Peer-to-Peer Based Parallel Web Computing

Dissertation

by

Joachim Gehweiler

Fakultät für Elektrotechnik, Informatik und Mathematik

Universität Paderborn

Acknowledgements

First and foremost, I would like to thank my supervisor Prof. Friedhelm Meyer auf der Heide for his continuous support and for giving me the opportunity to choose the direction of my research. It was a pleasure to work in his research group. Furthermore, I wish to thank all (former) members of this research group with whom I worked together in such a nice atmosphere.

I am also grateful to Prof. Burkhard Monien and Ulf-Peter Schroeder with whom I collaborated in the EU integrated project AEOLUS in a very pleasant ambience. I also wish to thank my (former) colleagues with whom I closely collaborated: Olaf Bonorden, Christiane Lammersen, Henning Meyerhenke, Gunnar Schomaker, and Michael Thies.

For their continuous technical support, I would like to thank the members of the IRB, especially Ulrich Ahlers and Heinz-Georg Wassing.

Last but not least, I wish to thank my family and friends for their moral support and for making life so pleasant.

Contents

1

Introduction

In recent years volunteer, grid, and web computing have received considerable attention. Since PCs with a good Internet connection have become affordable for everybody, the world's computing power is distributed in — meanwhile — hundreds of millions of PCs all over the world. Most of these PCs are only partially utilized, so that the world-wide unused computing power easily sums up to hundreds of PetaFLOPs, whereas the fastest supercomputer currently has less than two PetaFLOPs. Even if only a few percent of the PC owners all over the world can be convinced to donate their unused computing power, this will already be competitive to the computing power of the top supercomputers. The idea of volunteer computing is utilize this immense amount of unused computing power for computing-intense applications. Famous examples are the *Great Internet Mersenne Prime Search (GIMPS)* [gim], the Internet's first general-purpose distributed computing project *distributed.net* [dis], or *Search for Extraterrestrial Intelligence (SETI@home)* [ACK+02, set]. Some years ago the *Berkeley Open Infrastructure for Network Computing (BOINC)* [boi, And04] was invented to provide a unified framework which greatly eases the implementation, software distribution to volunteers, and maintenance of such projects. BOINC currently has more than 300,000 volunteers who participate with more than 600,000 computers and donate more than five PetaFLOPs of computing power, which is more than 2.5 times the power of the fastest supercomputer.

Grid computing is another kind of distributed computing: Many trusted, networked computers are used to build a virtual supercomputer. These computers may be both supercomputers or large-scale clusters themselves, or ordinary desktop computers. In the latter case, these grid computing systems are referred to as desktop grids; a prominent example is the general-purpose experimental platform *XtremWeb* [xtr, FGNC01]. Though grids are usually also managed by a central master node, they are — in contrast to many volunteer computing systems — not dedicated to one special application, but they process various kinds

of applications of different users in batch mode.

Web computing combines aspects of both volunteer computing and grid computing: Like in the volunteer computing approach, web computing means to utilize the idle times on lots of PCs connected via the Internet to a virtual supercomputer. And like in the grid computing approach, web computing means to provide a general purpose middleware, where multiple users can run various kinds of applications.

Over time, many prominent distributed computing projects have added features to expand their application areas. Meanwhile, the terms "volunteer computing", "desktop grid", and "web computing" cover so many aspects and overlap so much, that some communities already use them synonymically. In the following, we use the term "web computing".

This thesis is about the web computing approach with two important constraints: First, we support the execution of coupled, massively parallel algorithms (rather than distributed data processing). And second, we organize the system in peer-to-peer (P2P) fashion.

1.1 Our Contribution

We present the *Paderborn University BSP-based Web Computing* (PUB-Web) library, which supports the execution of parallel programs in the bulk-synchronous style (BSP) on networked computers, utilizing only their idle times. This computer network is organized in P2P fashion and is dynamic not only with respect to the set of active peers, but also concerning the idle times continually changing on the particular machines in an unpredictable way. Since its first prototype implementation presented in [BGM05b, BGM06], PUB-Web has become a stable and mature system with several new features. In this thesis, we will focus on the following major building blocks:

- As the donated computing power on the particular host continually changes in an unpredictable fashion, we need to migrate threads to other (faster) hosts at runtime. And because computing nodes may crash or leave the network suddenly at any time, we need to create process state backups at regular times, which we can use to restore crashed processes. As JavaGo — the most promising, existing thread migration and checkpointing library for Java — only was a prototype implementation not suitable for production use and could not be extended to Java 5 or later versions, we designed a new language specification and developed a new library (PadMig) and compiler. In particular, using annotations instead of additional keywords, our new specification sticks to the Java standard instead

of deriving a new programming language. As a side effect, this allows developers to keep using their favorite IDEs without any drawbacks. Furthermore, we were able to design PadMig such that developers do not need to maintain two versions if they like to have a non-migratable and a migratable version of their code — they can now simply skip the intermediate compilation step with our compiler to obtain a non-migratable version of their code. Finally, we based our implementation solely on tools in Java for portability reasons, so that PadMig is available for all important platforms.

- Once we were able to migrate threads, the key challenge was to find a suitable distributed load balancing technique. For this, we adapted a well-established decentralized data distribution method using distributed heterogeneous hash-tables from the storage community to our setting. This new load balancer is able to fairly assign the processes of parallel programs to computing nodes and to balance the load on changes in the available computing power. The nodes running this distributed algorithm need to continually exchange information about the available computing nodes, but not about the running processes, i.e., each process can be scheduled without the knowledge of other processes. This additionally guarantees that, in case of a crash, the schedule can be reconstructed by an arbitrary node, which did not collect any information in advance. Although not all features are implemented, our implementation is stable and suitable for production use. In order to adequately judge the quality of the schedules produced by our load balancer, we performed extensive experiments. At first, we collected the utilization data of more than 250 PCs for a period of several months in order to both feed our load balancer with realistic input data and conduct reproducible experiments. We then performed different kinds of experiments to separate the influences of the external work load and the stream of parallel jobs to execute, and we compared several variants of our load balancer with the well-established Work Stealing algorithm.

- Despite the fact that the available computing power is the most important resource to fairly share among all parallel processes, there are also other criteria to consider such as the network bandwidth, for example, as BSP programs communicate a lot. If a BSP program would be scheduled entirely on a fast connected component of the network although there would be a few faster processors available outside this component, it will run faster due to the reduced networking delays than it would have run when just scheduled according to processing power. Thus, another

challenge was to cluster the PUB-Web network according to bandwidth. For this, we employed a novel, fault-tolerant, adaptive, and scaling distributed clustering algorithm called DiDiC. Although DiDiC is not yet integrated into PUB-Web, we experimentally compared DiDiC to the well-established MCL algorithm using a simulator.

1.2 Bibliographic Notes

Most of the results in this thesis have been presented and published in a preliminary version in conference proceedings, journals, or as technical reports. In particular, the first (hybrid peer-to-peer) version of PUB-Web has been published in:

[BGM05b] Olaf Bonorden, Joachim Gehweiler, and Friedhelm Meyer auf der Heide. A web computing environment for parallel algorithms in Java. In *Proceeedings of International Conference on Parallel Processing and Applied Mathematics (PPAM)*, pages 801–808, Poznan, Poland, 2005.

[BGM06] Olaf Bonorden, Joachim Gehweiler, and Friedhelm Meyer auf der Heide. A web computing environment for parallel algorithms in Java. *Scalable Computing: Practice and Experience*, 7(2):1–14, 2006.

The new thread migration and checkpointing library and its compiler have been presented in:

[GT10] Joachim Gehweiler and Michael Thies. Thread migration and checkpointing in Java. Technical Report tr-ri-10-315, Heinz-Nixdorf-Institute, June 2010.

The architecture of the load balancer has been proposed in:

[GS06] Joachim Gehweiler and Gunnar Schomaker. Distributed load balancing in heterogeneous peer-to-peer networks for web computing libraries. In *Proceeedings of 10th IEEE/ACM International Symposium on Distributed Simulation and Real Time Applications (DS-RT)*, pages 51–58, Torremolinos, Malaga, Spain, 2006.

An experimental comparison of different load balancing strategies is currently under submission:

[GM11] Joachim Gehweiler and Friedhelm Meyer auf der Heide. An experimental comparison of load balancing strategies in a web computing environment. In *SPAA '11: Proceedings of the 23rd ACM Symposium on Parallelism in Algorithms and Architectures*, under submission, 2011.

The distributed clustering algorithm has been published in:

[GM10] Joachim Gehweiler and Henning Meyerhenke. A distributed diffusive heuristic for clustering a virtual P2P supercomputer. In *Proceeedings of 24th International Parallel and Distributed Processing Symposium (IPDPS, HPGC)*, Atlanta, USA, 2010.

Finally, the large-scale distributed environment used for running and debugging PUB-Web has been presented in:

[GMS10] Joachim Gehweiler, Friedhelm Meyer auf der Heide, and Ulf-Peter Schroeder. A large-scale distributed environment for peer-to-peer services. Technical Report tr-ri-10-317, Heinz-Nixdorf-Institute, June 2010.

1.3 Organization of the Thesis

In the next chapter, we discuss the computing model and present the architecture and some basic features of the web computing library. In Chapter 3, we describe technical aspects of the web computing library and its development, with a strong focus on the new thread migration and checkpointing library its compiler. Our load balancing algorithms and their analysis are discussed in Chapter 4. In Chapter 5, we present the clustering algorithm and its evaluation. Concluding remarks and a look ahead are given in Chapter 6. Finally, the appendix provides detailed results of our extensive experiments in numerous additional figures. Related work on the different topics is discussed in the particular sections.

2

The Web Computing Library

In this chapter we discuss architectural aspects of the PUB-Web library. Distinguishing characteristics of PUB-Web are its ability to support the execution of coupled, massively parallel algorithms (rather than distributed data processing), and the fact that this web computing system is organized in P2P fashion. Thus, we first discuss the parallel computing model and compare against related work. Then we present the architecture, some core features, the application programming interface (API), and the interoperability interface of PUB-Web.

2.1 The Bulk-Synchronous Parallel Model

In order to handle communication, synchronization, and dependencies between the processes of a parallel program in a very heterogeneous computing environment, we restrict the parallel applications to a round-based model with computing, communication, and synchronization phases. Rather than inventing a new variant, we stick to the well-established *Bulk-Synchronous Parallel (BSP)* model [Val90, Bis04], which has been introduced by Leslie G. Valiant in order to simplify the development of parallel algorithms. It forms a bridge between the hardware to use and the software to develop by giving the developer an abstract view of the technical structure and the communication features of the hardware to use (e.g., a supercomputer with shared memory, a cluster of workstations or PCs connected via the Internet).

A *BSP computer* is defined as a set of processors with local memory, interconnected by a communication mechanism (e.g., a network or shared memory)

capable of point-to-point communication, and a barrier synchronization mechanism.

A *BSP program* consists of a set of *BSP processes* and a sequence of *supersteps* — time intervals bounded by the barrier synchronization. Within a superstep each process performs local computations and sends messages to other processes; afterwards it indicates by calling the `sync()` method that it is ready for the barrier synchronization. When all processes have invoked the `sync()` method and all messages are delivered, the next superstep begins and the messages sent during the previous superstep can be accessed by its recipients. Fig. 2.1 illustrates this.

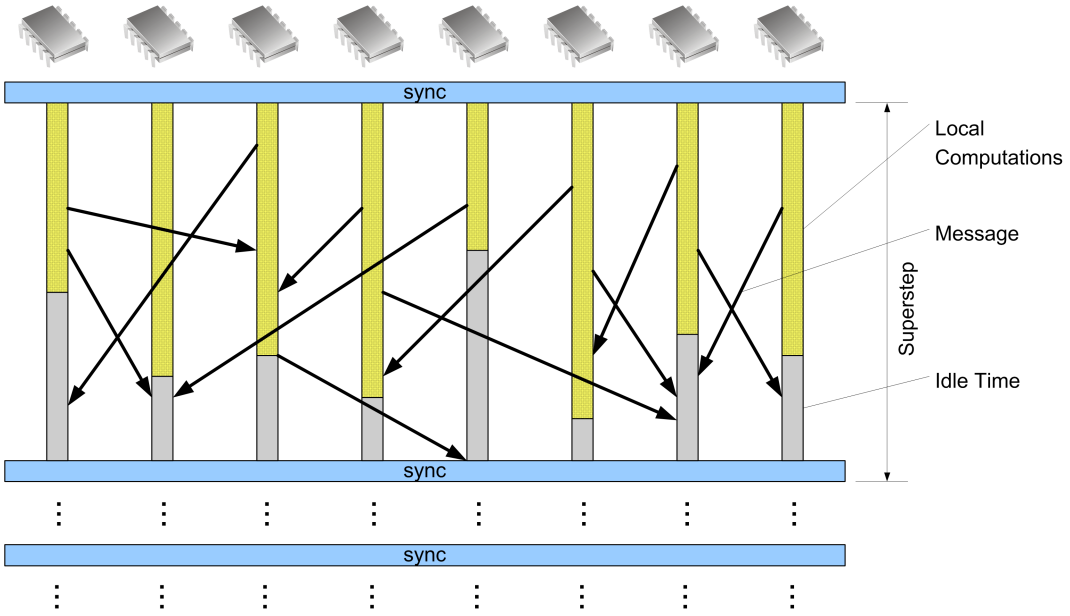

Figure 2.1: *A superstep in a parallel program consisting of 8 processes.*

2.2 Related Work

Well-known BSP implementations are the Oxford *BSP programming library (BSPlib)* [HMS+98] and the *Paderborn University BSP (PUB)* library [puba, BJvOR03]. The *Bayanihan* BSP implementation [Sar99] is a first attempt to support BSP programs in a volunteer computing context: The central master node decomposes the BSP program to be executed into pieces of work, each consisting of one superstep in one BSP process. The worker nodes download a work package

consisting of the current process state of one BSP process and its incoming messages for the current superstep, execute the superstep, and send the resulting state together with the outgoing messages back to the master. When the master has received the results of the current superstep for all BSP processes, it moves the messages to their destination work packages. Then the workers continue with the next superstep. With this approach all communication between the BSP processes passes though the server — additionally to the overhead generated by starting / stopping the processes and saving / restoring their process states.

Our approach removes this bottle-neck at the master node: Organized as a P2P network, all computing nodes communicate directly with each other; supernodes are only involved when a computing node has to look up another computing node before their first interaction, or when an error occurs. In order to handle varying donations of computing power in a dynamic P2P environment, we have developed novel approaches to distributed load balancing and clustering.

2.3 The Architecture of PUB-Web

Distinguishing architectural aspects of our *Paderborn University BSP-based Web Computing (PUB-Web)* library are its support for parallel programs in the BSP style and its P2P structure. Though PUB-Web was only a hybrid P2P network in its first prototype version [BGM05b, BGM06], using a central server for load balancing, user management, etc., it is now designed as a pure P2P system, consisting of a dynamically changing set of maybe worldwide distributed computers temporarily donated to be used for web computing; in addition, a few supernodes are employed for the management of the system (cf. Fig. 2.2). Though PUB-Web is a stable and mature system, ready for production use, not all features discussed in this thesis are fully implemented yet.

PUB-Web only utilizes the left-over computing power on the donated computers, in order not to disturb other activities on these machines. The donated computers may be very different (e.g., desktop PCs, notebooks, etc.), both with respect to their computing power and the dynamics of their availability, i.e., their left-over computing power. Note that the availability depends on the way the computers are used by other activities of their users. This might be very different for different computers. For the remainder of this thesis we refer to these other activities as *external workload*. PUB-Web does not require any guaranties on the availability of the peers, i.e., the donated computers may be switched on and off at arbitrary times, may crash, and may be subject to an arbitrary external workload changing in an unpredictable fashion.

Basically, everybody can join the PUB-Web network with his computers: In order to just donate the unused computing power, it is sufficient to download and install the peer software available at [pubb]; in order to also run your own BSP programs, a login for the PUB-Web network is required.

When logging in to the PUB-Web network, a peer searches for an arbitrary trusted supernode and sends a login request. As the network is dynamic and peers may leave the network without signing off (e.g., in case a computer crashes), the login requests are granted on a lease base, i.e., in order to stay valid, a login has to be renewed at regular intervals. There is no upper bound on the lifetime of such a lease.

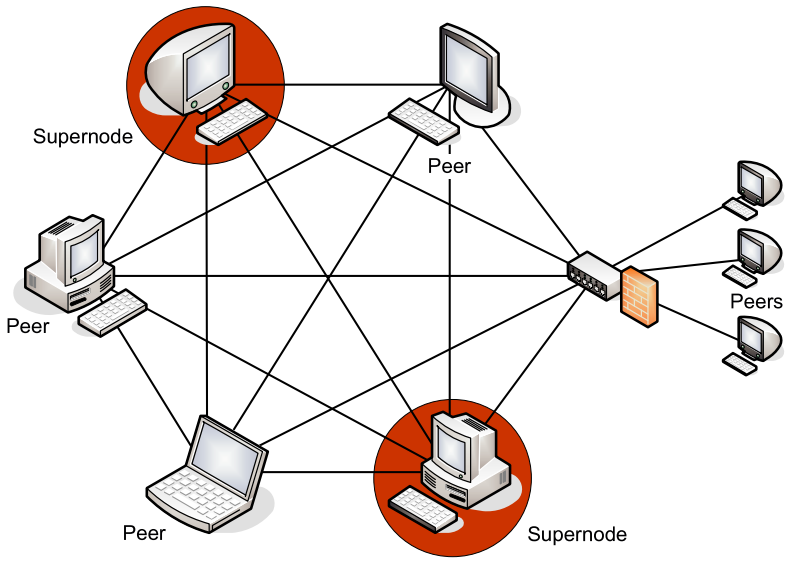

Figure 2.2: *The network architecture of PUB-Web.*

2.3.1 Running Parallel Programs

When a user wants to run a BSP program, it has to be copied into a special directory specified in the configuration file. After the user has provided the required information, e.g., the name of the program and the requested number n of parallel processes, his peer contacts a supernode in order to request a set of up to n peers, where to run the program (cf. Fig. 2.3). The supernode then selects an appropriate subset of the P2P network; it may return less then n peers because the P2P network is very heterogeneous with respect to the available computing power and there may exist peers that are more than twice as fast as other peers. From now on, the execution of the parallel program is supervised

by the user's peer. On each of the assigned peers a PUB-Web runtime environment is started and the user's parallel program is obtained via dynamic code downloading. The output of the parallel program and, possibly, error messages including stack traces are forwarded to the user's peer.

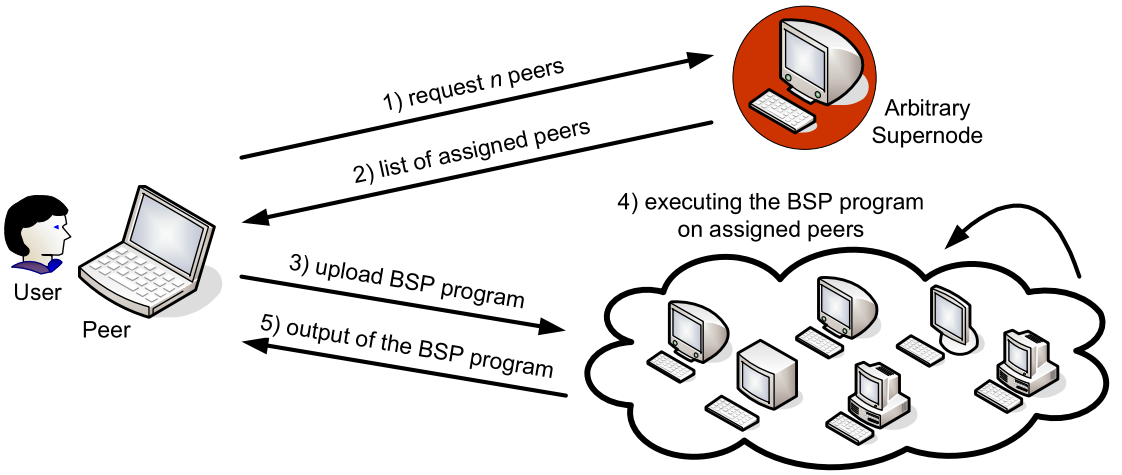

Figure 2.3: *Executing a parallel program in PUB-Web.*

2.3.2 Core Features

Since the computing power available on the assigned peers continually changes depending on how intensively the people, who donate their unused computing power, currently utilize their computers, the supernode's prediction where to optimally schedule the BSP processes may appear to be wrong after some time. Fig. 2.4 illustrates that the whole BSP program is delayed if only one peer does not provide the expected amount of computing power. Thus, we migrate such a BSP process to another, faster peer.

As not only the available computing power is dynamic, but also the P2P network itself, i.e., peers may disappear out of a sudden, PUB-Web creates backup copies of the process states during each synchronization phase and stores them at different nodes across the network. Thus, we are able to restore processes of a BSP program on-the-fly.

Because the computers in our scenario are not only highly heterogeneous with respect to its hardware, but also run various types and versions of operating systems, PUB-Web needs to be platform independent. The choice of Java does not only fulfil this requirement, but also provides a basic security

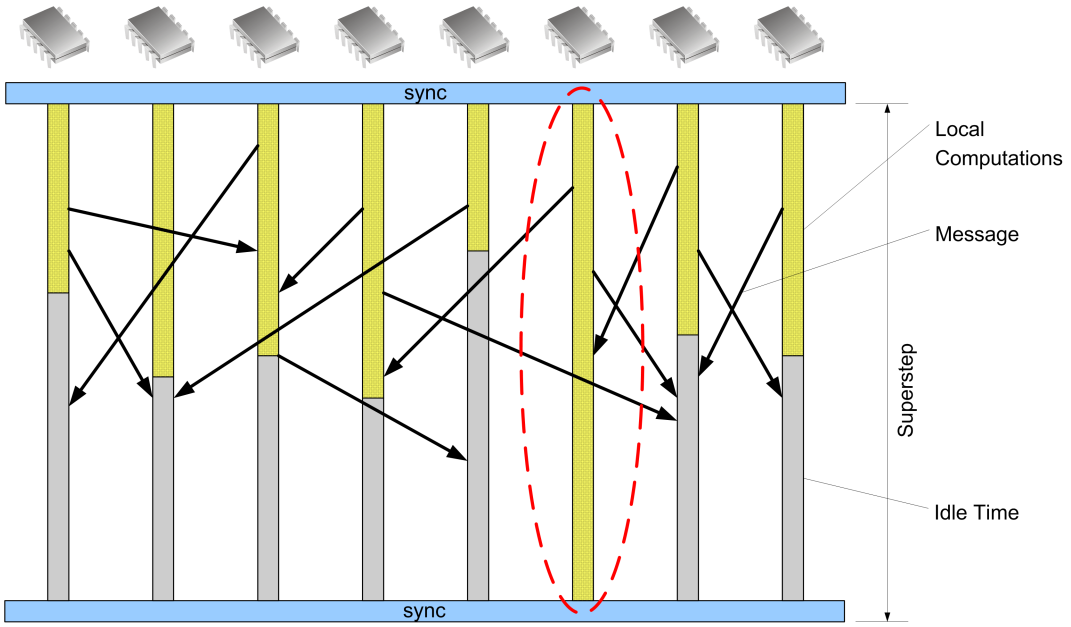

Figure 2.4: *BSP program delayed by one slow process.*

model out of the box: The *Java Sandbox* allows to grant code specific permissions depending on its origin. For example, access to (part of) the file system or network can be denied. In order to guarantee a high level of security, we grant user programs only read access to a few Java properties which are needed to write completely platform independent code (e.g., `line.separator` etc.).

2.3.3 The PUB-Web API

User programs intended to run on PUB-Web have to be BSP programs ([Bis04] is an excellent guide to parallel scientific computation using BSP). Thereto the interface `BSPProgram` must be implemented, i.e., the program must have a method with this signature:

```
public void bspMain(BSP bspLib, Serializable args)
    throws AbortedException
```

Its first parameter is a reference to the PUB-Web runtime environment in order access BSP API methods; the second parameter holds the arguments passed to the BSP program, which is a `String[]` if the BSP program is started from a command prompt, or any serializable Java object in case the interoperability API (cf. Section 2.3.4) is used.

In order to enable BSP programs to be migrated at runtime by the load balancer, they need to be compiled using the PadMig compiler (cf. Section 3.2) and need to implement the `BSPMigratableProgram` interface instead, which means that the main method has this different signature:

```
@Migratory public void bspMain(BSPMigratable bspLib,
    Serializable args) throws AbortedException
```

In the following, we discuss the BSP library functions which can be accessed via the `BSP` and `BSPMigratable` interface, respectively, which is implemented by the PUB-Web runtime environment. In non-migratory programs, the barrier synchronization is entered by calling:

```
public void sync()
```

The migratory version additionally creates a backup copy of the execution state and performs migrations if suggested by the load balancer:

```
@Migratory public void syncMig()
```

A message, which can be any serializable Java object, can be sent with these methods; thereby the latter two methods are for broadcasting a message to an interval and an arbitrary subset of the BSP processes, respectively:

```
public void send(int to, Serializable msg)
    throws IntegrityException
public void send(int pidLow, int pidHigh,
    Serializable msg) throws IntegrityException
public void send(int[] pids, Serializable msg)
    throws IntegrityException
```

Messages sent in the previous superstep can be accessed with these methods, where the `find*` methods are for accessing messages of a specific sender:

```
public int getNumberOfMessages()
public Message getMessage(int index)
    throws IntegrityException
public Message[] getAllMessages()
public Message findMessage(int src, int index)
    throws IntegrityException
public Message[] findAllMessages(int src)
    throws IntegrityException
```

When receiving a message, it is encapsulated in a `Message` object. The message itself as well as the sender ID can get obtained with these methods:

```
public Serializable getContent()
public int getSource()
```

In order to terminate all the processes of a BSP program, e.g. in case of an error, the following method has to be called; the `Throwable` parameter will be transmitted to the PUB-Web user who has started the program:

```
public void abort(Throwable cause)
    throws AbortedException
```

Any output to *stdout* or *stderr* should be printed using the following methods as they display it on the command prompt of the user who has started the BSP program rather than on the computer where the processes are actually running:

```
public void printStdOut(String line)
public void printStdErr(String line)
```

Beside writing to *stdout* and *stderr*, BSP programs can output any serializable Java object, which is especially useful when a BSP program is started using the interoperability API (cf. Section 2.3.4):

```
public void writeRawData(Serializable data)
```

By default the output is sent back to the user asynchronously in the background so that these methods immediately return, even if delivering the output takes some time due to network delays. However, in some cases it might be necessary to ensure that the output is delivered before proceeding, e.g., before a program is terminated using the `abort()` method. This can be achieved using this method:

```
public void flush()
```

To access data from files, the following method should be used. In particular, any file in the BSP program folder of the PUB-Web user's peer can be read with it:

```
public InputStream getResourceAsStream(String name)
    throws IOException, MalformedURLException
```

In migratable programs there is also a method available which may be called to mark additional points inside long supersteps where a migration is safe (i. e. no open files etc.):

```
@Migratory public boolean mayMigrate()
```

Furthermore, there are some service functions to obtain the number of processes of the BSP program, the own process ID, and so on.

Listing 2.1 shows an example program which demonstrates the basic BSP features, especially how to send and receive messages.

Listing 2.1: *Example program demonstrating message passing.*

```
1  import pubweb.*;
2  import pubweb.bsp.*;
3
4  public class MessagePassing implements BSPProgram {
5
6    public void bspMain(BSP bsp, Serializable args) throws
       AbortedException {
7      // calculate neighbours
8      int pid = bsp.getProcessId();
9      int n = bsp.getNumberOfProcessors();
10     int left = (pid - 1 + n) % n;
11     int right = (pid + 1) % n;
12
13     try {
14       bsp.send(left, new Integer(1));
15       bsp.send(right, new Integer(2));
16     } catch (IntegrityException ie) {
17       bsp.printStdErr("sending failed: " + ie.getMessage());
18     }
19
20     bsp.sync();
21
22     // get all my messages, method 1
23     n = bsp.getNumberOfMessages();
24     for (int i = 0; i < n; i++) {
25       try {
26         Message msg = bsp.getMessage(i);
27         bsp.printStdOut("got " + msg.getContent() + " from pid " +
             msg.getSource() + " in superstep " + msg.getSuperstep());
28       } catch (IntegrityException ie) {
29         bsp.printStdErr("receiving failed: " + ie.getMessage());
30       }
31     }
32
33     // get all my messages, method 2
34     Message[] msgs = bsp.getAllMessages();
35     bsp.printStdOut("got in total " + msgs.length + " messages");
36
37     // get messages from some specified pid, method 1
38     try {
39       int i = 0;
40       Message msg;
```

```
41        while ((msg = bsp.findMessage(0, i++)) != null) {
42          bsp.printStdOut("received " + msg.getContent() + " from pid 0
              in superstep " + msg.getSuperstep());
43        }
44      } catch (IntegrityException ie) {
45        bsp.printStdErr("receiving failed: " + ie.getMessage());
46      }
47
48      // get messages from some specified pid, method 2
49      try {
50        msgs = bsp.findAllMessages(0);
51        bsp.printStdOut("got " + msgs.length + " messages from pid 0");
52      } catch (IntegrityException ie) {
53        bsp.printStdErr("receiving failed: " + ie.getMessage());
54      }
55    }
56  }
```

2.3.4 The Interoperability Interface

A BSP program needs not necessarily be launched by a user. Using the interoperability interface, BSP programs can be started out of other stand-alone software just like an (asynchronous) function call: the PUB-Web peer software can be embedded in GUI-less mode into other applications using the interoperability API. Amongst others, the API provides a function to start a BSP program and an interface for callback-functions to receive the output and, possibly, error messages of the parallel program. Parameters passed to the BSP program as well as the output sent back are not restricted to strings, but can be any (serializable) object.

In particular, one first needs to create an instance of a PUB-Web peer in consumer mode:

```
Consumer consumer = new Consumer(
    "/path-to/config-file.conf")
```

BSP programs are started using the following method, where `desc` is the description of the BSP program appearing in the process list, `nProcs` is the number of requested BSP processes, `mainClass` is the fully qualified name of the BSP program's main class, and `progArgs` can any serializable Java object to pass as argument to the BSP program:

```
public Job newJob(String desc, int nProcs,
    String mainClass, Serializable progArgs)
    throws ClassNotFoundException, IntegrityException,
```

```
    InternalException, MalformedURLException,
    NotConnectedException, NotEnoughWorkersException,
    PpException
```

In order to asynchronously receive the output and status updates of the BSP program, an event listener is required, which can be added / removed using these methods:

```
public void addConsumerEventListener(
    ConsumerEventListener cel)
public void removeConsumerEventListener(
    ConsumerEventListener cel)
```

The event handler interface contains methods to receive output and status messages of BSP programs:

```
public void printToStdOut(Job job, int pid,
    String line)
public void printToStdErr(Job job, int pid,
    String line)
public void writeRawData(Job job, int pid,
    Serializable data)
public void printStatusLine(Job job, int pid,
    String line)
public void processExited(Job job, int pid)
public void jobExited(Job job)
public void processRolledBack(Job job, int pid,
    int superstep)
public void jobDiedOnError(Job job, int pid,
    Throwable cause)
public void jobAborted(Job job, int pid,
    Throwable cause)
public void jobListChanged()
```

To keep track of pending, running, and finished jobs and to kill running or dispose finished jobs, the following methods of a `Consumer` instance be can used:

```
public synchronized Job[] getWaitingJobs()
public synchronized Job[] getActiveJobs()
public synchronized Job[] getFinishedJobs()
public synchronized void updateJobList()
    throws PpException, NotConnectedException
```

```
public synchronized void killJob(Job job)
    throws PpException, IntegrityException,
    InternalException, NotConnectedException
public synchronized void disposeJob(Job job)
    throws IntegrityException
```

Listing 2.2 shows a minimalistic example how to run a BSP program using the interoperability interface.

Listing 2.2: *Running a BSP program using the interoperability interface.*

```
1  import padrmi.*;
2  import pubweb.*;
3  import pubweb.user.*;
4
5  public class IopExample implements ConsumerEventListener {
6
7    private Consumer consumer;
8
9    public IopExample() {
10     try {
11       consumer = new Consumer("/path-to/config-file.conf");
12       consumer.addConsumerEventListener(this);
13       consumer.newJob("Example", 16, "myPackage.MyBspProgram", null);
14     } catch (ClassNotFoundException e) {
15       System.err.println("BSP program not found: " + e.getMessage());
16     } catch (IntegrityException e) {
17       System.err.println("integrity violated: " + e.getMessage());
18     } catch (NotConnectedException e) {
19       System.err.println("no supernode connection available: " + e.
           getMessage());
20     } catch (NotEnoughWorkersException e) {
21       System.err.println("not enough peers available: " + e.
           getMessage());
22     } catch (PpAuthorizationException e) {
23       System.err.println("authentication failed: " + e.getMessage());
24     } catch (Exception e) {
25       System.err.println("operation failed:");
26       e.printStackTrace();
27     }
28   }
29
30   public void printToStdOut(Job job, int pid, String line) {
31     System.out.println(job + "[" + pid + "]: " + line);
32   }
33
34   public void printToStdErr(Job job, int pid, String line) {
35     System.err.println(job + "[" + pid + "]: " + line);
36   }
```

```
37
38    public void writeRawData(Job job, int pid, Serializable data) {
39      System.out.println(job + "[" + pid + "]: data: " + data);
40    }
41
42    public void printStatusLine(Job job, int pid, String line) {
43      System.out.println(job + "[" + pid + "]: status: " + line);
44    }
45
46    public void processExited(Job job, int pid) {
47      System.out.println("process " + job + "[" + pid + "] completed");
48    }
49
50    public void jobExited(Job job) {
51      System.out.println("job " + job + " completed");
52    }
53
54    public void processRolledBack(Job job, int pid, int superstep) {
55      System.out.println("process " + job + "[" + pid + "] restored in
          superstep " + superstep);
56    }
57
58    public void jobDiedOnError(Job job, int pid, Throwable cause) {
59      System.err.println("job " + job + " crashed at pid " + pid + "
          because of:" + cause.getMessage());
60      try { consumer.killJob(job); } catch (Exception any) {}
61    }
62
63    public void jobAborted(Job job, int pid, Throwable cause) {
64      System.err.println("job " + job + " aborted at pid " + pid + "
          because of:" + cause.getMessage());
65      try { consumer.killJob(job); } catch (Exception any) {}
66    }
67
68    public void jobListChanged() {
69    }
70  }
```

3

Technical Aspects

In this chapter, we discuss essential technical aspects of PUB-Web: In the first section, we present our communication library PadRMI, which is a lightweight replacement for Java RMI. Its development became necessary due to drawbacks of Java RMI. Though PadRMI is tailored to the needs of PUB-Web, we designed it in the form of a stand-alone module as a general purpose communication library. In Section 3.2, we present our thread migration and checkpointing library PadMig including its compiler, describing its architecture, language specification, technical background, and internals. Like PadRMI, PadMig is a stand-alone general purpose library. In Section 3.3, we briefly discuss security and trust mechanisms. Finally, we present our large-scale distributed environment used for running and debugging PUB-Web in Section 3.4. This distributed environment is a general purpose large-scale testbed for P2P software written in Java using the JXTA framework.

3.1 The Communication Library

To enable computers in a P2P network to locate each other and to communicate with each other is a non-trivial task. In order to comply with established standards (and for interoperability reasons within the EU FP6-IST project "Algorithmic Principles for Building Efficient Overlay Computers" (AEOLUS)), PUB-Web uses JXTA [jxt, Gon01] to locate peers and establish the first contact between two peers. Once two peers know each other, they continue their communication for efficiency reasons using direct TCP connections.

As PUB-Web spawns a special runtime environment in a separate process for each BSP process due to security and fairness reasons, one server socket is necessary per runtime environment. But since PUB-Web must be able to run behind firewalls and routers, possibly using network address translation (NAT), with

a reasonably amount of (network) resources, a PUB-Web peer cannot request one open TCP port per BSP process; instead, it has to act as a proxy for its child runtime environments.

In the first prototype implementation of PUB-Web, we employed standard Java RMI for remote method invocations. Unfortunately, Java RMI does not support high-level routing. Thus, in order to "route" a remote method invocation in a peer to its correct child runtime environment, the peer must provide a corresponding remotely invokable method that passes the request on to the particular child runtime environment. In addition to the implementation overhead, this involves a deserialization and re-serialization overhead for each remote method call. In order to overcome this drawback, we invented a lightweight replacement for Java RMI that supports port forwarding: the *Paderborn Remote Method Invocation* (PadRMI) library [padb].

During the following description of PadRMI, we refer to a peer invoking a remote method as *client*, and to the peer whose method is remotely invoked as *server*.

PadRMI communicates using TCP connections and its own text-based protocol: the *PadRMI Protocol* (`pp://`). In order for this to work, the Java VM property `java.protocol.handler.pkgs` has to be set to the value `padrmi`. Internally, PadRMI uses asynchronous messages for communication; for this, an instance of the PadRMI server has to run at both the server and client side.

Like in Java RMI, the server's remotely invokable methods must be defined in a special interface: it has to extend the `padrmi.PpRemote` interface, and all its methods must declare the `padrmi.PpException` (cf. Listing 3.1).

Listing 3.1: *Example for a PadRMI remote interface.*

```
1 import padrmi.*;
2 import padrmi.exception.*;
3
4 public interface MyRemoteService extends PpRemote {
5   public Object computeSomething(Object input) throws PpException;
6 }
```

In contrast to Java RMI, PadRMI does not support remote objects, so the implementation of the server only requires the implementation of the remote interface and its registration at the PadRMI server (cf. Listing 3.2). The description "MyRemoteService" needs to be a locally unique identifier. Username and password are optional and may be `null` in order to be disabled.

Listing 3.2: *Example for a PadRMI remote interface implementation.*

```
1  import padrmi.*;
2  import padrmi.exception.*;
3
4  public class MyRemoteServiceImpl implements MyRemoteService {
5    public Object computeSomething(Object input) throws PpException; {
6      // ...
7    }
8
9    // ...
10
11   public static void main(String[] args) {
12     Server.startDefaultServer();
13     MyRemoteServiceImpl impl = new MyRemoteServiceImpl();
14     Server.getDefaultServer().addObject("MyRemoteService", impl,
         MyRemoteServiceImpl.class, "user", "password");
15   }
16 }
```

The PadRMI server needs a few properties to be set correctly in order to start: `padrmi.guid` must be a globally unique identifier (GUID) string. `padrmi.bind.address` and `padrmi.bind.port` are the IP address and TCP port to be used, respectively. When using NAT, the router's IP address and its forwarding port can be specified as `padrmi.address` and `padrmi.port`, respectively.

In contrast to Java RMI, no stubs and skeletons need to be generated. Instead, PadRMI internally uses `java.lang.reflect.Proxy` to create local proxy objects at the client side. Listing 3.3 illustrates an invocation of the method defined in Listing 3.1.

Listing 3.3: *Example for a PadRMI remote interface.*

```
1  import java.net.*;
2  import padrmi.*;
3  import padrmi.exception.*;
4
5  public class MyRemoteMethodInvocation {
6    public static void main(String[] args) {
7      Server.startDefaultServer();
8      URL url = new URL("pp://user:password@there:1234/itsguid/
         MyRemoteService");
9      MyRemoteService service = (MyRemoteService) Server.
         getDefaultServer().getProxyFactory().createProxy(url,
         MyRemoteService.class);
10     Object result = service.computeSomething(args[0]);
11   }
12 }
```

A distinguishing feature of PadRMI is its support for routing; all messages (i.e., serialized remote method invocations or requested resources) are routed using their GUIDs according to the routes defined. Listing 3.4 shows how to set a route for GUID "myguid" to `there:1234`.

Listing 3.4: *Example for setting a PadRMI route.*

```
1 import padrmi.*;
2
3 // ...
4
5 Server.getDefaultServer().addRoute("myguid", "there", 1234);
```

The PadRMI server can also be used to host class files and other resources for downloading (e.g., as a remote codebase). The base directory, where the resources are located, needs to be specified via the `padrmi.path` property; also multiple directories can be supplied, separated by the platform-specific path separator character. The files can be accessed on the client side for dynamic code downloading or as shown in Listing 3.5.

Listing 3.5: *Example for resource downloading in PadRMI.*

```
1 import java.net.*;
2 import padrmi.*;
3
4 // ...
5
6 Server.startDefaultServer();
7 URL url = new URL("pp://there:1234/itsguid/subdirectory/file.txt");
8 url.openStream();
```

Further features and limitations of PadRMI include: only serializable arguments and results are allowed; for these, standard Java serialization is employed. Passing references as arguments or results is not supported. Methods are matched only by their names and number of arguments, but not by their argument and return types. PadRMI is compatible with the standard Java classloader. The remote method calls are in a different protection domain with restricted privileges and can be timeouted using annotations.

3.2 Thread Migration and Checkpointing in Java

As already pointed out in Chapter 2, it is necessary to migrate threads of lengthy calculations at runtime to other hosts because the donated computing power continually changes in an unpredictable fashion and hosts may even become unavailable. Furthermore, it is desirable to regularly create checkpoints of the

execution state, so that a certain state of a thread can be restored in case of a system crash rather than restarting all the calculations from the beginning.

There are three ways to migrate threads in Java: modification of the Java Virtual Machine (VM) [MWL00], bytecode transformations [SSY00, TRV+00], and sourcecode transformations [jav, Fün98]. Modifying the Java VM is out of the question because everybody would have to replace his installation of the original Java VM with one from a third party, just to run a migratable Java program. Approaches of this kind do not only have limited success due to their installation overhead, but also because of trust matters: people would need to trust that a third party VM does not have any security defects. Additionally, from the developers' point of view, this approach would result in a lot of maintenance work to adapt all future releases and updates of Sun's VM. An obvious alternative to modifying Sun's VM is of course to develop an own VM, but this results in even more implementation work. A quite well-known approach of this kind is the Jikes Research VM, which provides — among other features — thread migration techniques [CLQ06]; but although a lot of man-power has been spent into this project, it is not suitable for production use.

The bytecode transformation approach is also less suitable in our case because we would need to re-synthesize high-level constructs such as loops or `try-catch-finally` blocks for our translation approach. Additionally, a bytecode transformer should be able to deal with all possible bytecode constructs, not only those found in well-shaped *javac* output, which means additional effort for the development of such a compiler.

Thus, we use the sourcecode transformation approach inside PUB-Web. There are two ways to accomplish this: using code unfolding [SMY99] or an artificial program counter [Fün98]. Using the former approach, nested loops and branches have to be unfolded, whereas additional code fragments have to be inserted for each statement to check whether or not the statement has to be skipped in the latter approach. Obviously, unfolding needs only be done if there are migratory sub-statements; similarly, successive statements to be skipped can be grouped in case there are no migratory statements in between. We started using a very promising prototype implementation [jav] of the unfolding technique, called *JavaGo*, which extends the Java programming language with three keywords: migrations are performed using the keyword `go` (passing a filename instead of a hostname as parameter creates a backup copy of the execution state). All methods, inside which a migration may take place, have to be declared `migratory`. The depth, up to which the call stack will be migrated, can be bounded using the `undock` statement.

The translation of this extended language into Java sourcecode is done using the JavaGo compiler *jgoc*. Migratable programs have a special `main` method

and are launched via a wrapper class. In order to continue the execution of a migratable program, an instance of a migration server has to run on the destination host.

But, unfortunately, this prototype implementation was not only incompatible with PadRMI, but could also not be extended to support Java 5 due to design issues. Thus and because of the following two more reasons, we decided to start from scratch with our own implementation, the *Paderborn Thread Migration and Checkpointing (PadMig)* library [pada, GT10]:

- Using annotations instead of additional keywords, we can stick to the Java standard instead of deriving a new programming language. As a side effect, this allows developers to keep using their favorite IDEs without any drawbacks. Furthermore, we are able to design PadMig such that developers do not need to maintain two versions if they like to have a non-migratable and a migratable version of their code — they can simply skip the intermediate compilation step with our *migc* compiler to obtain a non-migratable version of their code.

- The prototype implementation of the JavaGo compiler was more or less a quick-hack written in Objective Caml [cam], which was terribly slow, did not produce useful error messages, was hard to debug and not available for all important platforms. For portability reasons we based our new implementation solely on tools in Java; in particular, we use the Java transformation framework Spoon [spo, Paw05], which provides a complete model of the abstract syntax tree where any element can be accessed both for reading and modification.

The PadMig API consists of special functions to initiate a migration to another machine or to save a checkpoint into a file. All methods, inside which a migration can occur or a checkpoint is created, need to be annotated. The PadMig compiler then transforms this code into migratable code. As our implementation does not modify the Java language, the original, non-migratable code is fully functional Java code, which just produces a warning rather than actually migrating.

In the following sections we describe our language specification, provide the technical background, give insight into the translation concepts, and finally evaluate our new approach.

3.2.1 The PadMig Language Specification

In order to migrate the calling thread or to create a checkpoint, migration points have to be inserted into a program. Only there — at statements consisting of a

call to the `migrate()` or `checkpoint()` method of the library class `padmig.PadMig` — the calling thread is migrated to a remote host or a checkpoint of the calling thread is created, respectively. In particular, the two methods have the following signatures:

```
public static void migrate(
    java.net.URL migrationServer)
    throws padmig.MigrationException;
public static void checkpoint(
    java.io.File backupFile)
    throws padmig.MigrationException;
```

When a checkpoint is generated, the execution continues locally. In case the migration or checkpointing fails, a `MigrationException` will be thrown.

In order to allow migrations or checkpoints inside a method, either directly by calling the `migrate()` or `checkpoint()` method, or transitively by calling some other migratable method, the particular method has to be annotated with `padmig.Migratory`. In some cases, it is desirable to migrate only a part of the call stack; in PUB-Web, for example, only the user program, but not the PUB-Web VM should be migated. The method, which forms the bottom element of the call stack to be migrated has to be annotated with `padmig.Undock` instead of `Migratory` (cf. Fig. 3.1). According integrity checks are performed at compile time: *migc* stops with an error if a migratable method is called from an ordinary method, i.e., a method which is neither migratable nor undockable. *migc* also ensures that migratory annotations are consistent when inheriting from a (possibly abstract) class or when implementing or inheriting interfaces.

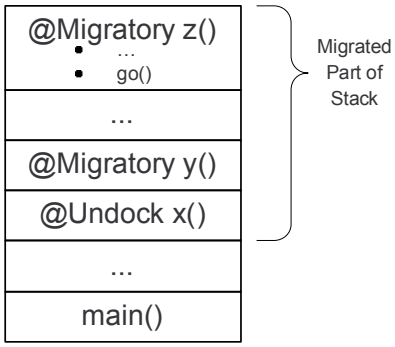

Figure 3.1: *Illustration of the migratory part of a call stack.*

We distinguish two kinds of migrations: if a method annotated with `Undock` has a return value, we call migrations on this call stack *synchronous* as the local execution has to wait for the undocked method to return; otherwise, if the `Undock` method has no return value, we call migrations on that call stack *asynchronous* because the local execution can already continue with the next statement after the call to the `Undock` method directly after the `Undock` method has migrated.

As all local variables are migrated by default, it is necessary to mark the locals which cannot be migrated (because they are not serializable) or should not be migrated (as they generate an unnecessary overhead). This can be done using the

`padmig.DontMigrate` Annotation. The PadMig compiler does not explicitly check for all possible kinds of side effects; however, the most prominent issues, such as open files or sockets, are often implicitly detected, e.g., because file or socket handles are not serializable. Note: In order to support, e.g., open socket connections in a transparent way with respect to inheritance etc., it is necessary to adapt the runtime environment appropriately by rewriting the according classes using proxies, which is out of the scope of this work. User programs inside PUB-Web are only allowed to open files for read access or to write output back to the user's peer via the PUB-Web API. Such open files or socket connections are not migrated but reside in the PUB-Web VM; after a migration, the user program needs to re-open files for read access or can continue to write output to the user's peer via the PUB-Web API.

The object whose method is to be migrated has to be serializable of course, i.e., it must implement the `java.io.Serializable` interface. The main class of an application is also required to implement a special main method defined in the `padmig.Migratable` interface:

```
public java.io.Serializable migratableMain(
    java.io.Serializable[] args);
```

For interoperability reasons (see next chapter), the parameters and return value of the main method can be any serializable object. Listing 3.6 shows an example for a migratable program, which demonstrates all the languages features.

Listing 3.6: *Example for a migratable program.*

```
1  import java.io.*;
2  import java.net.*;
3  import padmig.*;
4
5  public class Example implements Migratable, Serializable {
6
7    public boolean migrationNecessary() {
8      // evaluate situation here ...
9      return true;
10   }
11
12   public URL getMigrationTarget() {
13     try {
14       return new URL("pp://some.host:1234/migration_server_name");
15     } catch (MalformedURLException mue) { /* ... */ }
16   }
17
18   @Migratory
19   public int someMethod(int n) throws MigrationException {
```

```
20      for (int i=n; i>=0; i--) {
21        // some complicated calculation here
22        @DontMigrate
23        File someFileHandle;
24        // complicated calculation continued
25        if (i % 10 == 0) {
26          PadMig.checkpoint(new File("/path/checkpoint-"+n+".bak"));
27        }
28        if (migrationNecessary()) {
29          PadMig.migrate(getMigrationTarget());
30        }
31      }
32    }
33
34    @Undock
35    public int syncUndockMethod(int n) throws MigrationException {
36      return 2 * someMethod(n);
37    }
38
39    @Undock
40    public void asyncUndockMethod(int n) throws MigrationException {
41      // remote result output
42      System.out.println("the result is " + (2 * someMethod(n)));
43    }
44
45    public Serializable migratableMain(Serializable[] args) {
46      try {
47        asyncUndockMethod(21);
48        System.out.println("thread has undocked");
49        // local result output
50        System.out.println("the result is " + syncUndockMethod(23));
51      } catch (MigrationException e) {
52        System.out.println("migration has failed");
53        e.printStackTrace();
54      }
55      return null;
56    }
57  }
```

3.2.2 Technical Background

In this section we give technical insight in how the PadMig compiler transforms the annotated Java code into migratable standard Java code.

When migrating the calling thread, we need to save its current call stack and an abstract representation of its program counter, transfer both to the remote side, reconstruct the stack, and jump back into the code to the equivalent posi-

tion. To provide the functionality of a program counter, the PadMig compiler surrounds the original body of a migratory method with a switch statement, whose cases are used as entry points to reenter the code after a migration; thus, before every migratory invocation, the code generated by PadMig increases program counter and the migratory invocation is put into a new case statement (cf. Listing 3.7). If migratory expressions occur in loops or conditionals, an unfolding technique needs to be applied, as described in detail in Section 3.2.3.

For each migratory method a method-specific subclass of `padmig.lib.StackFrame` is generated, which stores all migratory locals and a symbolic program counter. When the current call stack is to be saved, it is represented as a list of instances of these stack frame classes. This list is created on demand by throwing a special `java.lang.Throwable`, namely `padmig.lib.SaveStack`, which holds the growing saved stack and has to be caught by every method and passed on after adding its own stack frame. To restore the call stack on the remote side, the corresponding stack frame is passed to each method on the stack via a non-`null` additional parameter called `__parentState` (this parameter is `null` during ordinary method invocations). The locals as well as the program counter are then restored by generated code inserted at the beginning of each method. A simplified example is provided in Listing 3.7.

