thereby delivering high performance for number-crunching tasks, like Fourier analysis, decoding and encoding of stream processing, real time ray tracing, etc. All these tasks require a huge amount of calculation and a certain supercomputing characteristic of the processor.

The Cell processor is composed of one 64 bit PPE and eight specialized co-processor SPEs. The PPE is a conventional power architecture core, for example it could be a PowerPC or other POWER processors. The PPE is good at running control-intensive tasks and quick at task switching and is usually used to run the operating system and most of the organizational work of an application. The PPE contains a PPU for calculation, an L1 cache for data and instruction and an L2 cache memory. SPEs, the SIMD processors of the Cell, are built to carry out intensive mathematical work. SIMD stands for Single Instruction Multiple Data, indicating SPE's capability of doing multiple operations simultaneously with a single instruction. An SPE consists of the SPU, the Local Store, and the MFC (Memory Flow Controller). The Local Store is a 256 KB on-chip memory that allows the local storage of data. MFC works as a gateway taking care of the communication between SPE and the other elements on the chip. Each SPU processor contains a dedicated DMA management queue capable of scheduling long sequences of data transfer between various endpoints without interfering with the SPU's computations. Furthermore the Cell/B.E. structure has a BIC (Bus Interface Controller) that takes control to all the i/o devices and a MIC (Memory Interface Controller) that supports two memory channels.

Elements are connected together with an internal high speed bus EIB (Element Interconnect Bus) and work intensively together like a small cluster of processors inside the chip. EIB is implemented as a circular ring with four 128 bit unidirectional channels. Each participant of the bus has a read port and a write port which allow a point to point communication that is easy to scale. EIB is optimized for transferring huge data streams.

This unique architecture of the Cell processor is named Cell/B.E., an abbreviation for Cell Broadband Engine and is illustrated in the following depiction:



**Graph 1: Cell Broadband Engine Architecture**

**Source: J. A. Kahle, Cell Broadband Engine Architecture [g01]**

The Cell processor's special structure contributes to its incredible data processing performance. The introductory design of 90 nanometer technology is able to reach a peak processing speed of over 200 billion floating point operations (200 Gflops) per second in comparison to 26 Gflops for a Pentium 4 and 77 Gflops for a XBox 360 [cell02]. IBM announced in 2007 the production of 65 nanometer version of Cell BE and in Feburary 2008 the production of 45 nanometer version. At 45nm the Cell processor will reach the processing speed of one teraflop per second.

### 2.1.4. Cell Programming

A typical programmer basically has two kinds of situations when starting a Cell project. One is when he wishes to write a completely new application to  run on the Cell processor; the other is, as is more often the case, when he has an existing application that runs on a PowerPC architecture core and wants to bring that application onto the Cell, thereby needing to rewrite part of the code in order to take advantage of the Cell's SIMD capabilities. Both situations require a detailed analysis of his code in order to identify where the compute-intensive repetitive tasks are and which of those could be offloaded onto SPEs. Too large algorithms or algorithms which jump randomly, accessing small pieces of data are not suitable to run on the SPEs. Vectorizable and parallelizable algorithms are in contrast well suited for them. Once this is figured out, he can start writing the code for the PPE and the SPE separately, or in the second situation, partition the movable code for the SPEs from the rest and move this over. The partitioning usually means quite some work and requires the year-long experience of a programmer. This is the so-called PPE-centric model, the most often used model for partitioning an application, with the main application running on the PPE and individual tasks off-loaded to the SPEs. The PPE then expects and coordinates the results returned from the SPEs. The tasks distributed to the SPEs could be multistage-pipelined, parallel-pipelined or service-oriented. In the multistage-pipelined model, one SPE's working result is sent to the next SPE stage to be processed and the SPE at the end of the stage completes the calculation circle and sends the final result to the PPE. In the parallel-pipelined model, the working data is divided into similar sizes and sent to the SPEs that all implement the same algorithms to process the data. Each SPE's output will be returned to the PPE, which is responsible to reconstruct the data in the right order. In the service oriented model, each SPE implement a different algorithm as their unique services and the PPE alone is responsible to the SPEs' returned data. The PPE-centric model is most suitable for an application working with streamed data with a need of parallel computation.

Another less often used model is the SPE-centric model where most of the application’s code is distributed among the SPEs. The PPE acts as a centralized resource manager. Each SPE fetches its next working item from the main storage (or its own local store) after completing its current work. This model is suitable for applications that need little organization from the PPE.

The SPEs are designed to be programmed in high-level languages, such as C/C++. They support a rich instruction set that includes extensive SIMD functionality. However, using SIMD data types is not mandatory - a rich set of language extensions that define C/C++ data types for SIMD operations are also available for the programmers. These extensions allow them great control over code performance, without having to deal with the complexity of assembly language.

A rife development environment already exists for Cell programming Beside code development tools,  there are debug tools, performance tools and miscellaneous tools like the IDL Compiler. There is an SPE Management Library that supports creating and destroying SPE threads and regulating them for inter-thread communication. There is also a hypervisor available that allows different operating systems to run as different partitions on the same Cell hardware. A system simulator is available which facilitates code development without Cell hardware.

### 2.1.5. Linux on Cell

Several versions of the Linux operating systems have already been brought to the Cell processor, RPM[2] based Fedora distribution amongst others. The Linux operating system has a built-in socket interface that enables the Cell system's networking with other computers based on the TCP/IP stack. Sockets communication on this level is fast and uncomplicated and it is the most preferable networking solution for a system whenever performance becomes a critical factor. Other ways of communication are also available for the Cell. On the higher layers of the ISO/OSI model, RPC (Remote Procedure Call), RMI (Remote Method Invocation) etc. can also be implemented depending on the system requirement. More on this will be found in Chapter four.

### 2.1.6. The Cell Blade

Since the introduction of the first Cell blade server QS20 in 2005, the Cell based blades have now reached their third generation. The experiments of this thesis were carried out on the second generation Cell blade QS21. Here is a brief overview of its key hardware features:

- two 3.2 GHz Cell/B.E. processors
- 2 GB XDR memory (1 GB per processor)
- two Gigabit Ethernet ports
- one high-speed expansion slot for two additional ports for 10Gigabit Ethernet or InfiniBand 4X
- InfiniBand adapter

## 2.2. Linux on System z

System z is the most trustworthy computing system to date. An increasing number of organizations are adopting the technology, many Linux environment users amongst others. Linux on System z is a big defender of Linux's open source values and the code is completely open to users under the GNU GPL (GNU General Public License). It is considered the leading driver that encourages adoption of the Linux environment among business and governments. Linux on z combines the advantages of System z with the flexibility of the Linux operating system, building a scalable, secure, highly available and cost-effective structure. It further

2 RPM stands for RPM Package Manager, a package management system originally developed by Red Hat Linux, now widely used in different Linux distributions.

helps to simplify the IT infrastructure which again reduces operating costs and promotes quicker deployment of new solutions to accelerate time to market. The less expensive hosting on z/Linux is achieved mainly through applying the much lower-priced IFL (Integrated Facility for Linux) processors dedicated to running Linux. Comparing with the traditional general purpose engines – the CPs (Central Processors), IFL has a simplified structure and is optimized for the Linux operating system.

## 2.3. Games on System z

### 2.3.1. Project Gameframe

In April 26, 2007 IBM announced a cross-company project “Gameframe” cooperated together with the Brazilian game developer Hoplon Infotainment. Hoplon is a leading developer of multi-player on-line games implementing complex real-world simulations. The project’s aim was to bring Hoplon's online science fiction massive social game - Taikodom to run under a hybrid system of z leveraged network-connected Cell/B.E..

**Graph 2: Massive Social Game Taikodom from Brazilian company Hoplon**

**Source: Jochen Roth, IBM, Nov. 2007. Gameframe_4AcademicDays_20071106.ppt [g03]**

### 2.3.2. MMOG Games Requirements

The number of players of MMOG (Massive Multi-player On-line Game) has been growing exponentially in the past ten years, reaching a registered number of 13 million. Taikodom, one of the MMOG games, also foresees a fast growth. The game simulates a virtual world of outer space with gamers playing with each other in order to fulfill certain missions under a persistent online environment. The gamers practice real-time interaction with each other and are actually playing the game “together” from their home client terminal. All MMOG games impose the following high demands on its server environment:

Requirement 1 - Real-time interaction and massive I/O throughput: As there are thousands of gamers connected by the Internet who interact with each other through the server-based service, there is an extremely high amount of data being sent back and forth between the two. Even though the data is usually compact, the huge number of players cumulates in an equally huge amount of data. Due to the user’s relatively low connection rate, the data is often first compressed at the local client level before it is sent to the server. The server therefore needs to decompress this before analysis can occur. Compression and decompression in this case reduce transfer time and help to improve system performance. The data analysis of the server consists mainly in evaluating the progress according to user input and calculating the interaction between the gaming users. The results are returned compressed to the client and the client program displays it uncompressed at its terminal graphically. The more players are involved in a game, the more voluminous the computation becomes and the more difficult it is to display the interaction in real-time. A game will not attract more gamers when noticeable latency begins to show up. The server's ability to handle the requirements of its gamers becomes the vital factor of its success.

Requirement 2 - On-demand scalability: Even if a server system can easily handle the peak amount of gamers today, it is already confronted with the performance levels it will need to handle several months later, as the number of on-line gamers daily grows, much faster than any architecture or technician can actually react to. This forces the system to recognize the high workload and scale on-demand by distributing the work vertically and horizontally to other server nodes in an intelligent manner. The system needs to dynamically support additional nodes in order to circumvent oversaturation. When during a holiday the number of players exponentially grow – a positive development, the server needs to have the added dynamic capacity so that gamers won't be disappointed – a negative development!

Requirement 3 - High availability:  As most games are intended for international clients that live in different time zones, the server is not supposed to be shut down to restart for maintenance, installation or for any other reasons. The server should not encounter crashes and is supposed to run stable without scheduled downtime - zero downtime is expected.

Requirement 4 - Security. With the evolution of the Internet, onl-ine crime is becoming a serious issue throughout the world. Gamers are with no exception confronted with the danger of data theft and the game servers will continue to be an attractive target for hackers. A game server should thus be technically heavy-armed and remain pervasive to avoid any loss of client data.

Requirement 5 - High Speed. The server should remain agile as it faces massive computation-intensive tasks. Taking Taikodom as an example, the server needs to implement for instance a significant amount of real-world simulation and security processing. Real-world simulation occurs when it needs to calculate the characteristics and reactions of the outer space objects based on physical law: A ball thrown in a virtual world must obey the laws of gravity. An explosion would only look “real” to the gamers when the objects fly the same way as they would in reality. When a spaceship moves forward at nearly light speed, the objects it passes over will look extended and the elapsed time shown on its devices will have to be calculated slower. The server also requires to do data compression to reduce transfer volume. Both the compression and the physical simulation involve much computation and consume a high percentage of the system's computing power. This requires from the the server a certain super-computing capability to qualify.

### 2.3.3. Cell Blade and System z: A Perfect Hybrid Platform

The hybrid platform of IBM’s System z connected with the Cell Blades is a optimal synergy that fulfills the above requirements. System z provides the highest level of security and massive workload handling, assuring the execution of its administrative tasks and guaranteeing an enduring connectivity to a huge number of clients. Cell/B.E. takes over the most resource demanding calculations thus enabling the System z to fulfill its job.

This combination is an effective and financially attractive game server system, as the most compute-intensive tasks are offloaded from the expensive CPU cycles of z and are carried out on the much more economical Cell blades. Without offloading, the server system required would will end up costing too much and would not be financially feasible.

System z and the Cell Blades of the testing environment were connected through gigabit Ethernet. Higher data transfer speeds at the physical layer through Infiniband are still under experimentation.

## 2.4. Hardware Compression Mechanism of System z

As data compression can be implemented in either software or hardware, there comes the question if the compression work could actually be done by the hardware mechanism of System z that already exist widely (for example under z/OS operating system), saving the effort of implementation in software. System z has an auxiliary processor that provide solutions for the compression requirements based on a LZ algorithm. By implementing compression on hardware it could bring advantages like:

- running faster
- being less expensive
- black box principle
- offloading the main processor

The followings is a short introduction to the hardware compression mechanism of System z. It illustrates the answer to the question, why the z hardware compression facility could not have been the solution for the compression job.

Two main components are required to activate hardware compression: the compression call instruction (CMPSC) and the compression dictionaries. CMPSC is based on Lempel-Ziv 2 (LZ2 or LZ78) algorithm and specifies via general registers the source operand address and its length, target operand address and its length, the location of the dictionary, and the indication of the operation (compression or decompression). According to IBM Redbook [z03] CMPSC can be used to compress any randomly or sequentially used data, as long as there is some degree of repetition of character strings. CMPSC also has a symbol translation option that allows the instruction to be used to compress network data. CMPSC uses two static dictionaries for compression and decompression, that must be prepared and read into the memory beforehand. This is usually done by a special program that read some sample data and create the dictionary out of it. The performance of the compression will directly be affected by the quality of the dictionary.

The hardware compression mechanism of CMPSC does not bring a solution to the current System z platform that runs z/Linux as the operating system. There are generally two main problems:

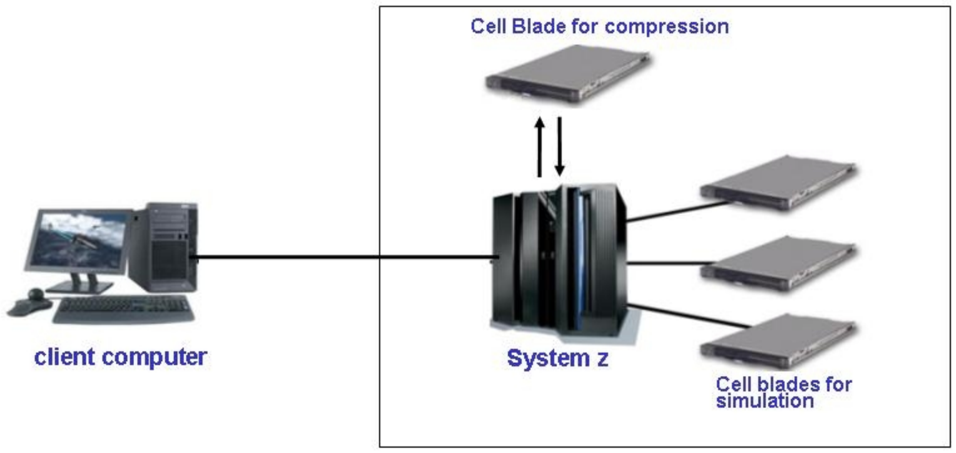
- Compatibility problem. Although CMPSC is used very much under z/OS, it can not be implemented on a Linux environment. The compression format that CMPSC instruction support, is not compatible to the standard compression algorithms of Linux. Linux environment supports compression and decompression standards like .gz, .bzip2, and .zip. Function gzip and zip implement LZ77 (LZ1) algorithm, a patent-free compression algorithm. Bzip2 implements a patent-free algorithm as well which combines several layers of compression for implementing on top of each other, including Borrows-Wheeler Transform and Huffman coding. CMPSC however, implement the patent-protected LZ78 (also named LZ1). This furthermore conflicts with Linux's patent-free principle and was thus never an implementation on Linux.

- Dictionary problem. Static dictionaries are applicable only when the content of the data to be compressed is known beforehand, or when most of the key words that would exist in the data are predictable. This is especially useful for example in database compression, since all field names are known beforehand and entries could be foreseen. It is very hard to construct an efficient static dictionary without knowing the contents of the data. In cases like this, the dictionary has to be built dynamically. They have to be created during compression to fit the input data. In the hybrid system of this project, since the data from the client could appear in any format and the content varies from case to case, there is no way to specify a general compression dictionary that works for all situations that guarantees a good compression result all the time.