Listing 3.7: *Simplified example of a generated method.*

```
1  public void foo(String param1, StackFrame __parentState) throws
     SaveStack, MigrationException {
2    FooStackFrame __state = null;
3    int __entryPoint = 0;
4    // locals declarations here...
5    if (__parentState != null) {
6      __state = ((FooStackFrame)(__parentState.child));
7      __entryPoint = __state.entryPoint;
8      // restore locals here...
9    }
10   try {
11     switch (__entryPoint) {
12     case 0 :
13       // original method body before migration here...
14       __entryPoint = 1;
15       throw new DoMigration(getMigrationDst());
16     case 1 :
17       // original method body after migration here...
18       // ...
19       // original method body before checkpointing here...
20       __entryPoint = 2;
21       throw new DoCheckpoint(getBackupFile());
22     case 2 :
23       // original method body after checkpointing here...
```

```
24      }
25    } catch (SaveStack __stack) {
26      __state = new FooStackFrame();
27      __state.self = this;
28      __state.entryPoint = __entryPoint;
29      // save locals here...
30      __state.child = __stack.bottomOfStack;
31      __stack.bottomOfStack = __state;
32      throw __stack;
33    }
34  }
```

Once you have compiled both your migratable program with the PadMig compiler *migc* and and the resulting code with the Java compiler *javac*, you are ready to run your migratable program — either standalone or as part of another application.

In order to execute a migratable program stand-alone, you need to run a migration server `padmig.standalone.Server` on every possible migration target and start the migratable program via `padmig.standalone.Client`, which is a wrapper around the migratable main method. As the internal communication of PadMig is performed via the *Paderborn Remote Method Invocation (PadRMI)* library [padb], the library `padrmi.jar` has to be included in the classpath. Java properties specify the IP address, port, etc. of the local machine and the codebase: either on a web server or file server as a `http:` or `file:` protocol URL or a `pp:` protocol URL pointing to one of the migration servers. Note that multiple cooperating migration servers must all point to the same codebase. Handlers for the PadRMI protocol (denoted by `pp:` in URLs) have to be installed into all participating JVMs as well via the Java Protocol Handler mechanism.

Instead of running your migratable application standalone, you can also integrate it into other Java applications — like we integrated it into PUB-Web — using the PadMig interoperability interface. In order to enable your application to accept incoming thread migrations, you need to start the PadRMI daemon and register a `padmig.iop.Service`, for example `padmig.iop.DefaultServiceImpl`:

```
padrmi.Server.startDefaultServer();
padrmi.Server.getDefaultServer().addObject(
    padmig.lib.PadMigLib.PADRMI_SERVICE_NAME,
    new padmig.iop.DefaultServiceImpl(),
    padmig.iop.Service.class, null, null);
```

Furthermore, you will probably want to implement the `padmig.iop.MigrationListener` interface and register it with the `DefaultService`

`Impl.addMigrationListener()` method in order to obtain references to incoming migratable objects and to be notified about migration failures; the code snippet in Listing 3.8 illustrates this.

Listing 3.8: *Example for a migration listener implementation.*

```
1 public void migratableObjectArrived(MigrationEvent e) {
2   System.out.println("incoming migration associated with object " + e
      .getObject());
3 }
4
5 public void migratableObjectContinuationFailed(MigrationEvent e) {
6   System.out.println("migration associated with object " + e.
      getObject() + " failed:");
7   e.getError().printStackTrace();
8 }
```

Supposed you have correctly set the PadRMI related properties like in the standalone case and your migratable program `HelloWorld` is located at the path specified in the `padrmi.path` property, you can start your program from an enclosing Java application like this:

```
Object returnValue = padmig.Launcher.launch(new URL(
    padrmi.Server.getDefaultServer().getURL() + "/"),
    HelloWorld.class.getName(),
    new Serializable[] { "hi", "there!" });
```

3.2.3 Translation Concepts

Before translating migratory methods, the PadMig compiler first checks if the provided code is syntactically correct Java code and obtains its abstract syntax tree using the Java transformation framework Spoon [spo, Paw05]. Then it performs some integrity checks on the code, in particular if every class (or interface) containing migratory methods implements (or extends, respectively) the `java.io.Serializable` interface, and if none of the reserved variable names `__state`, `__tmpState`, `__parentState`, `__entryPoint`, `__stack`, `__t`, `__gen`, `__tryNestingDepth`, `__cFlowBreakLevel` is used inside migratory methods. When overriding methods, either all or none of them can be migratory due to the additional stack frame parameter; thus the PadMig compiler verifies that a `@Migratory` or `@Undock` annotation of a method is compatible with all possibly existing overridden methods. Finally, it ensures that there are no migratory methods inside anonymous classes because stack frame containers can only be created for named classes.

Then every migratory method is translated (other code passes through unchanged). First, the signature of the method is changed in order to allow the

special throwable `padmig.lib.SaveStack` to be thrown on a migration, and to get the transmitted call stack passed in as a parameter on the remote side. In particular, a parameter `__parentState` of type `padmig.lib.StackFrame` is added to the parameter list and `padmig.lib.SaveStack` to the throws clause.

Before explaining the actual translation of a migratory method body in detail, we have to regard two special cases: First, if the method to be translated is an `@Undock` method, a helper method with the original signature is additionally required, which invokes the translated migratory method and handles the `padmig.lib.SaveStack` throwable (see listings 3.9 and 3.10 for an example). Second, if the method to be translated is abstract or declared inside an interface, or if the body of an ordinary `@Migratory` method contains no migratory invocation, no further processing of the method body is required.

Next, an inner class extending `padmig.lib.StackFrame` is created for each migratory method, which contains a field for each parameter of the method and for each local variable that is not excluded from migration (see Listing 3.10).

Each translated method is structured as follows (see Listing 3.10): at the beginning some PadMig specific variables and locals of the original method are declared. In case the `__parentState` parameter is not `null`, the method call is a continuation of the execution after a migration, which means that the values of the locals have to be restored from the saved call stack, and that the entry point, from where on to resume the execution, has to be set. The original method body is enclosed in a `try` statement, whose catcher creates a new stack frame and saves all the locals in case of a migration. The purpose of the endless `while(true)` loop around the original method body will be explained later together with the unfolding technique. Finally, the `switch` statement is used to jump back into the original code at the correct entry point.

Listing 3.9: *A simple example (Java code with PadMig annotations)*

```
1  import java.io.*;
2  import padmig.*;
3
4  public class Example implements Serializable {
5    @Undock
6    public int foo(Object bar) throws MigrationException {
7      double myDoubleLocal;
8      @DontMigrate
9      long myLongLocal;
10     // method body
11     return 42;
12   }
13 }
```

Listing 3.10: *Output of the PadMig compiler for the example in Listing 3.9*

```
1  import java.io.*;
2  import padmig.*;
3  import padmig.lib.*;
4
5  public class Example implements Serializable {
6    public int foo(Object bar) throws MigrationException {
7      try {
8        return foo(bar, null);
9      } catch (SaveStack stack) {
10       return ((Integer)(padmig.lib.PadMigLib.syncTransmit(stack)));
11     }
12   }
13
14   public class FooStackFrame extends StackFrame {
15     public Object bar;
16     public double myDoubleLocal;
17
18     public Object continueExecution() throws Exception, SaveStack {
19       StackFrame frame = new EmptyStackFrame();
20       frame.child = this;
21       return ((Example)(self)).foo(null ,frame);
22     }
23
24   }
25
26   public int foo(Object bar, StackFrame __parentState) throws
       SaveStack, MigrationException {
27     FooStackFrame __state = null;
28     FooStackFrame __tmpState;
29     int __entryPoint = 0;
30     int __cFlowBreakLevel;
31     double myDoubleLocal = 0;
32     long myLongLocal = 0;
33     if (__parentState != null) {
34       __state = ((FooStackFrame)(__parentState.child));
35       __entryPoint = __state.entryPoint;
36       bar = __state.bar;
37       myDoubleLocal = __state.myDoubleLocal;
38     }
39     try {
40       while (true) {
41         __cFlowBreakLevel = -1;
42         switch (__entryPoint) {
43           case 0:
44             // method body
45             return 42;
46         }
47       }
```

```
48      } catch (SaveStack __stack) {
49        __state = new FooStackFrame();
50        __state.self = this;
51        __state.entryPoint = __entryPoint;
52        __state.bar = bar;
53        __state.myDoubleLocal = myDoubleLocal;
54        __state.child = __stack.bottomOfStack;
55        __stack.bottomOfStack = __state;
56        throw __stack;
57      }
58    }
```

Now we are ready to have a look at the actual translation of a migratable method. It is organized in two traversals of the syntax tree. During the first pass, the following tasks are done:

- If migratable code occurs in places where it is not allowed or not supported, the compilation is aborted. In particular, no migratable code is allowed inside `assert` statements, as the left-hand side of an assignment, in looping expressions, in `synchronized` sections, in catchers, and in finalization blocks.

- Local variable declarations are moved to the beginning of the method, i.e., their scope is widened to the whole method. This is necessary because we need to access them when saving their values in a stack frame upon a migration and when restoring their values from the stack frame on the remote side (see Listing 3.10). Java ensures disjoint life ranges for local variables with the same name of the same type; so such variables can be unified. However, variables of different types must be disambiguated just to ensure static type safety; thus if two or more variables with the same name but different types exist, we will distinguish these variables by different appendices to their name, uniquely identifying their types as well as their type arguments and array dimensions if applicable.

  The only variable declarations not moved are those declared as parameters in catchers. On the one hand, it is not possible to move them due to the Java language specification; on the other hand, it is also not necessary to do so because catchers are not allowed to contain migratable code.

- Ordinary `for` loops are converted into `while` loops by moving the initial assignment(s) before the loop and the increment operation(s) to the end of the loop.

  Enhanced `for` loops (also known as *foreach* loops) are handled similarly: If the looping expression is a subtype of `java.lang.Iterable`, a compiler

generated variable of type `java.util.Iterator` parameterized with the appropriate type element is initialized before the loop; the `hasNext()` operation is used as the new looping expression, and the value obtained via the `next()` operation is assigned to the respective local variable in a new first statement of the loop's body.

Else, if the looping expressing has an array type, a compiler generated variable to hold a reference to the array and a second variable of type `int` to iterate through the array are initialized before the loop; this iteration variable is compared to the length of the array in the new looping expression; as a new first statement of the loop's body the particular array element is assigned to the respective local variable, and as a new last statement the iteration variable is increased.

All three cases are illustrated in Listings 3.11 and 3.12.

- Loops, `if`-, `switch`-, and `try`-statements containing migratory invocations are marked for unfolding during the second traversal of the syntax tree.

- Statements containing one or more migratory invocations are expanded. This is illustrated in Listings 3.13 and 3.14. If such statements would not be expanded, this could lead to redundant execution of already executed code. In the example, the call to `foo()` would be executed a second time at the migration destination, if a migration occurs and the statement were not expanded using temporary variables.

Listing 3.11: *A simple example of `for` loops to convert into `while` loops*

```
1  for (i = 0, j = 42; i < 3; i++, j-= 2) {
2    // loop body
3  }
4
5  Set<String> set;
6  // ...
7  for (String s : set) {
8    // loop body
9  }
10
11 String[] array;
12 // ...
13 for (String t : array) {
14   // loop body
15 }
```

Listing 3.12: *Converted loops from Listing 3.11*

```
i = 0;
j = 42;
while (i < 3) {
  // loop body
  i++;
  j -= 2;
}

Set<String> set;
// ...
String s;
Iterator<String> __gen_java_util_Iterator_java_lang_String = set.
  iterator();
while (__gen_java_util_Iterator_java_lang_String.hasNext()) {
  s = __gen_java_util_Iterator_java_lang_String.next();
  // loop body
}

String[] array;
// ...
String t;
String[] __gen_java_lang_String_array = array;
int __gen_int = 0;
while (__gen_int < __gen_java_lang_String_array.length) {
  t = __gen_java_lang_String_array[__gen_int];
  // loop body
  __gen_int++;
}
```

Listing 3.13: *A simple example for expansion of a statement*

```
result = foo() + bar(myMigratoryMethod());
```

Listing 3.14: *Expanded statement from Listing 3.13*

```
tmp1 = foo();
tmp2 = myMigratoryMethod();
result = tmp1 + bar(tmp2);
```

During the second traversal of the syntax tree, marked control structures are unfolded. The idea behind unfolding is to reduce the control flow to basic blocks with migratory invocations and then convert it into a finite automaton implemented as a Java `switch` statement surrounded by an endless loop; the state of the automaton, implemented as an `int`, serves as a symbolic program counter during migration.

The loops, `if`-, `switch`-, and `try`-statements identified to contain migratory invocations during the first syntax tree traversal are now rewritten as follows:

In order not to duplicate all code fragments of a loop body before, between, and after migratory invocations or nested statements to be unfolded, the loop is moved to the top level, and a `switch` statement is used to jump to the correct entry point. At the end of a `do` loop, the looping condition is checked using an `if` statement, and if the looping condition is still fulfilled, we jump back to the top of the loop's body by setting the correct entry point and using the `continue` statement. In case loops subject to unfolding are nested, the hierarchy is flattened this way, i.e., we only have one outer `while(true)` loop and `switch` statement.

When unfolding a `while` loop, the negated looping condition is additionally checked in the beginning to skip over the loop body in case the looping condition is already initially false. Unfolding of both loop types is illustrated in Listings 3.15 and 3.16.

Listing 3.15: *A simple example for loop unfolding*

```
1  while (i < 3) {
2    // while loop body
3  }
4
5  do {
6    // do loop body
7  } while (j < 5);
```

Listing 3.16: *Unfolded code from Listing 3.15*

```
1  __entryPoint = 0;
2  while (true) {
3    switch (__entryPoint) {
4    case 0:
5      if (!(i < 3)) {
6        __entryPoint = 2;
7        continue;
8      }
9    case 1:
10     // while loop body
11     if (i < 3) {
12       __entryPoint = 1;
13       continue;
14     }
15   case 2:
16     // code between while and do loop
17   case 3:
18     // do loop body
19     if (j < 5) {
```

```
20        __entryPoint = 3;
21        continue;
22      }
23    case 4:
24      return;
25    }
26  }
```

`if` statements are handled by inverting the conditional expression to jump into the `else` part (if present) in case the condition is not fulfilled. An Example is provided in Listings 3.17 and 3.18.

Listing 3.17: *A simple example for unfolding an `if` statement*

```
1  if (i == 0) {
2    // if body
3  } else {
4    // else body
5  }
```

Listing 3.18: *Unfolded code from Listing 3.17*

```
1  __entryPoint = 0;
2  while (true) {
3    switch (__entryPoint) {
4    case 0:
5      if (!(i == 0)) {
6        __entryPoint = 2;
7        continue;
8      }
9    case 1:
10     // if body
11     __entryPoint = 3;
12     continue;
13   case 2:
14     // else body
15   case 3:
16     return;
17   }
18 }
```

When processing a `switch` statement, a separate entry point is generated for each `case` label (see Listings 3.19 and 3.20).

Listing 3.19: *A simple example for unfolding a `switch` statement*

```
1  switch (i) {
2  case 0:
3    // case 0 body
```

```
4     break;
5   case 1:
6     // case 1 body
7     // fall-trough into case 2
8   case 2:
9     // case 2 body
10    break;
11  default:
12    // default case
13  }
```

Listing 3.20: *Unfolded code from Listing 3.19*

```
1   __entryPoint = 0;
2   while (true) {
3     switch (__entryPoint) {
4     case 0:
5       switch (i) {
6       case 0:
7         __entryPoint = 1;
8         continue;
9       case 1:
10        __entryPoint = 2;
11        continue;
12      case 2:
13        __entryPoint = 3;
14        continue;
15      default:
16        __entryPoint = 4;
17        continue;
18      }
19    case 1:
20      // case 0 body
21      __entryPoint = 5;
22      continue;
23    case 2:
24      // case 1 body
25    case 3:
26      // case 2 body
27      __entryPoint = 5;
28      continue;
29    case 4:
30      // default case body
31    case 5:
32      return;
33    }
34  }
```

If there are migratory invocations inside a `try` block, the block has to be split before each migratory invocation. Each such code fragment is surrounded by its own copy of the original `try` block. The exception handling code can simply be copied for each of the new `try` statements, but finalization blocks and labels require special treatment (cf. Listings 3.21 and 3.22). The finalization code is only to be executed when the whole `try` block is executed to completion without errors, when an exception occurs, or when the control flow is diverted using `return`, `break`, or `continue`. For this purpose, the nesting-depth of each `try` statement is determined, and the special local variables `__tryNestingDepth` and `__cFlowBreakLevel` are introduced to indicate whether or not to run the finalization code when leaving a copy of a split `try` statement. `__tryNesting Depth` is increased at the beginning of a copy of a split `try` statement and decreased at its end (except for the last copy) or when a `SaveStack` throwable is caught as a result of a migration. `__cFlowBreakLevel` is set to -1 by default and to the number of the outer-most finalization block to run if the control flow is diverted. The finalization code is eventually surrounded by an `if` statement to ensure that it is only executed if the `__tryNestingDepth` has not been decreased or the `__cFlowBreakLevel` has been set accordingly.

Each copy (except the last one) of a split `try` statement is succeeded by code to skip over the remaining copies in case of an abnormal termination, i.e., when the `__tryNestingDepth` has not been decreased.

Finally, we need to remove unused catchers as not all the catchers might be necessary in every copy of the `try` statement.

Listing 3.21: *A simple example for unfolding a `try` statement*

```
1  myLabel: try {
2    foo();
3    if (x == 1) {
4      break myLabel;
5    }
6    myMigratoryMethod();
7  } catch (MyException e) {
8    // exception handling code
9  } finally {
10   // finalization code
11 }
```

Listing 3.22: *Unfolded code from Listing 3.21*

```
1  __entryPoint = 0;
2  while (true) {
3    __cFlowBreakLevel = -1;
4    __tryNestingDepth = -1;
5    switch (__entryPoint) {
```

```
6     case 0:
7       try {
8         __tryNestingDepth = 1;
9         foo();
10        if (x == 1) {
11          __cFlowBreakLevel = 0;
12          __entryPoint = 2;
13          continue;
14        }
15        __entryPoint = 1;
16        __tryNestingDepth = 0;
17      } catch (MyException e) {
18        // exception handling code
19      } finally {
20        if ((__cFlowBreakLevel == -1 && __tryNestingDepth >= 1) || (
            __cFlowBreakLevel >= 0 && __cFlowBreakLevel < 1)) {
21          // finalization code
22        }
23      }
24      if (__tryNestingDepth > 0) {
25        __entryPoint = 2;
26        continue;
27      }
28    case 1:
29      try {
30        __tryNestingDepth = 1;
31        myMigratoryMethod(__state);
32        __state = null;
33        // __tryNestingDepth NOT decreased here because this is the
            last part of a split try block
34      } catch (MyException e) {
35        // exception handling code
36      } catch (SaveStack __t) {
37        __tryNestingDepth = 0;
38        throw __t;
39      } finally {
40        if ((__cFlowBreakLevel == -1 && __tryNestingDepth >= 1) || (
            __cFlowBreakLevel >= 0 && __cFlowBreakLevel < 1)) {
41          // finalization code
42        }
43      }
44    case 2:
45      return;
46    }
47  }
```

After the second traversal of the syntax tree we finally have to determine and link the correct entry points for the structured non-local jumps caused by

`break` and `continue` statements. This completes the translation of a migratory method.

3.2.4 Evaluation

In this section, we will discuss the impact of our translation approach in terms of code growth and increased running time. The basic structure of an unfolded method causes a small constant overhead by

- an additional method parameter,
- a few additional local variables (`int`s and references), including their initial assignment,
- an `if` statement, whose body is only executed in case of a migration,
- a `try` statement, whose catcher will only be triggered in case of a migration,
- a `while` loop, which will be iterated more than once only if the translated method contains at least one unfolded element or migration, and
- a `switch` statement, which will be evaluated once per iteration of the main loop and which contains more than one `case` label only if the translated method contains at least one unfolded element or migration.

The methods subject to unfolding are usually the big, central ones in a software library; however, there are typically only a few of them, so that only a small part of the total amount of code is concerned.

Only in case of a migration

- an exception is thrown, caught, and passed on,
- a stack frame object is instantiated, consisting of a few basic member variables (`int`s and references) and additional members corresponding to all locals on the stack that are not excluded from migration,
- all members of the stack frame object are initialized using flat copies of the locals on the stack, and
- the stack frame object is inserted into a linked list.

When the execution continues after a migration, all members of the stack frame object are copied back into the local variables using flat copies.

Altogether, this overhead is negligible for an ordinary execution of a method and minimal for a migration. In the following we will discuss the additional code growth and running time increase caused by unfolding.

When `if` or `switch` statements are unfolded, no overhead is generated for the `if` block or the first `case` block, respectively, and little constant overhead (increase of symbolic program counter, one iteration of the main loop, and one evaluation of the main `switch` statement) for the `else` block or all subsequent `case` blocks, respectively.

In unfolded loops the loop condition is checked using an `if` statement at the end of the loop body, so we have the same little constant overhead as for an `if` statement per loop iteration (except the last iteration, where we fall through into the code below the loop without overhead).

Only when unfolding `try` statements, we experience a notable overhead because the catchers and finalization code are copied (and transformed causing a small constant overhead) once for every migratory invocation in the `try` block. However, this affects typically only quite short portions of error handling code; in particular, only about 3% of the code is usually enclosed by `try` statements according to [Thi01]. Furthermore, the running time does not increase because at most one of the copied catchers is actually executed.

Nesting of unfolded elements does not cause any additional overhead. Altogether, the overhead caused by unfolding is acceptable in terms of code growth and negligible in terms of the running time. For Example, the total size of the bytecode of PUB-Web grows from 413 KB to 423 KB, which is an increase of 2.4%.

The data transferred during a migration consists of the instance of the class, whose method is subject to migration, all serialization dependencies, and the contents of the stack from the topmost element up to the `@Undock` boundary. Recall, that the developer can exclude stack elements from migration using `@DontMigrate` and thus reduce the volume of the transferred stack contents to the minimum. The absolute data volume transferred depends on the efficiency of serialization. Beside manual fine tuning using the `Externalizable` interface, Java's default serialization mechanism can be substituted by more efficient drop-in replacements. The "UKA-serialization" [PH99], for example, reduces the serialization overhead for objects, which are similar to our stack frame objects, notably by 81% to 97% compared to JDK.

3.3 Security and Trust Mechanisms

In a system like PUB-Web there are several challenges concerning security and trust. First, the people donating their unused computing power have to be protected: As already pointed out in Section 2.3.2, we use the *Java Sandbox* to restrict the permissions of parallel programs, such that they are only allowed to access the absolutely necessary Java properties for writing platform independent code (e.g. `line.separator` etc.). Especially the file system and the network cannot be accessed. BSP programs can only communicate, read files from the user's peer, or write output back to the user's peer via the PUB-Web API methods.

Second, the PUB-Web network should be protected against malicious peers — both untrustworthy computing nodes and supernodes. A way to achieve this is to build a web of trust and isolate the remaining peers. Supernodes only accept properly authenticated connections from peers, and peers only connect to supernodes which they trust. As trust mechanism are not our primary research goal, peers are authenticated by a username and password by default, but we also made PUB-Web ready to plug-in suitable trust mechanisms; exemplarily, the SybilGuard [YKGF06] protocol has been implemented during a cooperation in the scope of the EU FP6-IST project "Algorithmic Principles for Building Efficient Overlay Computers" (AEOLUS).

Finally, users may wish to protect their data processed by their BSP programs. Although there recently are already some first promising theoretical results in this direction [Gen10], this approach currently is not yet practical as it generates an overhead which is much larger than the benefit of parallel computing.

3.4 A Large-Scale Distributed Environment

Running and debugging P2P software such as PUB-Web on a large number of computers generates an immense overhead of copying files, launching processes, and collecting output. In order to almost completely automate this work and, at the same time, create a large-scale distributed testing network across several universities, we developed a general purpose large-scale distributed environment for P2P software written in Java, which became part of the EU FP6-IST project "Algorithmic Principles for Building Efficient Overlay Computers" (AEOLUS). Beside supporting the development of PUB-Web, it has been used for testing an overlay computing platform within this project.

The current setup of our testing environment within the AEOLUS project at `http://aeolus.cs.uni-paderborn.de/` consists of more than 100 computers (and 1000 virtual nodes) located at 13 different European universities and research institutes. Its architecture is kept so general and simple that it is

suitable for running and debugging any (P2P) services implemented in Java using the JXTA framework with minimal effort. Developers can request a login for the existing installation or easily setup an own installation. Via a web frontend they can upload their code and (possibly parameterized) configuration files, which are then automatically deployed to a (large) number of computers (at possibly different physical locations), which are configured as a JXTA P2P network. Via the web interface the developers can easily control their software and view its output for debugging purposes. Also, they can easily connect further computers to the environment, which, e.g., act as a frontend to their software, as an interface to other software, are equipped with special hardware, or work as a bridge to other hardware such as wireless sensor networks, for example.

Our approach differs from PlanetLab [pla, CCR+03] in that it provides an environment for P2P software written in Java using the JXTA framework, whereas PlanetLab simply provides a network of virtual machines to the developer, i.e., PlanetLab is a more general tool, allowing to run and debug almost any kind of distributed software; but, at the same time, such a general approach involves quite some work to setup and configure everything properly. For the purpose of developing P2P software in Java we provide a comfortable ready-to-use tool, i.e., our approach significantly simplifies and speeds up the developers' work. Furthermore, our environment can be installed on existing Windows and Linux computers, where it runs with lowered priority in order not to disturb other users / processes, whereas PlanetLab must be installed on additional hardware dedicated solely for this purpose.

3.4.1 Architecture

The architecture of our system is depicted in Fig. 3.2. Its main components are:

- the server which includes a web server, management software, and a database with the system configuration;
- edge peers: these are essentially the computing nodes donated to the system by several different partners;
- rendezvous peers: these nodes support special functionalities of JXTA, which enable the edge peers to locate each other and communicate;
- user nodes: through these nodes, developers access the system. Beside using it through the web interface and JXTA shell, specialized edge peers can be employed as user frontends or interfaces to other software outside the system.

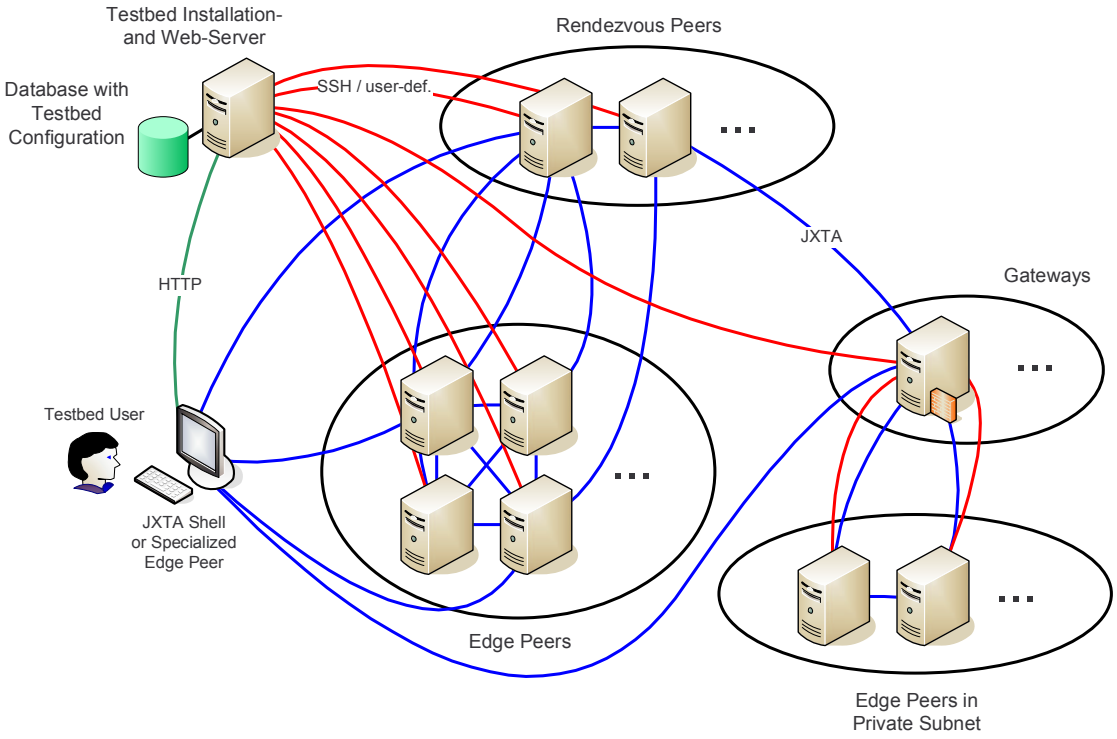

Figure 3.2: *The System / Network Architecture.*

The server stores the configuration of the whole system in its database; the configuration does not only include setup parameters of all the edge peers, but also the software and all testing parameters of all registered developers. This ensures that the system is able to automatically recover the configuration of edge and rendezvous peers after crashes, reboots, or even reinstallations of the operation system. Additionally, it eases developers' lives as they only have to upload and configure their software once through the web interface of the server; the server then automatically deploys the software to edge peers, configures, and controls it, and collects the output (stdout + stderr) on demand. The communication between the server and the edge / rendezvous peers is secured by SSH for the Linux peers; on Windows peers, a user-defined protocol is employed in order to circumvent security issues related to retrofitting SSH into Windows. Peers, which are part of the same testing scenario, communicate using JXTA (or a user-defined communication library); beside the peers assigned to the developer's test case, he / she can also add one or more own edge peers — either a JXTA shell or specialized edge peers which work as a user frontend or interface to other software outside the system.

The rendezvous peers are a subset automatically selected out of the edge peers by the server. The edge peers are computers running Windows or Linux, donated by different project partners; these computers need not be exclusively dedicated to our system as the software to test is executed on them with lowered priority in order not to disturb the owner of the computer in his activities.

An Example for specialized edge peers is given by gateways to special hardware. In the AEOLUS project, e.g., 289 wireless sensors of heterogeneous types located at 11 different sites have been connected to the system via gateway edge peers.

3.4.2 Requirements

For the server a Linux machine is required. In the particular case of our setup, a multiprocessor machine with a gigabit backbone connection and a soft-RAID system running the Ubuntu Server Edition is used, hosted at the Heinz Nixdorf Institute.

In order to setup an own installation, the main required software components (which are usually part of every modern Linux distribution) are the following: ssh, cron, bash, perl, gcc, libc, Java (Sun JDK version 6.0 or later), MySQL database, Apache web server, Apache PHP plugin. Instructions on how to properly configure these software components are usually shipped together with them and can additionally be found on the distributors' particular web pages. Specific instructions on how to install the server are included in the distribution.

Once the server is setup, edge peers can be added to the system. These can be either Windows or Linux computers. In order to comply with the strict security regulations of different networks, the requirements for integrating computers into the system are kept as low as possible. In particular, the requirements for Windows nodes are as follows: Windows 2000/XP/2003/Vista/7, a public IP address, open TCP port 2022 for system management and ports 9700-9799 for JXTA (these default values can be changed in case these ports are already occupied by other software). The installation procedure using the Windows installer is very simple: during the setup you are asked to enter your login and password, and you may change the default values for the TCP ports; the rest is done automatically.

For Linux nodes, the requirements are as follows: Kernel 2.6.x, ssh, bash, perl, tar, gzip (typically present on every distribution), a public IP address (or a gateway), open TCP port 22 (SSH) for system management and ports 9700-9799 for JXTA. Since an automated installer is not yet available, the installation procedure involves the manual setup of a dedicated ordinary user account (without root privileges); you need to provide the IP address, ports (when deviating from

the default values), and the login to the testbed administrator and install an SSH key for automated login.

We remark that there is no need to install Java since our software comes with its own Java installation in order to assure that all edge peers are running the same Java version and have all required libraries installed.

3.4.3 Security Aspects

The testing environment is secured against different types of attacks. On the one hand, to prevent outside attackers from gaining unauthorized access, a multilevel security model is applied. First, all management / installation / configuration traffic with the Linux nodes is tunneled using SSH with DSA keys (or RSA keys if DSA is not supported) of a reasonable strength; second, our software completely runs in user space, i.e., even if a connection would be hijacked, nothing outside this dedicated user account could be damaged. For the Windows nodes, all connections to the management port from another IP address than the server are dropped; second, a fixed set of reactive commands is defined, i.e., a command can only be triggered from outside without any parameters and the local process then collects the required options itself (thus, no invalid options can be used).

On the other hand, the system is also secured against internal attacks. In case, for example, a user does not properly protect his password, an attacker can neither hack other user accounts nor hijack any computers of the system. This is due to the security policy of the Java Sandbox which is configured in such a way that the uploaded code can only access files in the local folder and only open network sockets on ports above 1024 (thus, computers "infected" by malicious uploaded code can only produce a high CPU and network load in the worst case).

3.4.4 Testing P2P Software

As already mentioned at the beginning of Section 3.4, P2P software to be tested has to be written in Java using the JXTA library. In order to both provide a uniform interface to our system and simplify developer's lives, the API of our testing environment already includes basic implementations of JXTA P2P services and service discoveries.

To execute and test a particular piece of software, a developer needs to perform the following steps:

1. Allocate a number of computing nodes;

2. configure them as edge peers of a (JXTA) P2P network;
3. upload his / her code to test;
4. run / debug the software.

To ease this procedure as much as possible, most of these tasks are done automatically. The server holds the complete configuration of all test setups of all users and automatically installs and configures the computing nodes. Moreover, it automatically recovers computers which were temporarily unavailable or even suffered a loss of data.

Computing nodes are allocated via the web interface of the server, and afterwards the software to test (and all its configuration files) are uploaded. The server then automatically delivers all components of the software to the allocated computing nodes, configures them properly, and runs the software. For debugging purposes, the output of all peers can be viewed via the web interface.

In order to allow scalability tests with respect to the number of computing entities, our system is able to simulate up to 10 virtual nodes per physical computing node; the number of requested virtual nodes has to be specified during the allocation step.

In order not to interfere with other users when reconfiguring or restarting the P2P network (or when your software to test appears to be buggy), a separate P2P network is setup for each test case.

As not all P2P applications consist of totally homogeneous peers, the code and configuration files to be uploaded to the edge peers can be individually configured on a peer basis. Furthermore, there is a set of scripted constants supported, e.g., `__IP__` which is automatically replaced by the edge peer's IP address in all configuration files.

Some applications require peers outside our system, e.g., as a user frontend or interface to other software outside our system. Our API provides a framework to implement a specialized edge peer for such a case. To connect these peers to a JXTA network in our system, the location and port of the rendezvous peer is displayed in the web interface.

While developing PUB-Web, our testing environment greatly simplified the process of distributed remote running and debugging: After uploading the code and the configuration files, we can easily configure a P2P network, start / reset the PUB-Web network, and view the debug output through the web interface of our environment. In order to actually run a parallel program, a specialized edge peer is required because we need access to the graphical user interface at one of the peers in the PUB-Web network. As a very simple example think of a parallel program rendering the Mandelbrot image: A user behind a PUB-Web

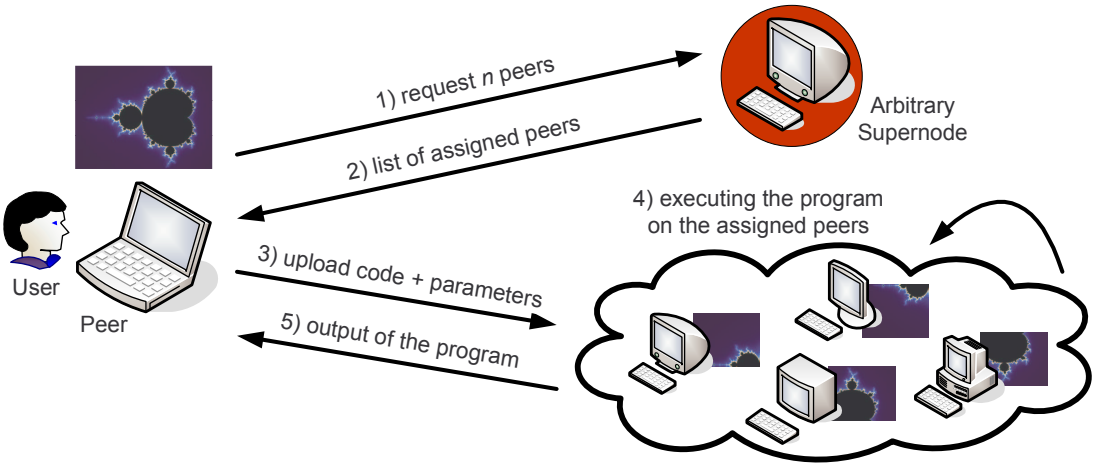

Figure 3.3: *Example for a Use Case.*

peer simulated on a specialized edge peer enters the parameters of the Mandelbrot image to draw (cf. Fig. 3.3). The PUB-Web software then requests a number of computing nodes from a PUB-Web supernode discovered through a lookup in our JXTA network. Then the PUB-Web software on the specialized edge peer uploads the parallel Mandelbrot code and the parameters to the assigned subset of our P2P network, where the Mandelbrot code is executed, and finally composes the image out of the output sent back from the peers.

Setting up such a testing scenario in a distributed environment without our system would require a lot of manual (remote) configuration work. But via the web interface of our environment it is just one click to upload code or to start / stop / reset the testing scenario, which significantly improves the workflow of a developer.

4

Load Balancing

Since the scheduling approach from the former, centralized version of PUB-Web [BGM05a] was no longer applicable to a P2P scenario, a novel distributed load balancer has been proposed in [GS06], which provides scheduling, migration, and fault tolerance for processes using Distributed Heterogeneous Hash-Tables (DHHT). In the first section of this chapter we give a description of the scheduling problem. In Section 4.2, we describe the DHHT-based load balancer and discuss some improved variants as well as two Work Stealing variants for comparison. In sections 4.3 and 4.4, we experimentally compare the different load balancers; as a precondition for this evaluation, we analyze the external workload on several hundred computers in order to obtain realistic load profiles for reproducible experiments.

4.1 Problem Description

Our scheduling problem can be formulated as follows: We are given a *stream describing the availability* of the peers and a *stream of jobs* consisting of independent processes. For a time t, the *availability* of a peer is described by a value in $[0;1]$ indicating the fraction of its computing power currently available. This value reflects removal / addition of peers (value changes to 0 / from 0 to something greater 0) or the power left over beside the current external workload. This stream may show an unpredictably changing behaviour. In order to reflect the heterogeneity of the hardware, we associate a static *weight* factor from $(0;\infty)$ describing the processor speed. This value is derived by a benchmark being part of PUB-Web. Thus, at any time, the *currently available computing power* of a peer can be expressed as the product of the peer's weight and its availability in a globally uniform way.

In a superstep of a BSP program (i.e., a job), all processes require approximately the same amount of computing power. By definition, the supersteps are subsequent, disjoint time intervals. Thus, scheduling a BSP program with ℓ supersteps is equivalent to scheduling ℓ successive BSP programs with one superstep each, where the release time of the second program is identical to the finishing time of the first program, and so on. As a consequence, each job in our stream of jobs is described by a release time, the number of its processes, and their lengths. These lengths are assumed to be identical within one job. We consider all processes of all jobs to be equally important, independently of their length and job-parallelism. Thus, the processes to be executed at a given time may belong to different jobs and may therefore have different lengths.

Our optimization goal is, at any time, to (re-) assign all currently running processes to peers such that

(i) all processes receive approximately the same amount of computing power, and

(ii) the total processor utilization is maximized without violating condition (i).

4.2 The Load Balancing Algorithms

In the following, we first describe two Work Stealing variants, adapted to the needs of our BSP-scenario, which we later use for comparison in our experimental evaluation. Then we focus on the new DHHT-based load balancer, describing the algorithm and some improved variants.

4.2.1 Work Stealing

The Work Stealing idea has been well studied over the past decades. In [RSAU91] for example, a Work Stealing strategy is presented for balancing the load of independent processes on a parallel computer, i.e., a set of homogeneous, fully available processors. In [BL94], a Work Stealing algorithm for well-structured, multi-threaded computations is presented. We have examined the following variants: Consider a randomized, distributed setting, where each computing node maintains a FIFO queue. Whenever a processor becomes idle, it removes the first process from its queue and executes it. In case its queue is empty, the processor "steals" a process from the queue of another, randomly chosen processor. When a job with parallelism k is released, its processes are added to the queues of up to k distinct, randomly chosen processors, with probabilities

inverse-proportional to the current queue-sizes. This, and the fact that a processor only steals a process when its queue has run empty, helps to prevent unnecessary migrations. Using a queue instead of a stack is useful for our BSP-scenario, because it makes sure that no jobs are disadvantaged. We belief that stealing processes from the head instead of the tail of the queues is also beneficial in our BSP-scenario, because this prevents BSP processes from starving; to verify this assumption, we consider both variants in our evaluation. In the remainder of this thesis, we refer to them as "WS (Head)" and "WS (Tail)", respectively. Work Stealing can be implemented in an efficient and fully distributed fashion.

4.2.2 Distributed Heterogeneous Hash-Tables

The new load balancer, proposed in joint work with Gunnar Schomaker [GS06], is based on *Distributed Heterogeneous Hash-Tables (DHHTs)* [SS05]. DHHTs are a heterogeneous generalization of the consistent hashing introduced by Karger et. al. [KLL$^+$97]. In order to map all running processes to the active peers, first all peers are hashed uniformly and independently at random into the $[0;1)$-interval, which is interpreted as a unit ring (cf. Fig. 4.1). There is a linear function associated with each node, whose gradient is the inverse of the currently available (globally normalized) computing power. The *lower envelope* for the $[0;1)$-interval is defined, at any point x, as the peer whose linear function has the minimum value at point x; this way we assign sub-ranges of the $[0;1)$-interval to the peers. Finally, all processes are also hashed uniformly and independently at random into the $[0;1)$-interval and are assigned to the peers associated with them through the lower envelope.

We performed the experimental evaluation presented in Section 4.4 using a non-distributed implementation of the DHHT load balancer. However, an exactly equivalent, distributed implementation can achieved as follows: when running multiple supernodes that trust each other and thus form one common (sub-)network, each supernode will hold a copy of the DHHT model. In order to guarantee that the lower envelope is consistent in all copies, each supernode communicates leaving or joined peers as well as changes in the available computing power of the peers. However, the assignment of a process to a peer does not depend on the presence or assignment of other processes. Thus, using pseudo-random or deterministic hash functions, the supernodes need not share information about the scheduled BSP processes; moreover, each supernode can schedule the BSP processes, which it is responsible for, without the knowledge of the other BSP processes in the system. Furthermore, if a supernode crashes, schedules can be easily reconstructed by any supernode.

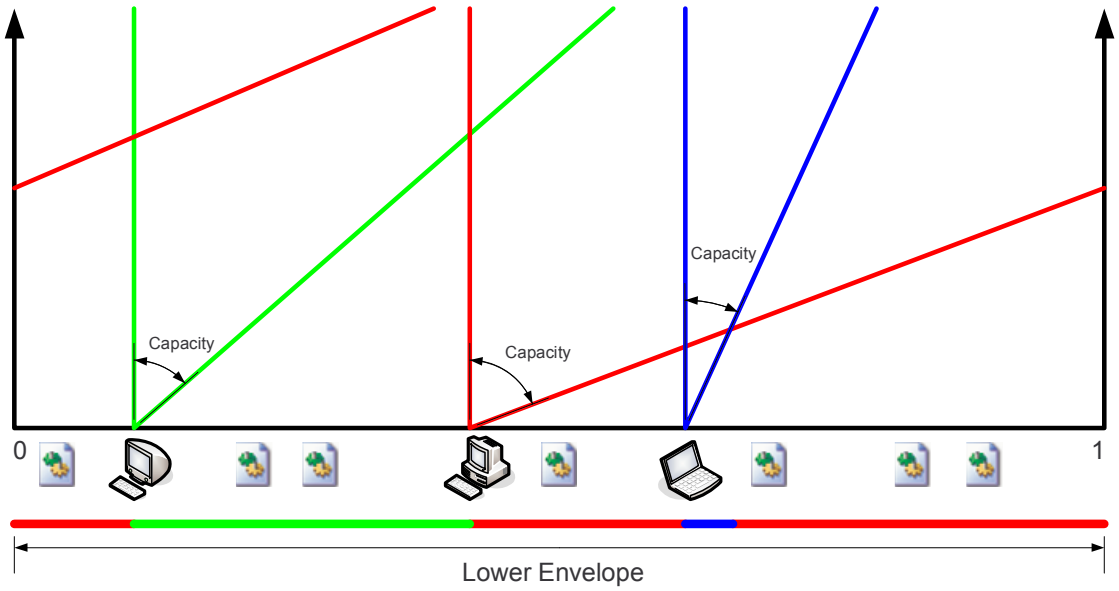

Figure 4.1: *Load balancing using DHHT.*

Our implementation uses SHA-1 as a pseudo-random uniform hash function. We refer to this DHHT variant as "DHHT (Rnd., Rnd.)" in the remainder of this paper.

4.2.3 Multiple Hashing

In [SS05] several possible improvements are discussed, among them a multiple hashing technique. With this approach, no lower envelope is used any more; instead, multiple hash functions are employed. When transferring this approach to our setting, an individual hash function is associated with every peer. In order to map a process x to a peer, first for each peer i the hash value $r_{i,x} \in [0;1)$ of process x is calculated using the individual hash functions of the peers. Then x is assigned to the peer, for which $\frac{r_{i,x}}{w_i}$ is minimal.

It is shown in [SS05] that a the probability that a process is assigned to peer i improves to $(1-\sqrt{\epsilon}) \cdot \frac{w_i}{W} \leq p_i \leq (1+\epsilon) \cdot \frac{w_i}{W}$ with this approach. An obvious drawback of this approach is a running time linear in the number of peers for assigning a process.

We also implemented this variant and refer to it as "DHHT (Multi)" in the following.

4.2.4 Deterministic Hashing

It is well-known that when hashing peers uniformly and independently at random, the longest interval has length $\Theta(\frac{\log n}{n})$ with high probability, and the shortest one has length $\Theta(\frac{1}{n^2})$ with constant probability. Thus, the ratio of the longest interval to the shortest one, called *smoothness*, is $\Theta(n \cdot \log n)$ with high probability.

In order to overcome this drawback, we implemented other, deterministic DHHT variants based on multiplicative hashing: Let $\phi := \frac{1}{2} \cdot (1 + \sqrt{5}) \approx 1.618$ denote the golden ratio, and $\Phi := \phi^{-1} = \frac{1}{2} \cdot (\sqrt{5} - 1) = \phi - 1 \approx 0.618$ its inverse. Furthermore, let $\{x\}$ denote the fractional part of x (namely $x - \lfloor x \rfloor$, or $x \mod 1$). Then the points $\{\Phi\}$, $\{2\Phi\}$, $\{3\Phi\}$, ..., $\{n\Phi\}$, consecutively added to $[0, 1)$, stay very well separated from each other; as proven in [Swi58], only three different interval lengths are produced and a smoothness of $\phi^2 \approx 2.618$ is obtained for any $n \in \mathbb{N}$.

Unfortunately, these properties only hold as long as all consecutively added points are present. Thus, whenever a process finishes or a peer leaves, its hash-value has to be reused. A way to accomplish this, would be to swap such an unused hash-value with the one of the latest released process or joined peer, respectively, and to migrate the processes accordingly. But since we consider a large system with a big amount of peers and running processes, new processes are released and peers join quite frequently; thus, we simply reuse such unused hash-values for them, suffering short periods of slight imbalance, but saving a lot of migration overhead. In the following, we refer to our DHHT implementation with deterministic hashing for peers as "DHHT (Det., Rnd.)", for processes as "DHHT (Rnd., Det.)", and for both peers and processes as "DHHT (Det., Det.)". In contrast to purely pseudo-random based DHHT variants, a distributed, crash-persistent implementation using this deterministic hash function is not trivial.

4.3 Experimental Setup

In order to perform a meaningful experimental evaluation of the load balancing, we need realistic information about the dynamics of the external workload on the peers, about the capabilities of the simulated hardware, and about the parallel programs to run.

4.3.1 Dynamics of the External Workload

In order to collect real usage data for the evaluation of our load balancers, we did a detailed, long-term analysis of the availability of several hundred computers. Beside using the collected data as input for our experiments, it will also help to understand the dynamic behaviour of the external work load — and thus the remaining computing power available to PUB-Web — as well as intervals, in which computers are not available at all (e.g., because they are switched off).

In total we have analyzed the CPU usage on more than 250 PCs for several months. Among these PCs there are different sets of office computers, PCs in computer pools available to students, notebooks of employees, and both home PCs and notebooks of individuals. The examined computers are very heterogeneous with respect to their hardware (old and recent machines, single and multi-core CPUs) and operating system (Windows 2000 / XP / Vista, Mac OS X 10.x, and different Linux distributions with 2.6.x kernel).

The data has been collected the following way: Every 30 seconds the average CPU idle percentage over the past 30 seconds is saved together with a timestamp. On Windows, the idle time values are obtained using the Microsoft *Performance Data Helper (PDH)* API; on Linux, the values are read out of the `proc`-filesystem; on Mac OS X, the `host_processor_info()` function is used. Once a day or at the next time when the computer is switched on and connected to the Internet again, the collected data is uploaded to our database. In order to respect the privacy of the computer users and owners, all data is collected in a completely anonymous way: when a computer uploads data for the first time, it requests a unique random string. Using this identifier all successive uploads are combined to a data series. The only information associated with the identifier is the operating system and the type of computer (notebook or PC; office, home, or shared use).

In the following, we analyze the collected data with respect to these criteria:

Availability How often a computer is switched on, and how much idle time is available, affects the total amount of computing power available in the PUB-Web network. This parameter describes how much computing power is available on a computer on average.

Diversity If all computers in the PUB-Web network would be switched on and off at the same time, and would have the same loss or gain of computing power at the same times, load balancing would be trivial because all running processes would experience the same slowdown or speedup. However, that is usually not the fact. The diversity parameter describes, how different the availability curves of the computers behave.

Uniformity If the availability of a computer is rather stable (at any level) for long time intervals, the load balancer rarely has to rebalance the assigned processes. This parameter describes how often the availability of a computer significantly changes (high uniformity = rare changes).

Cumulative Pattern When averaging over a big amount of computers, the aggregated data may show a certain pattern even if the diversity is high and the uniformity is low. Typical patterns are daytime / nighttime or workday / weekend differences, and may help to improve the load balancing.

Analyzing the collected data, we identify distinguishing parameters for three types of computers: office PCs, notebooks, and pool PCs used by students (cf. Table 4.1).

Table 4.1: *characterization of the availability of the peers*

	Office PCs	**Notebooks**	**Pool PCs**
Availability	high	low	medium
Diversity	low	high	medium
Uniformity	high	low	low
Daytime Pattern	significant	significant	weak
Weekend Pattern	significant	significant	no

When the computers are switched on, the availability is very high: above 90% with a probability of approximately 90% for office PCs and pool PCs; and above 80% with a probability of approximately 75% for notebooks. If additionally taking into account the times, when the computers are switched off, we have an average availability of approximately 70% for office PCs, 45% for pool PCs, and 30% for notebooks.

Not surprisingly, the office PCs are used in a quite homogeneous way with respect to both time of usage and kind of usage, leading to a low diversity, a high uniformity, and clear daytime and workday patterns. Notebooks are often used in a mixed fashion for different purposes, leading to a high diversity and low uniformity; however, they still show clear daytime and workday patterns on average. The pool PCs are used by lots of different students for very heterogeneous tasks. Despite this fact, they show only a medium diversity; but as expected, they have a low uniformity and not even a clear daytime pattern.

For our evaluation of the load balancers, the diversity property will be most interesting as a high diversity generates a lot of imbalance. In order to cover different levels of diversity, we will use the collected data of all three types of computers (office PCs, notebooks, and pool PCs) as realistic inputs for repro-

ducible experiments. In the following, we call a group of computers of one of these three types a *profile group*.

4.3.2 Capabilities of the Processors

In order to preserve the anonymity of the people who allowed us to analyze their CPU availability, the collected availability data does not contain any other information about the computer where is has been collected except the fact whether is was a notebook, an office PC, or a computer pool PC. Thus, we especially do not have specific information about the CPU such as its speed or number of cores. In order to also take this kind of heterogeneity of the hardware into account using realistic values, we have independently collected information about all CPUs in the AEOLUS testbed (cf. Table 4.2), and randomly choose one of these factors for each simulated peer, where the probability that a factor is chosen is weighted by the number of its occurrancies (cf. column "count"). Of course, we guarantee this random assignment to be the same for all experiments, using a fixed seed.

4.3.3 Properties of the Job Stream Model

Beside the parameters for the hardware capabilities and the external workload, where we will use collected data as input for our experiments, we will create synthetic job input streams because we do not have access to traces of comparable Web Computing systems. As already outlined in Section 4.1, a job stream consists of a sequence of jobs with certain release times, parallelism, and lengths.

The *release times* of the jobs are chosen uniformly and independently at random within the total simulation duration, except for a short warm-up phase at the beginning of the experiments.

Realistic values for the length of a superstep in a coarse-grained BSP program are lower bounded by a couple of minutes (otherwise the synchronization overhead becomes unreasonably large) and upper bounded by roughly one hour (otherwise communication would occur unrealistically rarely). Remember from Section 4.1 that a BSP program consisting of ℓ supersteps is split into ℓ successive jobs. Thus, our jobs have *lengths* of 5, 10, 15, 20, 30, 45, or 60 minutes (on a CPU with weight factor 1 and 100% availability). For each job, one of these lengths is randomly chosen, where we use probability distributions derived from traces of supercomputers available through the *Parallel Workloads Archive* [wor]; we adapt these probability distributions for our scenario keeping the quantitative balance between short, medium, and long jobs. The total

Table 4.2: *CPUs on the AEOLUS testbed.*

Speed	# Cores	Factor	Count
667 MHz	1	0.667	1
733 MHz	1	0.733	2
800 MHz	1	0.8	1
866 MHz	1	0.866	1
900 MHz	1	0.9	1
933 MHz	1	0.933	2
1.3 GHz	1	1.3	3
1.33 GHz	1	1.33	1
1.4 GHz	1	1.4	1
1.7 GHz	1	1.7	7
1.8 GHz	1	1.8	1
2 GHz	1	2	3
1 GHz	2	2	2
2.4 GHz	1	2.4	2
2.53 GHz	1	2.53	2
2.6 GHz	1	2.6	2
2.8 GHz	1	2.8	4
700 MHz	4	2.8	1
3 GHz	1	3	2
3.2 GHz	1	3.2	2
1.67 GHz	2	3.34	1
2 GHz	2	4	11
2.13 GHz	2	4.26	1
2.66 GHz	2	5.32	32
2.8 GHz	2	5.6	2
2.83 GHz	4	5.66	5
3.1 GHz	2	6.2	1

number of jobs is chosen such that there are sufficiently many processes ready to run in all of our experiments. While we only use the probability distribution of the job lengths to derive realistic values for our model, we directly copy the probability distribution of the *parallelism*; all parallelism values occurring are powers of 2, up to 1024.

In particular, we use the logs of these five parallel machines to derive our input profiles: *HPC2N* refers to a 120 node Linux cluster with two AMD Athlon MP2000+ processors per node, located at the High-Performance Computing Center North in Sweden. *LLNL Atlas* is a 1152 node Linux cluster with eight AMD Opteron processors per node, located at the Lawrence Livermore National Lab in California, US, and *LLNL Thunder* consists of 1024 nodes with four Intel Itanium processors per node. *SDSC BLUE* is a 144 node IBM-SP system with 8 processors per node, located at the San Diego Supercomputer Center in California, US, and *SDSC DataStar* is a 184 node IBM eServer pSeries system, consisting of 176 8-way SMP and 8 32-way SMP nodes. Table 4.3 shows the probability distributions of the parallelism and the number of jobs of the particular lengths for the five input profiles.

Table 4.3: *Parameters of the input profiles.*

Profile Name		*HPC2N*	*LLNL Atlas*	*LLNL Thunder*	*SDSC BLUE*	*SDSC DataStar*
Prob. of Parallelism [%]	1	42				
	2	18				
	4	14		5		
	8	10	33	13	5	4
	16	8	10	15	16	7.5
	32	7	7	5	12	28
	64	1	15	1	14	17
	128		13	5	4	2.5
	256		11	1	3.5	2.5
	512		7	0.2	0.5	2
	1024		4	0.8		0.5
# Jobs of Length	5 min	3040	550	3550	2900	2600
	10 min	3040	500	3050	2500	2200
	15 min	3040	450	2550	2100	1800
	20 min	6080	400	2050	1700	1400
	30 min	6080	350	1550	1300	1000
	45 min	6080	300	1050	900	600
	1 h	9120	250	550	500	200

Since all of our experiments are performed under a fully loaded or slightly overloaded system, and because we need to control the load as the system would collapse at some time under very heavy overload, we keep all released processes in a *ready queue* if the total system load exceeds a certain limit. This limit is given by the number of peers in the PUB-Web network times a so-called *overload factor*. This procedure allows for a fair comparison of the load balancers because each load balancer has to deal with the same total system load, independently of its throughput, and gets exactly the same sequence of jobs as input stream.

Though the job streams are generated in a randomized fashion according to the probability distributions of the parallelism and job lengths, we ensure, by using fixed seeds, that exactly the same job stream is used in a series of experiments for comparison of the particular load balancers.

4.4 Evaluation of the Load Balancers

Using the collected data and the job stream model described in the previous section, we are now able to conduct reproducible experiments: From each of the three profile groups (office computers, PCs in computer pools, notebooks) we use an interval of two weeks of collected data as input for our simulations. In total we simulate more than 10^8 processes in more than 700 experiments. In order to separate the influences of the external work load from potential drawbacks of the load balancers, we perform three types of experiments:

A: To focus on the dynamics of the external workload, we feed the load balancers with the measured availability in the different profile groups, but do not generate additional load imbalances by a changing number of processes in the system; instead, we consider $x = 1, 2, 4$ times as many processes as processors, which are simultaneously released at the beginning of the experiment and run throughout the whole duration of the experiment. Thus, we measure with our experiments how well processes are distributed among the available peers, and how often migrations occur.

B: In order to focus on the processing of the jobs, we fix the availability of all peers to 100% throughout the whole experiment and feed the load balancers with the different synthetic job streams. Thus, we measure how fast and how fair the processes are executed.

C: We run the experiments of type B again, but now apply the same profiles of the external workload as in the experiments of type A. Thus, we

measure the quality of the load balancers under dynamically changing availabilities of the processors and for realistic job streams.

4.4.1 Experiments of Type A

Since we run a fixed number of processes throughout the whole duration of the experiment, this setup is not suitable for Work Stealing. Thus, we only compare the different DHHT variants.

Because, on the one hand, the performance of a peer is most efficient if it is running only one process at a time (no overhead due to context switching and swapping), but on the other hand, the overall balancing quality typically improves with an increasing number of processes (due to a reduced fragmentation overhead), we run experiments with x, $2x$, and $4x$ processes, where $x = 100$ is the number of peers. Tables 4.5 – 4.7 show the aggregated results of our experiments: the CPU time is the running time of a process, weighted by the availability and the weight factor of the CPUs where it was running and the number of other BSP processes concurrently running on the same CPU.

Table 4.4: *The available CPU time in the particular profile groups.*

Profile Group	14-day CPU Time [s]		
	min	*avg*	*max*
Notebooks	6048	790228	5452258
Office PCs	197	4132389	6740009
Pool PCs	390170	3005205	6334383

As we ran all processes throughout the whole duration of the experiment, the following criteria are good indicators for how well a load balancer works:

- The closer the utilized average CPU time is to the available average CPU time (cf. Table 4.4), the better a load balancer utilizes the available computing power. The average utilization percentage is shown in column 4 of tables 4.5 – 4.7.
- The factor between the minimum and maximum CPU time a process receives (cf. column 7 in tables 4.5 – 4.7) should be as low as possible. A value of 1 would mean a perfectly balanced schedule.
- The number of migrations needed should be as low as possible.

The results using the notebooks availability profile (cf. Table 4.5) show that the average utilization significantly improves when running $2x$ or $4x$ processes.

Table 4.5: *Results of experiments of type A on notebooks (Windows).*

Load Balancer	# Procs.	CPU Time					# Migs. (Def.)			# Migrations (Bulk)			
		avg [s]	*avg* [%]	*min* [s]	*max* [s]	*max* / *min*	*avg*	*min*	*max*	*avg*	[% Def.]	*min*	*max*	[% Def.]
DHHT (R/R)	x	654191	83	294247	2418193	8.2	310	35	497	275	89	35	439	88
DHHT (R/R)	$2x$	360542	91	137455	2225923	16.1	307	35	543	274	89	35	485	89
DHHT (R/R)	$4x$	186511	94	73755	1219776	16.5	304	35	543	269	88	35	485	89
DHHT (R/D)	x	699446	89	298187	2518742	8.4	295	37	485	261	88	34	438	90
DHHT (R/D)	$2x$	368389	93	153519	1854472	12.0	296	37	529	262	89	34	474	90
DHHT (R/D)	$4x$	187740	96	78204	1010878	12.9	296	35	529	262	89	34	474	90
DHHT (D/R)	x	680869	86	297043	1501345	5.0	292	54	673	251	86	50	507	75
DHHT (D/R)	$2x$	378599	95	164065	1826258	11.1	294	54	673	253	86	50	525	78
DHHT (D/R)	$4x$	194566	98	85817	785340	9.1	296	47	749	253	85	46	534	71
DHHT (D/D)	x	790228	100	347341	2217005	6.3	265	47	614	229	86	46	467	76
DHHT (D/D)	$2x$	395114	100	166385	1107819	6.6	275	47	670	236	86	46	498	74
DHHT (D/D)	$4x$	197557	100	89610	480834	5.3	280	47	670	240	86	44	527	79
DHHT (Multi)	x	756912	96	446634	1336989	2.9	296	51	704	262	89	13	614	87
DHHT (Multi)	$2x$	391342	99	268151	672179	2.5	295	51	704	260	88	13	614	87
DHHT (Multi)	$4x$	197297	99	152556	378468	2.4	301	28	741	262	87	13	614	83

Table 4.6: *Results of experiments of type A on office PCs (Windows).*

Load Balancer	# Procs.	CPU Time					# Migs. (Def.)			# Migrations (Bulk)					
		avg [s]	*avg* [%]	*min* [s]	*max* [s]	*max* / *min*	*avg*	*min*	*max*	*avg*	[% Def.]	*min*	[% Def.]	*max*	[% Def.]
DHHT (R/R)	x	2601920	63	1108057	6682838	6.0	27	0	252	26	96	0		252	100
DHHT (R/R)	$2x$	1672058	81	599578	6606004	11.0	37	0	348	35	95	0		339	97
DHHT (R/R)	$4x$	924323	89	306667	6255070	20.4	36	0	449	35	97	0		443	99
DHHT (R/D)	x	2886232	70	925479	6681205	7.2	39	0	247	38	97	0		246	100
DHHT (R/D)	$2x$	1783931	86	459220	6287462	13.6	41	0	425	39	95	0		413	97
DHHT (R/D)	$4x$	936947	91	278870	5118833	18.3	40	0	425	38	95	0		413	97
DHHT (D/R)	x	2969277	72	846060	6683676	7.9	29	0	243	28	97	0		241	99
DHHT (D/R)	$2x$	1886265	91	533219	6391941	11.9	29	0	376	28	97	0		354	94
DHHT (D/R)	$4x$	1012264	98	275388	6264394	22.7	23	0	376	22	96	0		354	94
DHHT (D/D)	x	4132389	100	849089	6682840	7.8	4	0	86	4	100	0		86	100
DHHT (D/D)	$2x$	2066194	100	522795	6284854	12.0	14	0	272	14	100	0		242	89
DHHT (D/D)	$4x$	1033097	100	345819	3092967	8.9	17	0	272	17	100	0		242	89
DHHT (Multi)	x	2931728	71	685161	6501076	9.4	31	0	283	29	94	0		275	97
DHHT (Multi)	$2x$	1754360	85	513264	6091513	11.8	34	0	731	33	97	0		703	96
DHHT (Multi)	$4x$	996401	96	407639	3619062	8.8	32	0	731	31	97	0		703	96

Table 4.7: *Results of experiments of type A on pool PCs (Linux).*

Load Balancer	# Procs.	CPU Time					# Migs. (Def.)			# Migrations (Bulk)			
		avg [s]	*avg* [%]	*min* [s]	*max* [s]	*max* / *min*	*avg*	*min*	*max*	*avg*	[% Def.]	*min*	[% Def.]	*max*	[% Def.]
DHHT (R/R)	x	1961483	65	333663	6090658	18.2	66	2	505	57	86	2	100	486	96
DHHT (R/R)	$2x$	1222308	81	217194	5975632	27.5	72	2	519	63	88	2	100	505	97
DHHT (R/R)	$4x$	673194	90	128862	3884332	30.1	78	2	1352	67	86	2	100	1336	99
DHHT (R/D)	x	2247914	75	903143	5697631	6.3	68	2	498	61	90	2	100	485	97
DHHT (R/D)	$2x$	1282713	85	338427	5148606	15.2	79	2	1220	68	86	2	100	1206	99
DHHT (R/D)	$4x$	704179	94	208052	5405922	25.9	74	0	1220	64	86	0		1206	99
DHHT (D/R)	x	2206398	73	502850	5737753	11.4	51	2	330	45	88	2	100	314	95
DHHT (D/R)	$2x$	1342575	89	330031	5691317	17.2	59	2	483	51	86	2	100	475	98
DHHT (D/R)	$4x$	733766	98	176987	5235588	29.5	68	2	1138	58	85	2	100	1098	96
DHHT (D/D)	x	3005205	100	818204	6268406	7.6	14	0	103	12	86	0		82	80
DHHT (D/D)	$2x$	1502603	100	538205	5131090	9.5	32	0	260	28	88	0		255	98
DHHT (D/D)	$4x$	751301	100	254238	2539233	9.9	46	0	917	41	89	0		903	98
DHHT (Multi)	x	2171853	72	780507	5701513	7.3	66	2	577	57	86	2	100	563	98
DHHT (Multi)	$2x$	1355383	90	375909	5095559	13.5	65	2	1171	56	86	2	100	804	69
DHHT (Multi)	$4x$	736835	98	331081	2691983	8.1	66	0	1276	56	85	0		1008	79

The DHHT (Det., Det.) variant achieves full and DHHT (Multi) almost full utilization. The smoothness of the employed hash functions noticeably effects the balancing quality: Using a standard hash function with $n \cdot \log n$ smoothness, the initial situation worsens with an increasing number of processes, whereas using the deterministic hash function with ϕ^2 smoothness is balancing is not only initially better but also further improves with an increasing number of processes. By far the best balancing however is achieved using the DHHT (Multi) load balancer. With respect to the average number of migrations per process there is not much difference between the particular algorithms, and with respect to the running time of 14 days the values are at a good low level of less than one migration per hour.

The bulk migration mode is defined as follows: When the load of a peer changes and the load balancer asks a process to migrate as a consequence, it may happen that the load of the peer at the migration target also changes very soon or even before the migration is completed. In such a case, the process has to migrate again within a very short time interval. Theoretically, this can even happen several times in succession. In reality however, it usually takes a bit of time until the next possible migration point of a process reached; when then actually migrating, intermediate migration targets can be skipped. As our simulation is clocked, the peers simultaneously update their load and thus, sometimes intermediate migration targets are produced. In the bulk migration mode, we skip these intermediate migration targets in order to identify artifacts caused by the clocked simulation and to resemble reality as close as possible. The results in Table 4.5 show that, independently of the load balancing algorithm, approximately 10–15% of the migrations can be saved.

The results of our experiments using the office and pool PCs availability profiles (cf. tables 4.6 and 4.7) show also a positive, but even heavier impact on the utilization percentage with an increasing number of processes. Again the DHHT (Det., Det.) variant achieves full utilization and — in terms of balancing quality — it is on roughly the same level as the DHHT (Multi) load balancer, which performs worse. In the office PCs availability profile, the number of migrations ranges approximately from one migration every 8 to 18 hours. These really good values confirm our assumption that the availability factor in the particular computers is quite stable for reasonably long time intervals. At this low level, not much can be saved any more in the bulk migration mode. In the pool PCs availability profile, the number of migrations ranges approximately from one migration every 3 to 12 hours, depending on the load balancer. Like in the notebooks case, approximately 10–15% of the migrations can be saved in the bulk mode. Altogether, the DHHT (Det., Det.) variant clearly performs best in both the office and pool PC scenarios.

4.4.2 Experiments of Type B

In this set of experiments we fix the availability of all peers to 100% and feed the load balancers with the synthetic job stream inputs described previously. We perform two runs of this set of experiments, one with overload factor 1 (i.e., with an ideal total system load) and one with factor 4.

In the following, we call the factor between the actual average running time and the length of the processes the *process stretch factor*; analogously, we define the *job stretch factor*.

For the HPC2N input profile (experiment 10), figures 4.2 and 4.3 depict the average and maximum actual duration, respectively, of the processes, grouped by their length, i.e., the duration they would have at 100% availability and a processor performance factor of 1. As one can see, both Work Stealing variants perform best (and almost the same), and the processes have to wait in the deques only for a very short time, independently of their length. The DHHT load balancer performs second best when using deterministic hashing both peers and processes, and it performs worst with a significantly bad slowdown when using a default hash function with $n \cdot \log n$ smoothness. The DHHT (Multi) load balancer yields a medium quality result.

The maximum duration of the processes, shown in Fig. 4.3, is important as the duration of a superstep is determined by the slowest process in a BSP program. The least possible deviation from the average value is desired. We clearly miss this goal using a default hash function, and — interestingly — the situation even worsens when applying the deterministic method only to the peers but not to the processes. As a result, the average duration of the jobs (cf. Fig. 4.4) increases only slightly compared to the average duration of the processes for Work Stealing, DHHT (Det., Det.), and DHHT (Rnd., Det.), whereas it increases by a factor of approximately 2–3 in the other cases. Table 4.8 shows the process and job stretch factors for all load balancers and all input profiles. Note that stretch factors of less than 1 are possible because the vast majority of the peers has a processor performance factor of more than 1. We see that, with the best of our candidates, DHHT (Det., Det.), we are able to keep up with the Work Stealing algorithms up to a factor of approximately 1.25.

Fig. 4.5 shows how long, depending on their release time, the jobs had to remain in the ready queue until the overall system load was sufficiently low so that they could be started. Thus, low curves mean a high overall throughput. The curves end where the release time and the pending time plus the job duration sum up to the duration of the experiment (2 weeks). The factors between these curves mainly depend on and correspond to the stretch factors of the particular load balancer.

To sum up, we are not able to keep up with the Work Stealing algorithms in

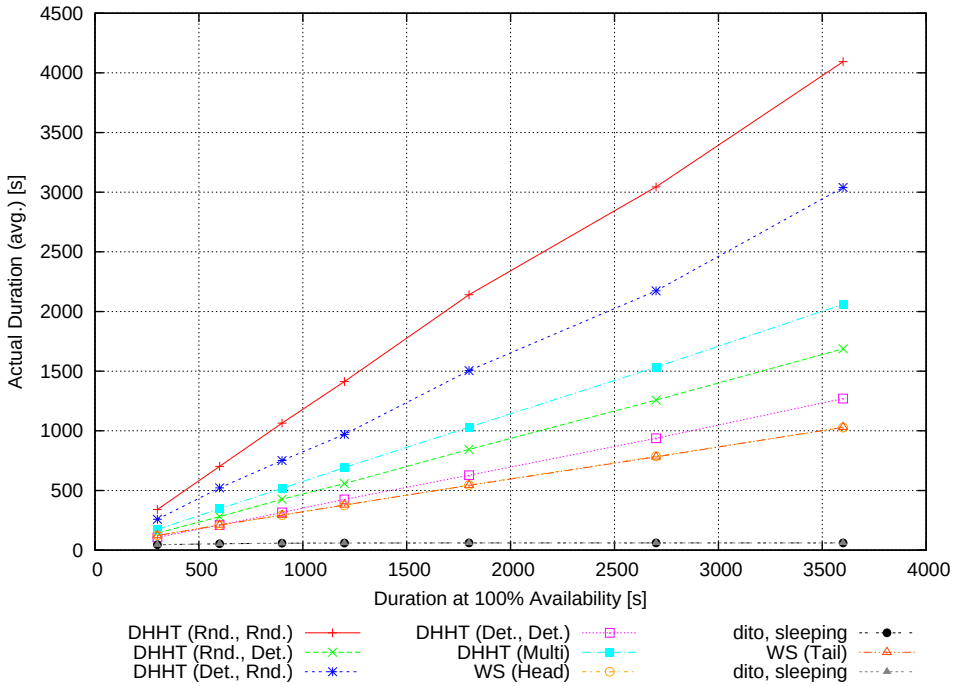

Figure 4.2: *Actual average duration of processes in experiment 10.*

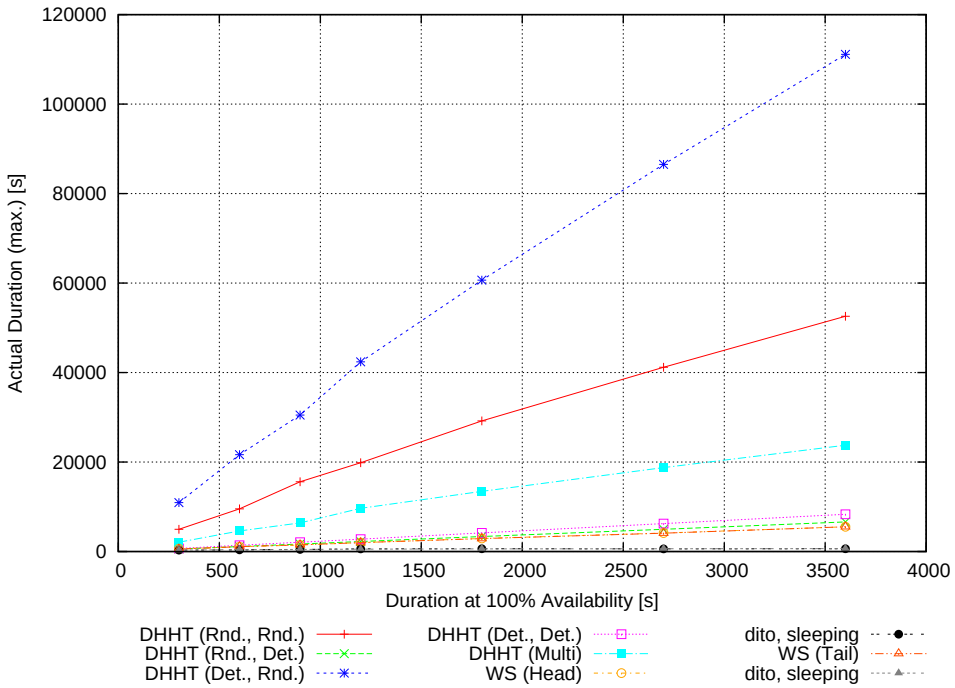

Figure 4.3: *Actual maximum duration of processes in experiment 10.*

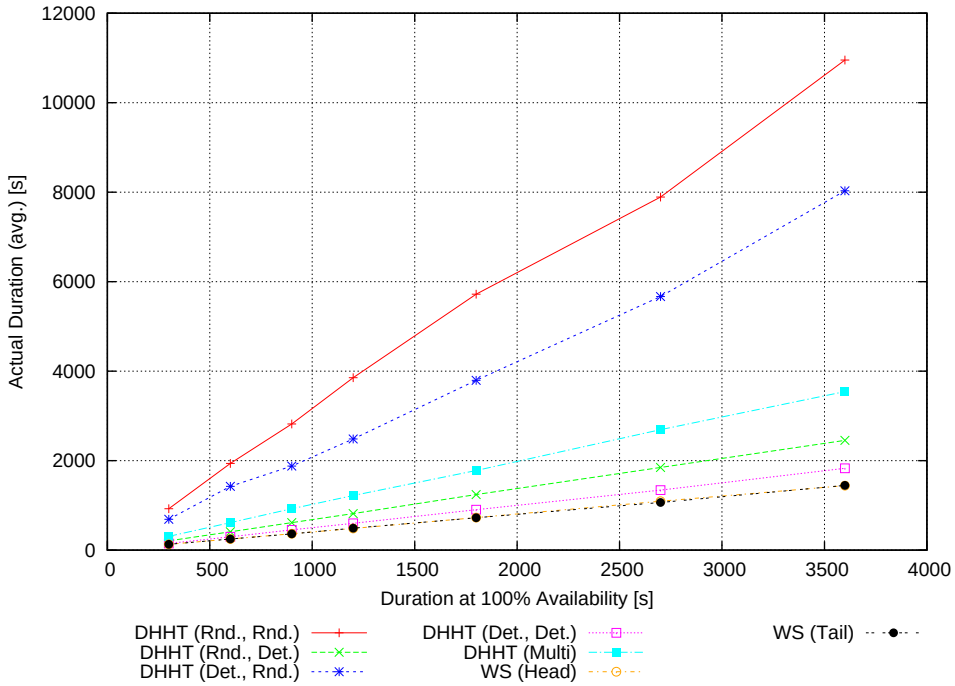

Figure 4.4: *Actual average duration of jobs in experiment 10.*

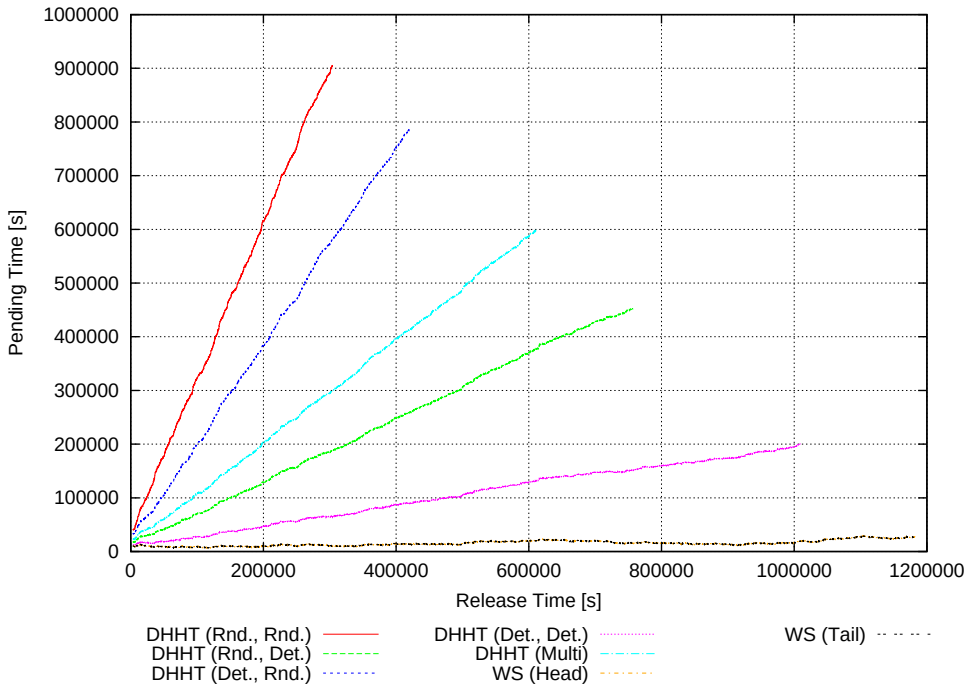

Figure 4.5: *Pending time of jobs in experiment 10.*

Table 4.8: *Results of experiments of type B on with ideal load.*

Exp.	Input Profile	Load Balancer	Stretch Factor	
			Procs.	Jobs
10	HPC2N	DHHT (Rnd., Rnd.)	1.157	3.100
		DHHT (Rnd., Det.)	0.469	0.684
		DHHT (Det., Rnd.)	0.834	2.168
		DHHT (Det., Det.)	0.351	0.502
		DHHT (Multi)	0.574	1.003
		WS (Head)	0.313	0.407
		WS (Tail)	0.313	0.405
11	LLNL Atlas	DHHT (Rnd., Rnd.)	3.203	11.823
		DHHT (Rnd., Det.)	2.293	3.373
		DHHT (Det., Rnd.)	2.348	13.305
		DHHT (Det., Det.)	1.726	2.368
		DHHT (Multi)	1.796	3.682
		WS (Head)	1.055	1.126
		WS (Tail)	1.056	1.120
12	LLNL Thunder	DHHT (Rnd., Rnd.)	2.381	6.284
		DHHT (Rnd., Det.)	1.700	1.591
		DHHT (Det., Rnd.)	1.765	5.435
		DHHT (Det., Det.)	1.329	1.109
		DHHT (Multi)	1.438	1.920
		WS (Head)	0.808	0.604
		WS (Tail)	0.808	0.608
13	SDSC BLUE	DHHT (Rnd., Rnd.)	1.687	8.035
		DHHT (Rnd., Det.)	0.919	1.619
		DHHT (Det., Rnd.)	1.202	6.419
		DHHT (Det., Det.)	0.673	1.167
		DHHT (Multi)	0.864	2.273
		WS (Head)	0.512	0.699
		WS (Tail)	0.512	0.700
14	SDSC DataStar	DHHT (Rnd., Rnd.)	2.137	8.981
		DHHT (Rnd., Det.)	1.497	1.953
		DHHT (Det., Rnd.)	1.584	6.939
		DHHT (Det., Det.)	1.088	1.396
		DHHT (Multi)	1.197	2.458
		WS (Head)	0.713	0.773
		WS (Tail)	0.713	0.773

this setting; DHHT (Det., Det.) is the best of our candidates, which only suffers a performance drawback of a factor of approximately 1.25.

The main difference of the other four input profiles is the parallelism ranging from 4–1024 (instead of 1–64). The results are similar, but the factors are higher. The corresponding graphs for these experiments (11–14) can be found in Appendix A.1. From Table 4.8 one can see that the performance loss of DHHT (Det., Det.) against Work Stealing is a factor of approximately 2 in terms of the job stretch factor. Note that it is possible to have a lower job stretch factor than process stretch factor when a few jobs with very high parallelism perform badly and lots of jobs with very low parallelism perform well.

As a second step, we repeated experiments 10–14 under the overloaded conditions. Figures 4.6 – 4.9 show the aggregated results for the scenario corresponding to Figures 4.2 – 4.5, and Table 4.9 sums up the results for all five input profiles, corresponding to Table 4.8.

The most noticeable difference in Fig. 4.6 is that the Work Stealing curves still are the ones that grow least, but are notably shifted upwards. At first glance, this may appear strange; but when looking at the Work Stealing sleeping curves, we notice that all processes wait in the deques approximately for the same time, independently of their length. As a consequence 5-minute-jobs take more than 5 times as long as when executed using DHHT (Det., Det.), and 1-hour-jobs are less than 1.5 times faster, leading to an overall performance loss for Work Stealing on average.

Concerning the maximum process execution duration (cf. Fig. 4.7), we do not only revisit this effect, but we also see a big difference between the two Work Stealing variants: when stealing processes from the tail of the queues, the worst case is almost five times as worse as compared to WS (Head). DHHT (Det., Det.) outperforms all other algorithms, and also does so concerning the average actual job duration (cf. Fig. 4.8) and with respect to the overall system throughput (cf. Fig. 4.9).

Note: From Table 4.9 one can see that WS (Head) is very close to — and in one case even better then — DHHT (Det., Det.) with respect to the process stretch factor. The corresponding graphs for the experiments 16–19 in Appendix A.2 seem to suggest that WS (Head) did even perform much better concerning the average actual process duration; but though a loss of WS (Head) is only apparent for short processes and a gain is present in many cases, we have to take into account the factors of the loss and the gain as well as the number of samples for the particular values (from Table 4.3 we see, for example, that the vast majority of processes is short in the input profiles of experiments 17–19); this leads to the average values in Table 4.9.

However, with respect to the job stretch factor, DHHT (Det., Det.) performs

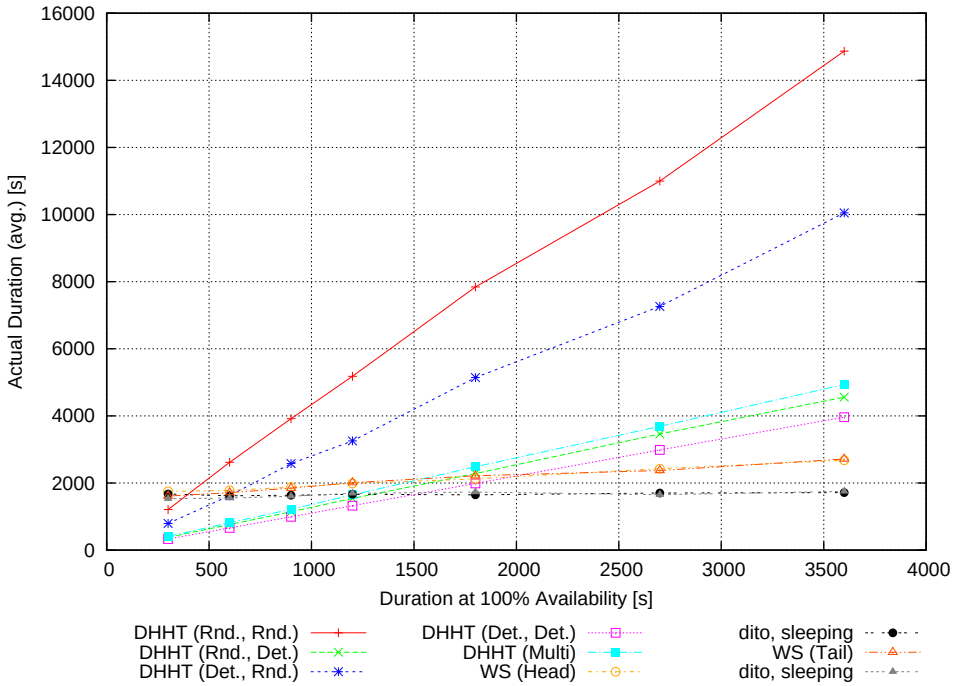

Figure 4.6: *Actual average duration of processes in experiment 15.*

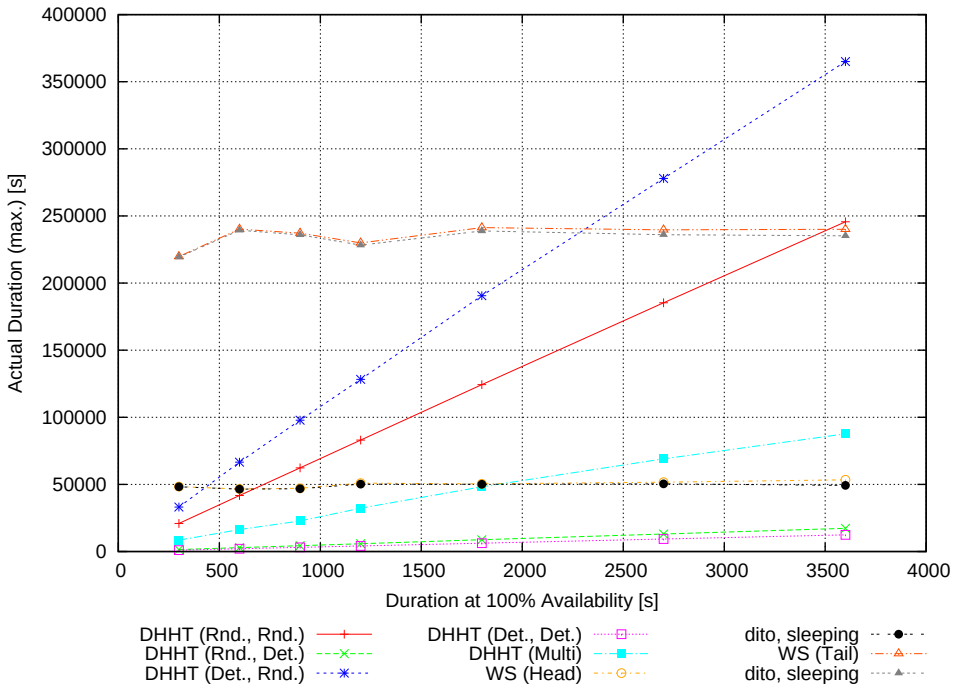

Figure 4.7: *Actual maximum duration of processes in experiment 15.*

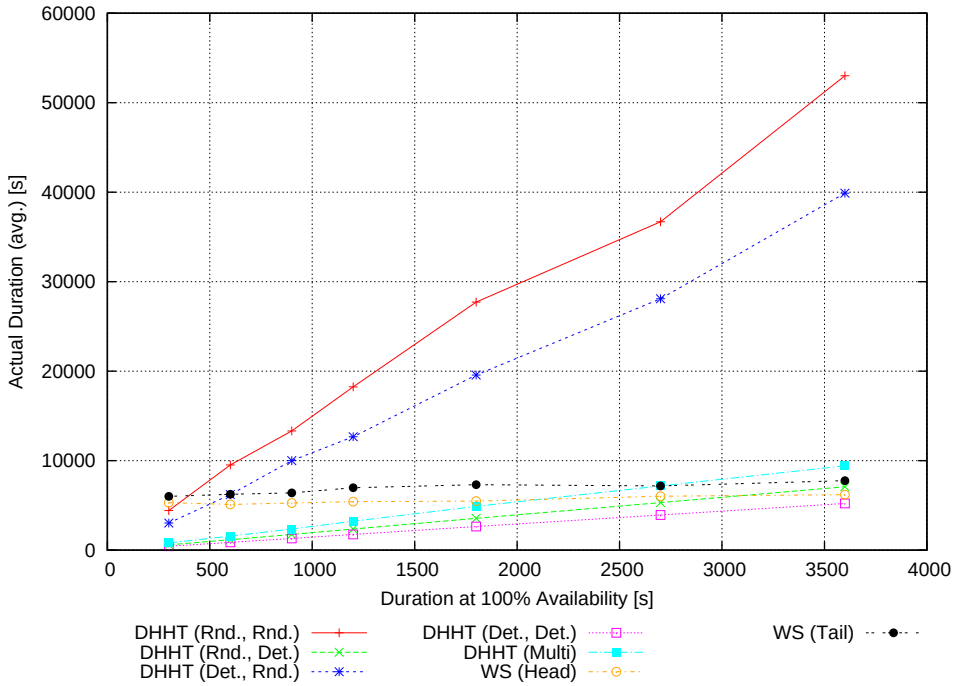

Figure 4.8: *Actual average duration of jobs in experiment 15.*

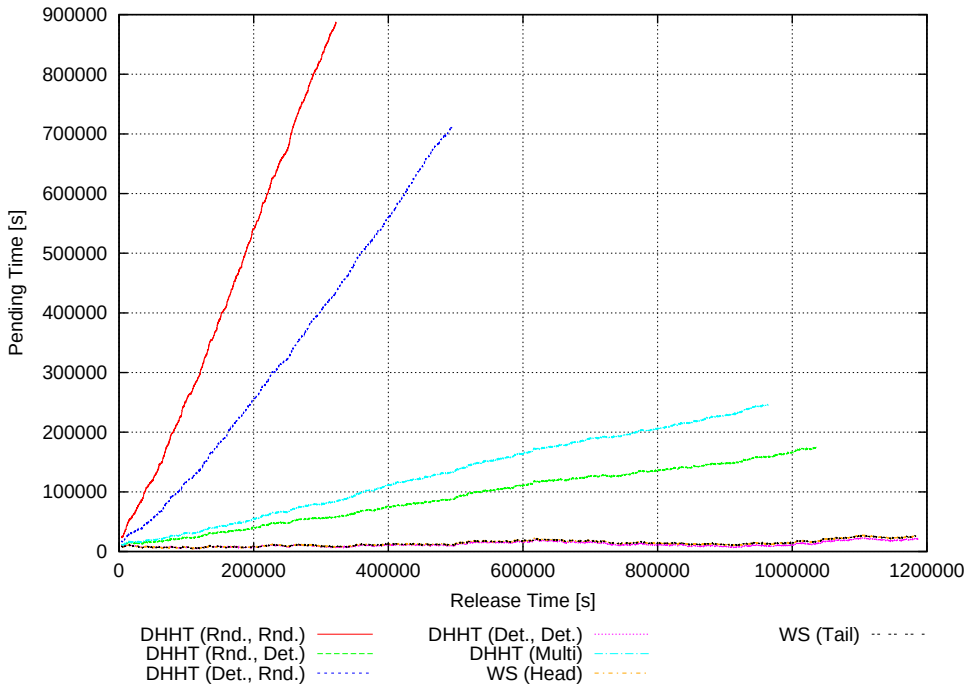

Figure 4.9: *Pending time of jobs in experiment 15.*

Table 4.9: *Results of experiments of type B on with overload.*

Exp.	Input Profile	Load Balancer	Stretch Factor	
			Procs.	Jobs
15	HPC2N	DHHT (Rnd., Rnd.)	4.220	14.834
		DHHT (Rnd., Det.)	1.273	1.964
		DHHT (Det., Rnd.)	2.759	10.703
		DHHT (Det., Det.)	1.102	1.454
		DHHT (Multi)	1.369	2.657
		WS (Head)	1.717	4.737
		WS (Tail)	1.676	5.760
16	LLNL Atlas	DHHT (Rnd., Rnd.)	6.119	46.178
		DHHT (Rnd., Det.)	2.959	6.272
		DHHT (Det., Rnd.)	4.120	56.115
		DHHT (Det., Det.)	2.359	4.264
		DHHT (Multi)	2.469	7.757
		WS (Head)	2.469	21.393
		WS (Tail)	2.266	44.587
17	LLNL Thunder	DHHT (Rnd., Rnd.)	5.425	30.816
		DHHT (Rnd., Det.)	2.496	3.668
		DHHT (Det., Rnd.)	3.664	26.385
		DHHT (Det., Det.)	1.957	2.530
		DHHT (Multi)	2.127	4.640
		WS (Head)	2.149	11.382
		WS (Tail)	2.042	16.123
18	SDSC BLUE	DHHT (Rnd., Rnd.)	4.729	38.136
		DHHT (Rnd., Det.)	1.662	3.926
		DHHT (Det., Rnd.)	3.063	31.851
		DHHT (Det., Det.)	1.366	2.633
		DHHT (Multi)	1.626	5.704
		WS (Head)	1.870	14.464
		WS (Tail)	1.802	21.098
19	SDSC DataStar	DHHT (Rnd., Rnd.)	5.193	43.326
		DHHT (Rnd., Det.)	2.190	4.668
		DHHT (Det., Rnd.)	3.481	37.111
		DHHT (Det., Det.)	1.759	3.156
		DHHT (Multi)	1.956	6.039
		WS (Head)	1.995	15.309
		WS (Tail)	1.889	22.900

best in all cases, and WS (Head) suffers a performance loss against DHHT (Det., Det.) of a factor of approximately 3–6.

4.4.3 Experiments of Type C

Now we repeat the experiments of type B again, but apply the three external workload profiles in order to obtain completely realistic circumstances.

The results for the ideal job load case using the notebook load profile are summarized in Table 4.10. Detailed results are available via figures A.33 – A.52 in Appendix A.3.1. As one can see, WS (Head) performs best concerning both process and job stretch factors for all input profiles. For the HPC2N input profile, the DHHT (Rnd., Det.), DHHT (Det., Det.), and DHHT (Multi) load balancers yield roughly the same quality at the second-best level; for all other input profiles, however, solely the DHHT (Multi) load balancer performs best among the DHHT variants concerning both process and job stretch factors. In all cases, the DHHT (Multi) load balancer needs roughly twice the time than WS (Head). So, the DHHT (Multi) load balancer obviously is able to handle the high diversity of the external workload in the notebook profile better the other DHHT load balancer variants. From the figures it can also be seen that Work Stealing achieves the best and DHHT (Multi) the second-best throughput. Concerning the number of migrations per hour, all DHHT-based algorithms are roughly at the same level, whereas Work Stealing produces approximately 3–4 times as many migrations (except for the LLNL Atlas input profile, where it causes only slightly more). In the bulk migration mode, we can save approximately 10–15% of the migrations.

The results of our experiments for the ideal job load case using the office PC load profile are very similar to the results obtained for our experiments with 100% availability, which is not very surprising since we found out in Section 4.3.1 that the availability is very high and the diversity low in this load profile. Table 4.11 shows that Work Stealing performs best again and DHHT (Det., Det.) second-best with a performance drawback of a factor of approximately 1.25 (for the HPC2N input profile) and approximately 2 (for the other input profiles), respectively. Detailed results are available via figures A.53 – A.72 in Appendix A.3.2. Concerning the number of migrations, DHHT (Det., Det.) performs best with only 0.016–0.052 migrations per hour, whereas Work Stealing needs 2.7–7.8 migrations per hour, which is 50–400 times as much in the particular cases. In the bulk migration mode, no noteworthy savings can be observed.

When now applying the pool PC load profile, Work Stealing again perform best (cf. Table 4.12). However, figures A.73 – A.92 in Appendix A.3.3 show that

Table 4.10: *Results of experiments of type C with ideal load on notebooks (Windows).*

Input Profile	Load Balancer	Stretch Factor		# Migs. / Hour		
		Procs.	Jobs	Def.	Bulk	[% Def.]
HPC2N	DHHT (Rnd., Rnd.)	1.345	1.881	1.007	0.898	89
	DHHT (Rnd., Det.)	0.864	0.987	0.902	0.807	89
	DHHT (Det., Rnd.)	1.286	1.912	0.961	0.83	86
	DHHT (Det., Det.)	0.836	1.006	0.807	0.688	85
	DHHT (Multi)	0.84	1.091	0.994	0.864	87
	WS (Head)	0.525	0.596	4.104		
	WS (Tail)	0.525	0.596	4.046		
LLNL Atlas	DHHT (Rnd., Rnd.)	12.233	10.026	0.934	0.842	90
	DHHT (Rnd., Det.)	10.978	7.887	0.967	0.862	89
	DHHT (Det., Rnd.)	14.118	9.923	0.919	0.781	85
	DHHT (Det., Det.)	9.854	8.11	0.828	0.707	85
	DHHT (Multi)	9.848	5.639	0.947	0.824	87
	WS (Head)	4.836	3.402	1.039		
	WS (Tail)	4.838	3.913	1.036		
LLNL Thunder	DHHT (Rnd., Rnd.)	9.927	4.47	0.953	0.857	90
	DHHT (Rnd., Det.)	6.866	2.97	1.02	0.905	89
	DHHT (Det., Rnd.)	7.676	5.101	0.958	0.829	87
	DHHT (Det., Det.)	7.217	2.989	0.885	0.762	86
	DHHT (Multi)	6.335	2.615	1.014	0.884	87
	WS (Head)	3.159	1.235	3.259		
	WS (Tail)	3.172	1.277	3.247		
SDSC BLUE	DHHT (Rnd., Rnd.)	4.188	4.821	1.036	0.927	89
	DHHT (Rnd., Det.)	3.508	2.946	0.928	0.831	89
	DHHT (Det., Rnd.)	4.296	5.13	1.014	0.875	86
	DHHT (Det., Det.)	3.521	3.23	0.858	0.741	86
	DHHT (Multi)	2.815	2.671	1.002	0.876	87
	WS (Head)	1.511	1.48	3.974		
	WS (Tail)	1.51	1.542	3.971		
SDSC DataStar	DHHT (Rnd., Rnd.)	6.377	5.725	0.955	0.851	89
	DHHT (Rnd., Det.)	5.936	3.71	0.999	0.903	90
	DHHT (Det., Rnd.)	7.076	6.467	0.931	0.798	86
	DHHT (Det., Det.)	6.381	3.792	0.878	0.761	87
	DHHT (Multi)	5.04	3.247	1.002	0.876	87
	WS (Head)	2.665	1.782	3.285		
	WS (Tail)	2.665	1.829	3.246		

Table 4.11: *Results of experiments of type C with ideal load on office PCs (Windows).*

		Stretch Factor		# Migs. / Hour		
Input Profile	**Load Balancer**	**Procs.**	**Jobs**	**Def.**	**Bulk**	[% Def.]
HPC2N	DHHT (Rnd., Rnd.)	1.084	2.65	0.119	0.115	96
	DHHT (Rnd., Det.)	0.479	0.686	0.11	0.106	97
	DHHT (Det., Rnd.)	0.813	1.943	0.071	0.069	97
	DHHT (Det., Det.)	0.372	0.538	0.016	0.016	100
	DHHT (Multi)	0.586	1.012	0.114	0.11	96
	WS (Head)	0.323	0.421	6.545		
	WS (Tail)	0.324	0.421	6.526		
LLNL Atlas	DHHT (Rnd., Rnd.)	3.229	9.769	0.109	0.107	97
	DHHT (Rnd., Det.)	2.462	3.431	0.122	0.118	96
	DHHT (Det., Rnd.)	2.454	11.64	0.074	0.072	97
	DHHT (Det., Det.)	1.883	2.51	0.052	0.05	97
	DHHT (Multi)	1.892	3.653	0.115	0.112	97
	WS (Head)	1.13	1.237	2.738		
	WS (Tail)	1.13	1.234	2.731		
LLNL Thunder	DHHT (Rnd., Rnd.)	2.363	5.488	0.116	0.111	96
	DHHT (Rnd., Det.)	1.791	1.566	0.12	0.117	97
	DHHT (Det., Rnd.)	1.826	4.826	0.074	0.072	97
	DHHT (Det., Det.)	1.481	1.206	0.045	0.043	97
	DHHT (Multi)	1.512	1.949	0.111	0.108	97
	WS (Head)	0.843	0.66	6.291		
	WS (Tail)	0.842	0.658	6.306		
SDSC BLUE	DHHT (Rnd., Rnd.)	1.639	6.815	0.114	0.11	97
	DHHT (Rnd., Det.)	0.967	1.658	0.112	0.108	97
	DHHT (Det., Rnd.)	1.217	5.423	0.074	0.072	97
	DHHT (Det., Det.)	0.733	1.271	0.033	0.032	97
	DHHT (Multi)	0.907	2.233	0.111	0.108	97
	WS (Head)	0.537	0.768	7.823		
	WS (Tail)	0.537	0.77	7.806		
SDSC DataStar	DHHT (Rnd., Rnd.)	2.117	7.77	0.123	0.118	97
	DHHT (Rnd., Det.)	1.557	1.971	0.115	0.112	98
	DHHT (Det., Rnd.)	1.636	6.278	0.076	0.074	97
	DHHT (Det., Det.)	1.183	1.479	0.04	0.039	97
	DHHT (Multi)	1.258	2.415	0.112	0.109	97
	WS (Head)	0.758	0.852	7.152		
	WS (Tail)	0.757	0.854	7.149		

Table 4.12: *Results of experiments of type C with ideal load on pool PCs (Linux).*

		Stretch Factor		# Migs. / Hour		
Input Profile	**Load Balancer**	**Procs.**	**Jobs**	**Def.**	**Bulk**	[% Def.]
HPC2N	DHHT (Rnd., Rnd.)	1.404	3.825	0.225	0.197	87
	DHHT (Rnd., Det.)	0.611	0.864	0.169	0.145	86
	DHHT (Det., Rnd.)	1.099	2.473	0.192	0.171	89
	DHHT (Det., Det.)	0.501	0.819	0.047	0.04	86
	DHHT (Multi)	0.748	1.339	0.216	0.193	89
	WS (Head)	0.422	0.661	4.944		
	WS (Tail)	0.423	0.667	4.966		
LLNL Atlas	DHHT (Rnd., Rnd.)	4.57	24.976	0.205	0.178	87
	DHHT (Rnd., Det.)	3.337	4.918	0.195	0.17	87
	DHHT (Det., Rnd.)	3.557	14.515	0.194	0.166	85
	DHHT (Det., Det.)	2.687	7.076	0.126	0.115	91
	DHHT (Multi)	2.511	5.982	0.229	0.197	86
	WS (Head)	1.497	3.254	2.296		
	WS (Tail)	1.492	2.887	2.293		
LLNL Thunder	DHHT (Rnd., Rnd.)	3.233	10.163	0.219	0.188	86
	DHHT (Rnd., Det.)	2.357	2.024	0.199	0.175	88
	DHHT (Det., Rnd.)	2.553	6.309	0.188	0.167	89
	DHHT (Det., Det.)	1.973	2.227	0.1	0.089	89
	DHHT (Multi)	1.91	2.716	0.232	0.204	88
	WS (Head)	1.139	1.204	4.858		
	WS (Tail)	1.137	1.147	4.892		
SDSC BLUE	DHHT (Rnd., Rnd.)	2.186	12.966	0.202	0.179	89
	DHHT (Rnd., Det.)	1.264	2.029	0.196	0.171	87
	DHHT (Det., Rnd.)	1.773	7.654	0.184	0.164	89
	DHHT (Det., Det.)	1.066	2.621	0.088	0.08	90
	DHHT (Multi)	1.191	3.229	0.219	0.191	87
	WS (Head)	0.705	1.579	6.188		
	WS (Tail)	0.706	1.565	6.164		
SDSC DataStar	DHHT (Rnd., Rnd.)	2.949	14.353	0.219	0.187	85
	DHHT (Rnd., Det.)	2.008	2.68	0.201	0.177	88
	DHHT (Det., Rnd.)	2.301	9.768	0.18	0.161	89
	DHHT (Det., Det.)	1.703	3.481	0.1	0.089	89
	DHHT (Multi)	1.708	3.761	0.228	0.201	88
	WS (Head)	0.988	1.74	5.759		
	WS (Tail)	0.989	1.827	5.783		

DHHT (Rnd., Det.) is the only DHHT variant that does not suffer from heavy outliers in the process duration (cf. maximum process duration figures). As a consequence, DHHT (Rnd., Det.) mostly outperforms our best DHHT candidates DHHT (Det., Det.) and DHHT (Multi) in this case. Its performance drawback in terms of the job stretch factor against Work Stealing ranges from approximately 1.25 to 2. Work Stealing again produces significantly more migrations, in particular approximately 10–30 times more than the second-best load balancer. Like in the notebook load profile, we can save around 10–15% of the migrations in the bulk migration mode.