# 3. Architecture and Compression Library

## 3.1. System Architecture

In the Gameframe project, the server system is a combination of mainframe and the Cell blades, both of which run Linux. The blades are plugged into System z and are connected with it through a one gigabit Ethernet connection. Several Cell blade servers takes over physics simulation. Another blade server is reserved for compression and decompression. The gamers access the game server from their home client through the Internet. Upon receiving client data, System z sends the packets to the Cell blade for unpacking and the Cell blade sends the decompressed data back to System z. After finishing its work, System z lets the Cell blade compress the data again before forwarding the results to the clients. The architecture of the whole System is represented in the graph below:



**Graph 3: System Architecture of Gameframe with Cell blades for offload**

The above is only a simplified structure. A demilitarized zone, the so-called DMZ, a subnetwork that exposes the services to the Internet, prevails between the client and System z. The DMZ is an additional layer of security for the organization's LAN (Local Area Network).

Although not illustrated in the graph, the demilitarized zone separates the Intranet from the Internet, is essential and cannot be omitted.

## 3.2. Zlib Library and the Deflate Algorithm

### 3.2.1. Background

Data compression is no new topic, and compression algorithms have seen little advance for the last years. Compression algorithms can be generally divided into

- lossless data compression
- lossy data compression

Lossless data compression allows the exact reconstruction of the original data from the compressed data and should be used when the source data and decompressed data have to be identical. Text-based data is mostly compressed in this way. Examples of lossless compression algorithms are Run-length encoding, Lempel-Ziv family (LZSS, LZW, etc.), Deflate algorithm, PNG, TIFF, etc.

Lossy data compression converts the data within defined tolerences. It is most commonly used to compress multimedia data (audio, video and graphic).

### 3.2.2. Deflate Algorithm

Zlib compression library was written by Jean-loup Gailly (compression) and Mark Adler (decompression) and is designed to be a free, general-purpose lossless compression library without being covered by payment liable patents. Zlib implements Deflate algorithm and achieves typical compression ratios between 2:1 and 5:1. Theoretically zlib can in extreme cases reach a compression factor of 1000:1 [zlib03]. Zlib is widely used under different platforms and for different languages, such as zip and gzip tools under Linux, Winzip under windows, java.util.zip package in the Java language, etc.

The Deflate algorithm is a lossless compression algorithm that uses a combination of LZ77 and Huffman Coding. Deflate algorithm is widely used in different libraries because of its good performance and its guarantee of never expanding the data, in comparison with LZW, which in extreme cases doubles or triples the file size.

LZ77 works based on a sliding window principle which slides through the whole text. The window consists of two buffers: one search buffer, and one preview buffer. The search buffer contains the text fragment that the program has worked on and serves like a dictionary. The preview buffer contains the text that needs to be compressed. To construct the compression data, LZ77 looks through the search buffer - when the next sequence of characters to be compressed in the preview buffer is identical to that can be found within the search buffer, the sequence of characters will be represented by two numbers: an offset, suggesting how far back into the search buffer the sequence starts, and a length, suggesting the number of repeated characters. During compression, a hash table is constructed to enable a faster searching process. The size of the buffers has to be set beforehand. A bigger buffer size makes compression of a higher ratio possible, but would take longer time. A smaller buffer size enables the compression algorithm work faster, but won't reach a ratio as high. The libraries that implement LZ77 usually gives the user the possibility to set the level of compression according to their wish. The following gives an example of compressing the byte sequence of “abcamanand” with LZ77.

| | Distance | Length | Symbol |
|---|---|---|---|
| abcamanand | 0 | 0 | 'a' |
| a bcamanand | 0 | 0 | 'b' |
| ab camanand | 0 | 0 | 'c' |
| abc amanand | 3 | 1 | 'm' |
| abcam anand | 2 | 1 | 'n' |
| abcaman and | 2 | 2 | 'd' |

**Table 1. LZ77 compression**

Huffman coding is based on the frequency of occurrence of the characters. The characters that occur more frequently will be encoded with a lower number of bits - the less frequently with higher - through a tree structure where each leaf indicates a character. The algorithm constructs at the end a so-called Code Book that serves as a reference for decoding. Huffman coding compresses typically between 20% and 90% of the original data.

### 3.2.3. Gzip File Format Specification RFC 1952

```
   +---+---+---+---+---+---+---+---+---+---+
   |ID1|ID2|CM |FLG|     MTIME     |XFL|OS | (more-->)
   +---+---+---+---+---+---+---+---+---+---+

(if FLG.FEXTRA set)

   +---+---+=================================+
   | XLEN  |...XLEN bytes of "extra field"...| (more-->)
   +---+---+=================================+

(if FLG.FNAME set)

   +=========================================+
   |...original file name, zero-terminated...| (more-->)
   +=========================================+

(if FLG.FCOMMENT set)

   +===================================+
   |...file comment, zero-terminated...| (more-->)
   +===================================+

(if FLG.FHCRC set)

   +---+---+
   | CRC16 |
   +---+---+

   +=======================+
   |...compressed blocks...| (more-->)
   +=======================+

     0   1   2   3   4   5   6   7
   +---+---+---+---+---+---+---+---+
   |     CRC32     |     ISIZE     |
   +---+---+---+---+---+---+---+---+
```

**Table 2: Gzip file format**

**Source: Gzip File Format Specification version 4.3, [rfc1952]**

The Deflate algorithm uses the advantage of both LZ77 and Hufmann algorithms and turns out to be one of the most popular lossless compression algorithms in wide use. Gzip utility implements the Deflate algorithm and the zlib library to compress and decompress data based on files. According to RFC 1952 the gzip file format is specified in the above Table 2. Here are some important notifications to the specification:

- ID1 and ID2 stand for identification 1 and 2, specifying the file as being in gzip format.
- CM stands for compression method. It is set to 8 when the Deflate compression algorithm is used.
- FLG are the flags
  - bit 0 FTEXT : Setting FTEXT usually denotes an ASCII text file. This flag is usually cleared if binary data is involved.
  - bit 1 FHCRC: If this is set, it denotes the use of the CRC16 version.
  - bit 2 FEXTRA : Ff this is set, it signifies the existence of optional extra fields.
  - bit 3 FNAME : Specify whether an original file name is available.
  - bit 4 FCOMMENT : When set, denotes the existence of a zero-terminated comment.
- MTIME stands for modification time.
- XFL stands for extra flags.
- OS stands for the operating system.
- CRC32 denotes the Cyclic Redundancy Check value of the uncompressed data.
- ISIZE stands for input size and contains the size of the original (uncompressed) input data.

## 3.3. The Zlib Library Optimized for the Cell Processor

### 3.3.1. Introduction

Zlib library is written for a sequential environment. By optimizing it for the Cell blade one needs to have a thorough understanding of the library’s functionality in order to exploit Cell's unique architecture. The library has been optimized by Seunghwa Kang, a Ph.D. student from Georgia Tech [zlib04]. An example application named minigzip demonstrates the performance and the use of the Zib library optimized for the Cell processor. Sourceforge [zlib05] presents the source code for both the library and the gzip utility. Two compiled binary executables for minigzip application are offered to run on a Cell blade environment with or without the SDK. The library offers a command-line user interface. User need to specify the name of the file to be compressed and the variables if needed. After the compression a new file with extension “.gz” is generated in the folder. The program offers the user the possibility of specifying the compression level (from 1 to 9), the block size (from 100 to 900 KB) and the maximum number of SPE threads that will be generated for compressing each file stream. For decompression user uses the same command with flag '-d' added ahead of the file name.