Eventually, we repeat these experiments under overload conditions. For the notebook load profile, Table 4.13 shows that Work Stealing still performs best in terms of the process stretch factor (except for the HPC2N input profile, where the DHHT (Multi) algorithm is best), but like in the second series of the type B experiments, Work Stealing is significantly outperformed concerning the job stretch factor: for all input profiles, jobs scheduled using the DHHT (Multi) algorithm only need approximately half the time compared to Work Stealing. Though DHHT (Det., Det.) does not keep up with the DHHT (Multi) algorithm, it is anyway still better then Work Stealing in this case. Compared to the type B experiments, the gain of the best DHHT-based algorithm against Work Stealing is lower, and there are already single cases where Work Stealing performs best (cf. figures A.93 – A.112 in Appendix A.4.1). Concerning the number of migrations, all DHHT-based algorithms produce roughly the same result than in the ideal load case, whereas Work Stealing now only needs less than half the number of migrations compared to the ideal load case; anyway, Work Stealing still requires more migrations than the DHHT-based algorithms (except for the LLNL Atlas input profile).

Using the office PC load profile, the results of our experiments for the overload case are again very similar to the results obtained for the same experiments with 100% availability: Table 4.14 shows that DHHT (Det., Det.) performs best not only with respect to the job stretch factor, but also even in most cases concerning the process stretch factor. WS (First) suffers a performance drawback of a factor of approximately 3–7 in terms of the job stretch factor (and the WS (Last) variant is even much worse). Detailed results are available via figures A.113 – A.132 in Appendix A.4.2. In terms of the number of migrations, all DHHT-based algorithms again produce roughly the same result than in the ideal load case; and again Work Stealing now only needs less than half the number of migrations compared to the ideal load case, which still is approximately 25–65 times as many migrations as the DHHT-based algorithms require.

For the pool PC load profile, Table 4.15 shows that DHHT (Det., Det.) performs best with respect to the process stretch factor for three input profiles,

Table 4.13: *Results of experiments of type C with overload on notebooks (Windows).*

		Stretch Factor		# Migs. / Hour		
Input Profile	**Load Balancer**	**Procs.**	**Jobs**	**Def.**	**Bulk**	[% Def.]
HPC2N	DHHT (Rnd., Rnd.)	3.205	5.460	0.968	0.865	89
	DHHT (Rnd., Det.)	1.883	2.573	0.842	0.754	90
	DHHT (Det., Rnd.)	3.050	5.617	0.961	0.818	85
	DHHT (Det., Det.)	1.738	2.458	0.738	0.631	85
	DHHT (Multi)	1.706	2.407	0.967	0.847	87
	WS (Head)	2.231	4.765	1.193		
	WS (Tail)	2.155	5.538	1.785		
LLNL Atlas	DHHT (Rnd., Rnd.)	12.193	19.005	0.970	0.861	89
	DHHT (Rnd., Det.)	12.139	12.544	0.939	0.831	89
	DHHT (Det., Rnd.)	11.008	19.382	0.969	0.836	86
	DHHT (Det., Det.)	10.442	13.135	0.909	0.784	86
	DHHT (Multi)	9.167	8.408	0.974	0.852	87
	WS (Head)	6.421	17.961	0.529		
	WS (Tail)	6.339	30.248	0.7		
LLNL Thunder	DHHT (Rnd., Rnd.)	9.086	11.976	0.980	0.872	89
	DHHT (Rnd., Det.)	8.446	6.179	0.927	0.825	89
	DHHT (Det., Rnd.)	8.756	12.541	1.006	0.852	85
	DHHT (Det., Det.)	7.739	5.843	0.853	0.725	85
	DHHT (Multi)	7.442	4.712	0.960	0.840	87
	WS (Head)	4.740	8.889	1.149		
	WS (Tail)	4.644	13.343	1.747		
SDSC BLUE	DHHT (Rnd., Rnd.)	5.652	12.031	0.984	0.885	90
	DHHT (Rnd., Det.)	4.224	5.909	0.918	0.820	89
	DHHT (Det., Rnd.)	5.569	12.898	1.001	0.852	85
	DHHT (Det., Det.)	4.331	6.108	0.798	0.684	86
	DHHT (Multi)	3.309	4.887	0.998	0.868	86
	WS (Head)	3.125	11.950	1.417		
	WS (Tail)	3.048	16.266	2.193		
SDSC DataStar	DHHT (Rnd., Rnd.)	8.247	14.349	0.966	0.864	89
	DHHT (Rnd., Det.)	6.680	7.649	0.909	0.811	89
	DHHT (Det., Rnd.)	7.349	14.693	1.009	0.863	86
	DHHT (Det., Det.)	7.867	7.615	0.861	0.739	86
	DHHT (Multi)	5.682	5.360	0.967	0.847	87
	WS (Head)	4.259	11.851	1.223		
	WS (Tail)	4.141	17.201	1.857		

Table 4.14: *Results of experiments of type C with overload on office PCs (Windows).*

		Stretch Factor		# Migs. / Hour		
Input Profile	**Load Balancer**	**Procs.**	**Jobs**	**Def.**	**Bulk**	[% Def.]
HPC2N	DHHT (Rnd., Rnd.)	3.851	13.390	0.119	0.116	97
	DHHT (Rnd., Det.)	1.279	1.973	0.107	0.103	96
	DHHT (Det., Rnd.)	2.495	8.844	0.067	0.064	95
	DHHT (Det., Det.)	1.132	1.524	0.039	0.038	96
	DHHT (Multi)	1.384	2.551	0.108	0.104	96
	WS (Head)	1.755	5.143	1.909		
	WS (Tail)	1.751	6.203	2.832		
LLNL Atlas	DHHT (Rnd., Rnd.)	5.866	42.188	0.113	0.108	96
	DHHT (Rnd., Det.)	3.068	6.320	0.118	0.113	96
	DHHT (Det., Rnd.)	4.023	38.987	0.071	0.069	97
	DHHT (Det., Det.)	2.491	4.407	0.056	0.054	96
	DHHT (Multi)	2.579	7.719	0.115	0.112	96
	WS (Head)	2.602	27.249	1.363		
	WS (Tail)	2.399	45.886	1.905		
LLNL Thunder	DHHT (Rnd., Rnd.)	5.148	28.178	0.119	0.115	97
	DHHT (Rnd., Det.)	2.657	3.744	0.112	0.108	96
	DHHT (Det., Rnd.)	3.551	21.788	0.075	0.073	97
	DHHT (Det., Det.)	2.092	2.638	0.048	0.047	97
	DHHT (Multi)	2.238	4.548	0.115	0.111	96
	WS (Head)	2.216	11.680	2.484		
	WS (Tail)	2.121	16.899	3.560		
SDSC BLUE	DHHT (Rnd., Rnd.)	4.390	34.714	0.117	0.112	95
	DHHT (Rnd., Det.)	1.692	3.922	0.109	0.105	96
	DHHT (Det., Rnd.)	2.910	28.603	0.075	0.072	96
	DHHT (Det., Det.)	1.427	2.739	0.046	0.044	97
	DHHT (Multi)	1.667	5.357	0.114	0.110	96
	WS (Head)	1.918	18.211	2.979		
	WS (Tail)	1.843	21.763	4.187		
SDSC DataStar	DHHT (Rnd., Rnd.)	4.912	39.536	0.115	0.111	97
	DHHT (Rnd., Det.)	2.238	4.620	0.109	0.105	97
	DHHT (Det., Rnd.)	3.371	34.204	0.072	0.070	97
	DHHT (Det., Det.)	1.862	3.241	0.050	0.048	97
	DHHT (Multi)	2.035	5.853	0.114	0.109	96
	WS (Head)	2.069	17.177	2.740		
	WS (Tail)	1.965	22.846	3.818		

Table 4.15: *Results of experiments of type C with overload on pool PCs (Linux).*

		Stretch Factor		# Migs. / Hour		
Input Profile	**Load Balancer**	**Procs.**	**Jobs**	**Def.**	**Bulk**	[% Def.]
HPC2N	DHHT (Rnd., Rnd.)	4.310	13.216	0.212	0.187	88
	DHHT (Rnd., Det.)	1.648	2.553	0.183	0.164	90
	DHHT (Det., Rnd.)	3.347	9.857	0.180	0.156	87
	DHHT (Det., Det.)	1.467	2.133	0.105	0.094	90
	DHHT (Multi)	1.765	3.168	0.210	0.187	88
	WS (Head)	2.280	5.987	1.646		
	WS (Tail)	2.233	7.826	2.554		
LLNL Atlas	DHHT (Rnd., Rnd.)	7.191	49.230	0.229	0.190	83
	DHHT (Rnd., Det.)	4.046	8.627	0.212	0.182	86
	DHHT (Det., Rnd.)	5.654	38.526	0.173	0.153	89
	DHHT (Det., Det.)	3.439	8.831	0.159	0.138	87
	DHHT (Multi)	3.387	10.601	0.212	0.189	89
	WS (Head)	3.363	25.112	1.142		
	WS (Tail)	3.191	60.710	1.628		
LLNL Thunder	DHHT (Rnd., Rnd.)	6.008	28.087	0.213	0.187	88
	DHHT (Rnd., Det.)	3.313	4.860	0.176	0.157	89
	DHHT (Det., Rnd.)	4.832	21.690	0.196	0.174	89
	DHHT (Det., Det.)	2.946	4.402	0.139	0.124	89
	DHHT (Multi)	2.942	5.788	0.212	0.188	88
	WS (Head)	2.933	14.104	2.099		
	WS (Tail)	2.841	21.745	3.033		
SDSC BLUE	DHHT (Rnd., Rnd.)	5.164	38.889	0.203	0.181	89
	DHHT (Rnd., Det.)	2.268	5.198	0.194	0.172	89
	DHHT (Det., Rnd.)	4.032	28.895	0.197	0.172	88
	DHHT (Det., Det.)	1.915	4.453	0.127	0.112	89
	DHHT (Multi)	2.162	6.561	0.222	0.198	89
	WS (Head)	2.512	16.475	2.452		
	WS (Tail)	2.408	26.093	3.605		
SDSC DataStar	DHHT (Rnd., Rnd.)	5.873	43.737	0.228	0.194	85
	DHHT (Rnd., Det.)	3.054	6.242	0.190	0.167	88
	DHHT (Det., Rnd.)	4.672	34.879	0.192	0.172	89
	DHHT (Det., Det.)	2.522	5.595	0.125	0.112	90
	DHHT (Multi)	2.623	7.860	0.222	0.196	88
	WS (Head)	2.692	19.479	2.386		
	WS (Tail)	2.612	32.419	3.413		

whereas in two cases it is slightly outperformed by Work Stealing and the DHHT (Multi) algorithm. Concerning the job stretch factor, DHHT (Det., Det.) nearly always performs best and has a performance gain of a factor of approximately 3–4 compared to Work Stealing (only for the LLNL Atlas input profile, the DHHT (Rnd., Det.) variant is best and slightly better than DHHT (Det., Det.)). For detailed results please see figures A.133 – A.152 in Appendix A.4.3. Concerning the number of migrations, we again notice that all DHHT-based algorithms produce roughly the same result than in the ideal load case, whereas Work Stealing now needs only approximately half the number of migrations as in the ideal load case, which anyway still means a loss of a factor of approximately 7–20 compared to the DHHT-based algorithms.

4.5 Summary

The evaluation of the load balancers reveals: as long as the total load of the jobs in the system almost ideally matches the number of available processors (cf. Table 4.16), the Work Stealing algorithm performs best. When the external workload has a high diversity, the DHHT load balancer with multiple hashing is able to keep up with the Work Stealing algorithm up to a factor of approximately 2. On medium or low diversity, the DHHT load balancer using deterministic hashing suffers a performance drawback of a factor of only approximately 1.25–2 against Work Stealing. On overloaded systems however (4 times as many processes as processors, cf. Table 4.17), the DHHT-based algorithms perform best; on high diversity, DHHT with multiple hashing is most efficient, outperforming Work Stealing by a factor of approximately 2; on medium or low diversity, DHHT using deterministic hashing works best with a performance gain by a factor of approximately 3–7 compared to Work Stealing. Thus, as future work, a distributed monitoring algorithm is conceivable which appropriately switches between the different load balancers according to the current overall job load and the diversity of the external workload.

Table 4.16: *Load balancer evaluation summary for ideal system load.*

Ext. Load Profile	Diversity	Best DHHT Variant	Loss against Work Stealing
Full Availability	none	DHHT (Det., Det.)	1.25–2
Office PCs	low	DHHT (Det., Det.)	1.25–2
Pool PCs	medium	DHHT (Rnd., Det.)	1.25–2
Notebooks	high	DHHT (Multi)	~ 2

Table 4.17: *Load balancer evaluation summary for overload factor 4.*

Ext. Load Profile	Diversity	Best DHHT Variant	Gain against Work Stealing
Full Availability	none	DHHT (Det., Det.)	3–6
Office PCs	low	DHHT (Det., Det.)	3–7
Pool PCs	medium	DHHT (Det., Det.)	3–4
Notebooks	high	DHHT (Multi)	~ 2

5

Clustering

Although the continually changing available computing power is the most important resource to fairly share among all parallel processes, there are also other criteria to consider such as the network bandwidth, for example, as the processes in BSP programs communicate with each other. If a BSP program would be scheduled entirely on a fast connected component of the network although there would be a few faster processors available outside this component, it will run faster due to the reduced networking delays than it would have run when just scheduled according to processing power.

Thus, it is convincing to cluster the PUB-Web network according to bandwidth. In the next section, we first describe the clustering problem, before we formalize it in Section 5.2. In Section 5.3, we explain the algorithm DiDiC. The experiments in Section 5.4 reveal that the clusterings computed by DiDiC converge to meaningful clusters; the results are comparable to or even slightly better than those computed by MCL [van00], an established non-distributed algorithm.

5.1 Problem Description

Taking a snapshot of the dynamic PUB-Web network structure, we notice: Typically, computers in the network either form a fast connected subnet (or a hierarchical structure of subnets), which in turn is connected to the Internet via a less powerful link, or they are isolated machines directly connected to the Internet. In either case, the link to the Internet is the component with the least bandwidth of a network path. The backbone of the Internet has such a high bandwidth that we model it as a single central node, to which all local subnets of computers and isolated machines are connected with an edge whose weight correlates to the bandwidth of their Internet links (cf. Fig. 5.1).

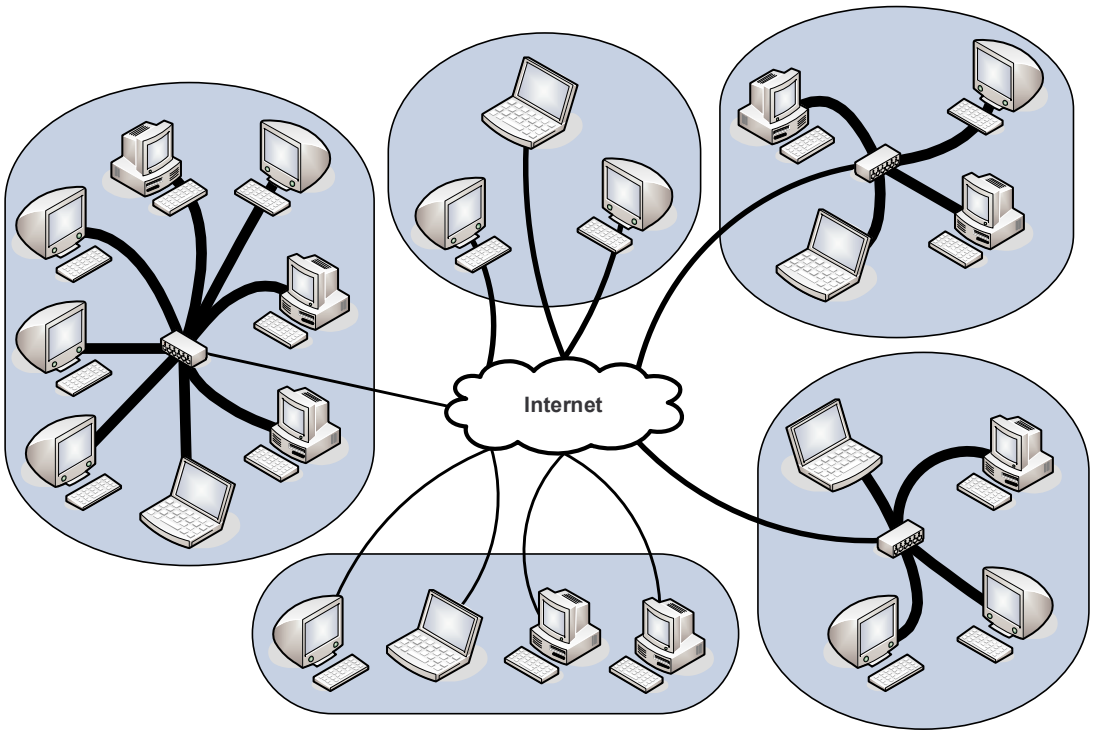

Figure 5.1: *Clustering the PUB-Web network.*

The goal now is to cluster the PUB-Web network such that all local subnets, whose size is above a suitable threshold, form its own cluster, and all other small subnets and isolated computers build a number of clusters such that, within each cluster, the bandwidth is as homogeneous as possible and that the cluster size fits between suitable upper and lower bounds. Since PUB-Web is a highly dynamic peer-to-peer network, it is not an option to just pick one of the existing centralized clustering algorithms; rather, we need a fault-tolerant, adaptive, and scaling distributed algorithm.

We model the machines in the PUB-Web network as vertices in a graph and choose the bandwidth as our similarity measure (or the inverse of the bandwidth as our distance measure), i.e., high edge weights correspond to high bandwidths and vice versa.

The algorithm must be able to start from an arbitrary initial configuration. Thus, we will use a random configuration if no initial configuration is provided. Using a random configuration is not only natural and easy to implement. Also, having in mind that a random cluster assignment is an extremely bad initial configuration, we can suppose that an algorithm will perform well on any configuration if it does so on a random one.

5.2 Problem Formalization

Let $\mathcal{G} := \bigcup_{i=0}^{T} G_i = (V_i, E_i, \omega_i)$ be a dynamic undirected and edge-weighted graph, i.e., a collection of static graphs G_i with vertex set V_i, edge set E_i, and corresponding edge weight set ω_i. The graph G_{i+1} is constructed from G_i by inserting and deleting certain vertices and/or edges. A k-way clustering of a graph is a function $\Pi_i : V_i \to \{1, \ldots, k\}$. Such a clustering divides the vertex set V_i into k disjoint subsets $V_i = \pi_{i,1} \dot\cup \pi_{i,2} \dot\cup \ldots \dot\cup \pi_{i,k}$.

For an undirected, edge-weighted graph $G = (V, E, \omega)$ with n vertices and its k-way clustering Π (as from now we omit the index i for ease of presentation), let

$$\deg(v) := \sum_{e=\{\cdot,v\}\in E} \omega(e)$$

be the *weighted degree* of vertex v and

$$N(v) := \{u \mid \{u, v\} \in E\}$$

its *neighbourhood*. Note: In order to keep the neighbourhood reasonably small for a local algorithm, only a certain number of the nearest vertices should be connected with an edge.

The popular clustering quality measure *modularity* [NG04] is defined as

$$\mathrm{Mod}(\Pi) := \sum_{c=1}^{k} \left(\frac{\sum_{e=\{u,v\}\in E | u,v\in\pi_c} \omega(e)}{\sum_{e\in E} \omega(e)} - \left(\frac{\sum_{v\in\pi_c} \deg(v)}{2 \cdot \sum_{e\in E} \omega(e)} \right)^2 \right)$$

This version of the modularity definition is derived from Brandes et. al. [BDG+08], who consider the unweighted case. They point out that maximizing modularity involves a trade-off between producing many intra-cluster edges (first part of the main sum) and producing a large number of clusters with small degree (second part), yielding more cut-cluster edges. It is $\mathcal{NP}$-hard to optimize modularity [BDG+08] and nearly all interesting clustering metrics [ŠS06] for general graphs.

Each connected component of the subgraph induced by the vertices of cluster π_c is called a *cluster-connected component* of π_c. The set of all cluster-connected components of π_c is denoted by $\mathrm{CCC}(\pi_c)$. The *nearly connected value* of π_c is then defined as

$$\mathrm{NCV}(\pi_c) := \frac{\max_{S\in\mathrm{CCC}(\pi_c)} |S|}{|\pi_c|}$$

A good clustering for the PUB-Web application is preferably connected and groups vertices with a high mutual bandwidth. Hence, our clusterings should have a high modularity and clusters with high NCV. In our dynamic setting, a

high-quality clustering should be obtained and maintained from a certain time step on. Moreover, local changes in the graph structure should entail also only local changes in the clustering.

5.3 The Iterative Process DiDiC

We now present a distributed, iterative diffusive algorithm, called DiDiC, which we will later compare against a well-known, established (but non-distributed) algorithm. DiDiC and its evaluation is joint work with Henning Meyerhenke, published in [GM10]. It is based on the framework in [MMS09], using additional techniques to make the algorithm work in a distributed setting. In particluar, we employ a secondary diffusion system inside DiDiC in order to make the algorithm work without global knowledge.

The algorithm uses the concept of disturbed diffusion to identify dense regions in the network graph weighted by bandwidth. The use of diffusion for graph clustering is motivated by its close relation to random walks [Lov93]. The intuitive idea both concepts have in common is that a random walk (or the related diffusion process) is likely to stay a very long time in a dense graph region before leaving it via one of the few outgoing edges.

Since diffusion on graphs can be realized as an inherently distributed process, the vertices executing the DiDiC algorithm need to communicate only with their direct neighbours. Moreover, the algorithm requires only a small amount of additional memory. In a very broad sense it can be regarded as self-stabilizing since the initial random clustering is transferred into a clustering with continuously high quality, also on dynamically changing graphs.

The general or first order diffusion scheme (FOS) [Cyb89] belongs to the class of local iterative algorithms for load balancing. Given a graph and a load for each vertex, these algorithms distribute the total amount of load stepwise among the vertices of the graph. Finally, in the convergence state of these algorithms, each vertex has the same average amount of load. However, in order to use this approach for clustering, some modifications are necessary: First, the diffusive method must result in an unbalanced load distribution. To achieve this goal, we send some load back to the source vertices in every step by subtracting it from the remaining vertices (this may lead to a negative load for some vertices) and we disturb the process by stopping it after very few iterations instead of iterating until all vertices have the same amount of load. Second, we need to assign the vertices to clusters. For this, we consider k distinct diffusion systems, i.e., we associate k diffusion load values with each vertex, and assign each vertex v to the cluster with the highest diffusion load on v.

Given the first snapshot G_0 of a dynamic graph $\mathcal{G}$, we initialize the load val-

ues appropriately. Then we run ψ diffusion iterations and compute the k-way clustering Π_0. For each time step i, $1 \leq i \leq T$, we again run ψ diffusion iterations on the current graph structure G_i and compute an updated clustering Π_i. In order to calculate the load drain for redirecting some load back to the source vertices in a fully distributed fashion, we employ a second, nested diffusion system, i.e. for each one of the ψ primary diffusion iterations, ρ diffusion iterations of the secondary system are performed.

We now formally describe the algorithm: Subsequently, let w_v and l_v denote two load vectors of length k, in which vertex $v \in V$ stores its load values for the k primary and secondary diffusion systems. By $w_v^{(t)}(c)$ we denote the load of v in the primary diffusion system c at time step t. The notation for the secondary load vector l is analogous. Then the primary diffusion system is defined as follows:

Definition 5.1 *Given a graph $G = (V, E, \omega)$, the cluster number c, and the load vectors $w^{(0)}, l^{(0)} \in \mathbb{R}^n$. Then the* truncated first order diffusion scheme (FOS/T) *with suitably chosen constants $\alpha(e)$ for each edge $e \in E$ (flow scale) performs the following operations on G in each iteration $0 < s \leq \psi$:*

$$x_{e=\{u,v\}}^{(s-1)}(c) = \omega(e) \cdot \alpha(e)(w_u^{(s-1)}(c) - w_v^{(s-1)}(c)),$$
$$w_u^{(s)}(c) = w_u^{(s-1)}(c) + l_u^{(s)}(c) - \sum_{u \in N(v)} x_{\{u,v\}}^{(s-1)}(c).$$

Note that in the definition of the flow $x_{\{u,v\}}$ between vertices u and v, the edge $\{u, v\}$ is directed implicitly (the flow changes its sign when viewed from the other direction). Also note that references to $l^{(s)}$ apply to the corresponding FOS/B time step $s \cdot \rho$ (see below).

Remember that the purpose of the secondary diffusion system is to redirect some load back to the source vertices. More precisely, in diffusion system c, vertices $u \notin \pi_c$ send most of their load to neighbouring vertices $v \in \pi_c$ using appropriate weights, while the load is balanced both among neighbouring vertices $u, v \in \pi_c$ and $u, v \notin \pi_c$. We call these weights benefits to express their purpose: A vertex of the corresponding cluster benefits from the secondary system. Now the second diffusion system is defined as follows:

Definition 5.2 *Given a graph $G = (V, E, \omega)$, its k-way clustering Π, the cluster number c, and the initial load vector $l^{(0)} \in \mathbb{R}^n$. Then the* first order diffusion scheme disturbed by vertex benefits (FOS/B) *with suitable $\alpha(e)$ for edge $e \in E$ (flow scale)*

*and B (*benefit*) performs the following operations on G in each iteration $0 < r \leq \rho$:*

$$y^{(r-1)}_{e=\{u,v\}}(c) = \omega(e) \cdot \alpha(e) \left(\frac{l^{(r-1)}_u(c)}{b_u(c)} - \frac{l^{(r-1)}_v(c)}{b_v(c)} \right),$$
$$l^{(r)}_u(c) = l^{(r-1)}_u(c) - \sum_{u \in N(v)} y^{(r-1)}_{\{u,v\}}(c),$$

where $y^{(r)}_{e=\{u,v\}}$ denotes the load exchange via edge e in iteration r and $b_u(c)$ denotes the benefit of vertex u in cluster π_c, which is defined as

$$b_u(c) := \begin{cases} 1 & u \notin \pi_c \\ B >> 1 & \textit{otherwise.} \end{cases}$$

Putting everything together, the DiDiC algorithm looks like shown in Listing 5.1. It is executed in a distributed way, parametrized by the respective vertex $v \in V$.

Listing 5.1: *The Distributed Diffusive Clustering (DiDiC) Algorithm*

```
1  DiDiC(v, N(v), π, k, T, ψ, ρ) → π
2  if (π is undefined)
3    π := RandomValue(1, k);
4  w_v := SetInitialLoad(π); l_v := w_v;
5  for time step t := 1 to T do
6    for each cluster system c do
7      for s := 1 to ψ do (* FOS/T *)
8        w̄_v(c) := w_v(c);
9        for r := 1 to ρ do (* FOS/B *)
10         l̄_v(c) := l_v(c);
11         for each u ∈ N(v)
12           l̄_v(c) := l̄_v(c) − α(e) · ω(e) · ( l_v(c)/b_v(c) − l_u(c)/b_u(c) );
13       for each u ∈ N(v)
14         w̄_v(c) := w̄_v(c) − α(e) · ω(e) · (w_v(c) − w_u(c));
15       w_v(c) := w̄_v(c) + l̄_v(c); l_v(c) := l̄_v(c);
16   π := argmax_{c=1,...,k} w_v(c);
17   N(v) := adaptToGraphChanges(v, N(v));
```

If the initial cluster affiliation of a vertex is undefined, the only information a vertex has in our distributed scenario (besides its neighbourhood structure and the loop durations) is the maximum number of clusters k. Thus, a vertex's initial affiliation is chosen as a random number between 1 and k. Note: While the possibility exists to assign an initial clustering, we assume that in our scenario it is usually not used and π is undefined until it is initialized randomly. Recall that the algorithm is expected to work with arbitrary initial clusterings. Even

with the simple strategy of random initialization DiDiC performs well in the experiments (if the number of time steps T is reasonably large), although it is difficult to derive meaningful clusters from such a configuration. While better initial settings would not raise the quality of the solution produced by the algorithm automatically, they can be expected to improve the speed of convergence to a good clustering.

After the clustering has been initialized, each vertex v sets its entries of the initial load vectors $w_v^{(0)}(c)$, $c = 1, \ldots, k$, to 0 with one exception. The load value corresponding to the own cluster is set to a high constant value, e.g., 100. The load vectors l of the secondary diffusion system are initialized in the same way.

Then the actual diffusive clustering process is started by the outermost loop, which runs for `T` time steps. At the end of each time step, the graph may be modified by local changes. Such changes include the addition or deletion of vertices and/or edges. In case of the deletion of a vertex v, the execution of the algorithm on v is stopped and its current load is distributed evenly among its neighbours. This simple strategy is why our diffusive method works well on dynamic graphs. Local changes in the graph affect the distribution of the diffusion loads only slightly. Thus DiDiC recovers quickly from small alterations of G and adapts the former clustering accordingly.

Within each time step the clustering is performed by calculating the diffusion systems for each cluster as explained before, i.e., by an outer FOS/T loop, into which an inner FOS/B loop is embedded. We set the flow scale constant to $\alpha(e) := 1/\max\{\deg(u), \deg(v)\}$ for an edge $e = \{u, v\} \in E$. This choice avoids large amounts of load being swapped back and forth.

A straightforward upper bound for the resulting time complexity per time step for each vertex v is $\mathcal{O}(k \cdot \psi \cdot \rho \cdot |N(v)|)$. Reasonable values for the parameters appearing in this expression can be found in the upcoming experimental section.

Note that DiDiC is not designed to deliver good clusterings from early time steps on. Instead, the random initial clustering needs to be refined from time step to time step. How long it takes to obtain reasonably good results, will be discussed during the experimental evaluation.

Improvements

After each time step the new cluster affiliation of each vertex v can be chosen generically as $\operatorname{argmax}_{c=1,\ldots,k} w_v(c)$ (as in line 19 of Listing 5.1). In the implementation of the algorithm, however, we use additional techniques to accelerate the clustering process. More precisely, after 10 time steps we enforce that a vertex

is assigned to the cluster with the number

$$\text{argmax}_{\{c=1,\ldots,k|N(v)\cap\pi_c\neq\emptyset\}}\, w_v(c)\,,$$

which ensures that isolated vertices (those without neighbours in the same cluster) vanish promptly and convergence to a good clustering is reached faster. Note that the order of cluster number updates is not fixed and queries to neighbours can yield data from different time steps. Yet, the final results are hardly affected by this behavior.

To eliminate isolated vertices, we migrate a vertex from one cluster to another not only based on the load in the primary diffusion system but also on the time step t. As an example, $v \in \pi_c$ changes only to $\pi_{c'}$ $(c \neq c')$ if v receives the highest amount of load from system c' and this amount is at least $1 + 0.0001 * t$ times higher than the load of system c. This way areas are only flooded by a new cluster if the diffusion process shows a really strong desire for this.

5.4 Experimental Evaluation

In this section we present experimental results on DiDiC. In order to perform reproducible experiments, we use a simulator that resembles a P2P environment occurring in PUB-Web. Except for the running time, the simulator computes the same results as if a real distributed system were executing DiDiC (under the assumption of precise floating point calculations). For generating the test graphs, the vertices are embedded into the two-dimensional unit square with wrap-around boundaries (i.e., into a torus). Such an embedding with the assignment to coordinates is not strictly necessary, but the generation of graphs with certain properties is simplified. Moreover, vertex coordinates do not affect the clustering results since the algorithm does not use the coordinates. Concerning the edges, remember that we assume the bandwidth (= edge weight) to be the inverse of the distance between two vertices. As a PUB-Web network graph is a complete graph, but DiDiC shall only operate within a local neighbourhood, we need to prune the network graph appropriately. Thus, we employ a slight variation of the disc graph model (e.g., [ALW+03]): It uses a uniform communication radius *rad* for all vertices; a vertex is connected to up to *maxNeigh* nearest neighbours within its communication radius, where *maxNeigh* is a user-defined parameter.

Recall from the problem description that we identify mainly two types of computers in the PUB-Web network (which is a subset of the Internet): home users with single computers and companies participating with lots of computers. In the latter case, the computers within a company are typically well connected, whereas the Internet connection is slower; additionally, there might be

medium speed connections between subnets or partner companies. Our goal is to identify subsets of the PUB-Web network that allow efficient communication and synchronization.

Evaluation criteria are modularity and the NCV value (the latter is averaged over all clusters). Note that empty clusters are seen as non-existing when computing these measures. The clusterings computed by DiDiC are compared with those computed by the graph clustering library [van00], which implements the well-known algorithm MCL [EvDO02]. As MCL is not a distributed algorithm, it should not be seen as a competitor. We use MCL only to validate DiDiC and it is not apparent that other algorithms are more suitable for this purpose. The inflation parameter of MCL strongly affects the granularity of clusters, which we set to 1.2. This choice avoids an extremely large number of small clusters and allows for a better comparison.

As a proof-of-concept we have started with experiments on graphs that have a well-separated cluster structure, i.e., we idealized the PUB-Web network by omitting the single home computers and by choosing quite high bandwidths for intra-company connections compared to inter-company links. The unit-torus is partitioned into $\tau \times \tau$ quadratic tiles, where τ^2 is the smallest square number greater than or equal to x, the number of dense regions to construct. Out of these τ^2 tiles x are randomly selected and filled with vertices, whose number is chosen randomly within certain limits for each tile, preventing extremely large or small clusters, which are not desired in our scenario. In each tile the vertices are distributed within a circle that is centered at the tile center and whose diameter is 95% of the tile side length. Fig. 5.2 (top) shows one of the graphs constructed this way with its initial clustering. The colors signify the cluster affiliation, grey edges are cut-edges. At the bottom and in Fig. 5.3 one can see the clusterings determined by DiDiC (after time step 209, which is the first one with NCV = 1.0) and MCL, respectively. The fact that DiDiC has determined fewer clusters than MCL is not at the expense of the quality as the modularity of DiDiC's solution (0.969) is better than MCL's (0.953). The results for the whole (seemingly easy) graph class can be summarized as:

- With the benefit parameter one can control somewhat how easily small clusters are dissolved. The higher the benefit is, the more resistant are small clusters against disappearance.

- Most importantly: The parameter k should be chosen larger than the number of clusters desired in the final solution. Then a reasonable number of clusters is determined by DiDiC since some colors vanish over time. Small values of k often result in disconnected clusters (and hence inferior quality) due to the random start and the strong separation of dense areas.

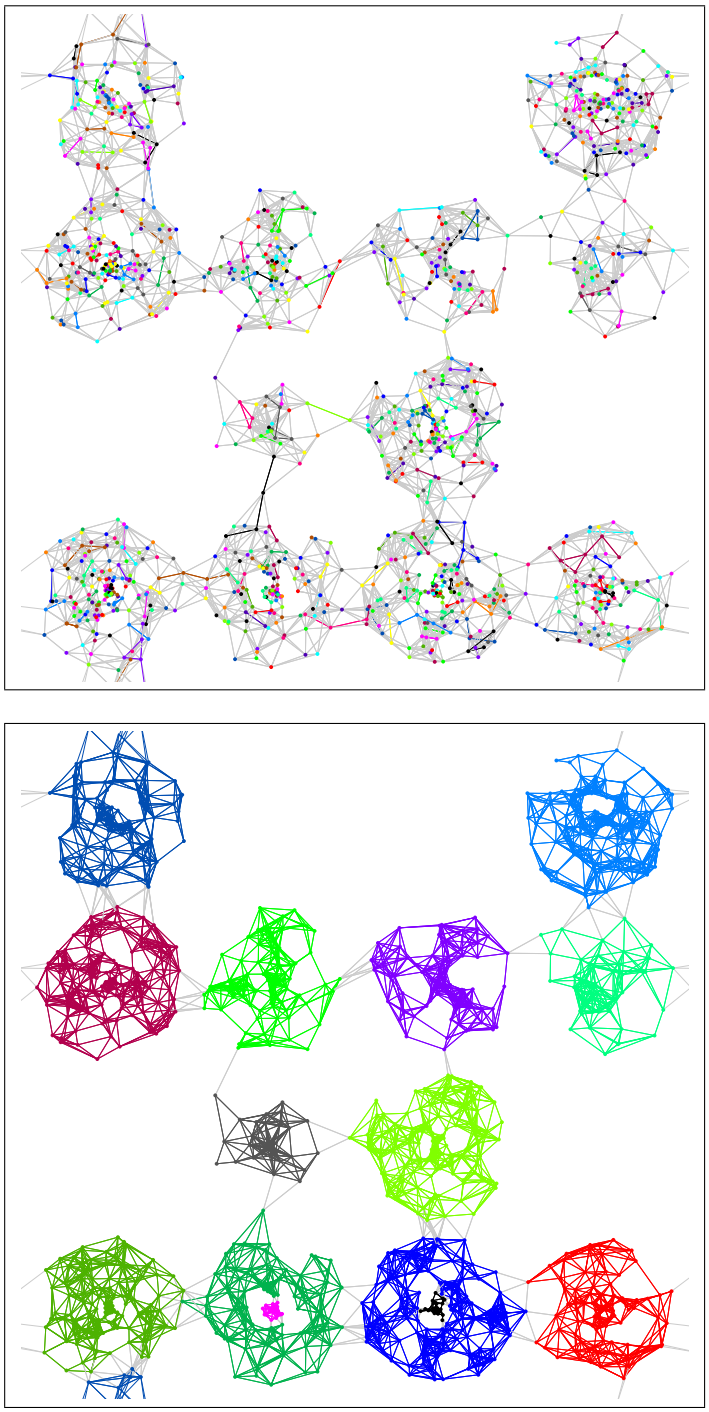

Figure 5.2: *Initial situation (top) and clustering computed by DiDiC after time step 209 (bottom) of a graph with twelve occupied tiles. Parameters:* $n = 1400$, $k = 20$, *maxNeigh* $= 16$, *rad* $= 0.25$, *no movement*, $\psi = 12$, $\rho = 8$, $B = 10$.

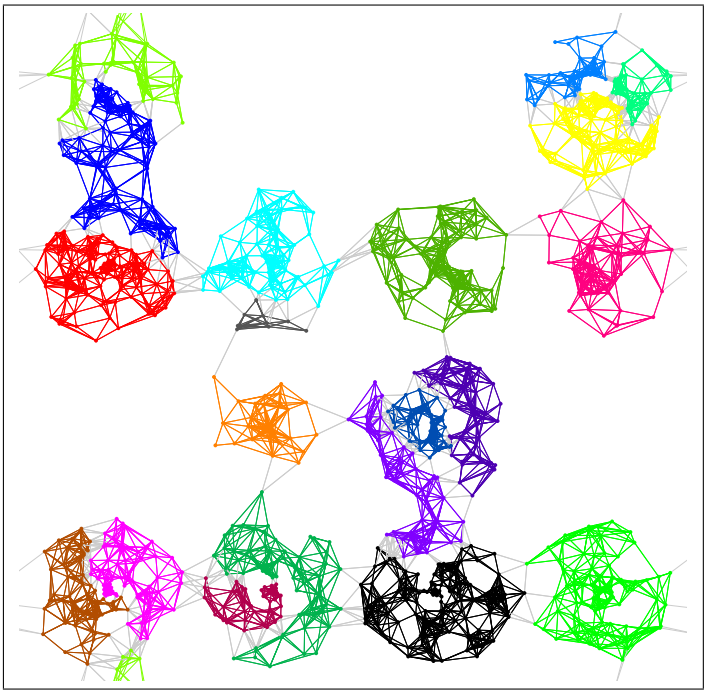

Figure 5.3: *The clustering computed by the non-distributed algorithm MCL for the same graph as in Fig. 5.2.*

To finally resemble the real PUB-Web network closely, we have built the following test scenario: The well connected subnets in companies etc. are represented by circular dense areas of nodes. The fact that the nodes, which are equally spread over the circular region, do not have a pairwise equal distance to each other, is a realistic assumption because big networks are usually organized hierarchically. Thus, computers connected to the same switch can simultaneously communicate with each other at full speed, whereas the bandwidth decreases when they have to share the same up- and downlinks while communicating over several hops with computers connected to another switch. The dense areas may overlap, which corresponds to different company divisions; or a small dense area may be situated inside a bigger one, which resembles a high-speed cluster within a company network. The coordinates of the x dense areas are chosen uniformly at random within $[0,1]^2$ and their radii range from 0.01 to 0.36 with an anti-quadratic probability distribution (small radii have higher probability).

In addition to these dense areas, single vertices representing private home computers are randomly spread over the unit-square. This is realistic as well: Not only world-wide but also within several major countries there are different Internet providers; customers of the same provider share a much higher

bandwidth than customers of different providers. To represent the single computers, $\frac{2}{x+2} \cdot n$ vertices are inserted uniformly at random into the unit-square. The remaining vertices are spread over all dense areas such that each area receives a fraction of $r_i^{1.5} / (\sum_{j \in 0,\ldots,x} r_j^{1.5})$, where r_i is the radius of dense area i, i.e., smaller areas have a slightly higher density. Inside each dense area the vertices are placed uniformly at random. Fig. 5.4 (top) shows such a graph with 800 vertices and 10 dense areas.

We did extensive tests with varying parameters and dozens of graphs with, e.g., 1600, 2400, and 3200 vertices. In order to simulate the dynamics of a P2P network, we randomly deleted and inserted 1%, 2%, or 5% of the vertices each, every second, or every fifth time step, respectively. We are aware that the real PUB-Web network may be magnitudes larger than just a few thousands of nodes, but it is not appearing out of a sudden with a random configuration. Rather, it will be dynamically growing or shrinking over time. Thus, we assume that it is sufficient to perform tests for initial instances with a few thousands of vertices.

In order to reduce the complexity of the illustration, the graph in the example in figures 5.4 – 5.8 consists of only 800 vertices; in particular, we constructed the graph and ran DiDiC using these parameters: $n = 800$, $k = 20$, $maxNeigh = 16$, $rad = 0.33$, $\psi = 11$, $\rho = 11$, $B = 10$, $x = 10$, 2% dynamics. Remember that the unit-square has wrap-around boundaries. Cut-edges are shown in grey. One can see from the clustering process illustrated in figures 5.4 – 5.7 how the solution computed by DiDiC converges quickly from an initially random situation; already after a few iterations a meaningful clustering emerges. In this particular example, all clusters are finally connected and the solution stabilizes around time step 60. Fig. 5.8 shows the clustering obtained using the MCL algorithm for the same graph (at time step 60). Both DiDiC and MCL produce a clustering consisting of 11 clusters. The modularity of the solution computed by DiDiC is 0.9044, which is of comparable quality to the result obtained using MCL (modularity 0.9078).

The plots in figures 5.9 – 5.11 are based on aggregated data of three experiment series and show how modularity and NCV evolve over time for graphs of the same class with 1600, 2400, and 3200 nodes, respectively. For validation purposes, we ran MCL every 50-th time step on the dynamic graphs and added its modularity and NCV values to the plots as single points in the same colors as the corresponding lines. As one can see, once DiDiC's clusterings stabilize, they have a modularity comparable to MCL's. Bigger instances take a bit longer to stabilize than smaller ones, but once the clustering stabilizes, DiDiC even yields a slightly better modularity than MCL in these cases. After the clusterings are rather stable, the quality in terms of connectedness is still increasing over time

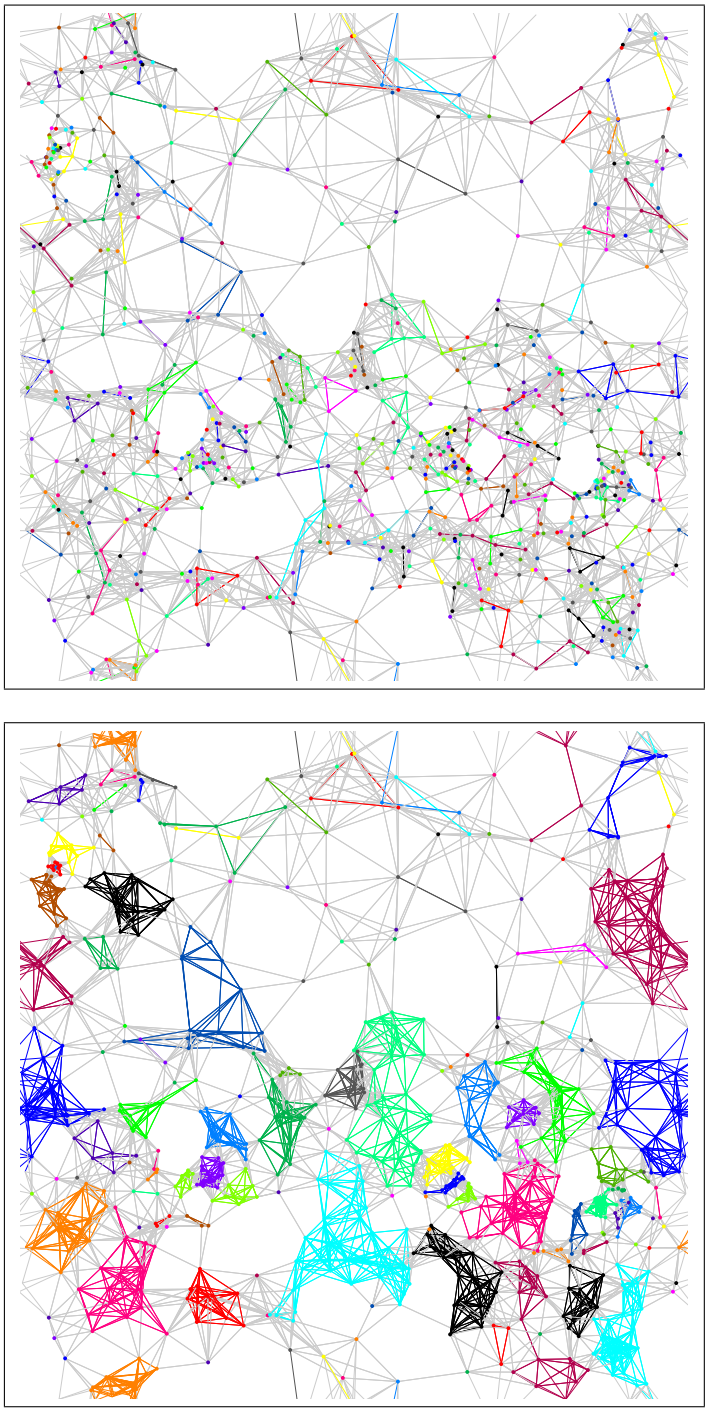

Figure 5.4: *Generated PUB-Web P2P graph with 800 nodes (top) and clustering computed with DiDiC after 5 time steps (bottom).*

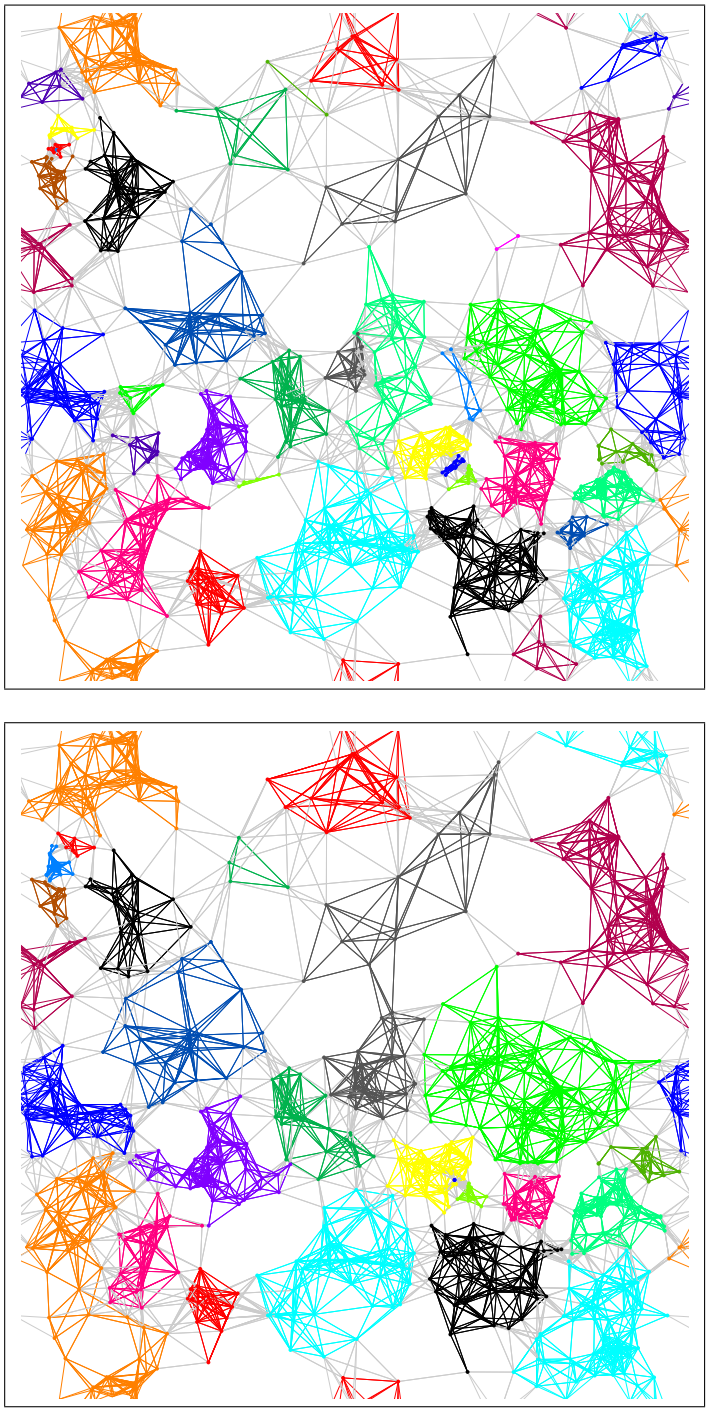

Figure 5.5: *Clusterings for the graph in Fig. 5.4 after 11 (top) and 21 (bottom) time steps.*

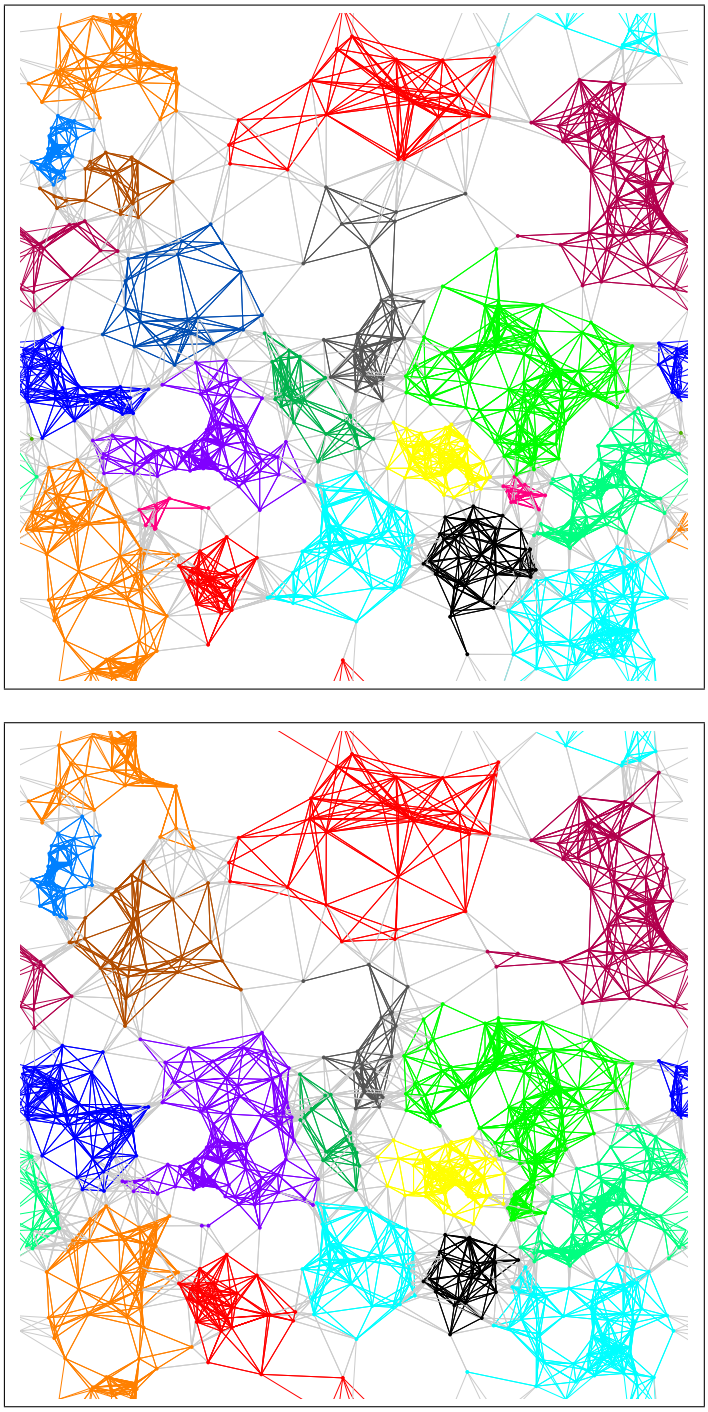

Figure 5.6: *Clusterings for the graph in Fig. 5.4 after 31 (top) and 41 (bottom) time steps.*

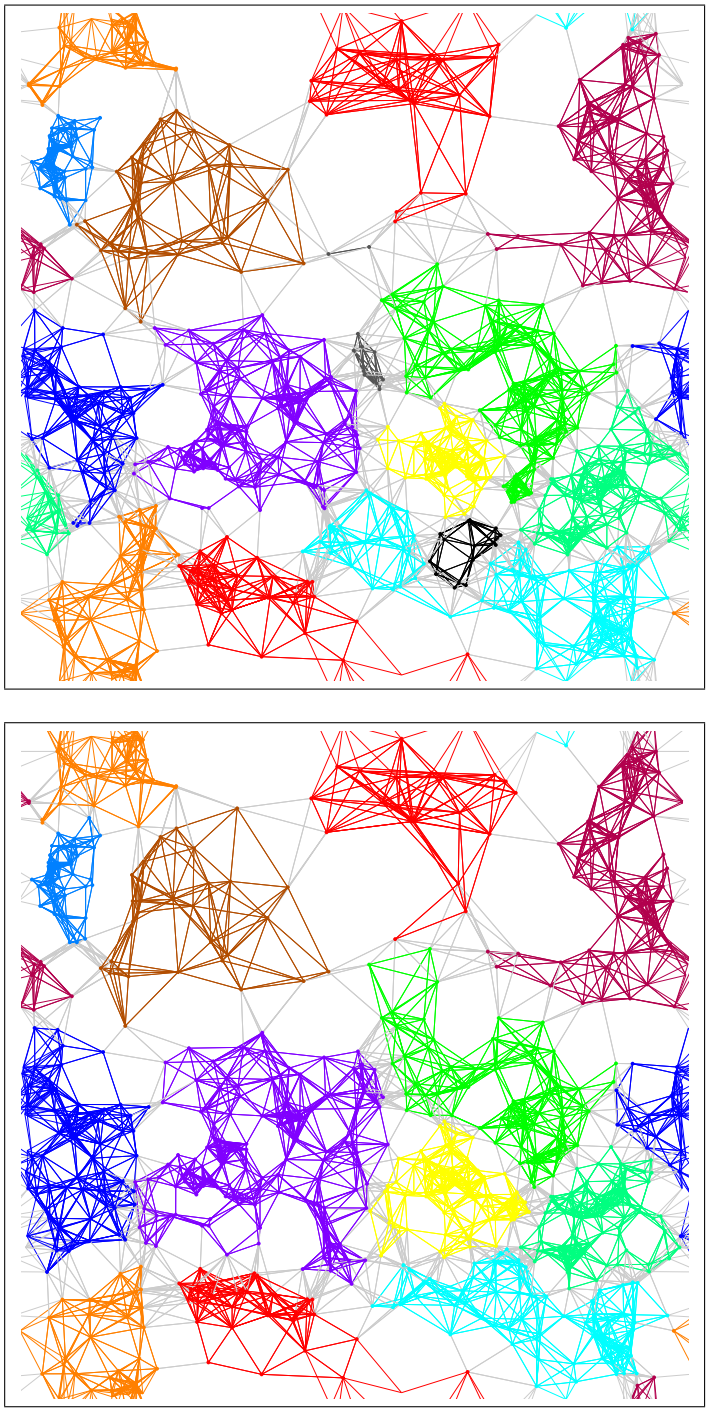

Figure 5.7: *Clusterings for the graph in Fig. 5.4 after 51 (top) and 61 (bottom) time steps.*

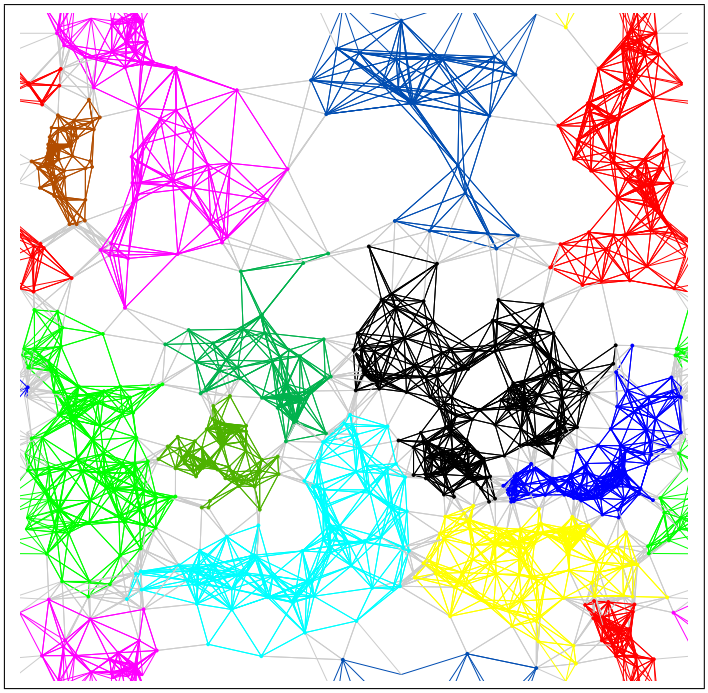

Figure 5.8: *The clustering computed by the non-distributed algorithm MCL for the same graph as in Fig. 5.7 (bottom).*

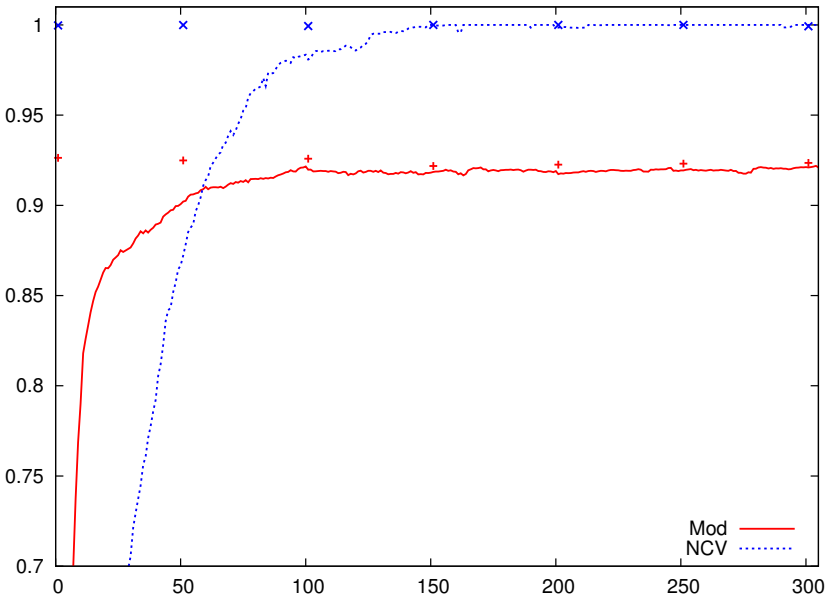

Figure 5.9: *Aggregated results (x-axis: time step, y-axis: modularity, NCV) for graphs with 1600 vertices.*

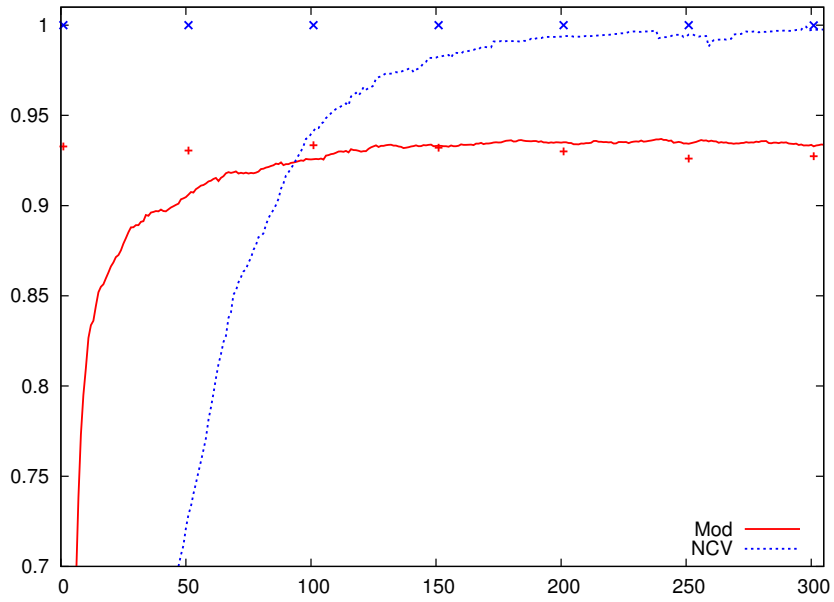

Figure 5.10: *Aggregated results (x-axis: time step, y-axis: modularity, NCV) for graphs with 2400 vertices.*

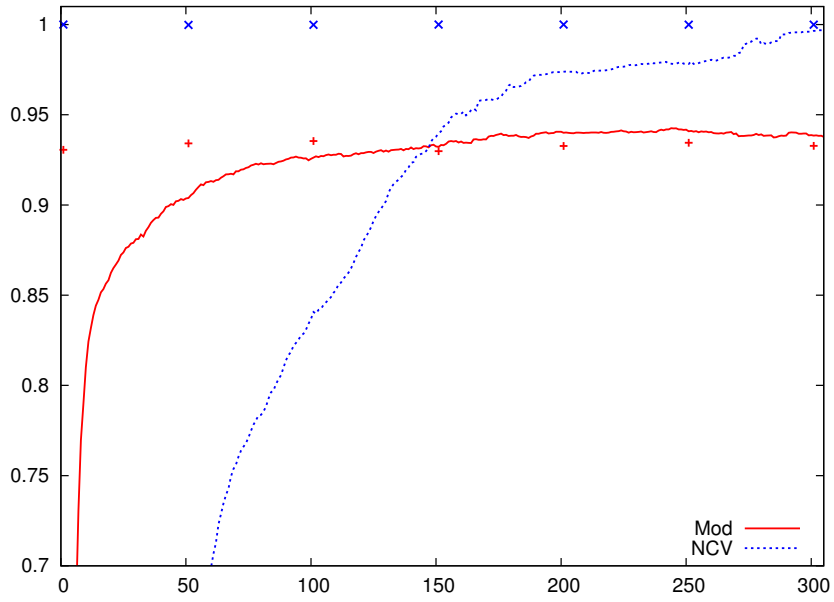

Figure 5.11: *Aggregated results (x-axis: time step, y-axis: modularity, NCV) for graphs with 3200 vertices.*

until a perfectly cluster-connected solution is reached.

Considering that each node sends roughly 100-150 very small messages in its local neighbourhood per time step, a time step would take a couple of seconds to complete in a real world scenario. Thus, a clustering would stabilize after a couple of minutes.

Experiments with smaller values of ψ and ρ made DiDiC faster, but often the quality worsened. Experiments using instances with the same parameters but with 1% or 5% (instead of 2%) dynamics in the graph led to similar results.

5.5 Summary

The experimental evaluation reveals that the iterative process DiDiC computes clusterings that are of comparable or even slightly better quality (in terms of the modularity measure) than the results obtained by the established non-distributed algorithm MCL, which is an excellent result. Since the PUB-Web graph permanently exists once the PUB-Web network has been set up, DiDiC's convergence speed is only of secondary importance — anyhow, it is a nice result that DiDiC's clusterings stabilize quite fast.

Thus, DiDiC is a perfect candidate for a clustering algorithm to integrate into PUB-Web. Note: When integrating DiDiC into PUB-Web, it has to be taken into account that the PUB-Web network is not only dynamically changing over time, which is handled well by DiDiC, but it is also expected to grow — and occasionally also to shrink. While a shrinking network is implicitly handled by DiDiC by completely dropping cluster colors once clusters become to small, there is no automatism for a growing network. However, a quite easy and natural approach is to split one of the largest clusters once the ratio of the network size and the number of clusters becomes to small.

6

Conclusion

We have presented the Paderborn University BSP-based Web Computing (PUB-Web) library — a middleware that supports the execution of parallel programs in the BSP style on a dynamic P2P network of computers, utilizing only their idle times. Since its first prototype implementation, PUB-Web has become a stable and mature system. In this thesis, we have focused on important technical and algorithmic aspects, in particular: In order to schedule processes with respect to the currently available computing power, which continually changes in an unpredictable fashion, we need intelligent load balancing algorithms and, as a basic precondition, the technical ability to migrate threads at runtime. To achieve the latter in a way suitable for production use, compatible with recent Java versions, available for all important platforms, and easy-to-use for developers, we have developed the PadMig thread migration and checkpointing library. Our implementation is stable, efficient, and user-friendly especially with respect to IDE integration and error reporting.

In order to tackle the distributed load balancing problem, we have adapted a decentralized data distribution method from the storage community to our setting. This new DHHT-based load balancer is able to fairly assign the processes of parallel programs to computing nodes and to balance the load on changes in the available computing power. In order to judge the quality of the schedules produced, we have performed extensive experiments, using realistic input data obtained by profiling the utilization of several hundred PCs for a period of several months. The comparison of several variants of the DHHT-based load balancer with the well-established Work Stealing algorithm reveals: as long as the total load of the jobs in the system almost ideally matches the number of available processors, the Work Stealing algorithm performs best and the best DHHT variant suffers a performance drawback of a factor of approximately 1.25–2. On overloaded systems however (4 times as many processes as processors, the DHHT-based algorithm performs best: on a high diversity

in the available computing power, DHHT with multiple hashing is most efficient, outperforming Work Stealing by a factor of approximately 2; on medium or low diversity, DHHT using deterministic hashing works best with a performance gain by a factor of approximately 3–7 compared to Work Stealing. Thus, as future work, a distributed monitoring algorithm is conceivable which appropriately switches between the different load balancers according to the current overall job load, the diversity of the external workload, and maybe other criteria such as the desired fairness level (Work Stealing delays all jobs approximately equally, whereas DHHT delays the jobs proportionally to their length); since the exchange of a load balancer at runtime involves a certain initialization and migration overhead, these costs need to be modeled adequately.

Beside the available computing power, we have also considered the network bandwidth as a secondary criterion for load balancing. As BSP programs communicate a lot, they should be scheduled entirely on fast connected components of the network. We have addressed this challenge by clustering the PUB-Web network according to bandwidth, employing a novel, fault-tolerant, adaptive, and scaling distributed clustering algorithm called DiDiC. Our experimental evaluation using a simulator reveals that DiDiC computes clusterings that are of comparable or even slightly better quality than the results obtained by the established non-distributed algorithm MCL, which is an excellent result. Furthermore, DiDiC's clusterings stabilize quite fast. Thus, DiDiC is a perfect candidate for a clustering algorithm to integrate into PUB-Web.

Now that we are able to cluster the PUB-Web network, promising future work could include a two-level load balancer (cf. Fig. 6.1): On the top level, one load balancer assigns BSP programs (i.e., groups of processes) to fast connected clusters of the PUB-Web network. On the lower level, one load balancer per cluster schedules the BSP processes on the particular peers within the clusters. As mentioned above, a distributed monitoring algorithm could keep track of the job load and the external workload — within each cluster and also globally on the top level. Using this monitoring information, not only a sophisticated choice of the load balancing algorithm variant is possible, but it can be also used to manage a global, distributed ready-queue that buffers BSP programs induced into the PUB-Web network in order to avoid an arbitrarily high job overload. For this, it is also conceivable to derive suitable availability prediction strategies from the cumulative pattern observed in our evaluation of the external workload in Section 4.3.1.

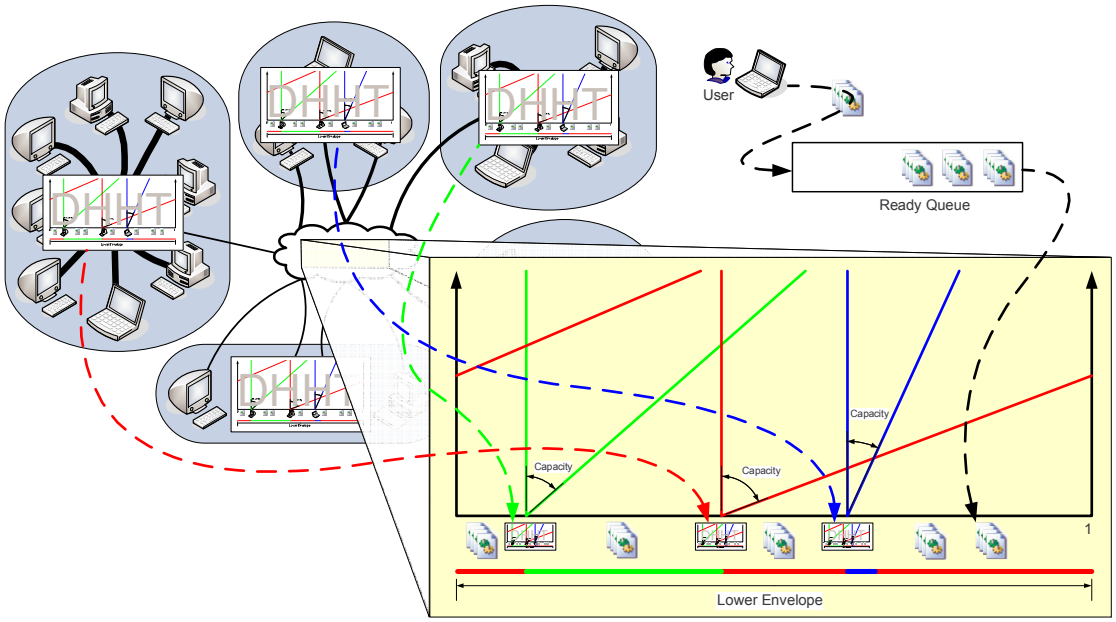

Figure 6.1: *Two-level load balancing.*

A

Detailed Results of the Evaluation of the Load Balancers

In Section 4.4, the results of the experimental evaluation of the load balancers have been presented in a compact way as aggregated data. In this appendix, we show the results of the particular experiment series in detail.

A.1 Type B, Ideal Load

This section contains the plots for the input profiles LLNL Atlas (figures A.1 – A.4), LLNL Thunder (figures A.5 – A.8), SDSC BLUE (figures A.9 – A.12), and SDSC DataStar (figures A.13 – A.16) for the experiments of type B with ideal total system load.

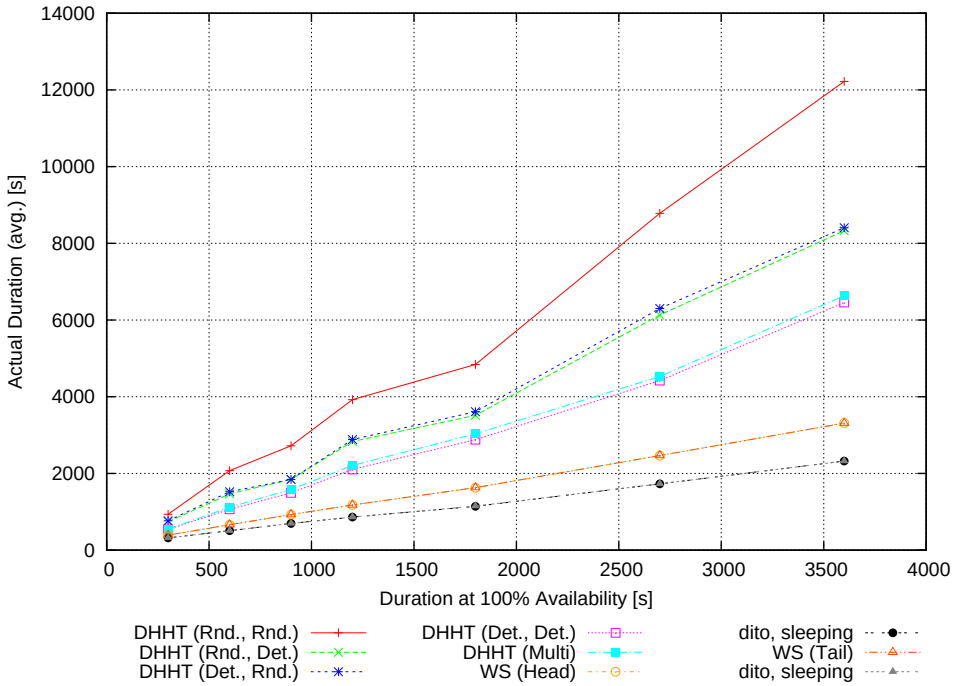

Figure A.1: *Actual average duration of processes in experiment 11.*

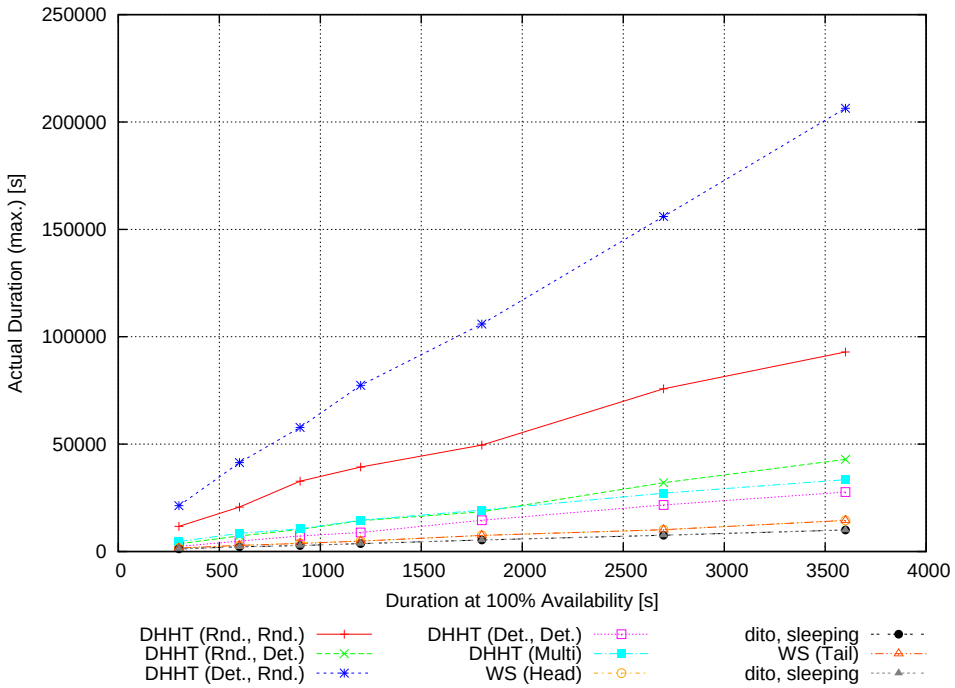

Figure A.2: *Actual maximum duration of processes in experiment 11.*

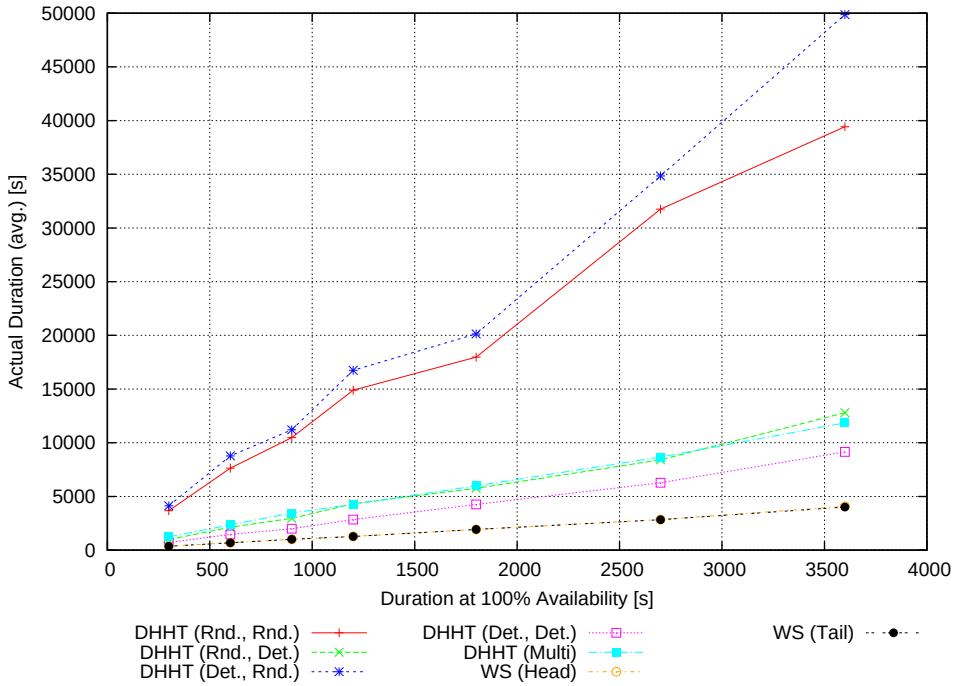

Figure A.3: *Actual average duration of jobs in experiment 11.*

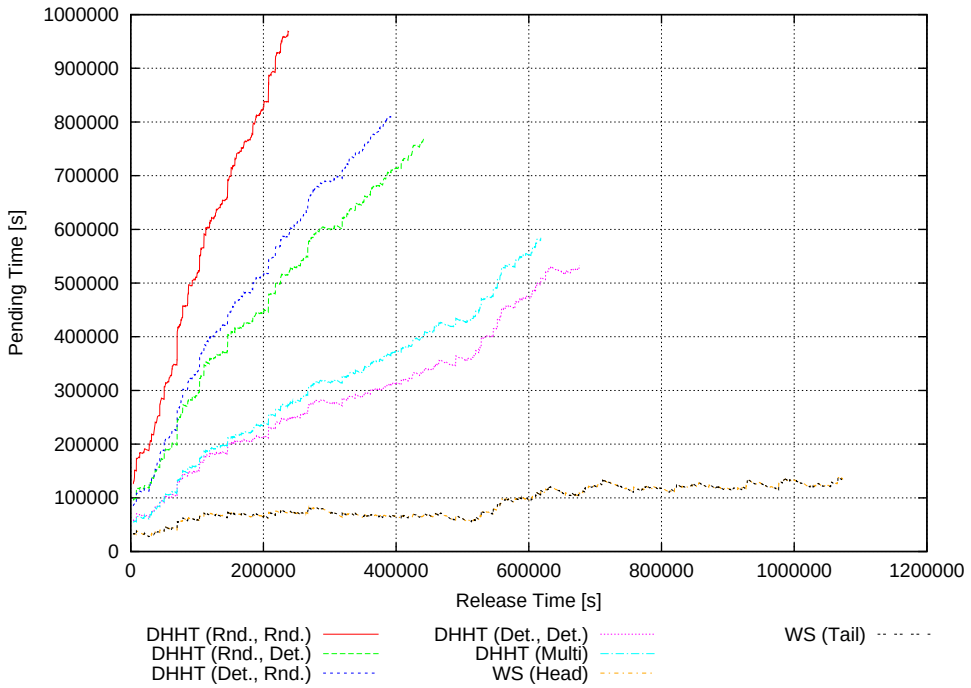

Figure A.4: *Pending time of jobs in experiment 11.*

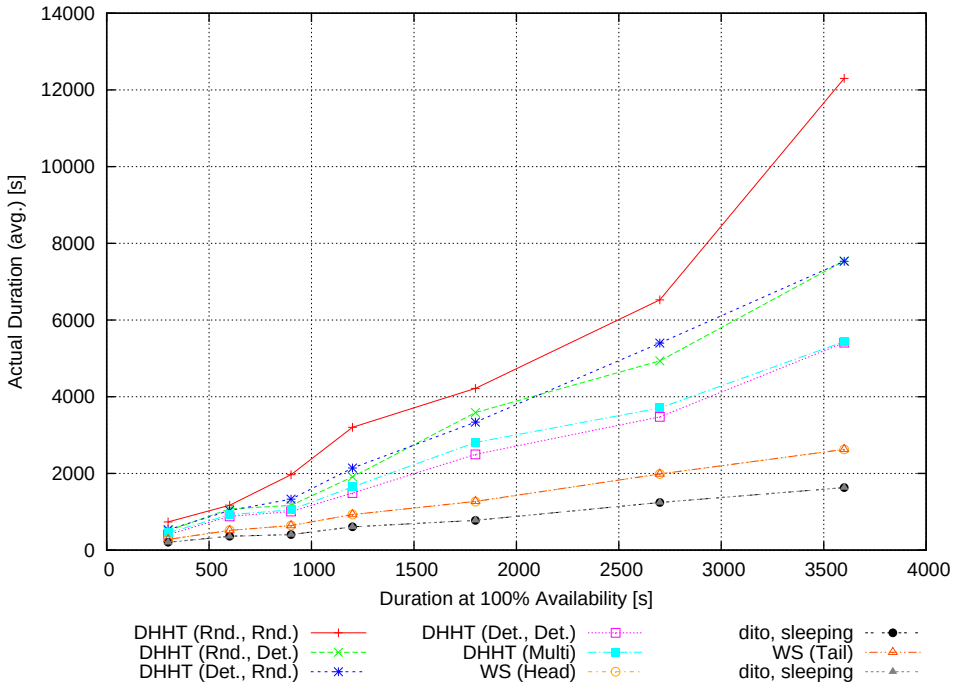

Figure A.5: *Actual average duration of processes in experiment 12.*

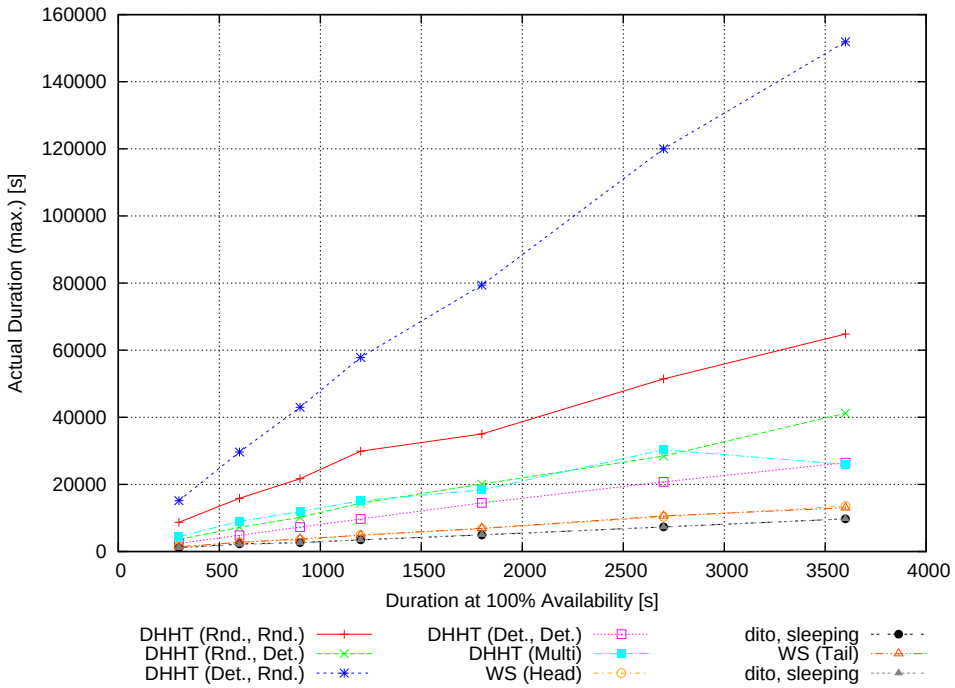

Figure A.6: *Actual maximum duration of processes in experiment 12.*

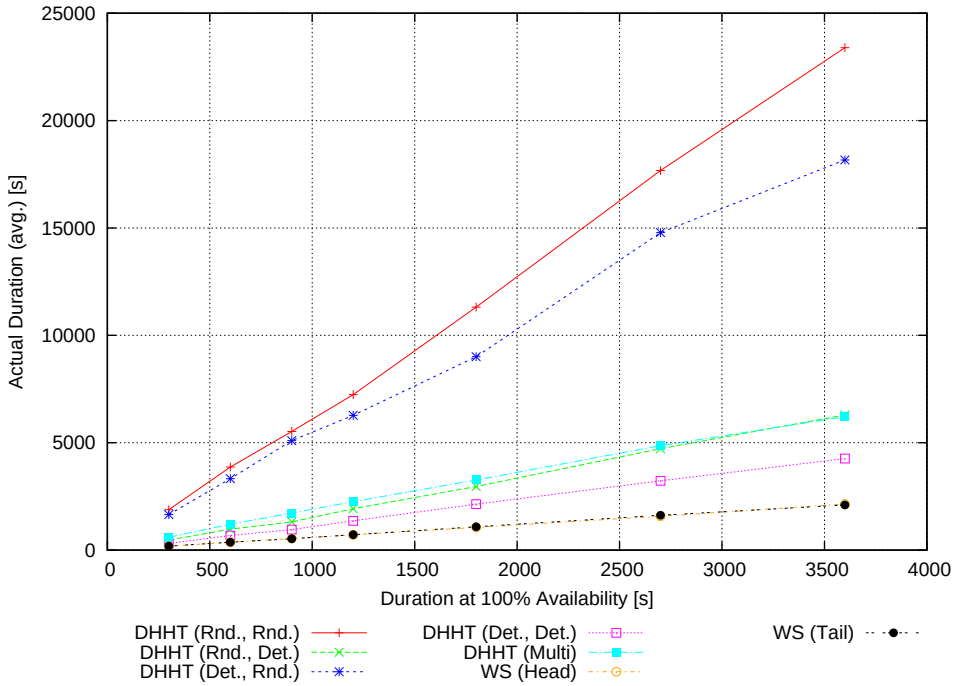

Figure A.7: *Actual average duration of jobs in experiment 12.*

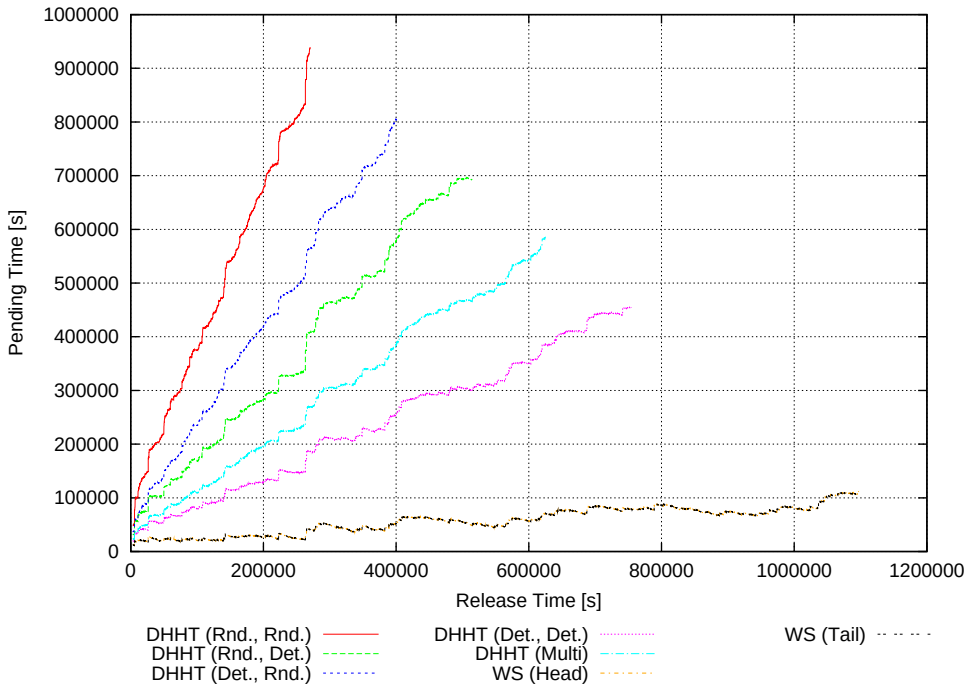

Figure A.8: *Pending time of jobs in experiment 12.*

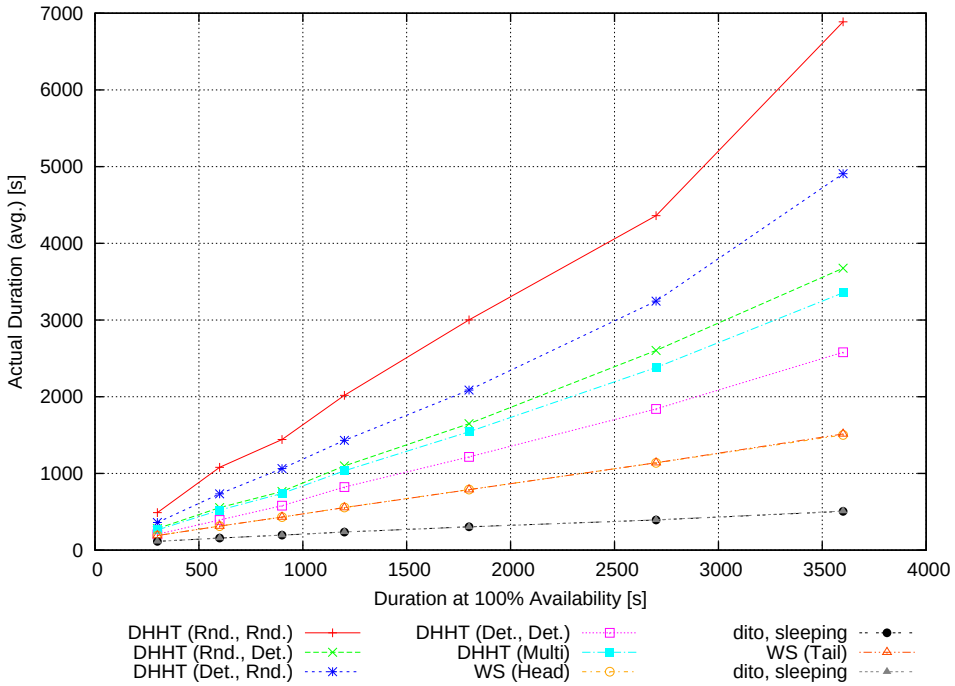

Figure A.9: *Actual average duration of processes in experiment 13.*

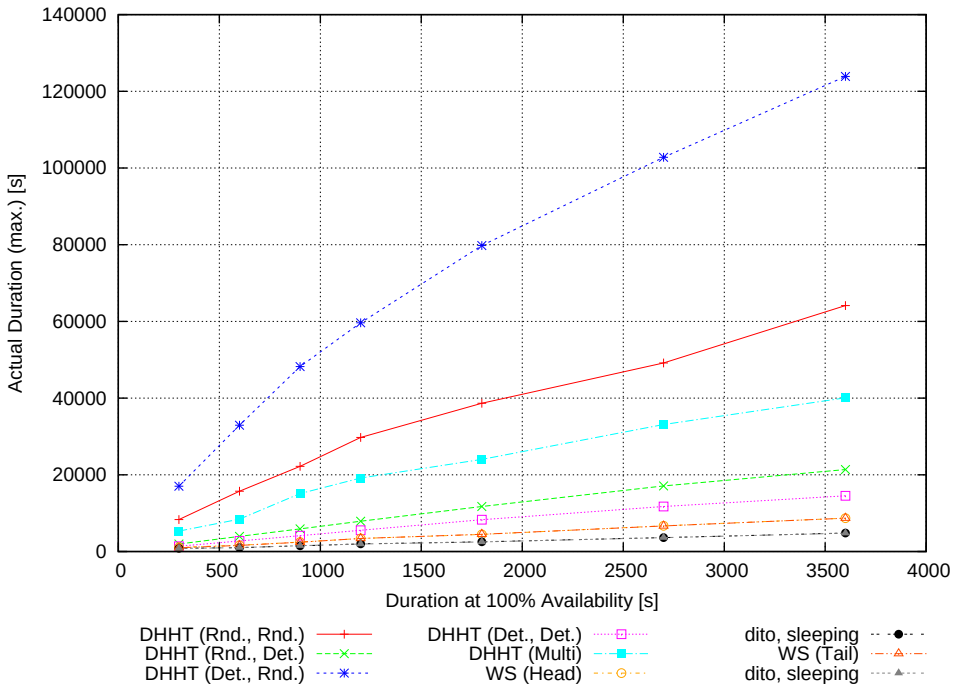

Figure A.10: *Actual maximum duration of processes in experiment 13.*

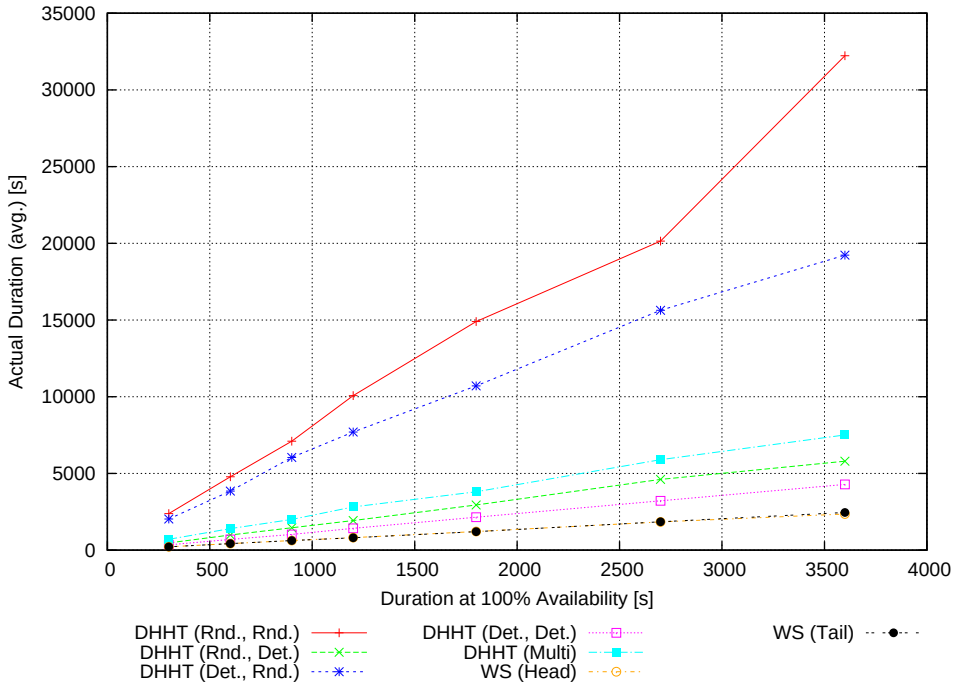

Figure A.11: *Actual average duration of jobs in experiment 13.*

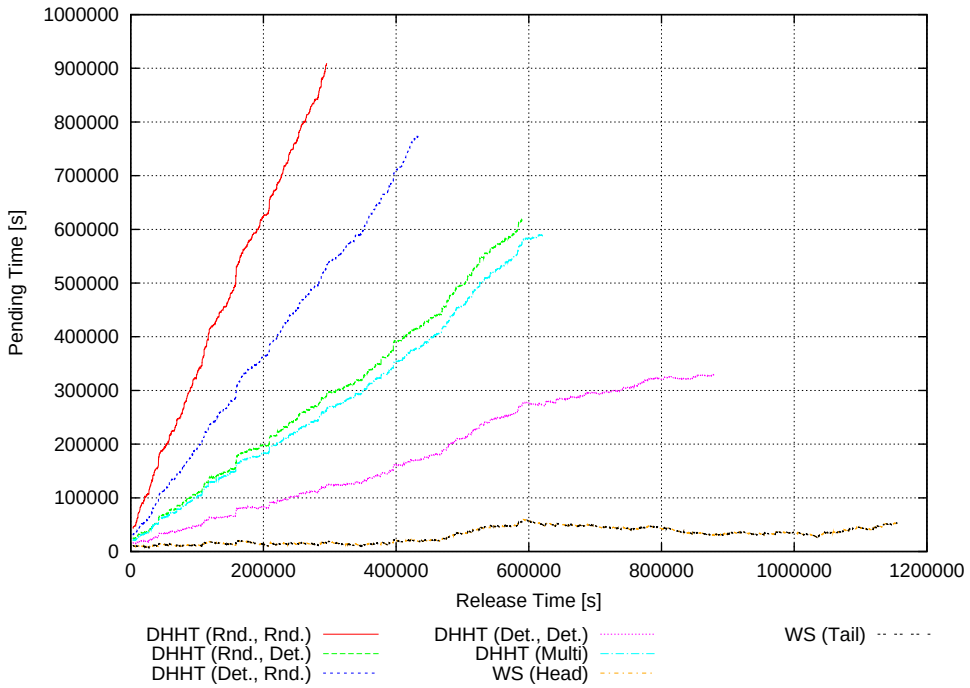

Figure A.12: *Pending time of jobs in experiment 13.*

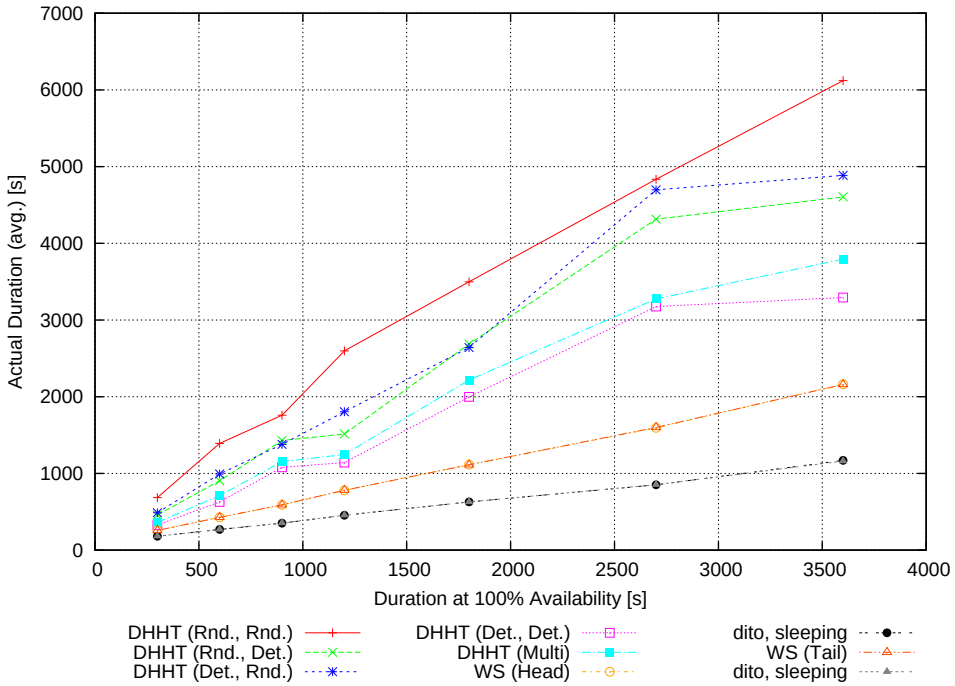

Figure A.13: *Actual average duration of processes in experiment 14.*

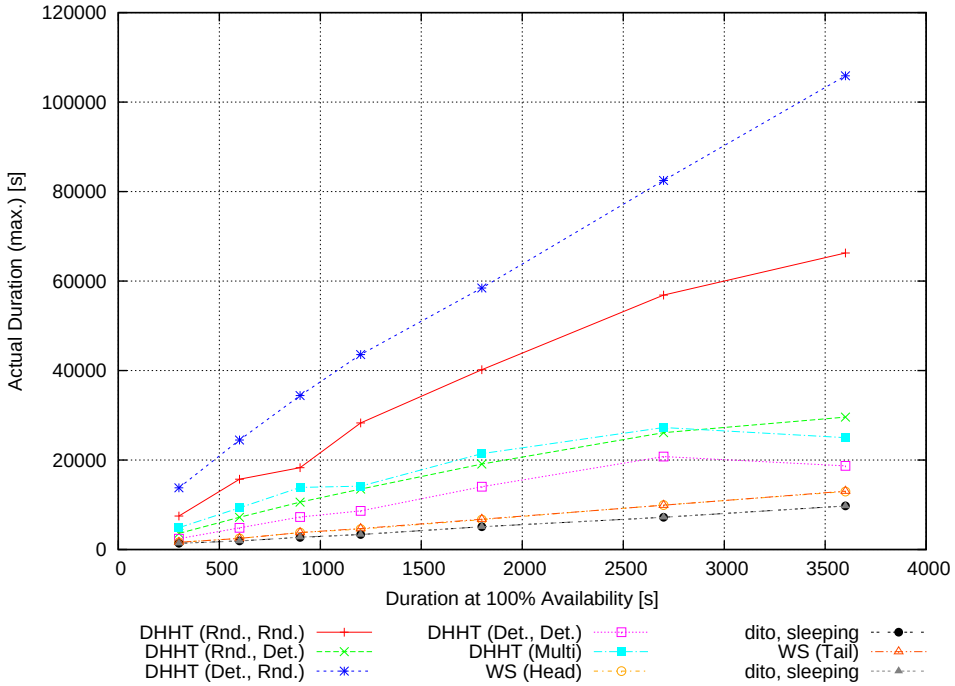

Figure A.14: *Actual maximum duration of processes in experiment 14.*

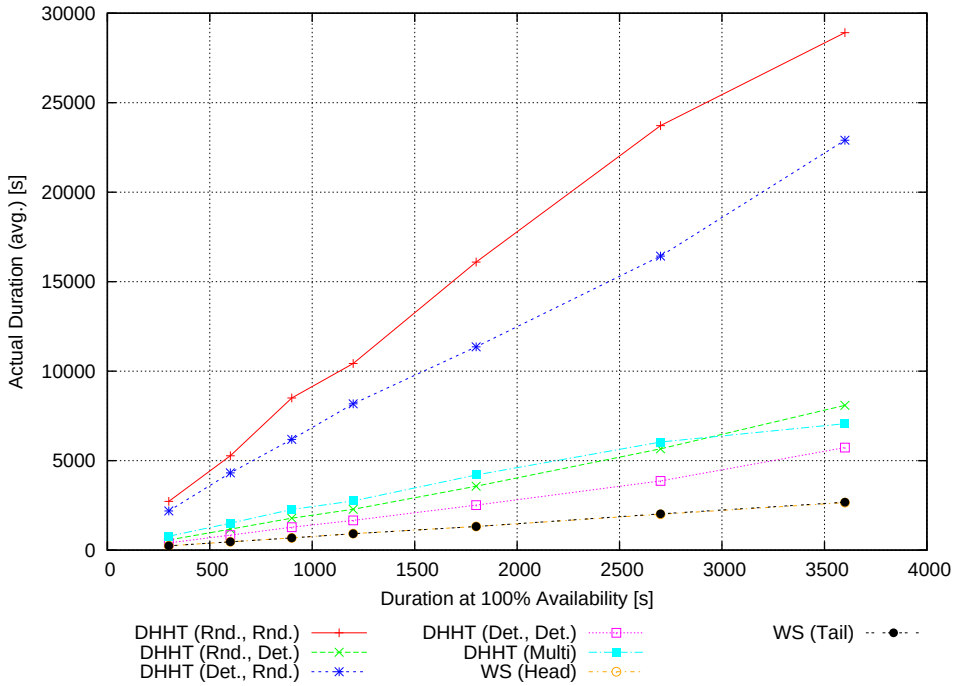

Figure A.15: *Actual average duration of jobs in experiment 14.*

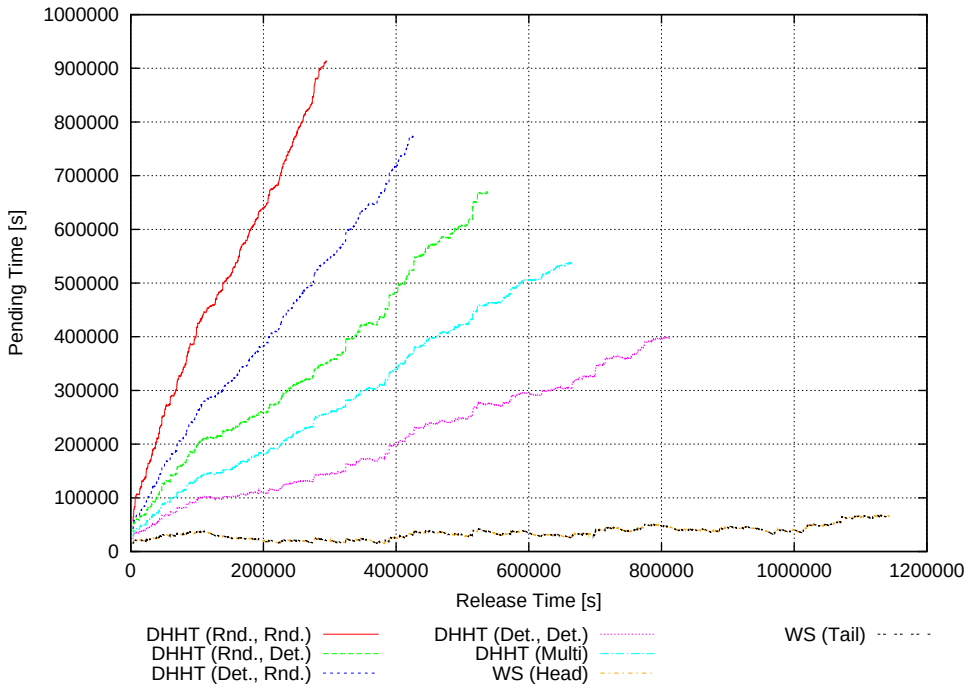

Figure A.16: *Pending time of jobs in experiment 14.*

A.2 Type B, Overload

This section contains the plots for the input profiles LLNL Atlas (figures A.17 – A.20), LLNL Thunder (figures A.21 – A.24), SDSC BLUE (figures A.25 – A.28), and SDSC DataStar (figures A.29 – A.32) for the experiments of type B with an overload of factor 4.