### 3.3.2. Optimization Analysis

Parallelizing the zlib library on the Cell processor is not easy. Zlib has a high dependency on data processing and the way the algorithm accesses the data makes it difficult to parallelelize. Basically the following points prohibit a good parallelization of the code [zlib06]:

- Data that is compressed by LZ77 is a mixture of literals and numbers that indicate length and distance (see table 1). In order to find out what the next symbol is, the symbol that is before it has to be identified first. This sets a limitation for the Cell to parallelize it.
- Decompression of data processed through the Huffman algorithm also builds on data dependency which needs to be processed sequentially. This is due to the character symbols that are encoded with different bit lengths. The length cannot be known when the characters before is not yet decompressed.
- Both the LZ77 and Huffman algorithms require a great amount of table lookups. This happens for instance when the LZ77 algorithm looks for the identical literals in the search buffer or when the Huffman-coded data runs the decoding process. This process cannot be vectorized since the Cell SPE does not support this type of random memory

accessing.

- Another point is the difficulty of branch prediction. Branches always strongly depend on the input. Well compressed data contains a great percentage of numbers (see table 1) indicating the length and distance of the indexed content in the sliding window. Poor compressed data, on the contrary, contains lots of literals and less numbers. This results in different branch behaviors and reduces the performance for the zlib library when running on the Cell.

According to Kang, the following optimization of zlib code for compression and decompression has been achieved for zlib running on the Cell bladeAccording to Kang, the following optimization of zlib code for compression and decompression has been achieved for zlib running on the Cell blade [zlib06]:

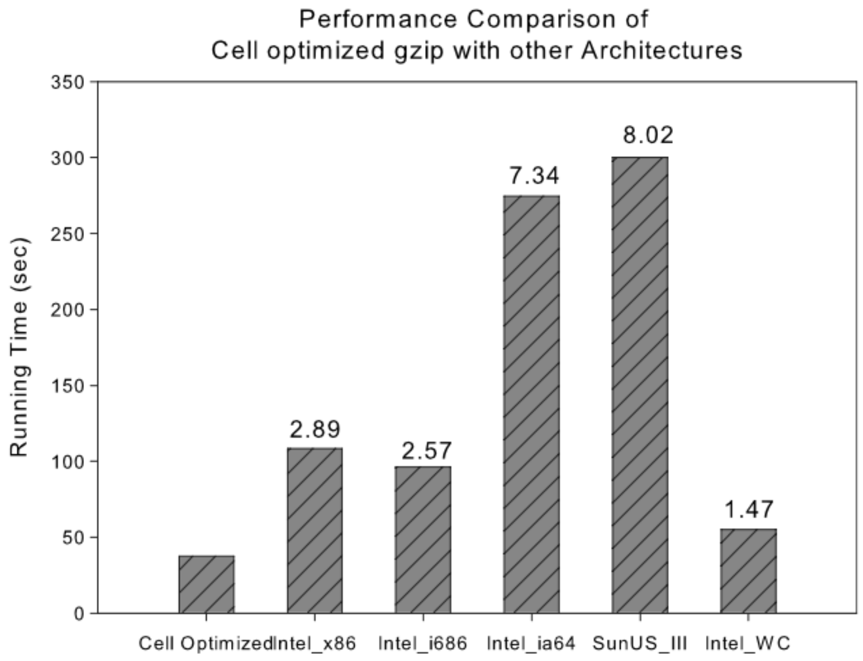
- Calculation of the hash key using the three bytes starting from the inserting byte at compression
- String comparison of LZ77 using SPE's 16 byte byte-wise vector comparison instruction at compression
- Vectorization of the window update loop of LZ77 at compression
- Vectorization of table construction at decompression
- Vectorization of CRC calculation algorithm and Adler32 algorithm (an alternative to CRC) of decompression
- Identification of computation-intensive loops and applying loop unrolling
- Static branch hinting
- Separation of compression and decompression routines to reduce memory usage

The gzip utility, by implementing zlib library, is data-dependent as well. According to Kang this tool has gone through the following optimization [zlib06]:

- Full flushing to break data dependency based on introducing an extra field in the header data of the compressed file
- Input file partitioning to multiple blocks

- File read and write threads

These efforts enables the gzip implementation on Cell BE in achieving an overall speedup of 2.89 for compression, as compared with a Intel Pentium4 system. Other comparisons are illustrated in the following graph:



**Graph 4. Performance comparison of Cell/B.E. optimized gzip compression with the original zlib implementation on other single processor architectures**
**Source: Paper HPC-Cell-ParCo2007.pdf [zlib06]**

To make it easy to distinguish between the zlib optimization done from Kang and the thesis author's further optimization for the hybrid system in the following chapters, Kang's version of the zlib library including the optimized gzip utility will be called “Georgia-zlib”. Georgia-zlib is further adapted and optimized to fit the requirements and specialty of the hybrid system, making a compression offload from System z onto the Cell blade profitable.

# 4. Program Analysis

## 4.1. System environment

The Georgia-zlib library and its gzip code were originally compiled with gcc provided by Cell SDK 2.0 and run on a IBM QS20 Cell blade with SPE library libspe. The tool run under SDK3.0 on a QS21 as well.

The System z runs a Java client that calls up the Cell blade server compression and decompression functions. The server functions are written in C. In the example of Hoplon's Taikodom, the client game code is written in Java. The client can also be implemented in any programming languages as long as the library supports socket.

## 4.2. Modification of Georgia-zlib for Offload

For the purpose of offloading System z's workload onto a Cell blade, the optimized gzip utility in the Georgia-zlib package is a helpful basis. Certain points of the utility need to be optimized so that the offloading can achieve optimal performance.

### 4.2.1. Interface and File I/O

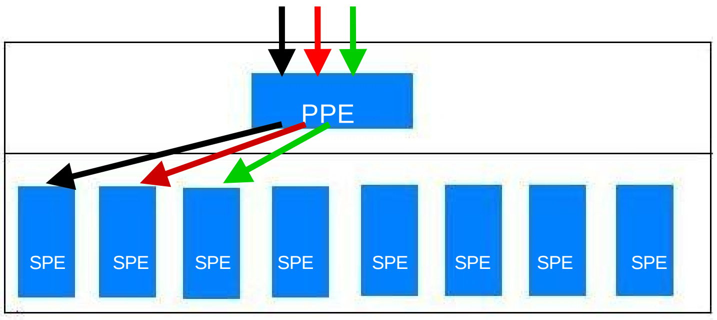
The gzip tool of Georgia-zlib offers a command line interface and requires a file name as a variable. The tool accesses the data through the standard file i/o functions in C, which, after being called, copies the data from the disk to a library buffer. The buffer size is usually set at 8192 bytes in the Linux operating system, that means, the file data is transferred from the disk to the memory in blocks of 8 KB. Accessing the disk creates a high i/o latency. Avoiding this would help to increase the performance of the Cell/B.E. server, therefore the following optimization to the current gzip utility was carried out:

- The new interface of the calling program receives a memory address and a number as its arguments instead of a file name. The memory address indicates the location of the data which needs to be compressed/decompressed; The number specifies the size of the data. This optimization avoids the original disk activity that is actually unnecessary. As System z would send the data to the Cell blade through a standard TCP socket, the Cell blade now receives the client data first in its main memory buffers. There is thus no need to write the data onto disk only to read it back sometime later. Through a much faster data access in the memory, the performance increases. This optimization involves changing the program code in the gzip utility, including that running in the SPE.

- Integrating the gzip function call in the TCP socket program enables calling the program directly without using command line. This is a trivial modification which requires a simple adaption of the main function name and arguments of Georgia-zlib.