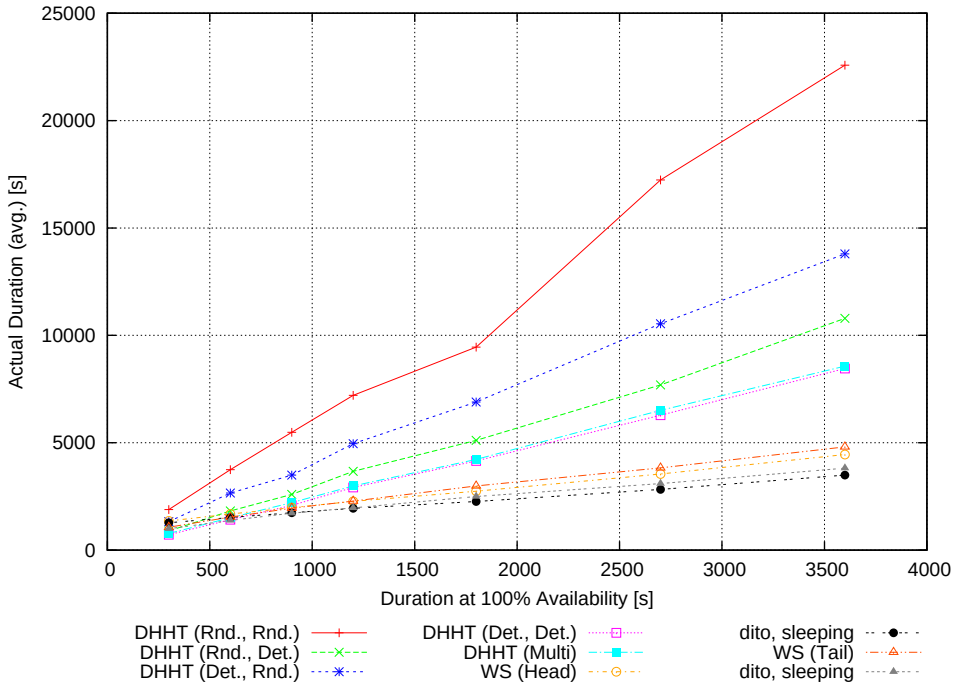

Figure A.17: *Actual average duration of processes in experiment 16.*

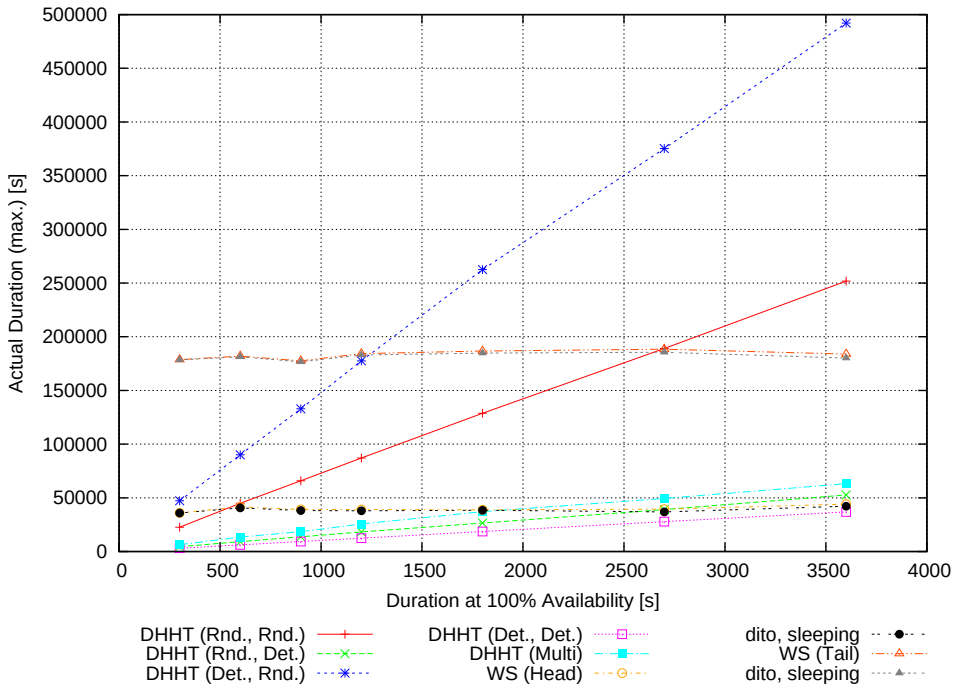

Figure A.18: *Actual maximum duration of processes in experiment 16.*

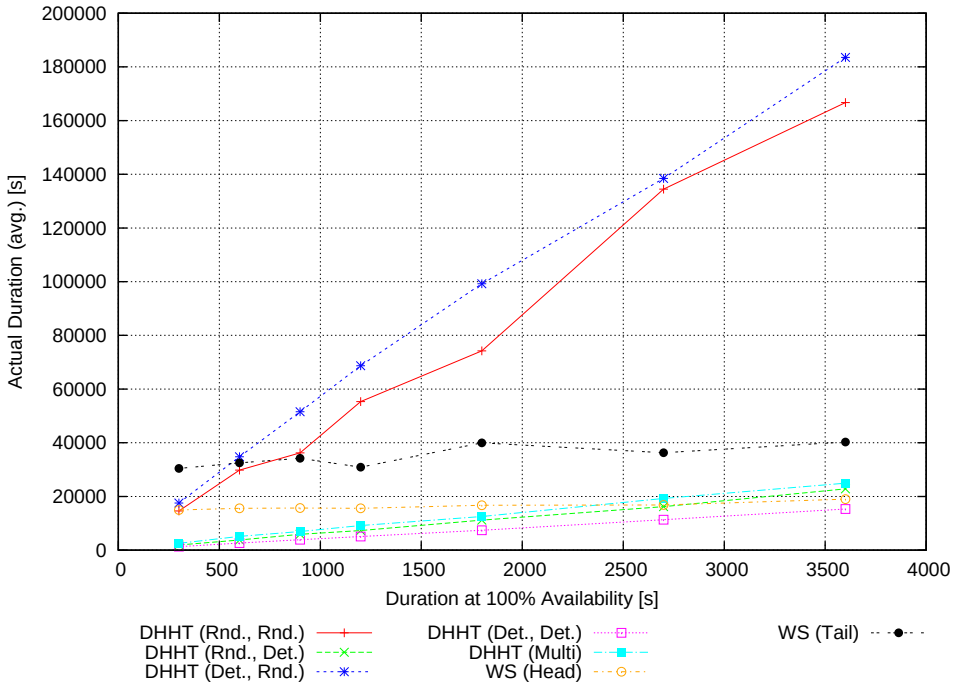

Figure A.19: *Actual average duration of jobs in experiment 16.*

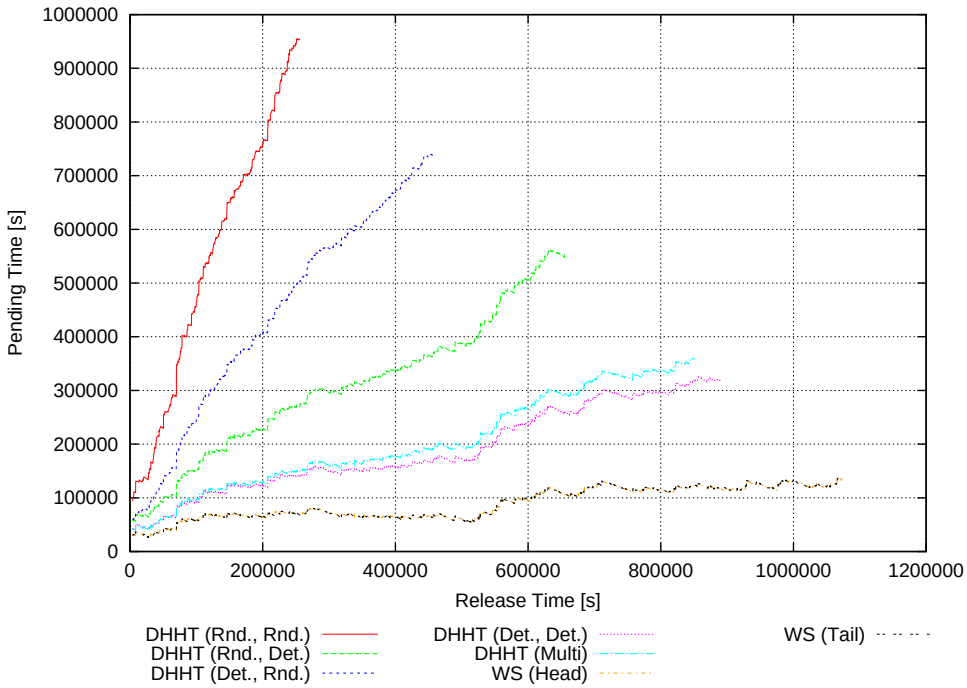

Figure A.20: *Pending time of jobs in experiment 16.*

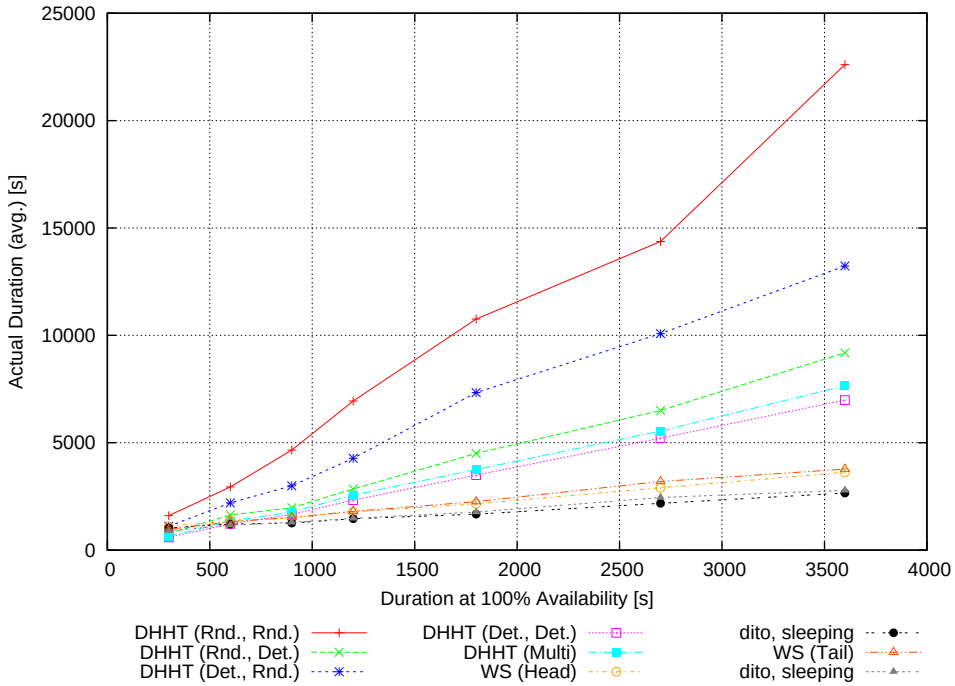

Figure A.21: *Actual average duration of processes in experiment 17.*

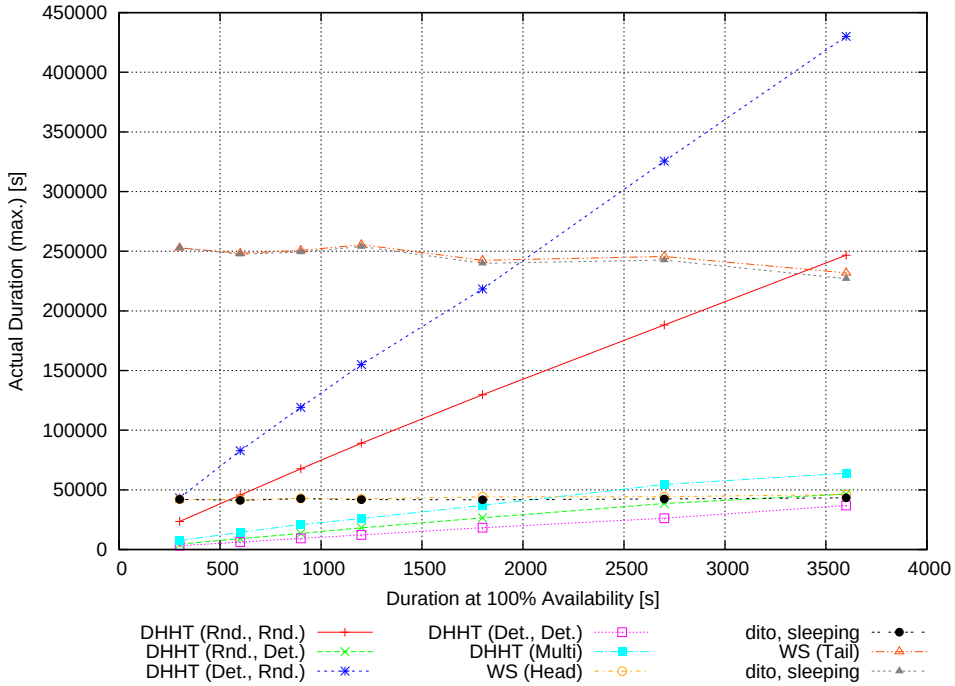

Figure A.22: *Actual maximum duration of processes in experiment 17.*

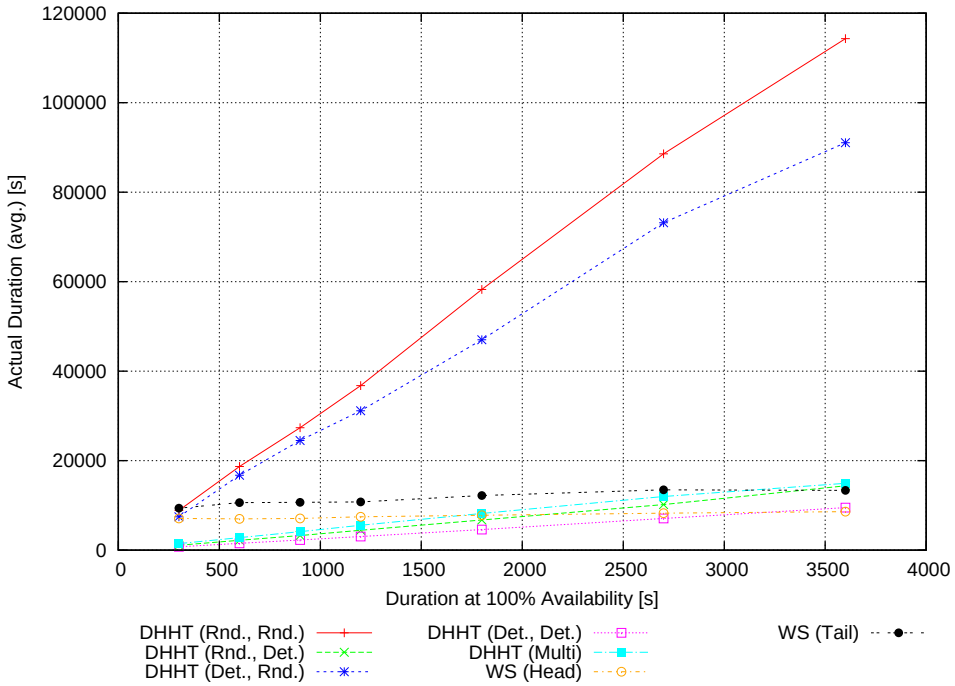

Figure A.23: *Actual average duration of jobs in experiment 17.*

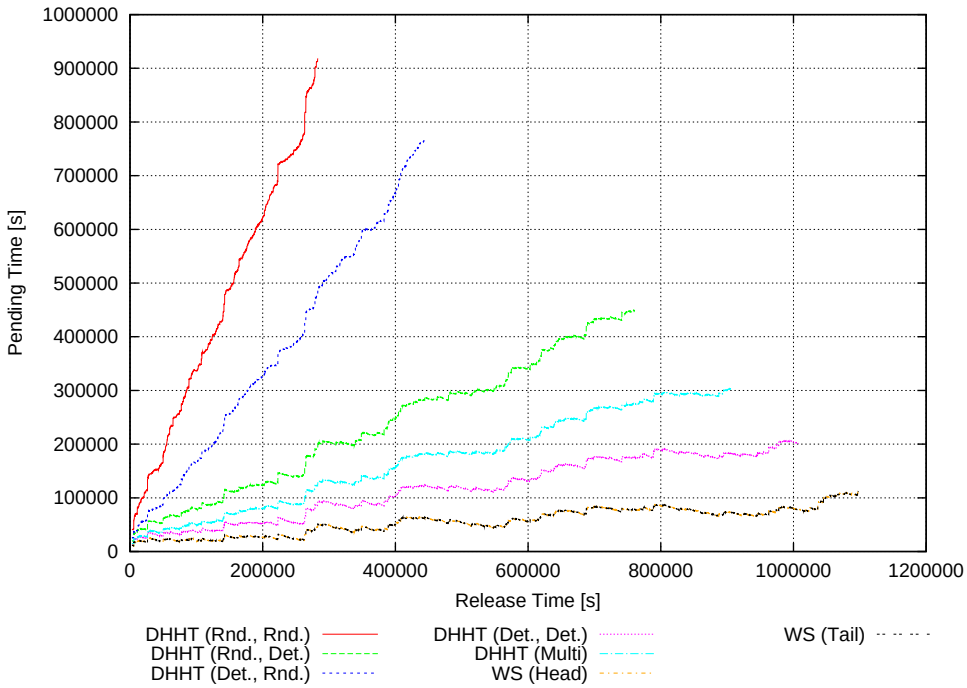

Figure A.24: *Pending time of jobs in experiment 17.*

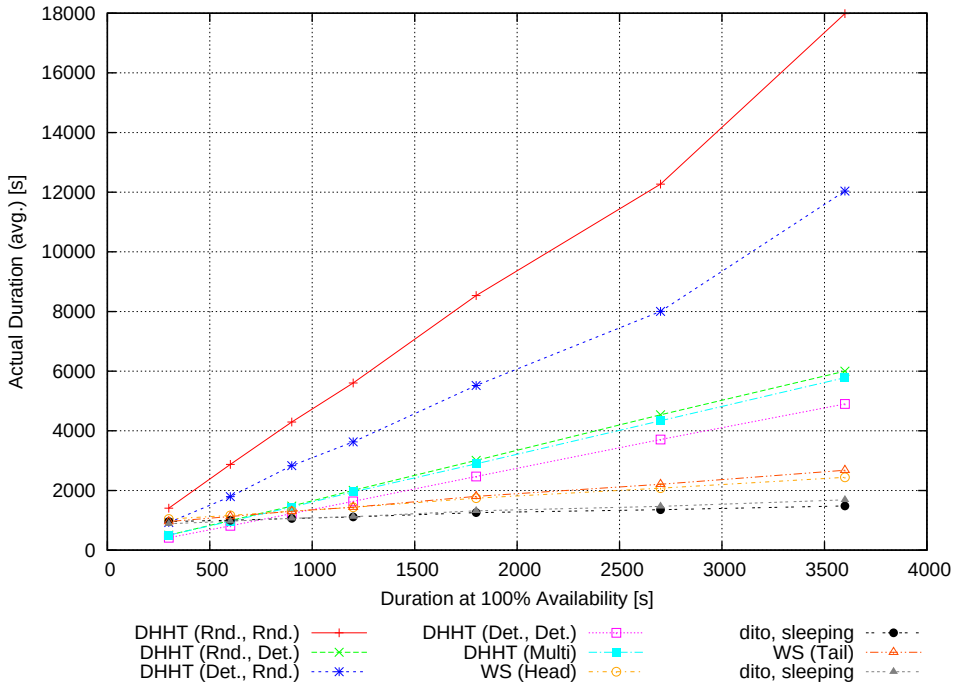

Figure A.25: *Actual average duration of processes in experiment 18.*

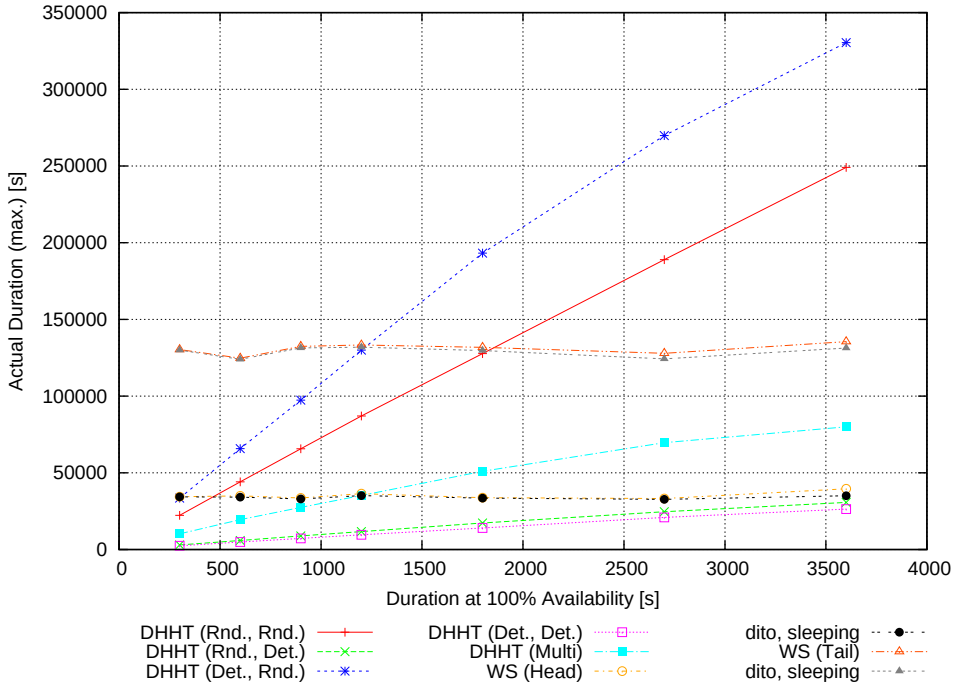

Figure A.26: *Actual maximum duration of processes in experiment 18.*

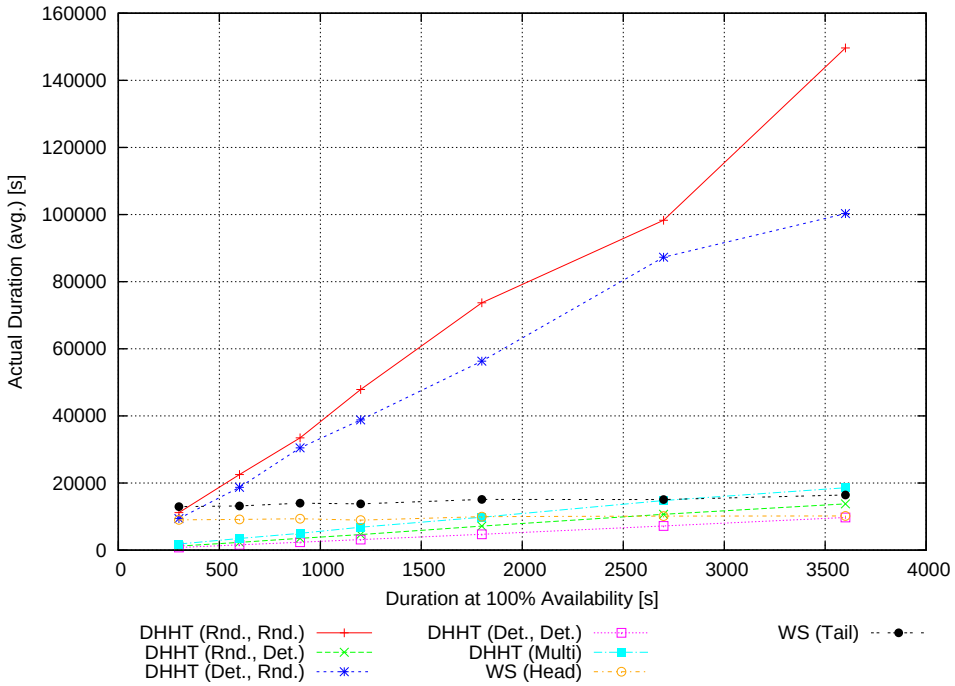

Figure A.27: *Actual average duration of jobs in experiment 18.*

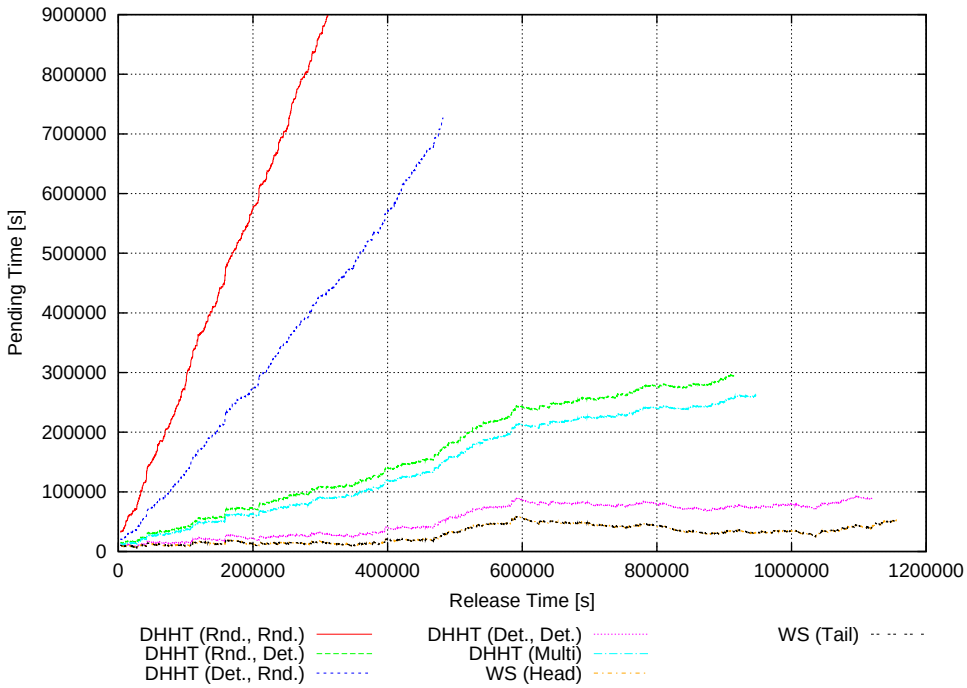

Figure A.28: *Pending time of jobs in experiment 18.*

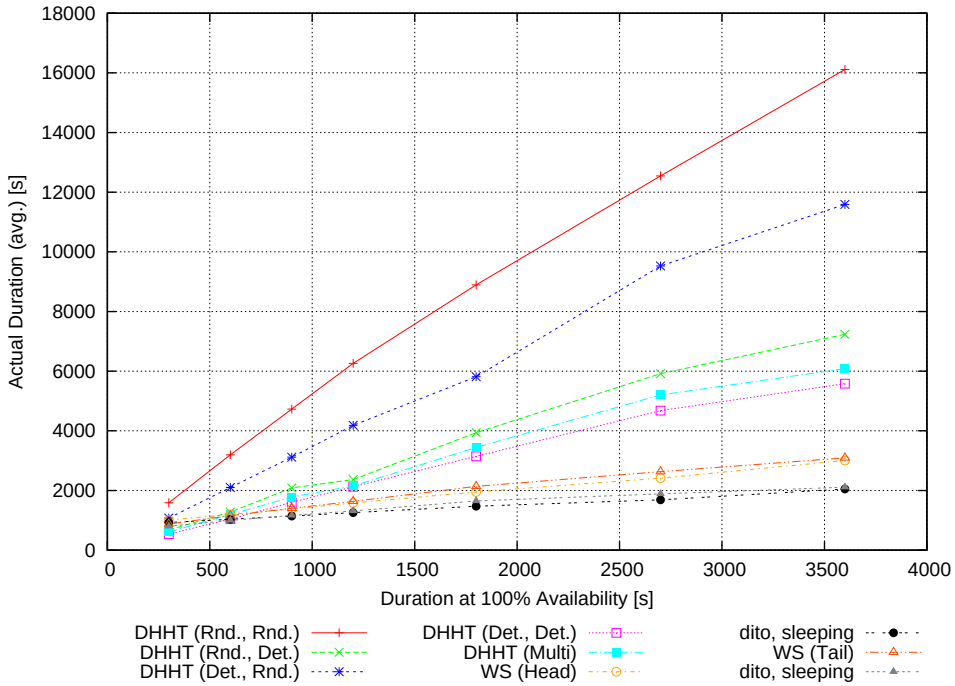

Figure A.29: *Actual average duration of processes in experiment 19.*

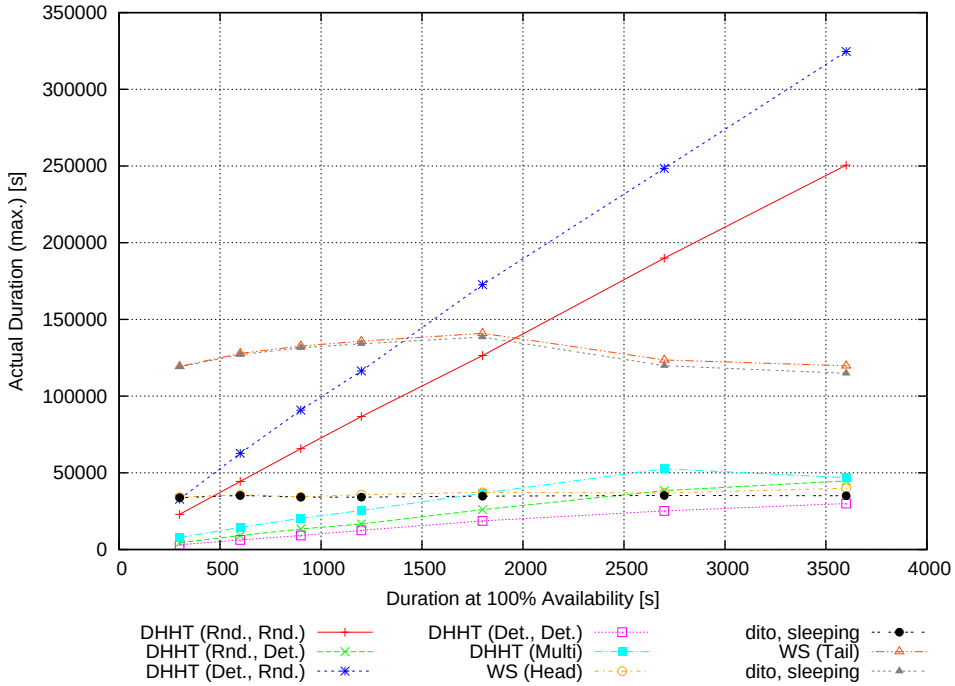

Figure A.30: *Actual maximum duration of processes in experiment 19.*

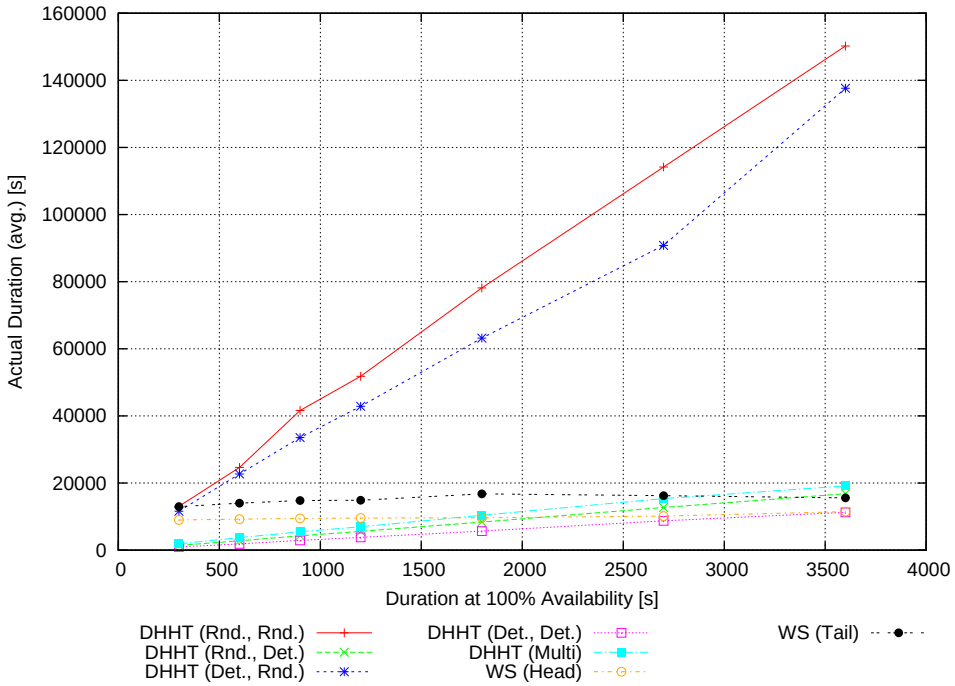

Figure A.31: *Actual average duration of jobs in experiment 19.*

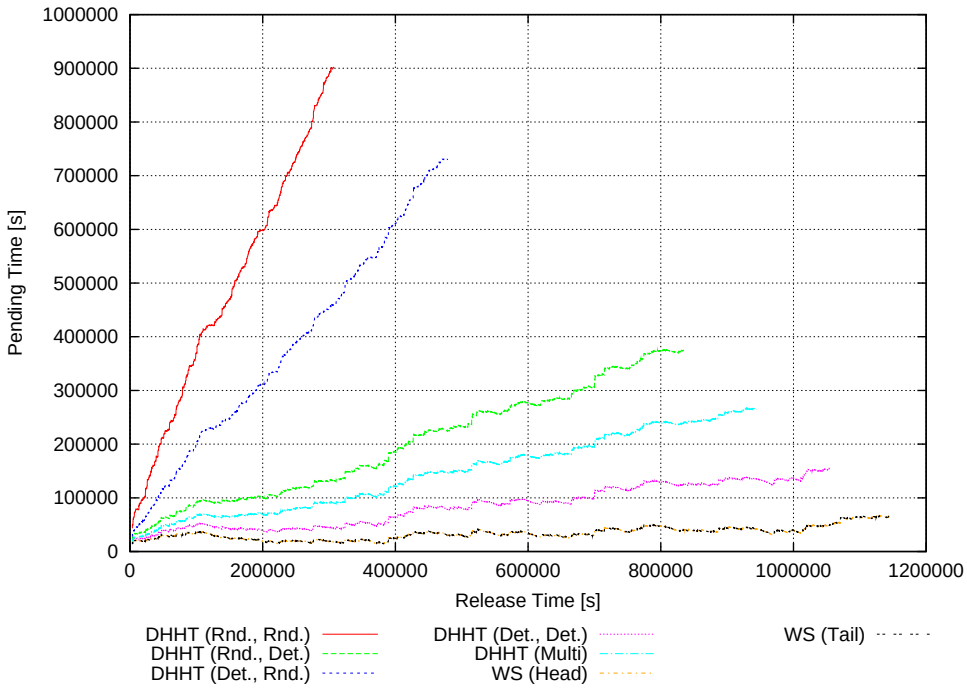

Figure A.32: *Pending time of jobs in experiment 19.*

A.3 Type C, Ideal Load

In this section, we present the plots for the experiments of type C with ideal total system load.

A.3.1 Notebooks

This subsection contains the plots for the input profiles HPC2N (figures A.33 – A.36), LLNL Atlas (figures A.37 – A.40), LLNL Thunder (figures A.41 – A.44), SDSC BLUE (figures A.45 – A.48), and SDSC DataStar (figures A.49 – A.52) for the experiments of type C with ideal total system load and using the notebook load profile.