### 4.2.2. SPE Thread Adaption

The SPE thread creation model of Georgia-zlib needs to be adapted for this particular hybrid system constellation as well. Georgia-zlib is optimized for data being equally distributed to each SPEs – the parallel-pipelined model discussed in chapter two. At compression it generates SPE threads on all of the available SPEs (or according to user definition) and sends each one a same amount of data to be processed. To realize this, the PPE carries out many organizational tasks including loading the basic workload, assigning vectors of data, initializing buffer variables and creating SPE threads. These tasks themselves take a lot of system resources and create a big overhead for the PPE for utilizing each SPE. After a closer look at the situation, it is obvious that the extra workload is actually only useful when the data to be processed is very large. By analyzing the realistic data sizes being transferred inside the hybrid system (in the Hoplon around 50 KB per stream and less than 100 KB), compressing the data that way actually takes longer and creates more costs for the system. Client data in this structure more often falls under the 100 KB size limit. For this reason, the data doesn't need to be distributed to run on different SPEs – one stream needs only one SPE so that the other ones can be reserved for the other streams. At every incoming data stream, the PPE creates a new child process to collect the data and then sends it to one free SPE to compute.



**Graph 5. SPE thread creation**

### 4.2.3. Workflow and Other Modifications

The optimized gzip utility begins its execution with argument parsing. Several issues require explanation:

- The compression level from the Cell's server program is set to 6 to achieve an average compression and time performance. This number can be set to anything between 1 and 9. The greater the number is, the higher the compression ratio will be and the more time it will cost. This number should not be set by the parsing function of gzip, but should be done by the Cell's server program through specifying one more compression argument variable.
- The number of SPU threads that should be generated per input stream is set to 1 (represented by '-t1'), indicating the concept of one stream being compressed with one SPU.
- The size of the compression block is set to the maximum block size represented by '-b9'. The maximum block size is set by the header file minigzip.h with a value of 900 KB. This size can also be set to 9 MB, allowing the highest input data stream size of this value. The current Cell blade QS21 does not support 90 MB due to its memory size.
- If the arguments contain a '-d', the decompression process will be invoked.
- The other arguments ('-f', '-h', '-r') are deactivated in order to achieve simplicity. They lack impact in compressing normal data. Modification could be undertaken in the Cell server’s program to reactivate the usage of these flags.

Depending on the existence of the input argument “-d”, the program sends the data for either compression or decompression. When compressing, it firstly starts the header writing procedure which fills up the first 22 bits with the header information that was originally created for a zipped file. The structure of the header data can be found in the gzip format specification (see table two). After defining the memory address pointer and the data size, the program then goes on to determine the workload that will later be processed by the SPE. A control block for the SPE is created, specifying the location and size of input, output and other information. The program then starts an SPE thread for each compression stream. It also creates a PPE thread which accumulate the result sent from the SPE. The PPE and SPEs communicate the processing status using mailboxes. After all blocks are processed, the PPE thread has already saved the result data in the right order. The program then starts a new procedure of header writing (as some information was missing when the first procedure took place) and substitutes the old header with the new one. The program finishes with a write trailer process that writes the CRC information at the end of the data buffer before passing its location to the Cell server program.

Decompression is similarly processed with header and trailer checking instead of writing them. After achieving the compression information from the header, the program lets the data be decompressed accordingly and finishes delivering a result stream that contains only pure data.

In both folders spu_compress and spu_decompress store the SPU codes, the parts of file access are modified to support reading the changed data structure that is sent from the PPE.

## 4.3. Networking through Sockets

### 4.3.1. Background Information of Socket Programming

Besides RPC, socket programming is one of the most often used mechanisms to build up networking between different systems as long as both kernel sides offer socket support. It is in fact the faster way of communication, as it avoids much system overhead of middleware implementation that would have been necessary for RPC.

Socket implementation happens on the TCP/IP stack of an operating system. It can usually achieve the upper bounds of bandwidth and speed that can be achieved between the two communicating systems. The simplest TCP/IP network test can be performed using the ping command. By giving the command “ping -c 10 cellhop”, the server named “cellhop” receives 10 TCP packages (based on standard mode) with one second interval between each package and generates the following output:

```
--- cellhop.boeblingen.de.ibm.com ping statistics ---
10 packets transmitted, 10 received, 0% packet loss, time 8992ms
rtt min/avg/max/mdev = 0.419/0.482/0.598/0.056 ms
```

**Table 3. TCP/IP network test**

Here “rtt” refers to the Round-trip Time – the time elapsed for the transmission of 64 bytes of a TCP packet between the two operating systems. Server cellhop offers an average transfer time of half a millisecond, which is also the system latency of TCP.

Socket programming is not language dependent, and it is not obligatory to have the client and the server implement the same programming language. The great flexibility of socket programming is one main reasons why socket use is so widespread.

### 4.3.2. Cell/B.E. Server Program

The Cell server program creates a usual AF_INET server socket that enables a  connection to it. INET stands for Internet, allowing a TCP/IP connection to the server with an IPv4 Internet address and a port number reserved for that service. The socket has the type of SOCK_STREAM that guarantees error-free arrival of the data stream in the right order.

The program continues with binding the socket with the address (sin_family and sin_port) and sets the incoming address (client address) to be INADDR_ANY, allowing connections from any client inside the network. This variable should be changed if the Cell server is located in an insecure network without firewall protection or other safety mechanisms. In the case of the hybrid system, Cell blades situated in the Intranet is protected by layers of firewalls. It can thereby trust any client that requests a connection to that port. After building up the real Gameframe infrastructure, this address is recommended to be configured into the IP-address of the System z in order to avoid unwanted connections and dedicate this Cell blade server only for compression offload.

The program is able to handle many connections at the same time. It creates a new child process when needed using fork() allowing the server to remain available for other requirements as well as processing incoming data. The program sets a BACKLOG number of 10, which is the highest allowed pending connections value. This number could also be set to other integer values including 0. A higher value allows more simultaneous connections but could delay compression processing since too many open connections take much of the available system resources away.

At the end of the execution process, the program reaps the dead process to free system resources. When a child process terminates, the parent process is informed through a SIGCHLD signal by calling the waitpid() system call. All the resources of the dead process are given back to the operating system and the process ID is deleted from the process table.

The data format that the Cell server program receives from System z is specified as follows: *size of data + argument + data for processing*. The size is an integer of four bytes that could represent over 4 billion bytes of incoming data (more than 4 GB).

The only argument that the System z can send the Cell server is '-d' indicating decompression. Other arguments can no longer be specified by the z client. This avoids a too complicated interface for the user (the one that calls up Cell compression), and helps to reduce mistakes. A user that has imported the class CellDeflater and CellInflater package in his client program can call up deflate(inputbytes) and inflate(inputbytes) functions by only specifying his input data with no knowledge of how the server system works.

The result data that the server sends back after finishing its compression or decompression has the format: *size* + *result data*. The size is also represented by an integer of four bytes as in the case of receiving. The result data is sent over to the client through the socket channel.

How the data is represented in the communicating systems deserve attention as well. In the hybrid system architecture, both System z and Cell blade implement big-endian structure, thus eliminating potential problems. Should the data presentation differ, a conversion between the TCP network byte oder and the host byte order might have to take place. Network byte order transmits data in big-endian format.

The server program is ready to run. It calls the modified gzip utility, passing the data and variables to it. Gzip processes the input and returns the result to the server in form of a struct value of a memory pointer and data size.

```
typedef struct _result_t{
      unsigned int size;
      char* buf;
} result_t;
```

**Table 4. Struct return value**

The Cell server sends the data to its client program on System z. The Cell's main program doesn't quit. The child process that handles each connection is reaped at exit.

```
void sigchld_handler(int s)
{
     while(waitpid(-1, NULL, WNOHANG) > 0);
}

int main(){
     struct sigaction sa;
     sa.sa_handler = sigchld_handler;
     sigemptyset(&sa.sa_mask);
     sa.sa_flags = SA_RESTART;
     if (sigaction(SIGCHLD, &sa, NULL) == -1) {
          perror("sigaction error");
          exit(1);
     }

     if (!fork()) {
     ...
     }
}
```

**Table 5. Reaping dead process**

## 4.4. CellDeflater and CellInflater for Java Client

Clients of all programming languages can talk with the Cell server. In fact, the offload does not have to be restricted to System z – other platforms can also profit from Cell's functionalities. As the game software is written in Java, a Java interface of compression offload on the Cell blade server is provided in the source code.

Java programmers that wish to take the advantage of the offload mechanism of the Cell blade server should import the Java class CellDeflater and CellInflater of his thesis. CellDeflater is equipped with the function deflate(inputbytes) that calls up the compression of the Cell blade. CellInflater likewise inflate(inputbytes) that brings the decompression function to running. In the example program, the Java client sends the byte data for compression and receives the compressed data from the server.

```java
byte[] compressedByte;
byte[] decompressedByte;
CellDeflater def = new CellDeflater();
compressedByte = def.deflate(inputByte);
CellInflater inf = new CellInflater();
decompressedByte = inf.inflate(compressedByte);
```

**Table 6. Java client**

The deflate function initializes a socket connection to the Cell blade server, sends over the working bytes and receives the result bytes before returning them to the user program.

```java
BufferedOutputStream writer = new
     BufferedOutputStream(Socket.getOutputStream());
DataOutputStream out = new DataOutputStream(writer);
out.writeInt(inputSize);
out.writeInt(mode);
out.write(input,0,inputSize);
```

**Table 7. Deflate class sending data**

Bytes received are processed in a similar way. The received size should be checked to see if it equals the number specified beforehand. When not, the function recv(1) needs to be called to run in a loop.

# 5. Efficiency Evaluation

## 5.1. Compression with Java.util.zip

The most obvious compression tool for a java programmer is the java.util.zip package. By importing this package the programmer can use the convenient compression classes and their comfortable functions. An example: the Deflater class offers a deflate(1)[3] function, the GZIPInputStream that offers a read(3). The performance of these classes is based on the performance of the Java language. They do not overcome the performance of machine-near languages like C. The test was carried out on a z/Linux system that runs on a z/VM operating system. It has 20 GB of main memory and utilizes 12 dedicated CPUs. The command “java -version” gives the following information:

```
java version "1.5.0"
java(TM)   2   Runtime   Environment,   Standard   Edition   (build
pxz64devifx-20071025 (SR6b))
IBM  J9  VM  (build  2.3,  J2RE  1.5.0  IBM  J9  2.3  Linux  s390x-64
j9vmxz6423-20071007 (JIT enabled)
```

**Table 8. Java version information**

The compression and decompression of java.util.zip was based on the following piece of code:

3 The number in the bracket indicates the number of parameters a function has.

```
Deflater java_def = new Deflater();
java_def.setInput(input.getBytes());
java_def.finish();
java_def.deflate(java_comp);

Inflater java_inf = new Inflater();
java_inf.setInput(java_comp);
java_inf.inflate(java_decomp);
java_inf.end();
```

**Table 9. java.util.zip compression and decompression**

The variable “input” is a string that contains the content of a file that was read from the disk into the main memory beforehand. The time used for disk access for java.util.zip is not calculated into the following graph since a programmer can also have his compression data in main memory instead of on the disk. If this is not the case, extra time has to be calculated in the consumption of java.util.zip. As disk accessing time is composed of Seek Time[4], Rational Delay[5] and Transfer Time[6], a good 20 milliseconds could be needed for accessing 50 KB of data on the disk.

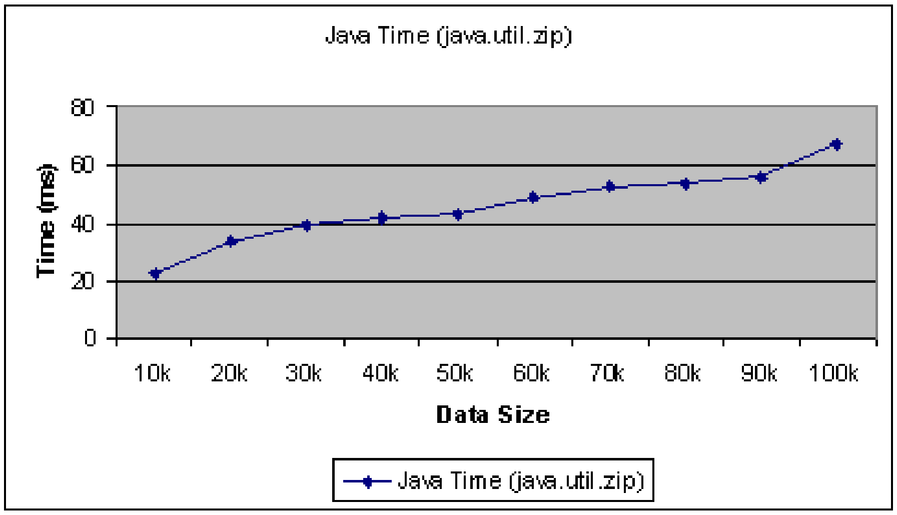
For every compression, 10 identical trials were conducted in order to canculate an average value. The data being compressed is normal English text[7]. The numbers form a rough representation of the working time of java.util.zip. For compressing and decompressing 60 KB of data, this original Java package takes about 50 milliseconds (without disk access time) as shown:

4 Seek Time is the amount of time needed for the access arm to reach the disk track. It depends on the spindle speed of the disk and the numbers between 10 to 20 milliseconds are common[java01].

5 Rotational Delay is the time needed for bringing the disk to a needed rotation speed. 7200 revolutions per minute (RPM) has a maximum rotational delay of 8 ms or an average rotational delay of 4ms.

6 Time during which data is actually read and written to the disk.

7 The text is chapters extracted from the Cell BE Programming tutorial.



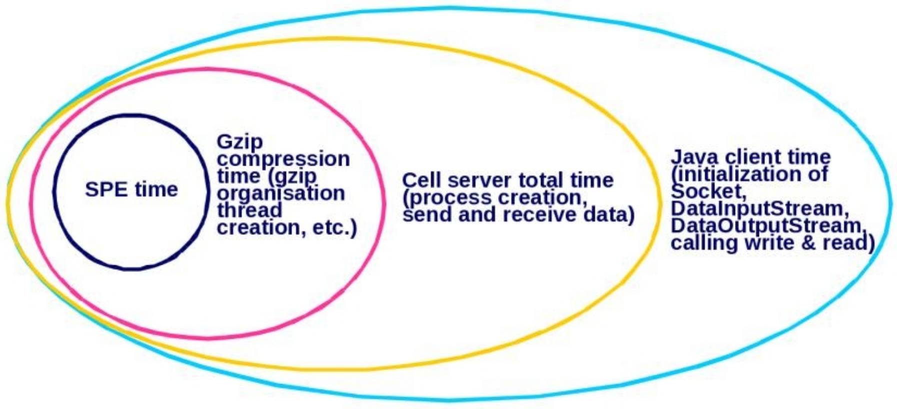
**Graph 6. Compression measurements with java.util.zip**

The values, as may be seen, are not always consistent. For an sample data size of 100 KB, the compression time was: 63ms, 52ms, 64ms, 50ms, 155ms, 49ms, 154ms, 50ms, 52ms, 49ms, 55ms, 51ms, 51ms, 48ms. The "usual" measurements were around 50ms, the two values above 150 milliseconds exceptionally high. This is caused by the virtual machine that the z/Linux operating system is built on. By coordinating a number of operating systems running together on one single physical machine, the z/VM has to distribute the resource fairly that the compression cycles sometimes have to wait to be rescheduled. If the same code were executed on Linux directly installed on a physical machine, the measurements would tend to stay constant.