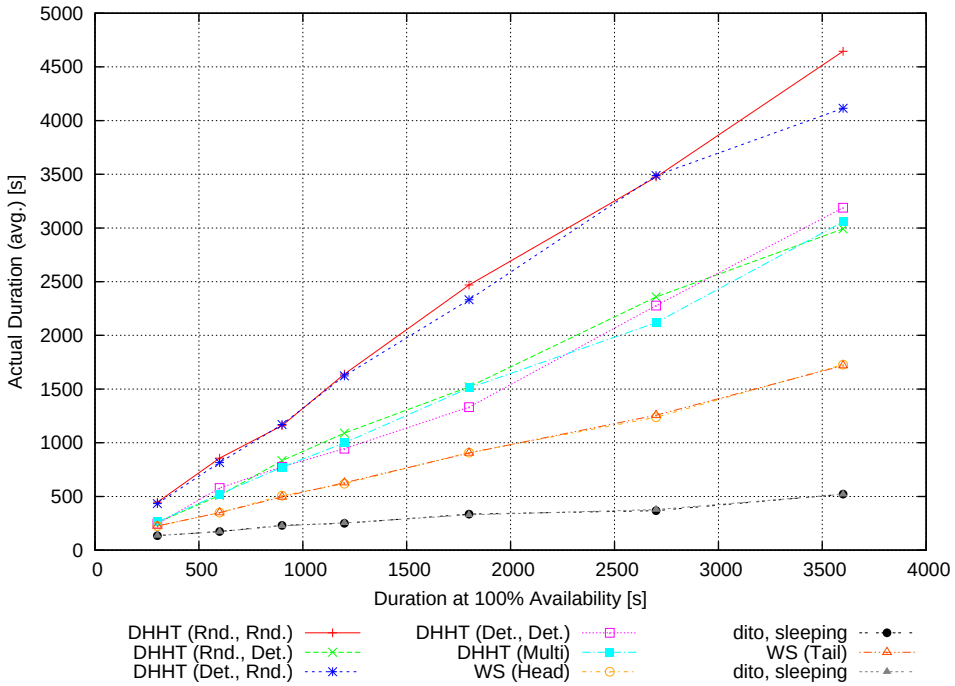

Figure A.33: *Actual average duration of processes in experiment 20.*

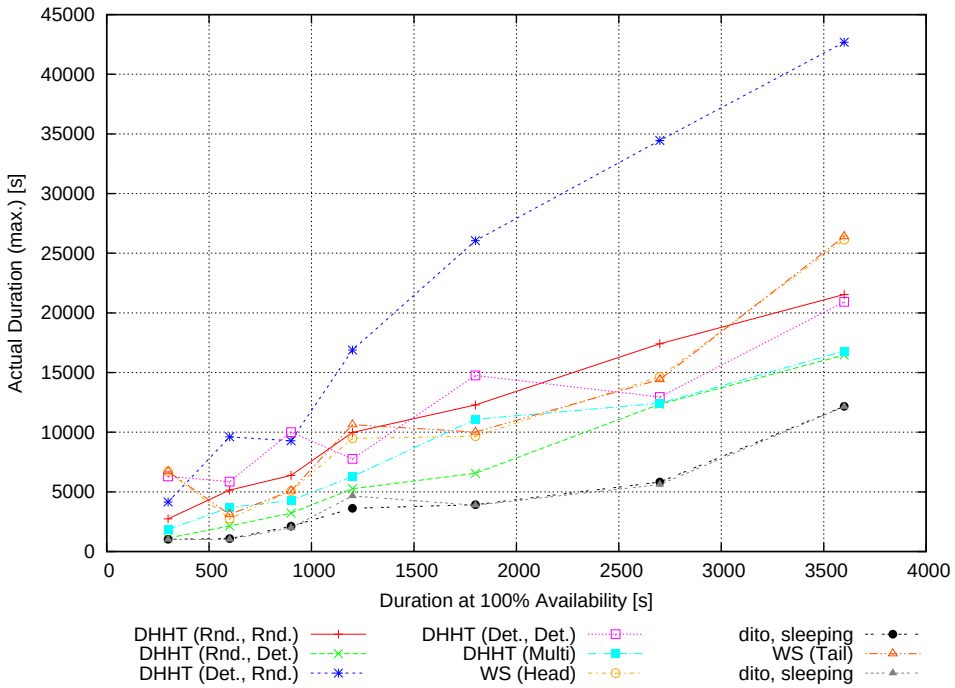

Figure A.34: *Actual maximum duration of processes in experiment 20.*

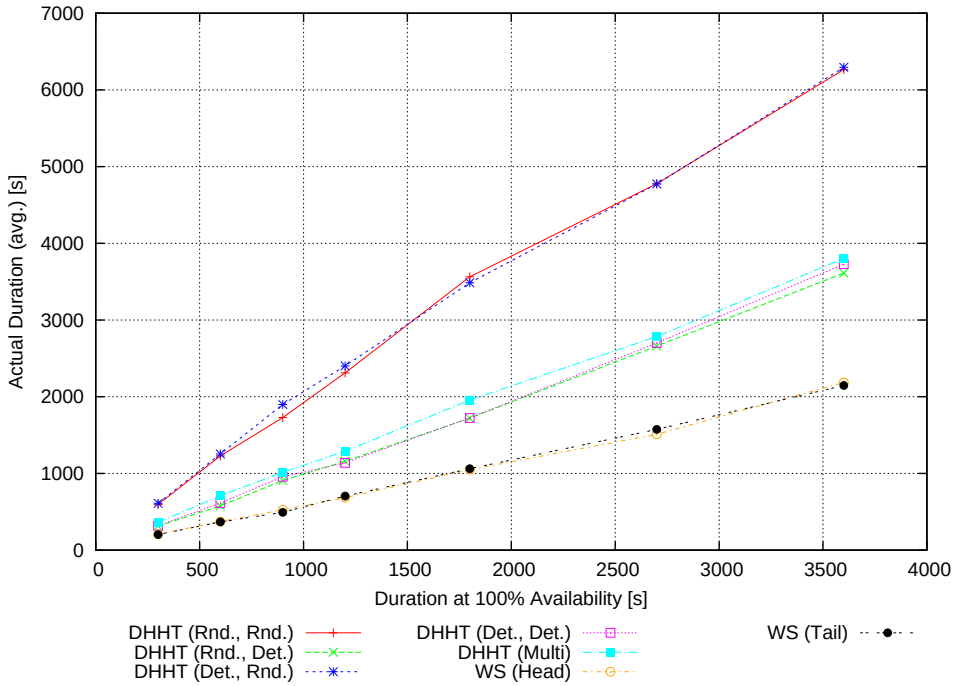

Figure A.35: *Actual average duration of jobs in experiment 20.*

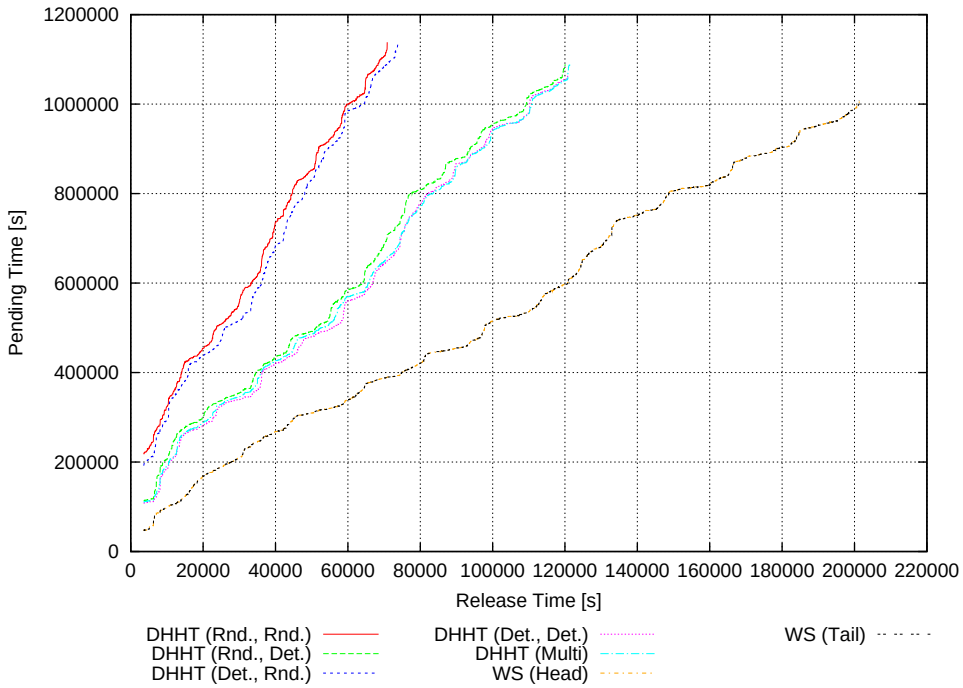

Figure A.36: *Pending time of jobs in experiment 20.*

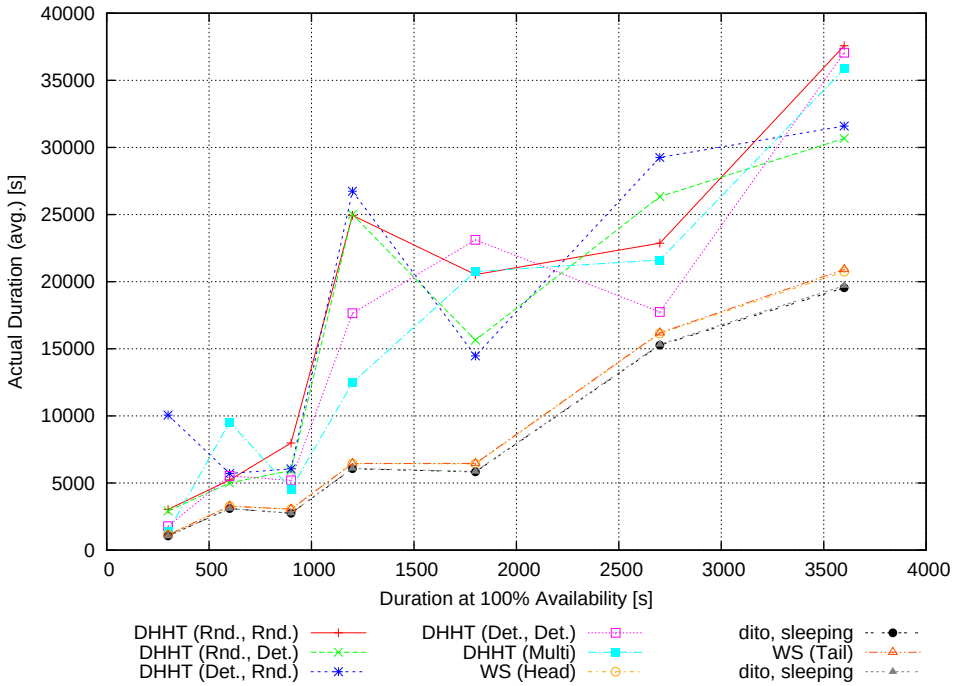

Figure A.37: *Actual average duration of processes in experiment 21.*

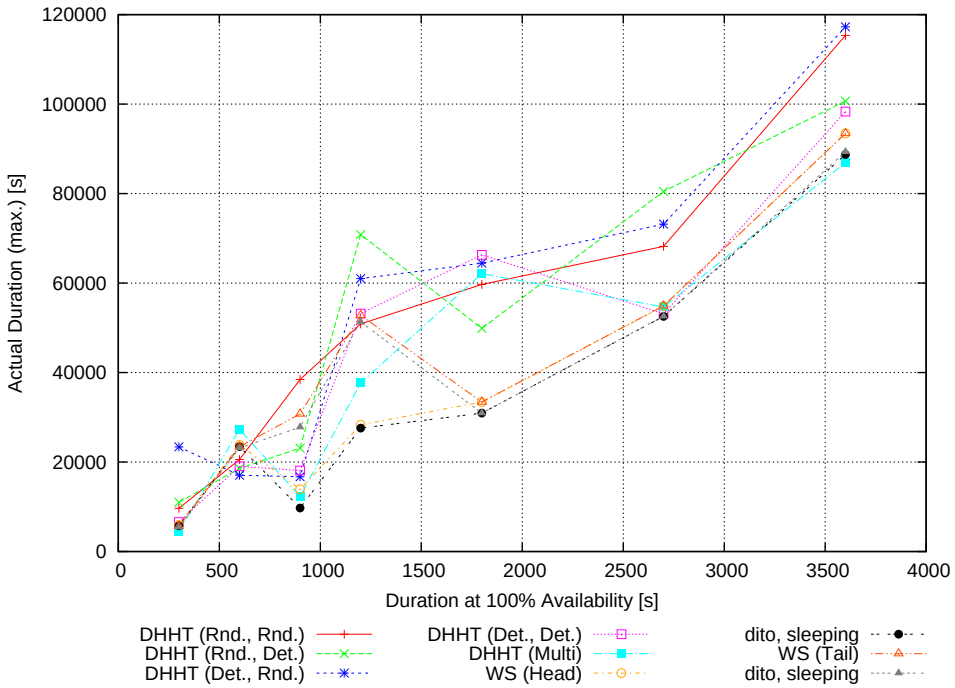

Figure A.38: *Actual maximum duration of processes in experiment 21.*

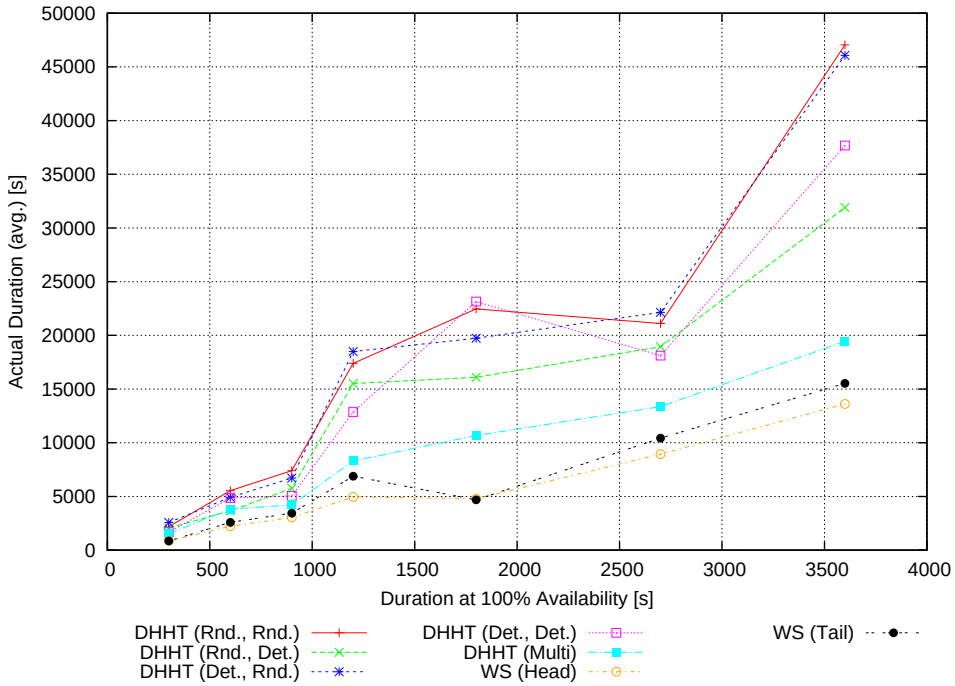

Figure A.39: *Actual average duration of jobs in experiment 21.*

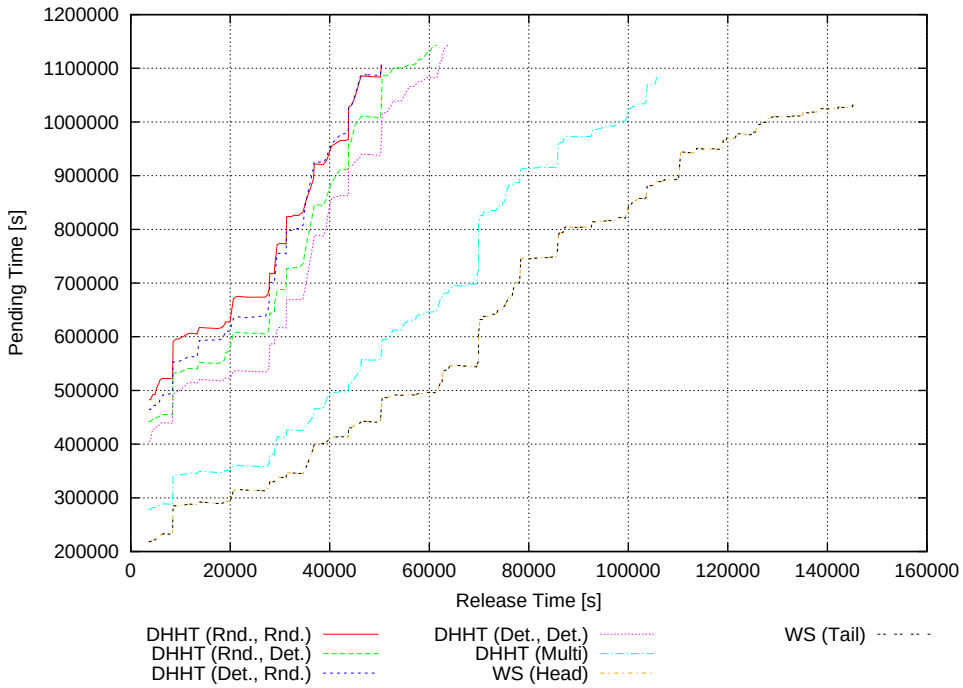

Figure A.40: *Pending time of jobs in experiment 21.*

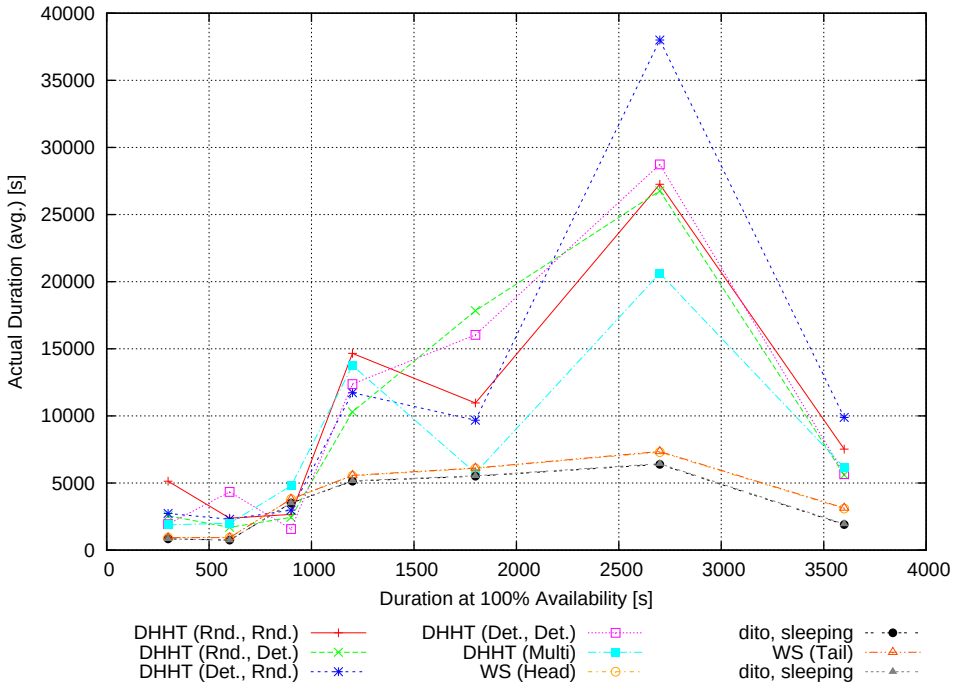

Figure A.41: *Actual average duration of processes in experiment 22.*

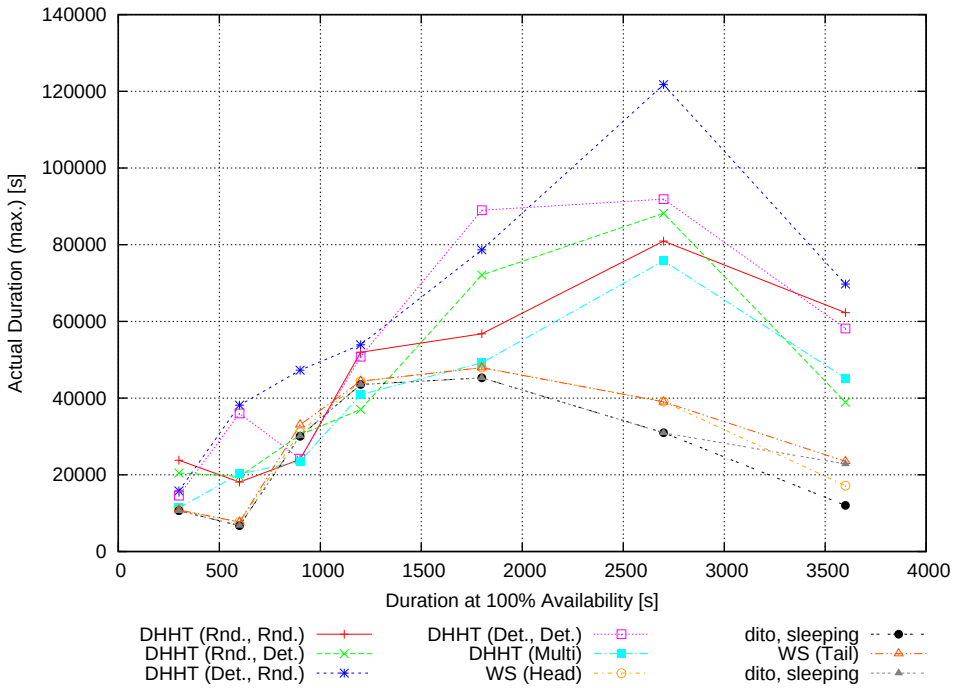

Figure A.42: *Actual maximum duration of processes in experiment 22.*

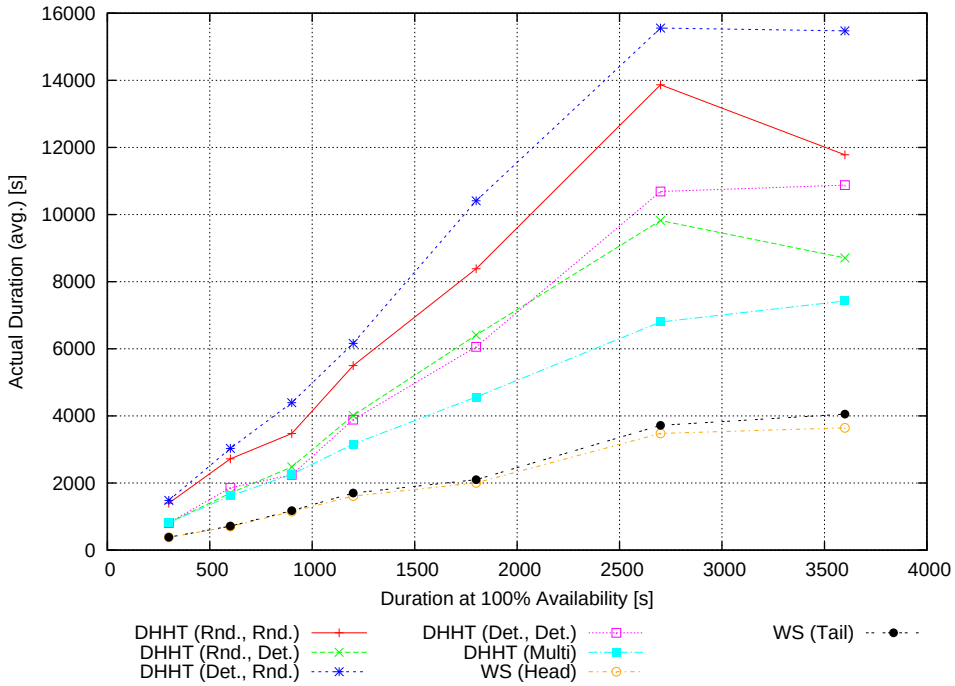

Figure A.43: *Actual average duration of jobs in experiment 22.*

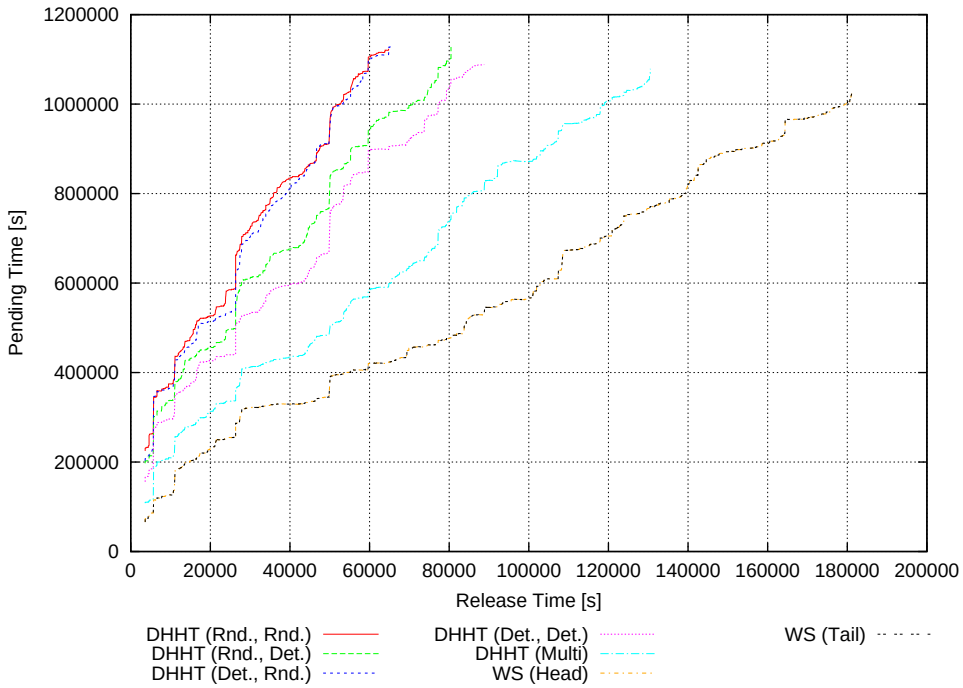

Figure A.44: *Pending time of jobs in experiment 22.*

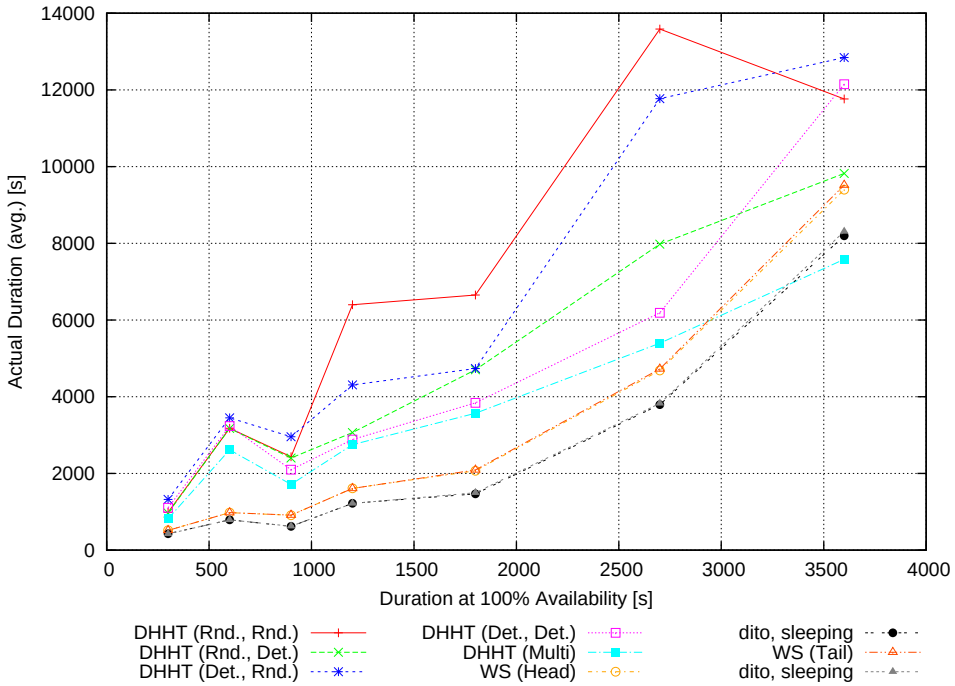

Figure A.45: *Actual average duration of processes in experiment 23.*

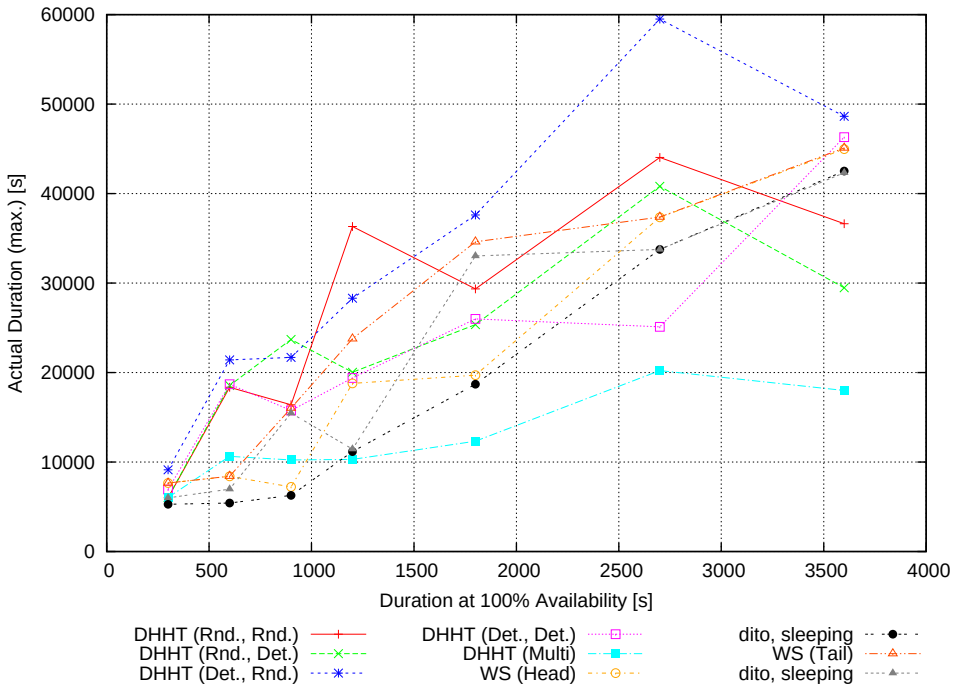

Figure A.46: *Actual maximum duration of processes in experiment 23.*

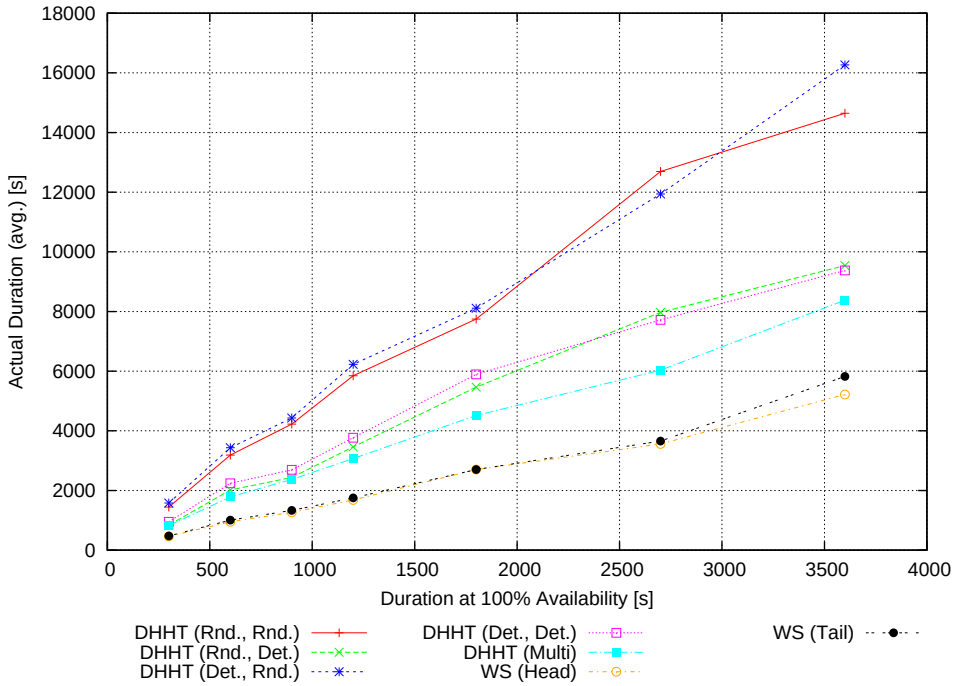

Figure A.47: *Actual average duration of jobs in experiment 23.*

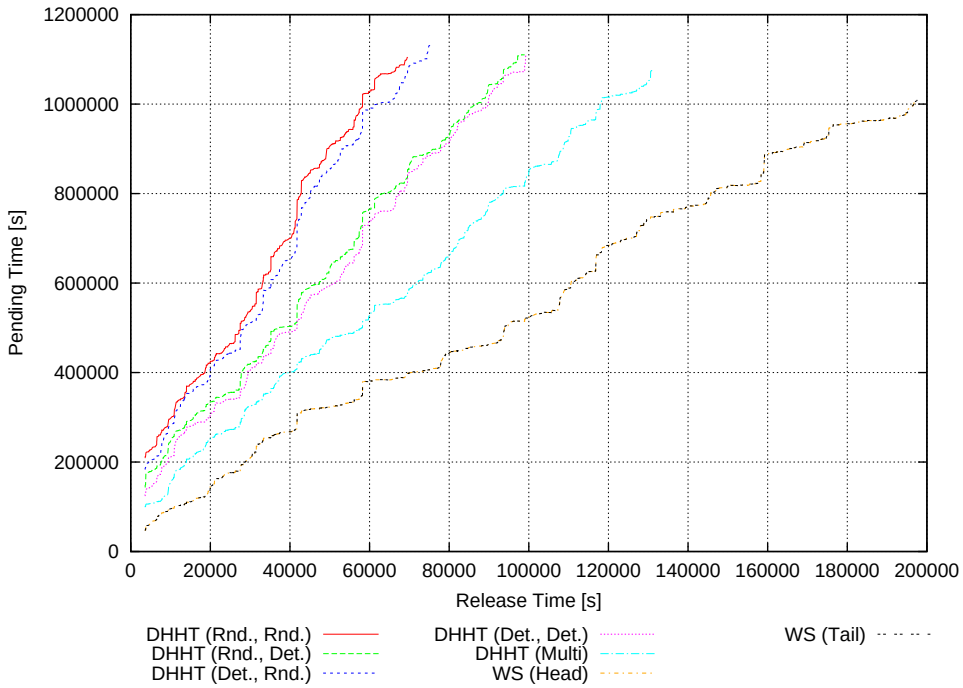

Figure A.48: *Pending time of jobs in experiment 23.*

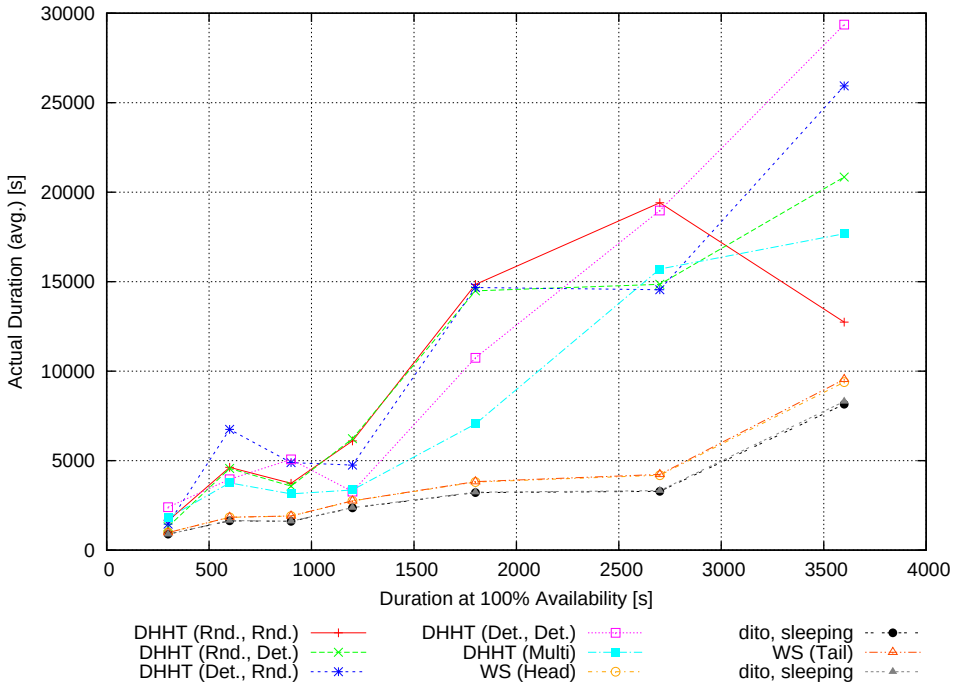

Figure A.49: *Actual average duration of processes in experiment 24.*

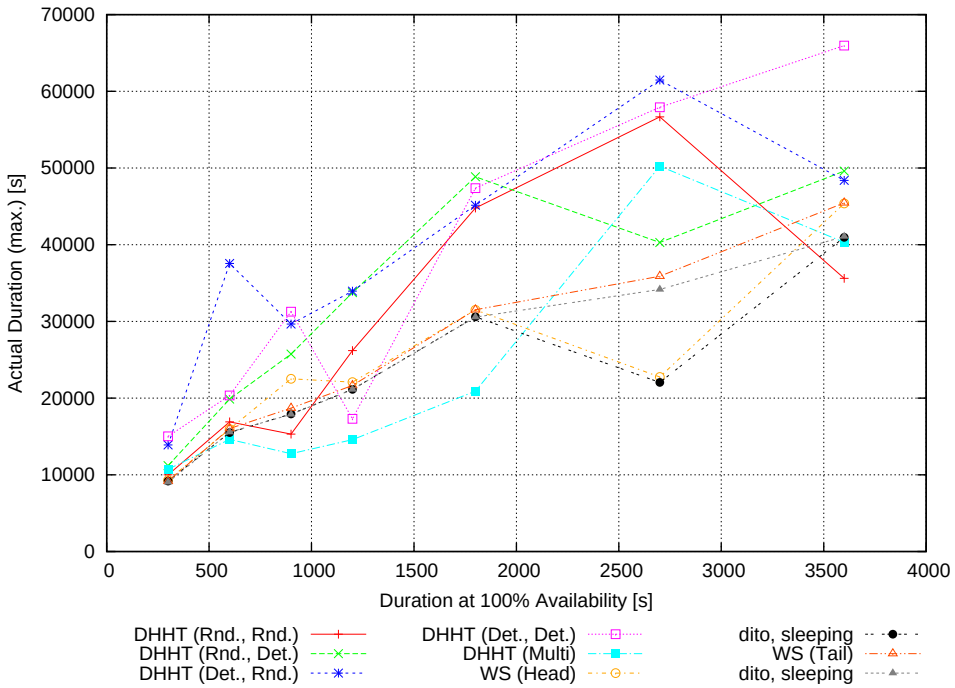

Figure A.50: *Actual maximum duration of processes in experiment 24.*

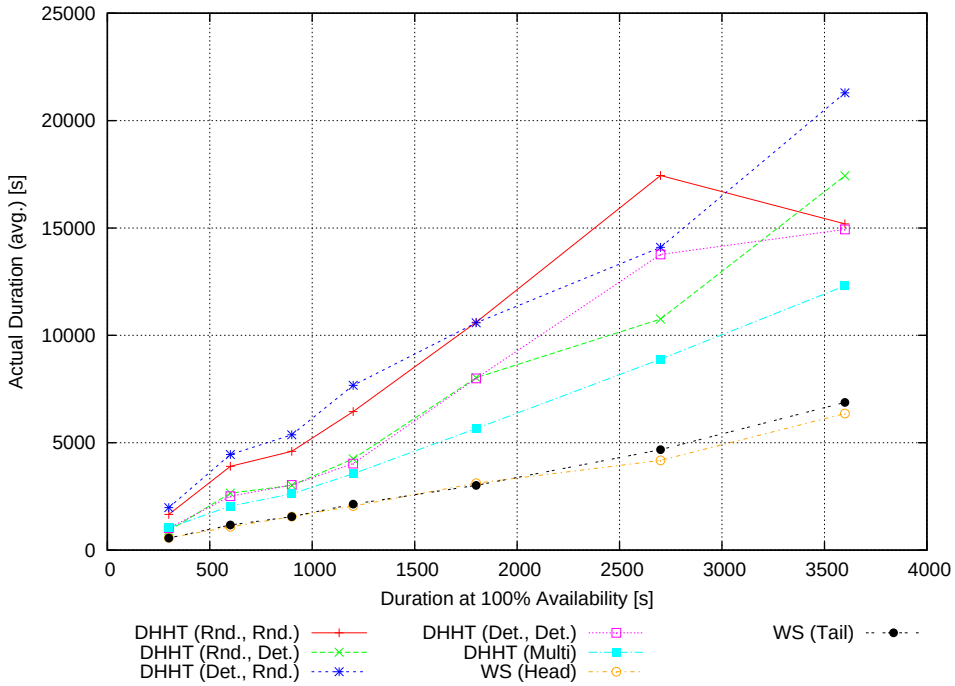

Figure A.51: *Actual average duration of jobs in experiment 24.*

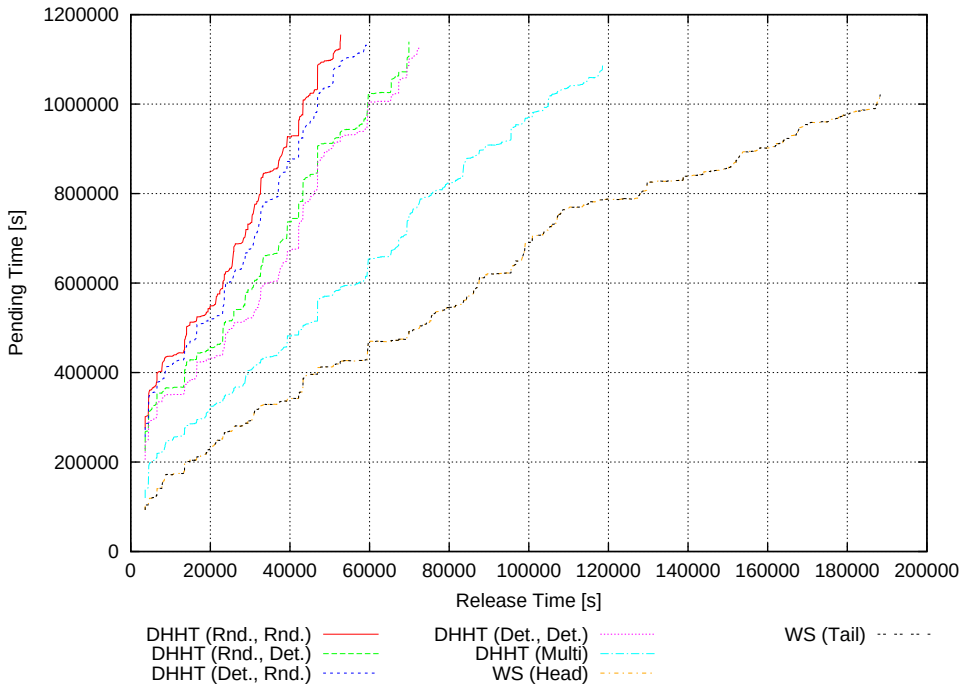

Figure A.52: *Pending time of jobs in experiment 24.*

A.3.2 Office PCs

This subsection contains the plots for the input profiles HPC2N (figures A.53 – A.56), LLNL Atlas (figures A.57 – A.60), LLNL Thunder (figures A.61 – A.64), SDSC BLUE (figures A.65 – A.68), and SDSC DataStar (figures A.69 – A.72) for the experiments of type C with ideal total system load and using the office PC load profile.