## 5.2. Compression with Cell/B.E. Server

### 5.2.1. Compression Time Components

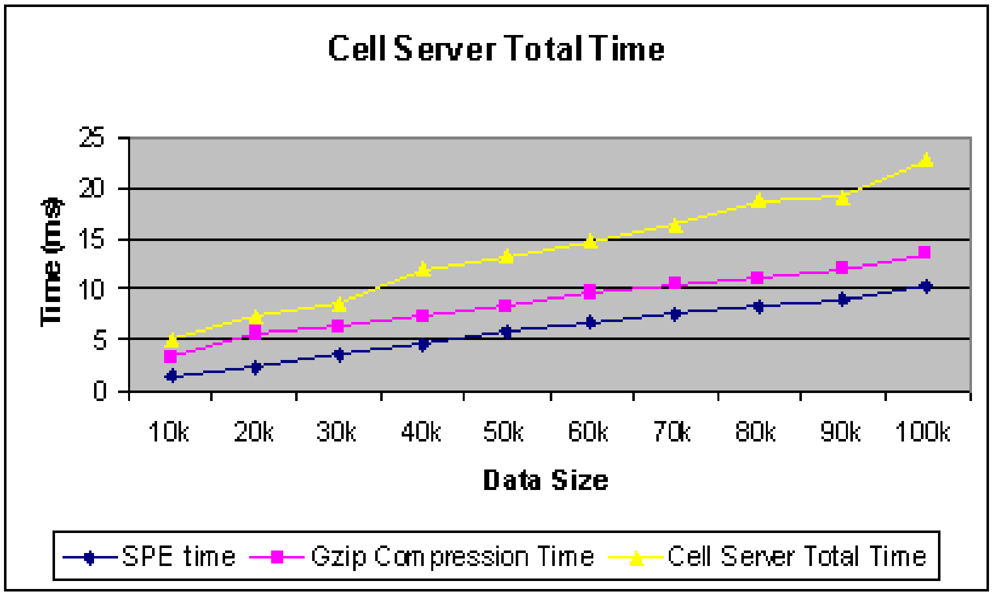
The time required for Cell/B.E. supported compression is split into several components. On top of the real SPU time that the SPEs need for doing the actual compression and decompression, the PPU uses a certain amount of time to carry out the gzip organizational tasks that are needed before the actual compression work can take place in the SPU. This is named gzip time; it includes processing the gzip header and trailer, assigning initial workload, creating the SPE thread and the PPE write thread etc. Beyond this gzip time, the Cell blade server is involved in other organizational tasks like creating new processes for the client connections, gathering client data packages, calling up compression / decompression and sending results back to the client. This is called the Cell server total time and encompasses the entire time consumption of the Cell blade server. The Java client at the other end of the network initializes the offloading by creating a new socket, new InputStream and OutputStream objects to call up the compression. This is demonstrated with the light blue colored circle. All four circles contribute to the total cost of time.


SPE time
Gzip compression time (gzip organisation thread creation, etc.)
Cell server total time (process creation, send and receive data)
Java client time (initialization of Socket, DataInputStream, DataOutputStream, calling write & read)

**Graph 7. Time composition of Cell-supported Compression**

### 5.2.2. Cell Server Total Time

The server programs run on the Linux operating system of the Cell blade QS21 with two Cell processors and 2 GB of main memory. The graph below shows the time consumption of the three smaller circles of Graph 7, with each represented in blue, pink and yellow. The values are an average of 10 measurements respectively.

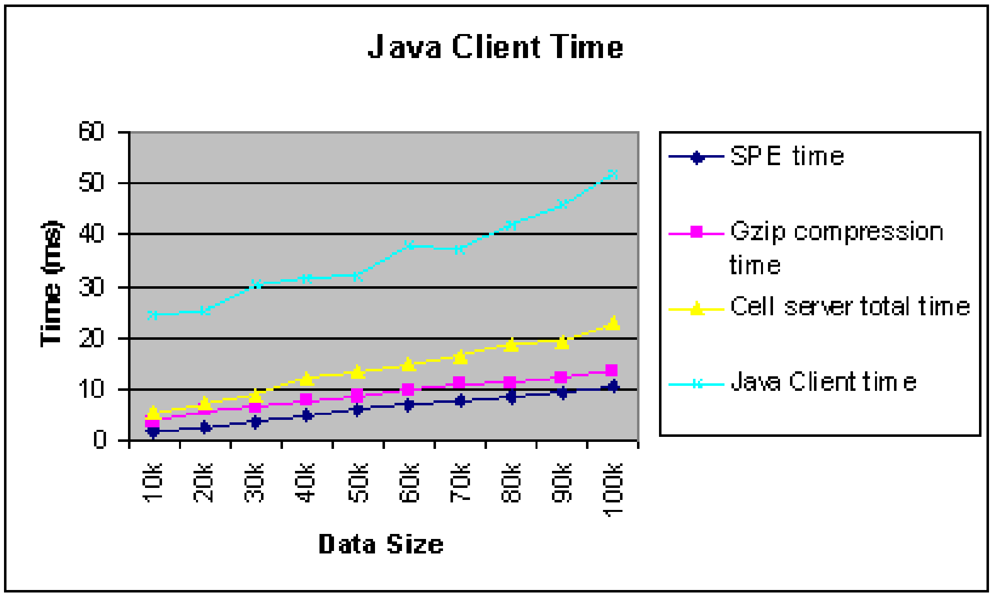


**Graph 8. Cell Server Total Time**

For compressing and decompressing 60 KB of data, the Cell blade only uses 15ms locally, which is a good acceleration compared to Java's 50ms. This shows a drastic contrast between the performance of the C language and Java. Compressing the same data using the same algorithms, the 35 milliseconds difference is pure overhead caused by the Java language and its virtual machine.

### 5.2.3. Java Client Time

The Java client that calls up the Cell blade's compression was tested on the same System z machine of the java.util.zip test. The systems are connected to each other through gigabit Ethernet, which allows a highest data transfer rate of 125MB/s. As shown in the following graph, the Java client adds on top a considerable amount of time indicated by the light blue colored path. The Java language again takes a lot of system resources and proves to be disadvantageous and costly. By compressing 60 KB of data, it consumes an extra 20 milliseconds only for creating the necessary data objects for receiving and sending. In comparison, the SPE only needs 8 milliseconds net time to do the actual compression work.

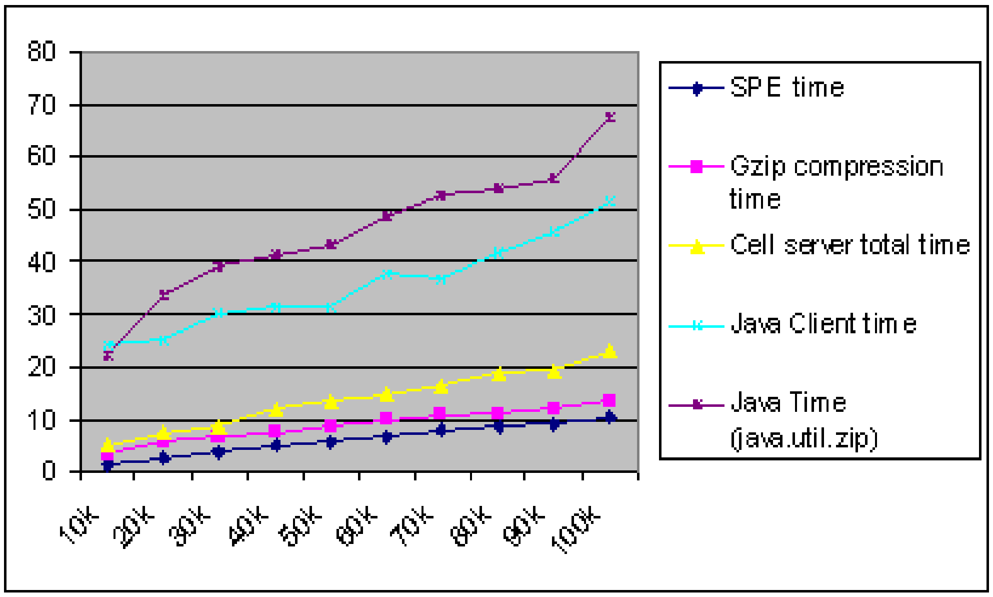


**Graph 9. Java Client Time**

An average calculation time with the Java client was from 10 trials. A typical phenomenon was that the time for the first compression invocation was always much higher than the rest. For 100 KB data again, the results were 299ms, 49ms, 45ms, 45ms, 56ms, 47ms, 56ms, 49ms, 54ms, 47ms and 54ms. This is mainly caused by Java's relative higher cost at initializing and implementing some expensive objects during the first call, in this case objects of Socket, DataInputStream and DataOutputStream. On the Cell blade's side, the gzip time, shown in pink, showed little divergence between trials – a few milliseconds at the most [mea01]. It is thus strongly recommended that a Java client programmer consolidate the number of compression invocations in one piece of program code in order to avoid unnecessary costly object initiation.

## 5.3. Comparison and Conclusion

Combining java.util.zip and the Cell blade results in the following. Java.util.zip, represented in purple, demonstrates a higher demand on time. The Java client in light blue proves it’s advantage over the java.util.zip starting at a data size of about 10 KB. Compression offloading from this size on is worthwhile and the extra costs of the Java client are well compensated for.



**Graph 10. Java and Java Client Comparison**

As long as time is concerned, compression tasks of less than 10 KB data should not conduct through the offload mechanism, as it will take longer for the little amount of data to be compressed on a network based server rather than using Java's local compression.

# 6. Outlook

The design of the workload offloading on System z leveraged network-connected Cell/B.E. servers proves to be a successful and well-profitable option as illustrated in chapter five. By implementing offloading, the CPU cycles of System z are greatly reduced and compression time is saved. The actual percentage of CPU cycles saved was not measured in the framework of this thesis. This would certainly be interesting to quantify and be worthy of further experimentation .

The data transfer of the test environment is based on a one gigabit Ethernet connection. The Infiniband communication system allows an even higher transfer rate and would raise the performance of the offload design even more. The improvement from Infiniband is currently being measured by another thesis student.

As discovered during testing, the client program implemented in Java isn't very efficient and has hindered achieving better performance. This raises the inherent question of the performance of other programming languages in the same context. This would indeed be an interesting comparison in future experiments.

# A. Appendix

## A.1 Source Index

- [z01] Robert Frances Group: Powerful Incentives - Using IBM System z to Realize Significant Operational Cost Savings , ZpowerJan2007.pdf
- [cell01] MPR Microprocessor Reward: http://www.mdronline.com/watch/watch_Issue.asp?Volname=Issue+%23013006&on=1 (last visited on 7th May, 2008)
- [cell02] Forbes, 2006: http://www.forbes.com/forbes/2006/0130/076.html (last visited on 7th May, 2008)
- [cell03] IEEE Spectrum Jan 2006: Winners & Losers 2006, IEEE Spectrum Jan 2006.pdf
- [cell04] Nicholas Blachford. 2005. Cell Architecture Explained: http://www.blachford.info/computer/Cell/Cell0_v2.html (last visited on 8th May, 2008)
- [cell05] IBM Developerworks Multicore: http://www.ibm.com/developerworks/power/cell/ (last visited on 15th, May, 2005)
- [cell06] A. Buttari, P. Luszczek, J. Kurzak , J. Dongarra, G. Bosilca . May 2007. A Rough Guide to Scientific Computing On the PlayStation 3. http://www.netlib.org/utk/people/JackDongarra/PAPERS/scop3.pdf
- [cell07] IBM, CBE_Programming_Tutorial_v3.0.pdf
- [z02] IBM Journal of Research and Development, Schwarz, E M, Check, M A, Shum, C-L K, Koehler, T, Et al, Jul/Sep 2002. Microarchitecture of the IBM eServer z900 processor. http://findarticles.com/p/articles/mi_qa3751/is_200207/ai_n9093756/pg_1 (last visited on 14 May, 2008)
- [z03] IBM Redbook, Paolo Bruni, Rama Naidoo, DB2 for OS/390 and Data Compression , Nov. 1998. SG245261.pdf

- [zlib01] Zlib library, http://www.zlib.net/ (last visited on 31th May, 2008)
- [zlib02] Antaeus Feldspar, An Explanation of the Deflate Algorithm, http://www.zlib.net/feldspar.html (last visited on 15th May, 2008)
- [zlib03] Zlib Technical Details, http://www.zlib.net/zlib_tech.html (last visited on 15. May 2008)
- [zlib04] Site of Prof. David A. Bader on Georgia Tech, http://www.cc.gatech.edu/~bader (last visited on 16. May, 2008)
- [zlib05] SourceForge.net, http://sourceforge.net/projects/cellbuzz (last visited on 15. May, 2008)
- [zlib06] D.A. Bader, V. Agarwal, K. Madduri, S. Kang, Sept. 2007. High performance combinatorial algorithm design on the Cell Broadband Engine processor . Paper HPC-Cell-ParCo2007.pdf
- [oth01] Henry Newman, Using Lib C and I/O and Performance, http://www.samag.com/documents/s=9365/sam0204h/0204h.htm (last visited on 19th May, 2008)
- [rfc1952] Gzip File Format Specification version 4.3, May 1996, http://www.faqs.org/rfcs/rfc1952.html (last visited on 19th May, 2008)
- [java01] Disk access time, http://en.wikipedia.org/wiki/Access_time (last visited on 16. July, 2008)
- [mea01] Huiyan Roy, 17th July, 2008. Performance Measurement General Information.odt
- [cell08] Roadrunner, http://www.top500.org/system/9485 (last visited on 16. July, 2008)
- [cell09] Cell Processor, http://en.wikipedia.org/wiki/Cell_%28microprocessor%29 (last visited on 16. July, 2008)
- [g01] J. A. Kahle, Cell Broadband Engine Architecture: kahle.pdf
- [g02] http://en.wikipedia.org/wiki/Taikodom
- [g03] Jochen Roth, IBM, Nov. 2007. Gameframe_4AcademicDays_20071106.ppt
- [g04] D.A. Bader, V. Agarwal, K. Madduri, S. Kang, Sept. 2007. High performance

combinatorial algorithm design on the Cell Broadband Engine processor . Paper HPC-Cell-ParCo2007.pdf

- [pf02] Huiyan Roy, 15th July, 2008. Java Compression Time.odt
- [pf03] Huiyan Roy, 15th July, 2008. Cell Supported Compression Measurement.odt
- [cell10] IBM Redbook, David Watts , Randall Davis , Ilia Kroutov, IBM BladeCenter Products and Technology, Feb. 2008. sg247523.pdf
- [cell11] IBM Redbook, Programming the Cell Broadband Engine Examples and Best Practices, Feb. 2008. sg247575.pdf

## A.2 Table Index

## A.3 Graph Index

## A.4 Abbreviations and Definitions

| | |
|---|---|
| Cell/B.E. | Cell Broadband Engine |
| CP | Central Processor, the general purpose processor of the System z |
| DMZ | Demilitarized Zone, a subnetwork staying between the untrusted network (usually Internet) and LAN. |
| EIB | Element Interconnect Bus |
| JVM | Java Virtual Maschine |
| GNU GPL | GNU General Public License, a widely used free software license |
| IDL | Interface Definition Language, a language that specifies the interface of a service component. It is widely used in RPC, CORBA, etc. |
| IFL | Integrated Facility for Linux, this is a System z processor dedicated for running Linux operating system. |
| LAN | Local Area Network |
| LZ1 | Lempel-Ziv 1, also called LZ77 |
| LZ2 | Lempel-Ziv 2, also called LZ78 |
| MIC | Memory Interface Controller |
| PPE | Power Processing Element |
| PPU | Power Processing Unit |
| RMI | Remote Method Invocation |
| RPC | Remote Procedure Call |
| RPM | Rotations per Minute |
| RTT | Round Trip Time, the elapsed time for the transmission of 64 byte data between a client and |

| | |
|---|---|
| | a server machine. |
| SPE | Synergistic Processing Element |
| SPU | Synergistic Processing Unit |
| STI | Sony, Toshiba, IBM |
| z/Linux | Also called Linux on System z, is the Linux operating system on the IBM System z. |
| z/OS | An operating system on the IBM mainframe. |
| z/VM | System z Virtual Machine operating system |