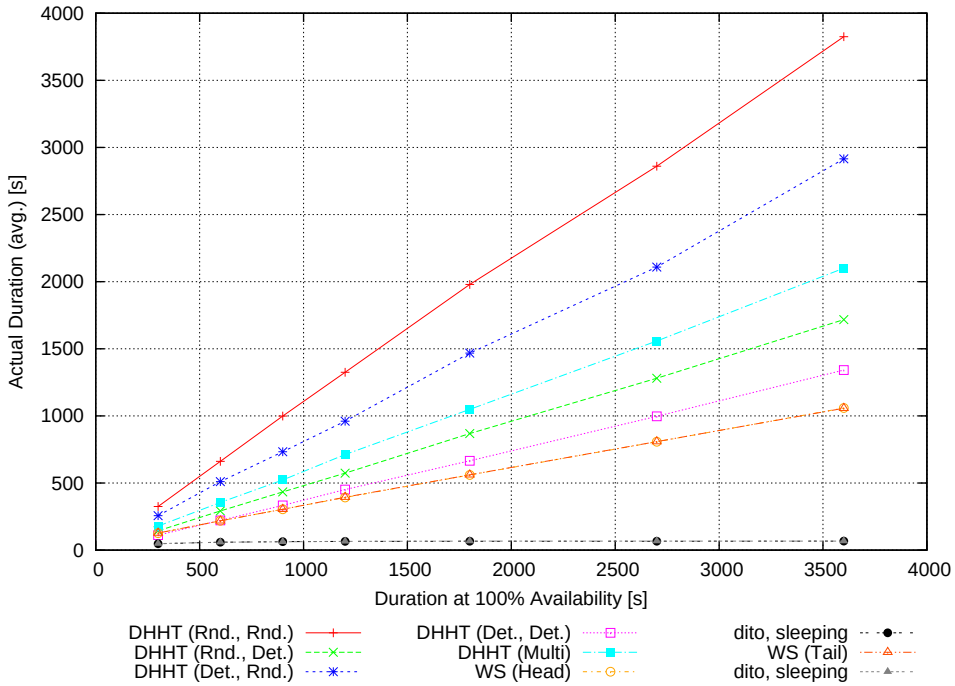

Figure A.53: *Actual average duration of processes in experiment 25.*

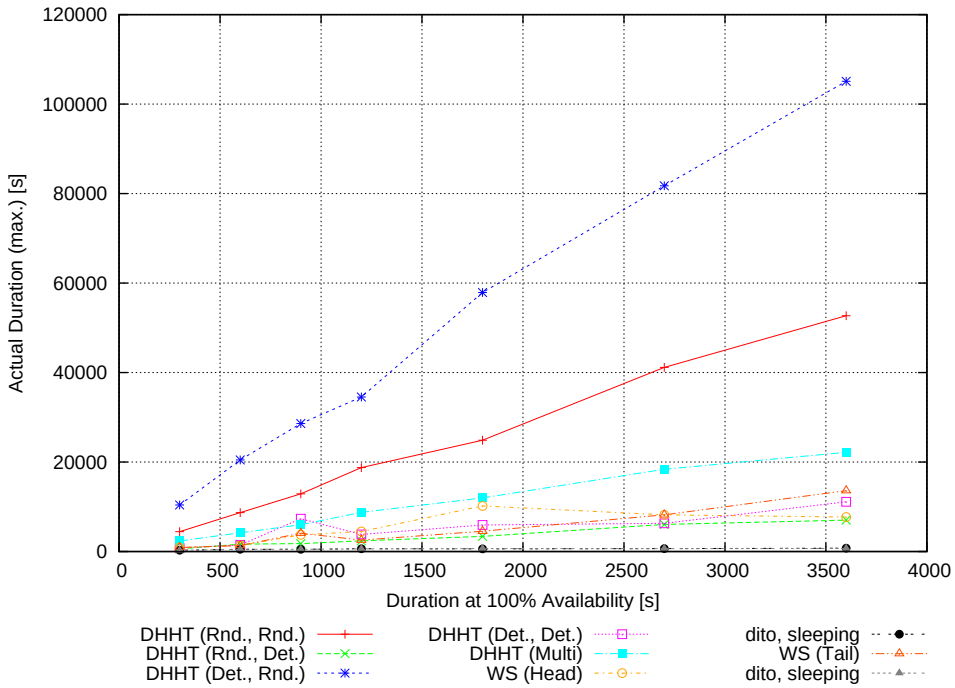

Figure A.54: *Actual maximum duration of processes in experiment 25.*

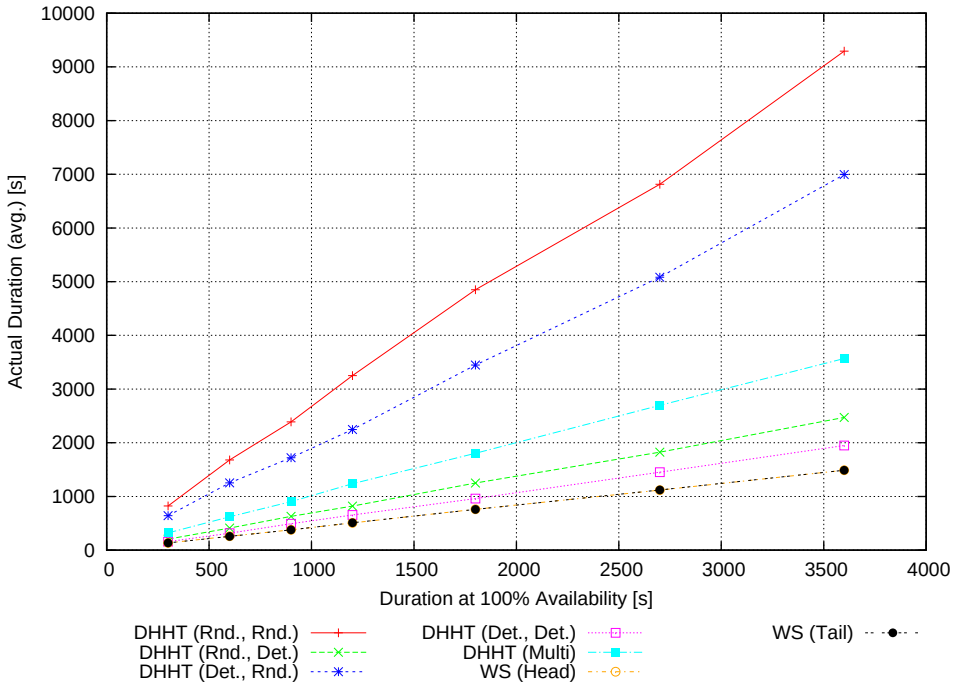

Figure A.55: *Actual average duration of jobs in experiment 25.*

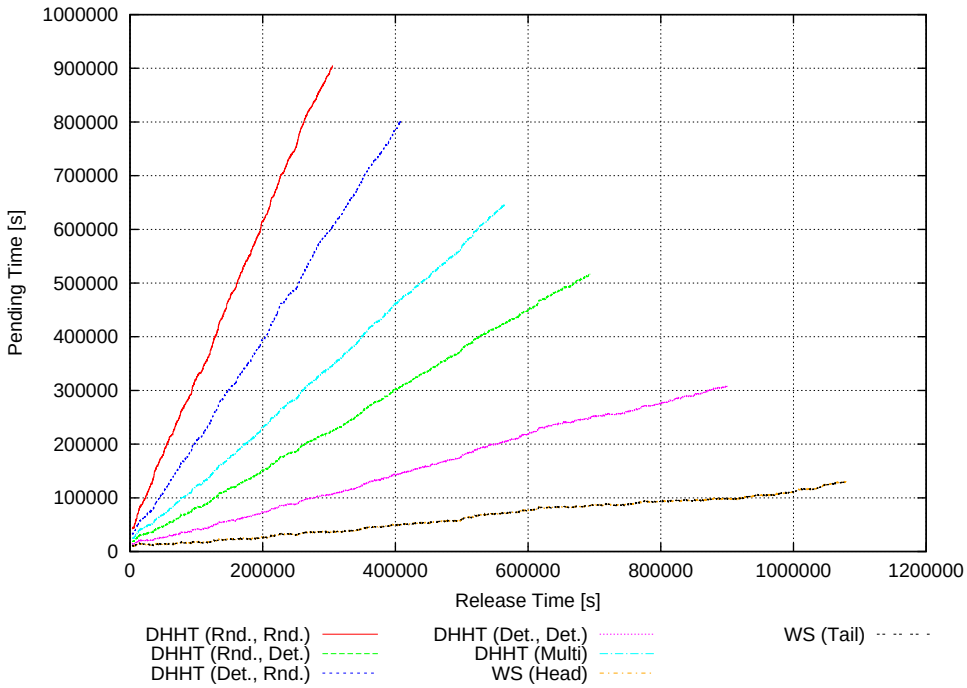

Figure A.56: *Pending time of jobs in experiment 25.*

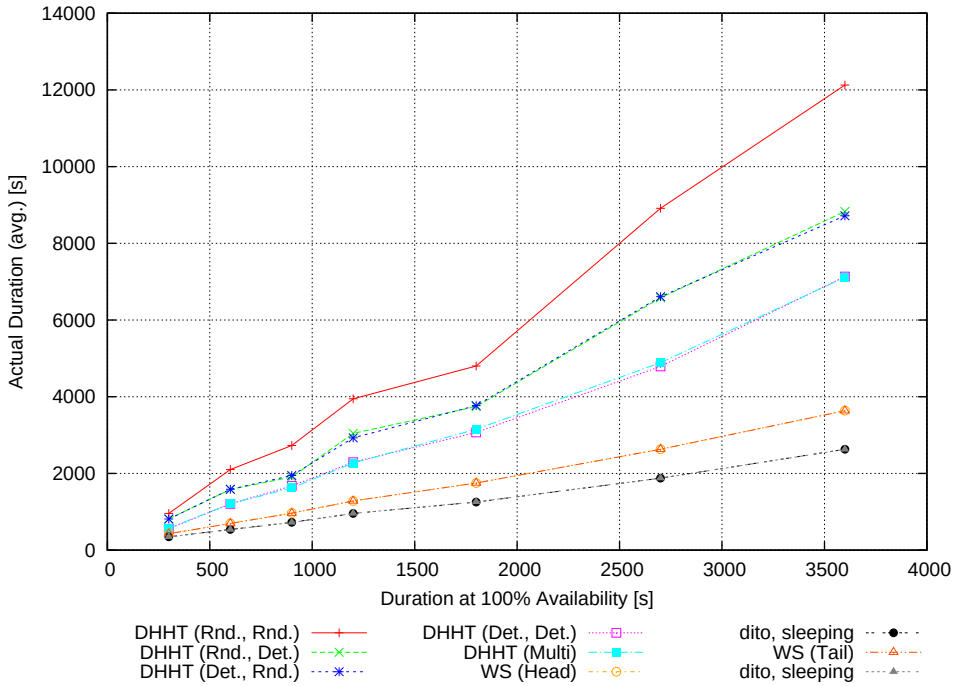

Figure A.57: *Actual average duration of processes in experiment 26.*

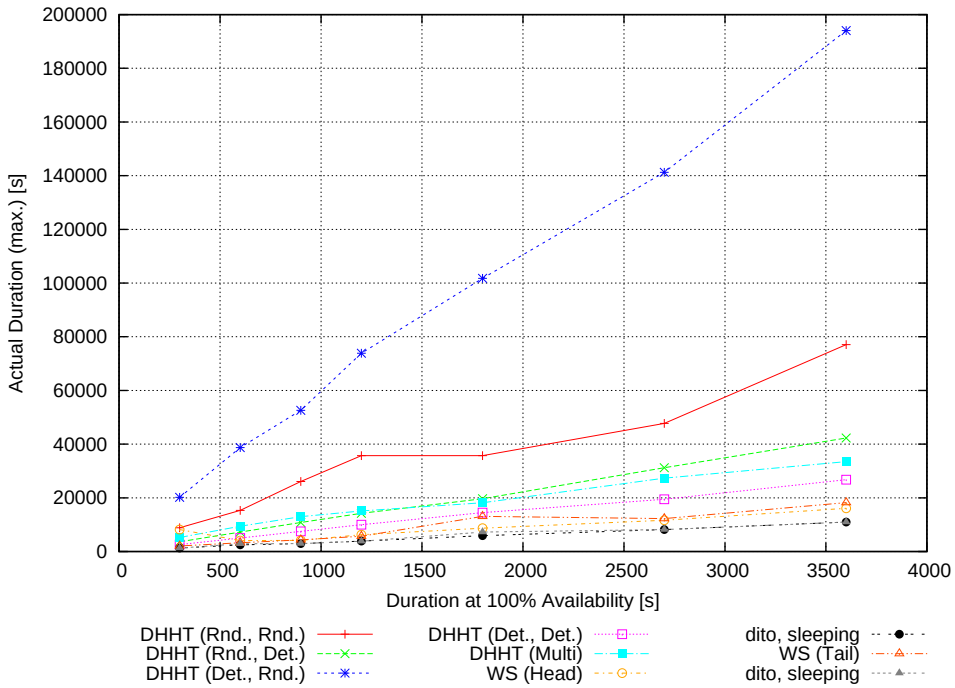

Figure A.58: *Actual maximum duration of processes in experiment 26.*

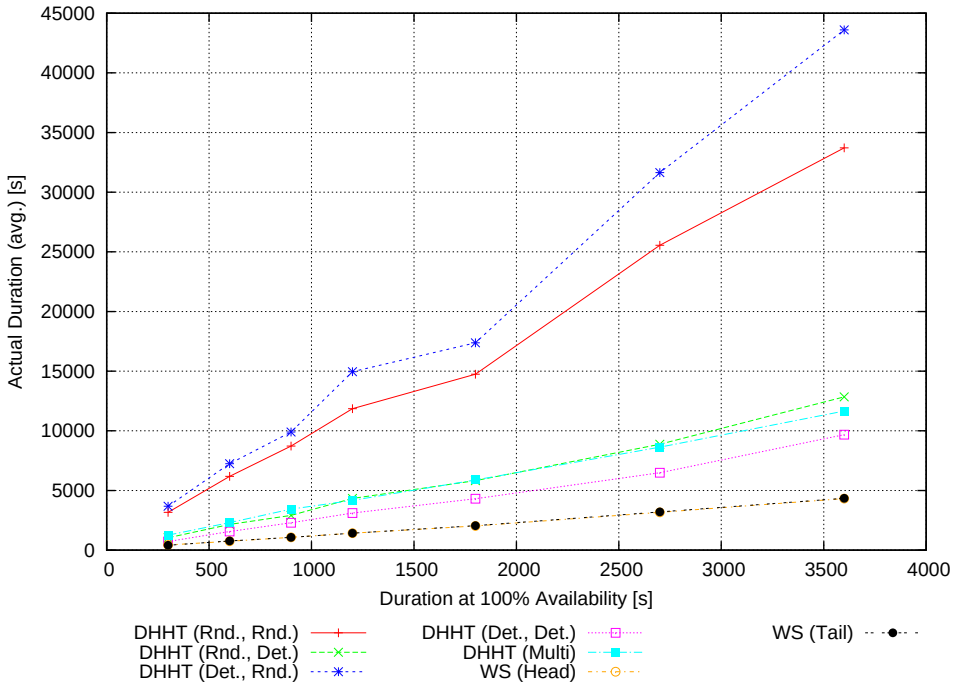

Figure A.59: *Actual average duration of jobs in experiment 26.*

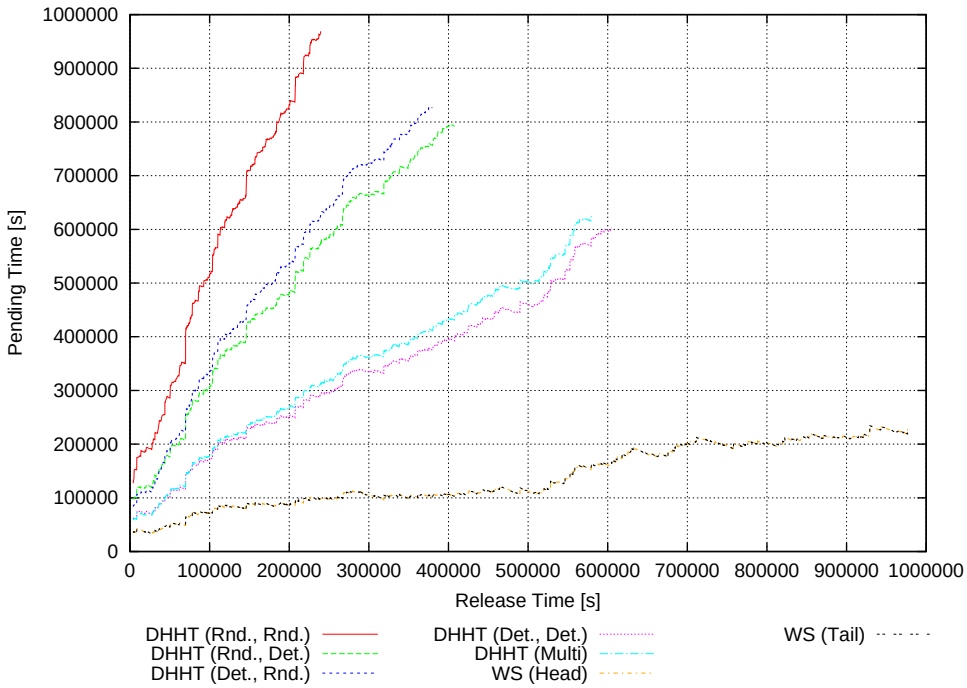

Figure A.60: *Pending time of jobs in experiment 26.*

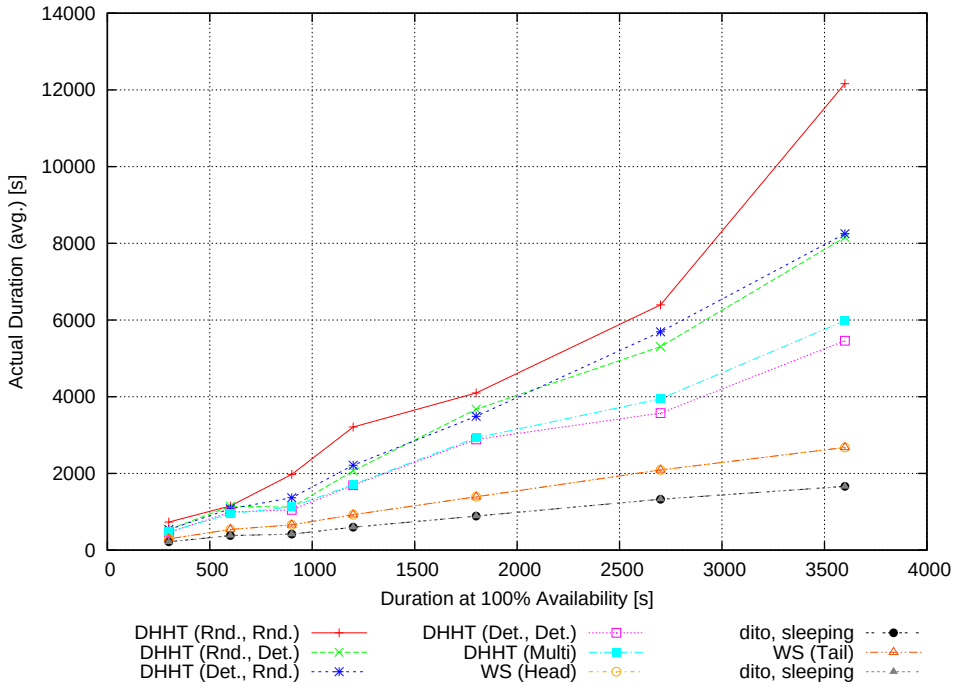

Figure A.61: *Actual average duration of processes in experiment 27.*

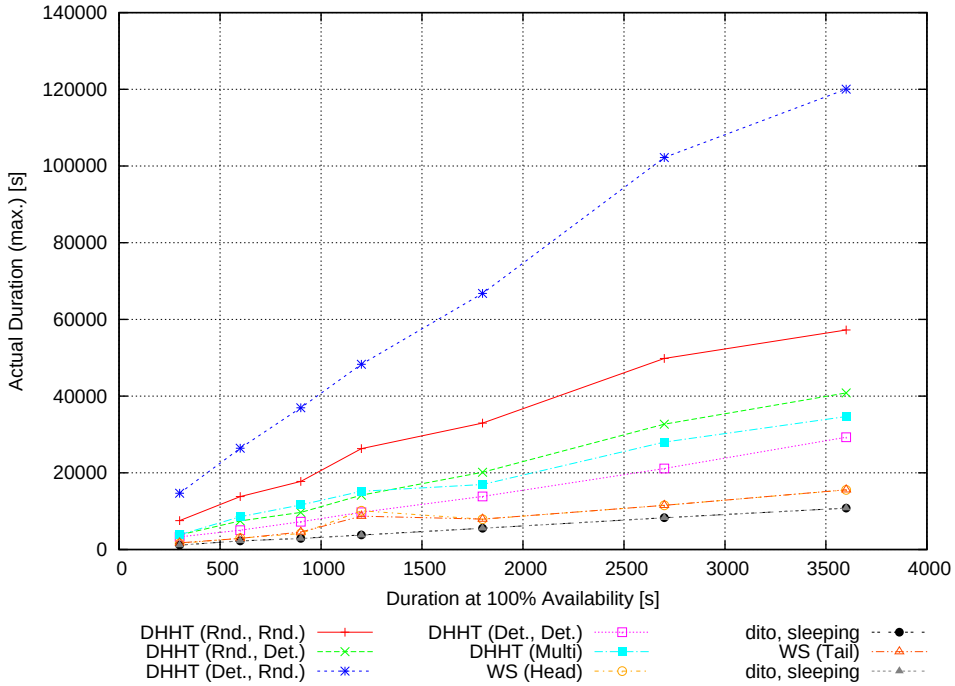

Figure A.62: *Actual maximum duration of processes in experiment 27.*

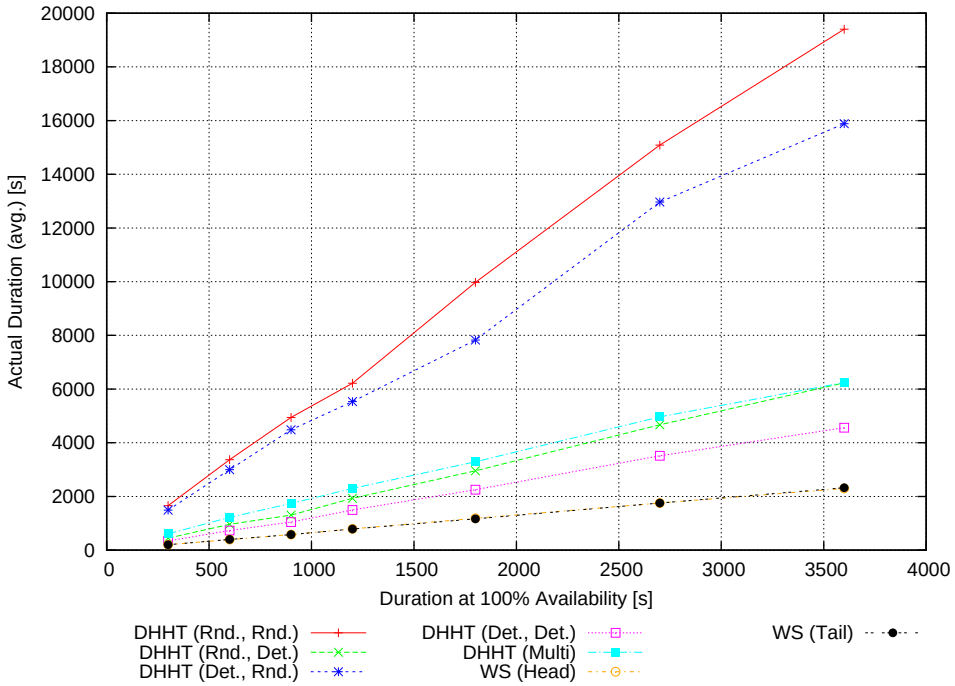

Figure A.63: *Actual average duration of jobs in experiment 27.*

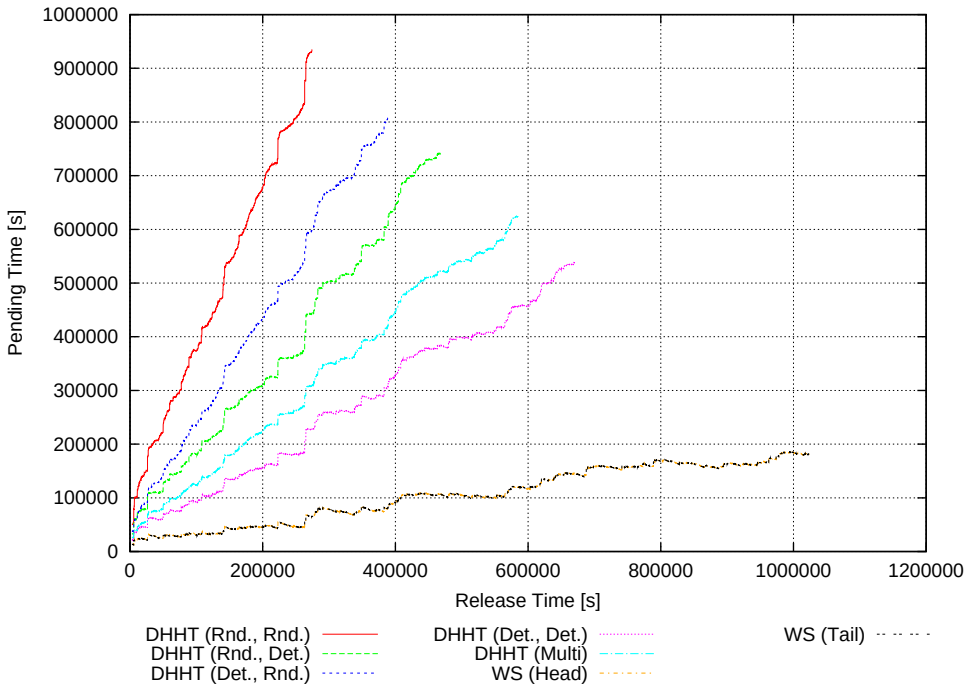

Figure A.64: *Pending time of jobs in experiment 27.*

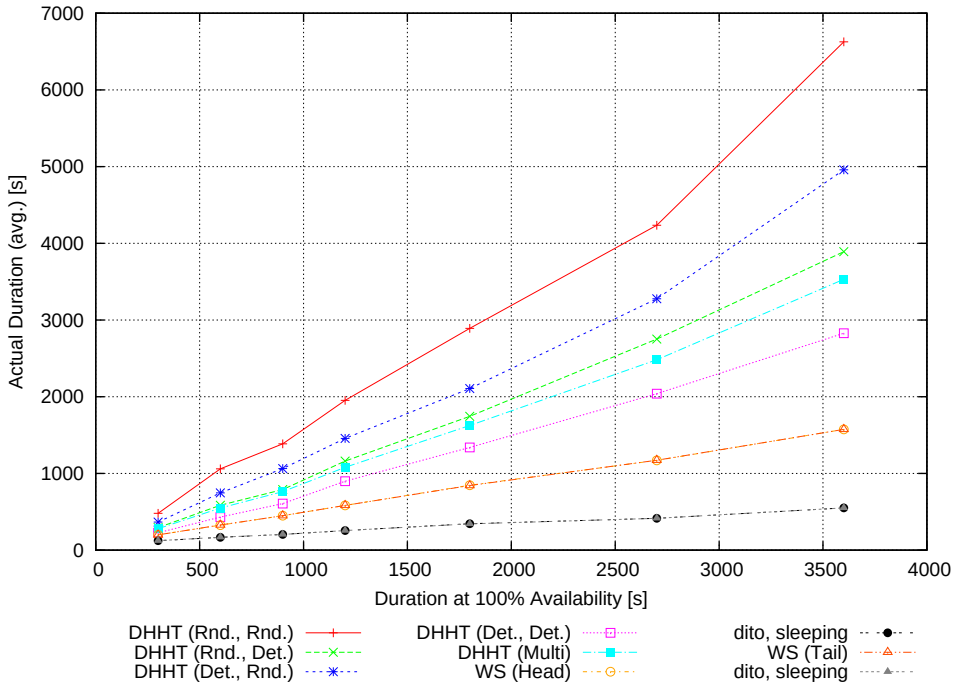

Figure A.65: *Actual average duration of processes in experiment 28.*

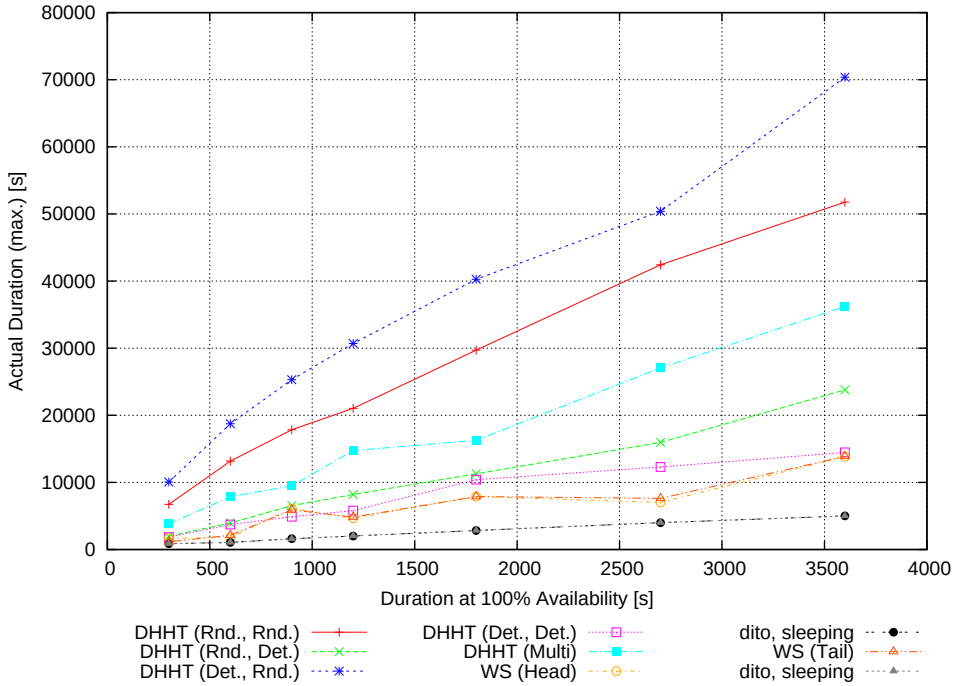

Figure A.66: *Actual maximum duration of processes in experiment 28.*

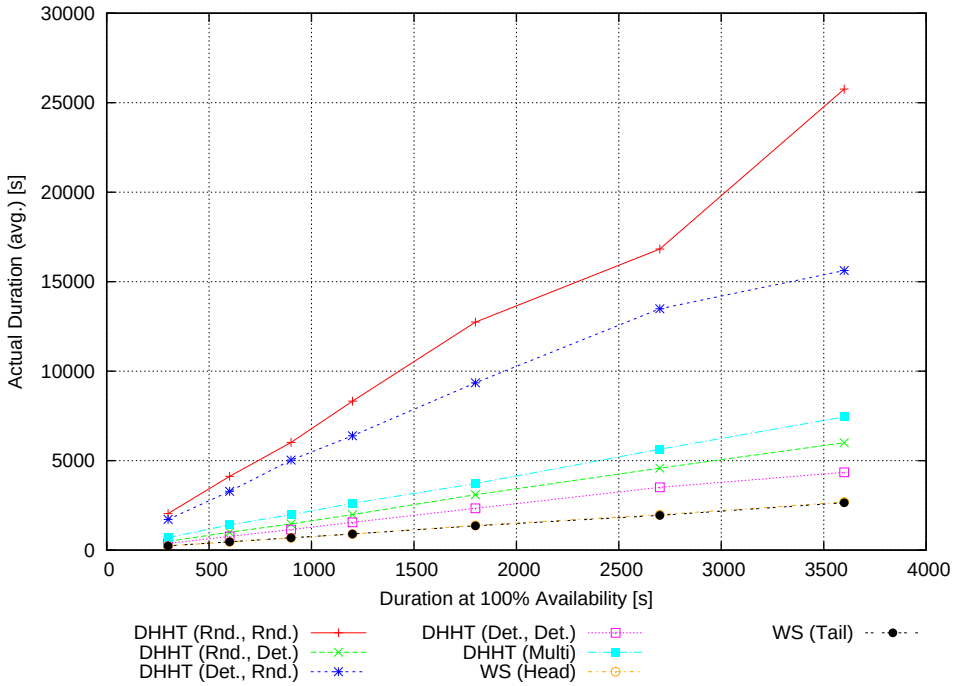

Figure A.67: *Actual average duration of jobs in experiment 28.*

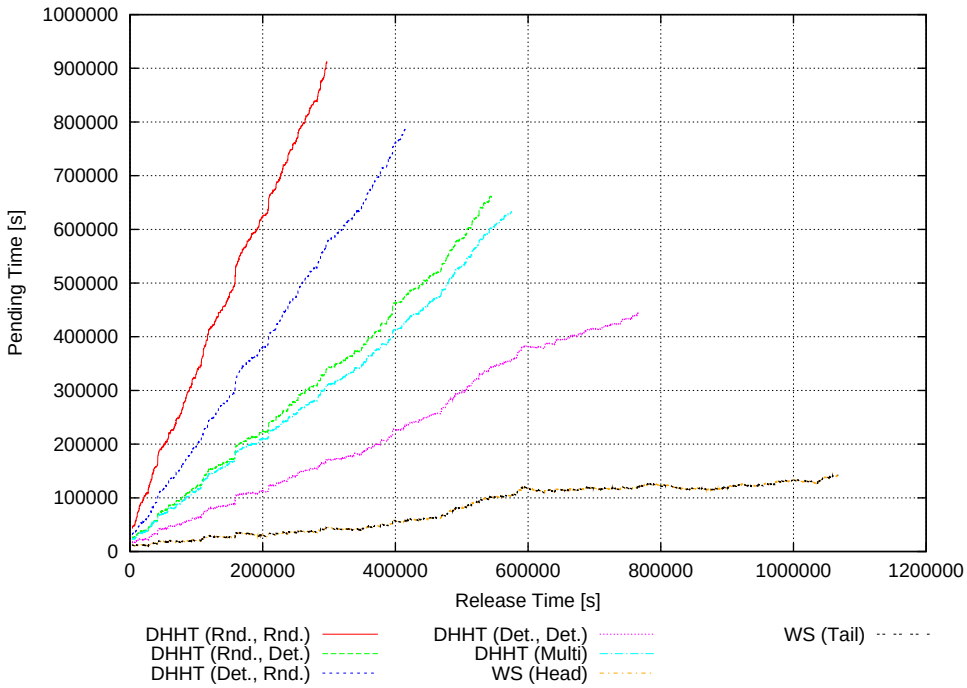

Figure A.68: *Pending time of jobs in experiment 28.*

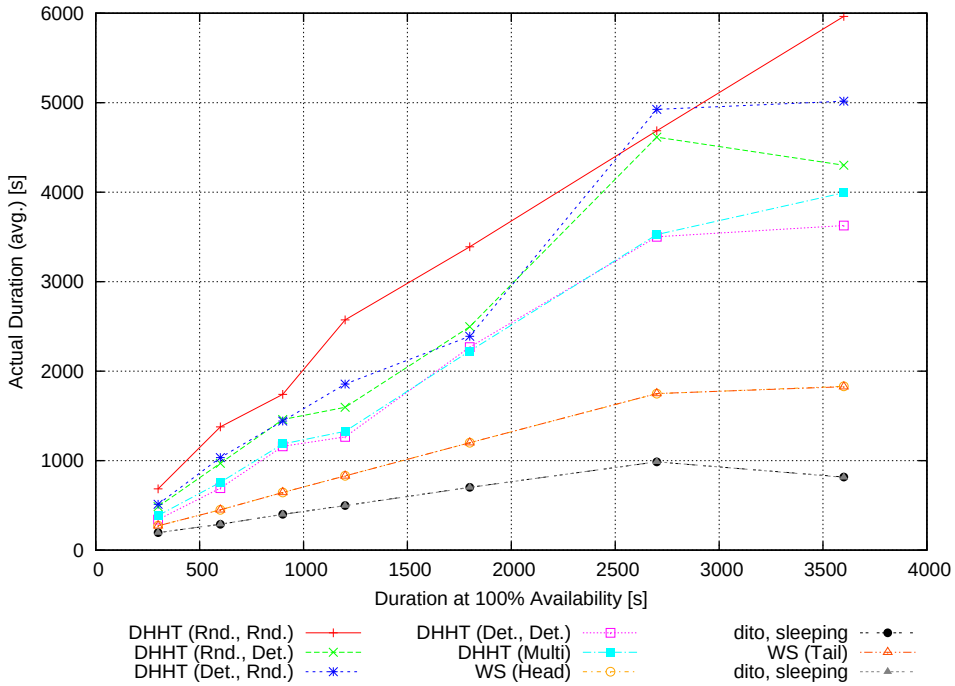

Figure A.69: *Actual average duration of processes in experiment 29.*

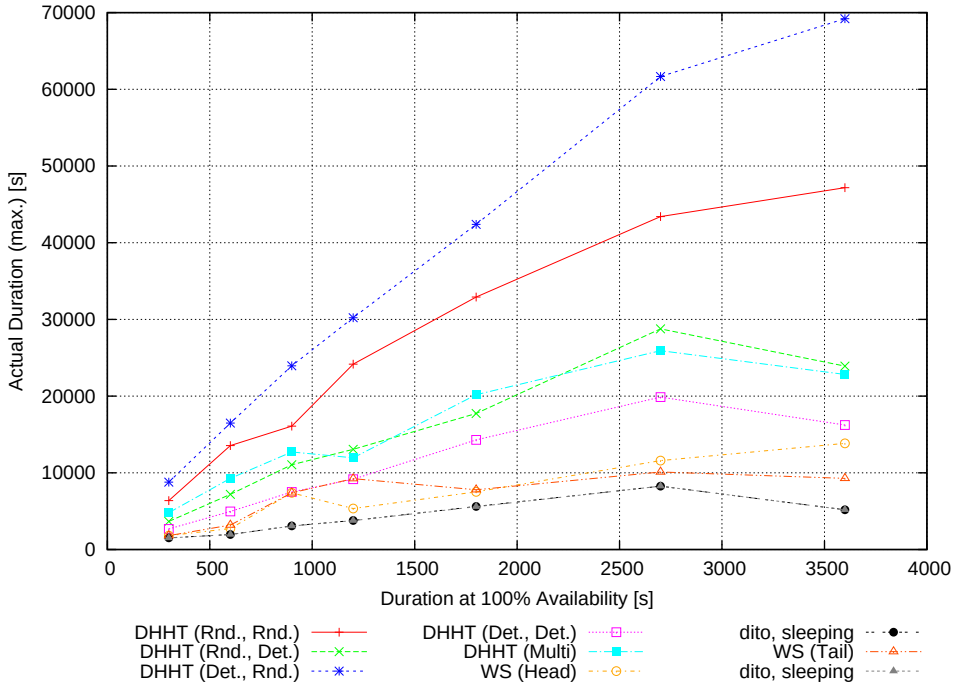

Figure A.70: *Actual maximum duration of processes in experiment 29.*

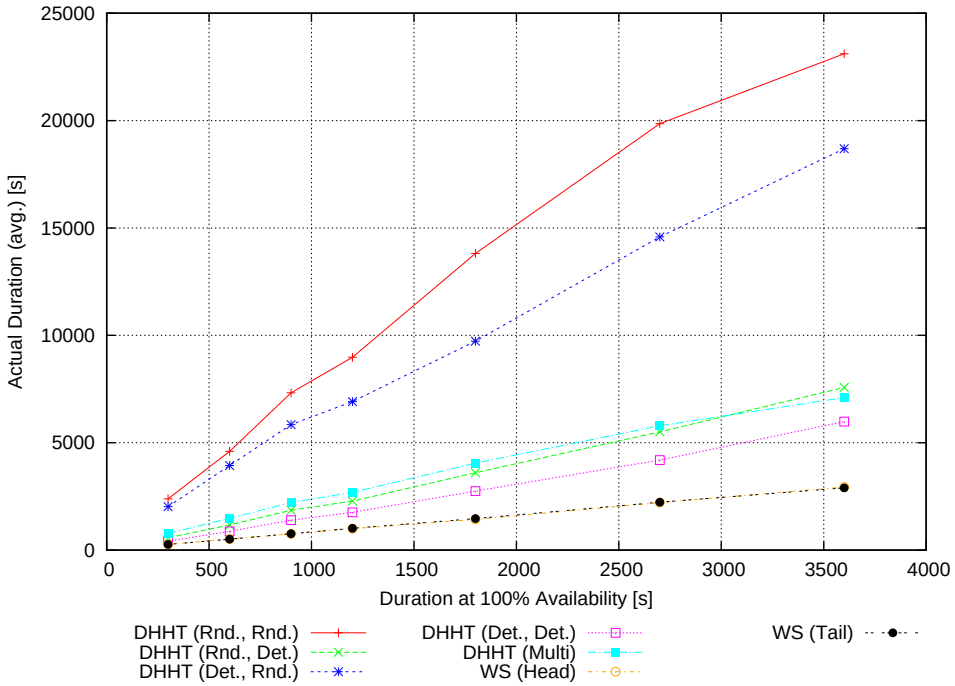

Figure A.71: *Actual average duration of jobs in experiment 29.*

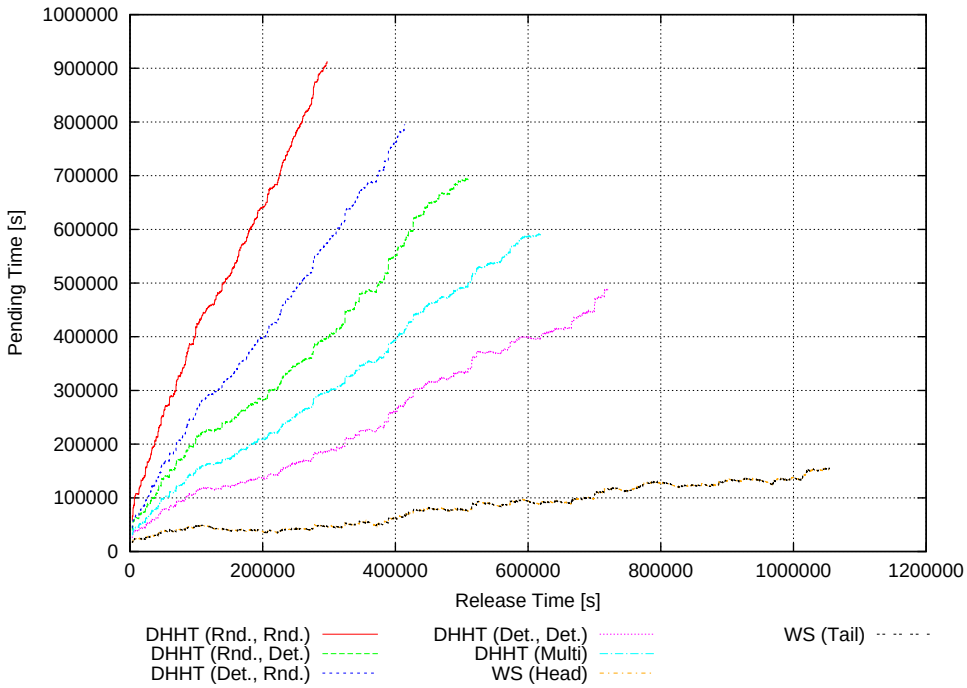

Figure A.72: *Pending time of jobs in experiment 29.*

A.3.3 Pool PCs

This subsection contains the plots for the input profiles HPC2N (figures A.73 – A.76), LLNL Atlas (figures A.77 – A.80), LLNL Thunder (figures A.81 – A.84), SDSC BLUE (figures A.85 – A.88), and SDSC DataStar (figures A.89 – A.92) for the experiments of type C with ideal total system load and using the pool PC load profile.

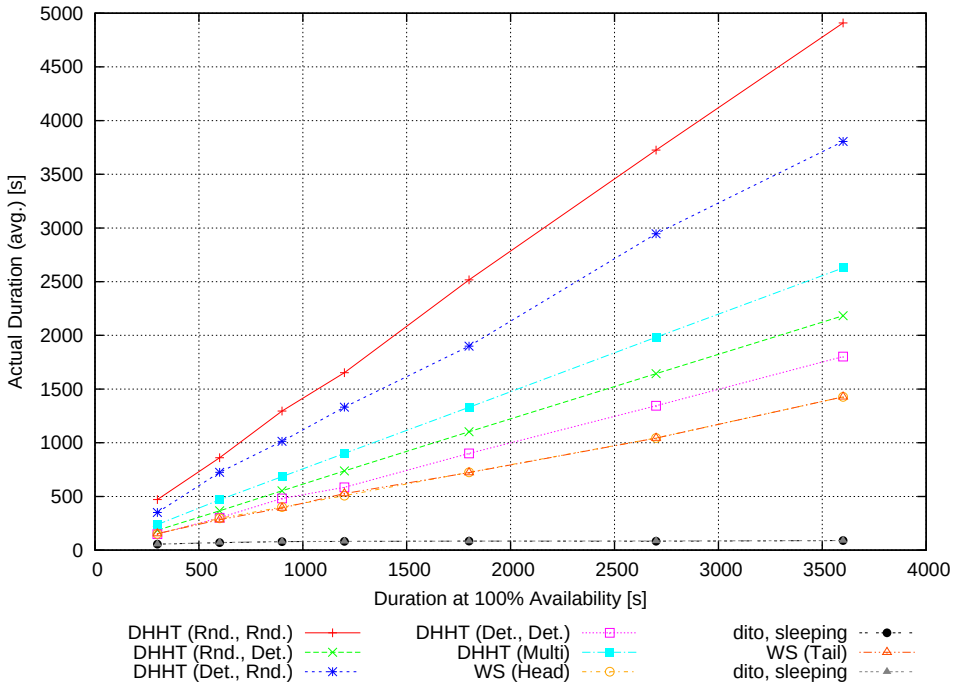

Figure A.73: *Actual average duration of processes in experiment 30.*

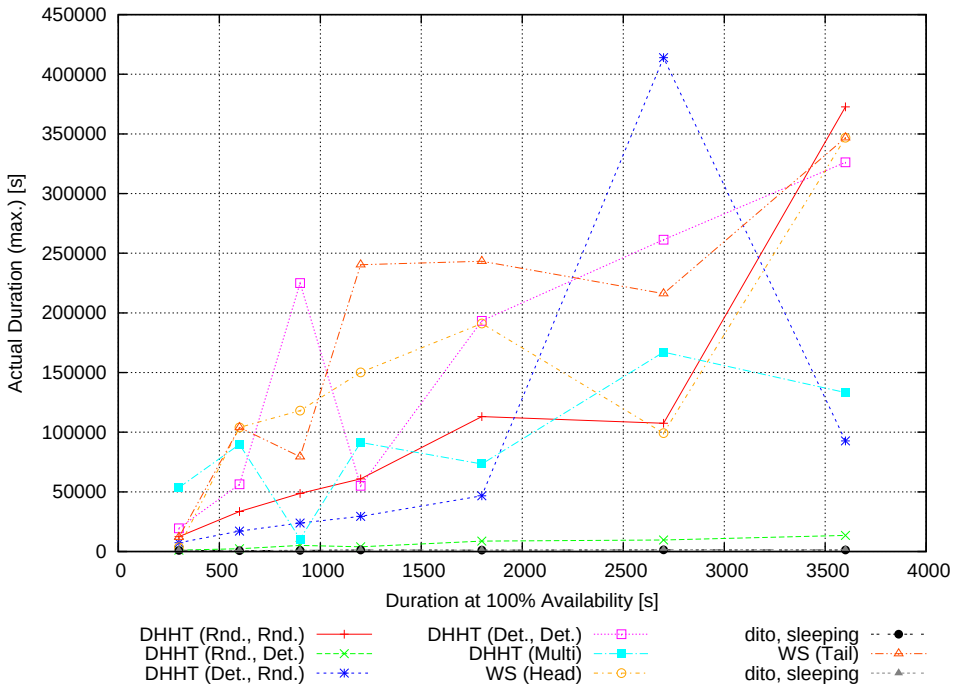

Figure A.74: *Actual maximum duration of processes in experiment 30.*

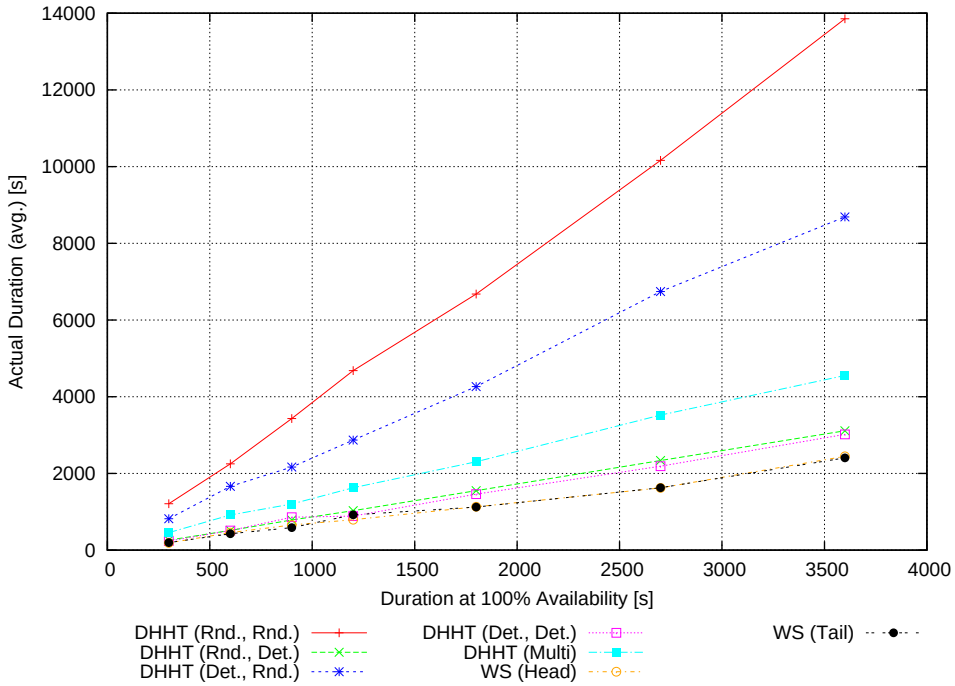

Figure A.75: *Actual average duration of jobs in experiment 30.*

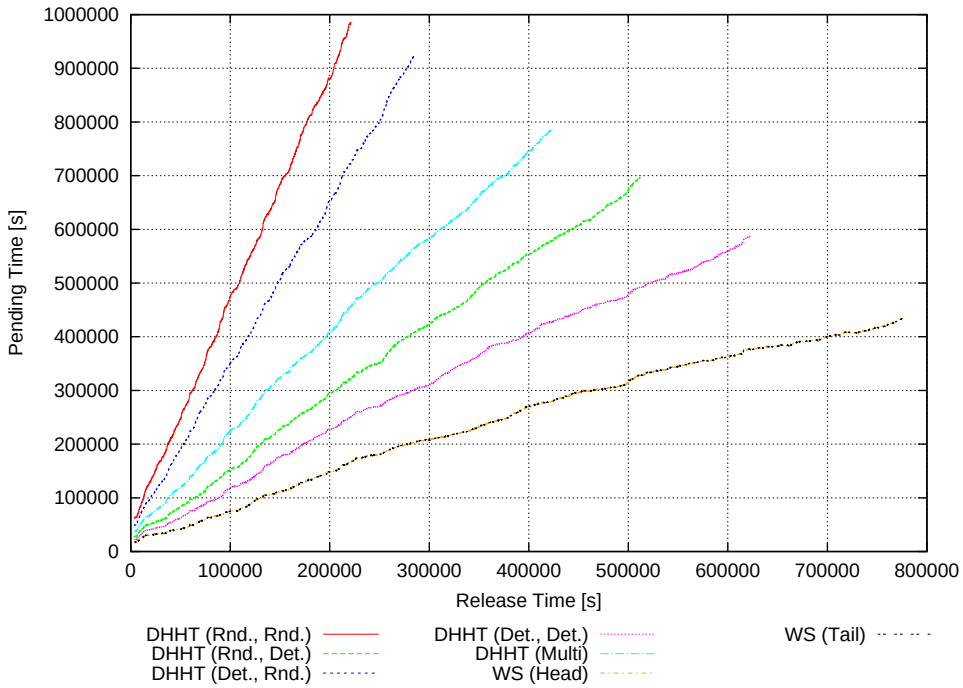

Figure A.76: *Pending time of jobs in experiment 30.*

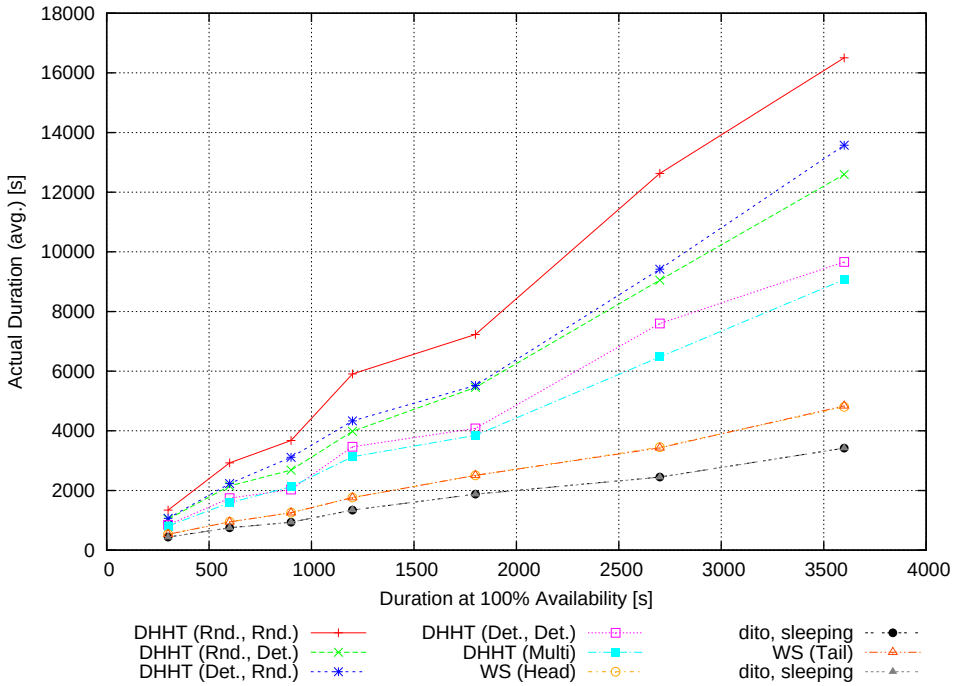

Figure A.77: *Actual average duration of processes in experiment 31.*

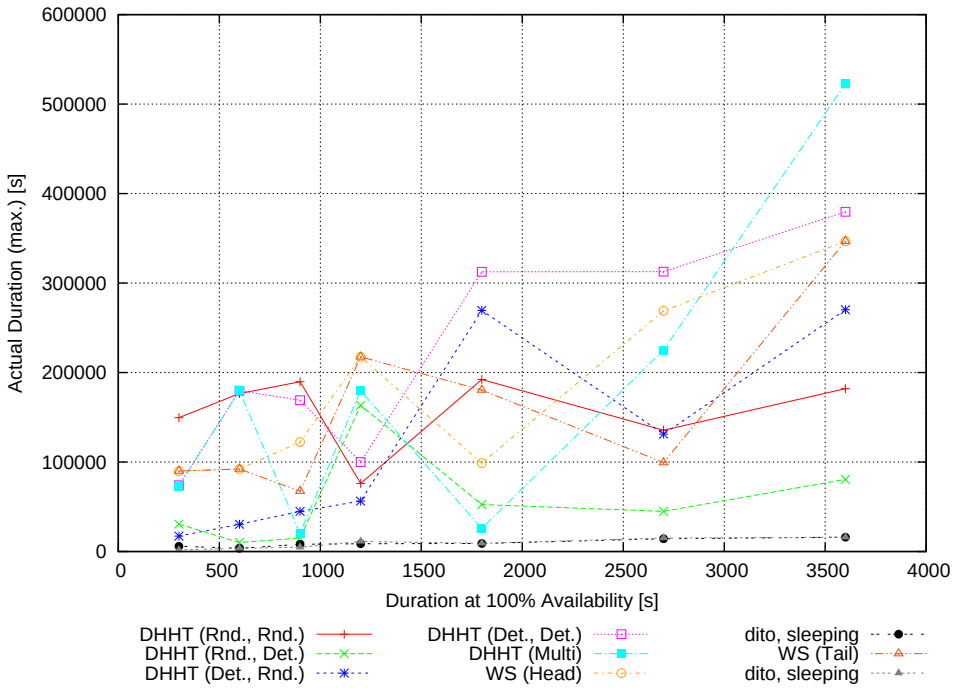

Figure A.78: *Actual maximum duration of processes in experiment 31.*

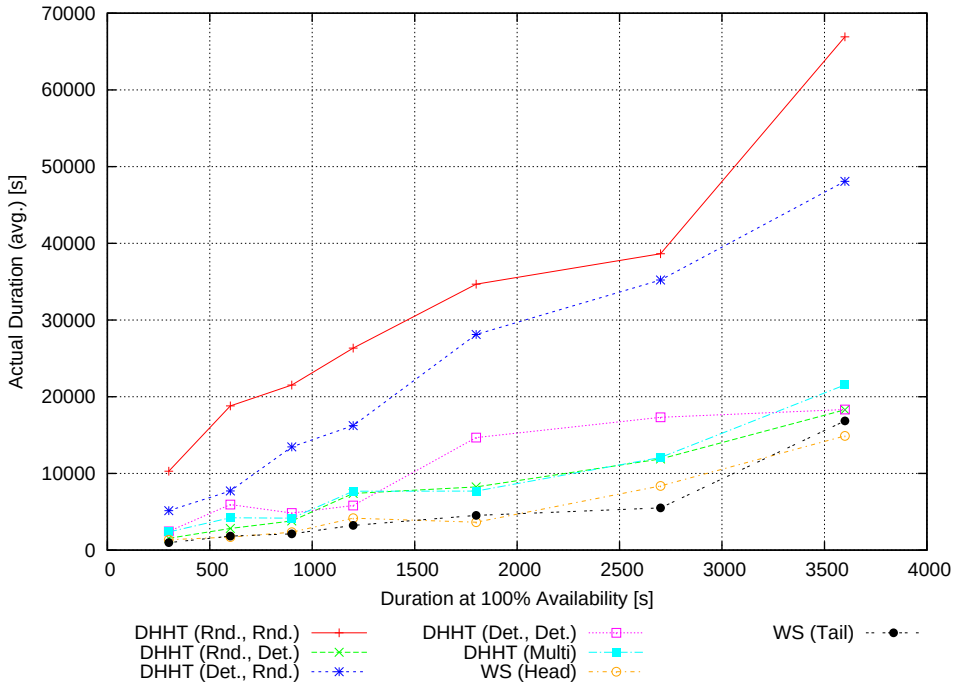

Figure A.79: *Actual average duration of jobs in experiment 31.*

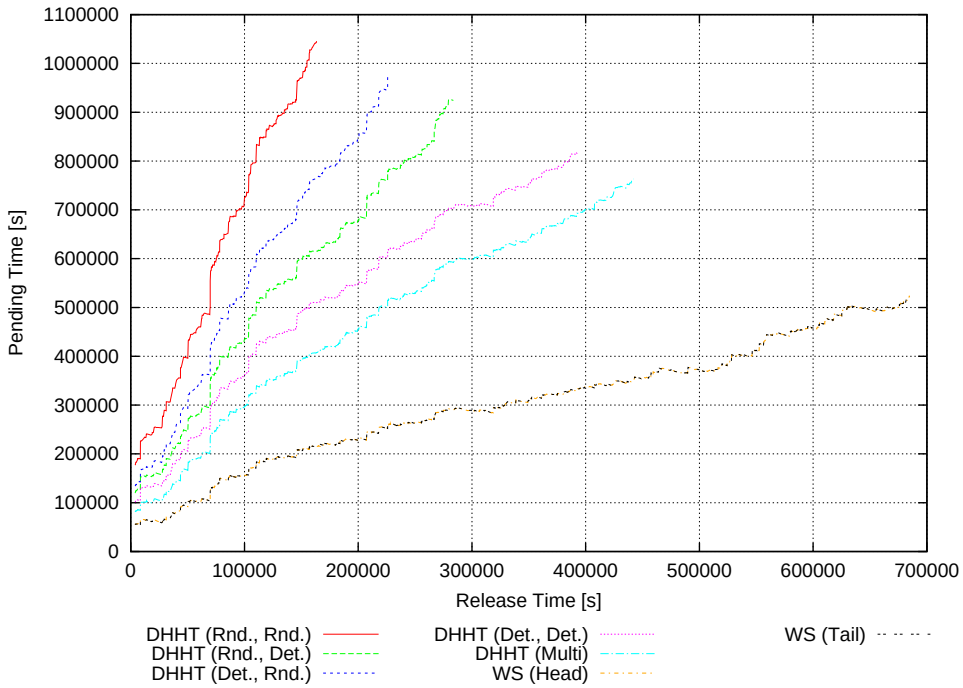

Figure A.80: *Pending time of jobs in experiment 31.*

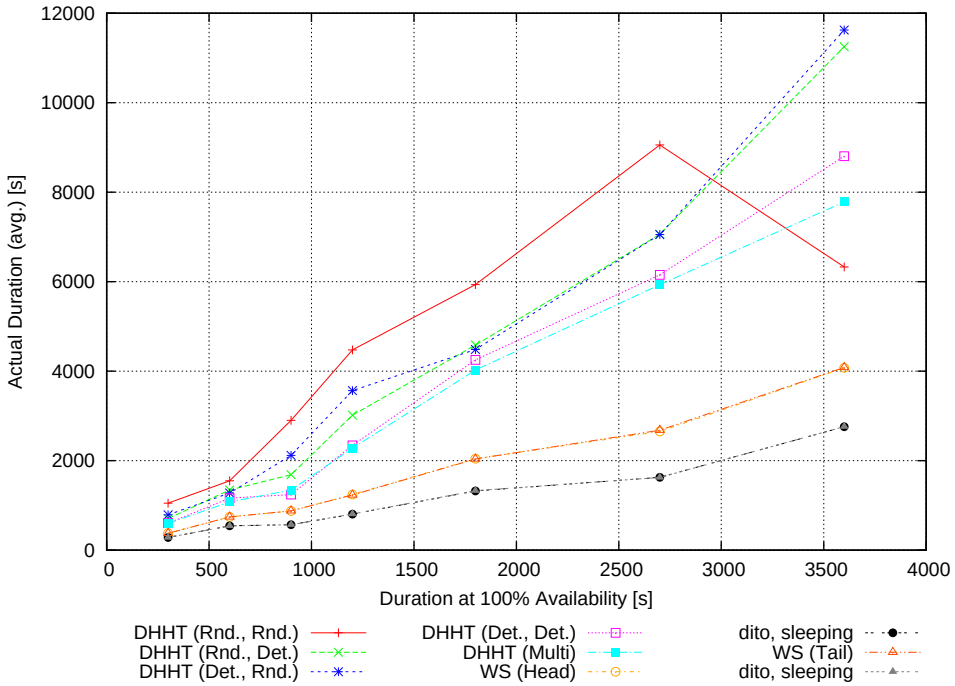

Figure A.81: *Actual average duration of processes in experiment 32.*

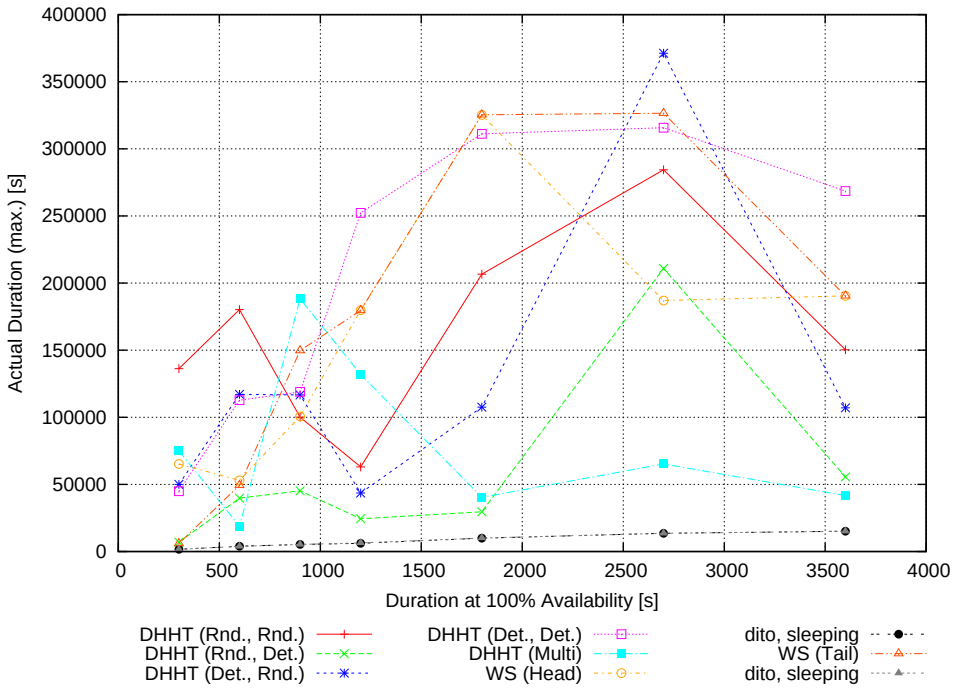

Figure A.82: *Actual maximum duration of processes in experiment 32.*

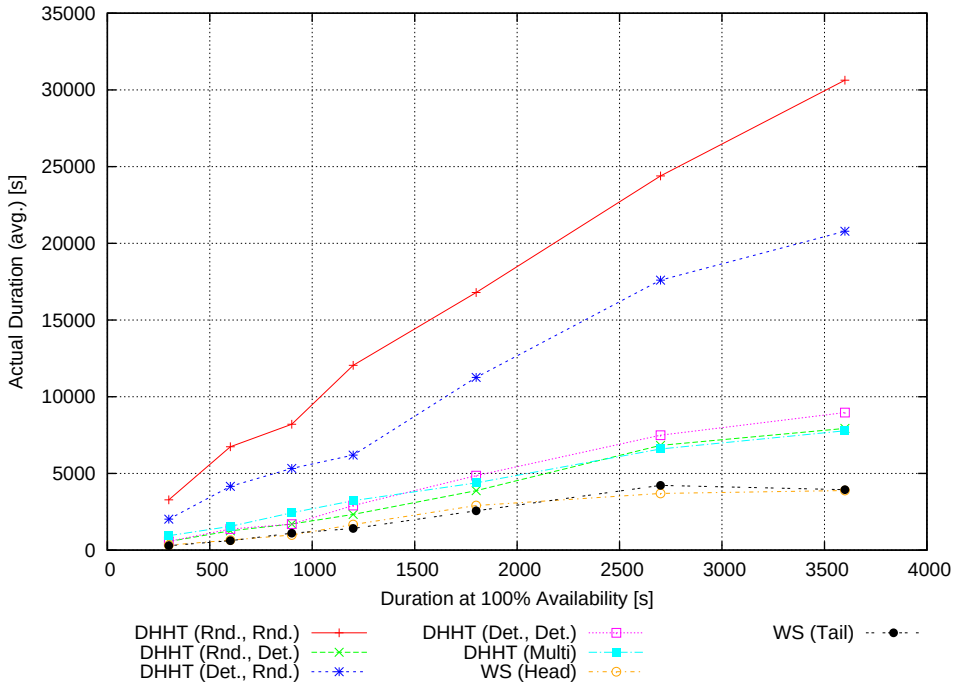

Figure A.83: *Actual average duration of jobs in experiment 32.*

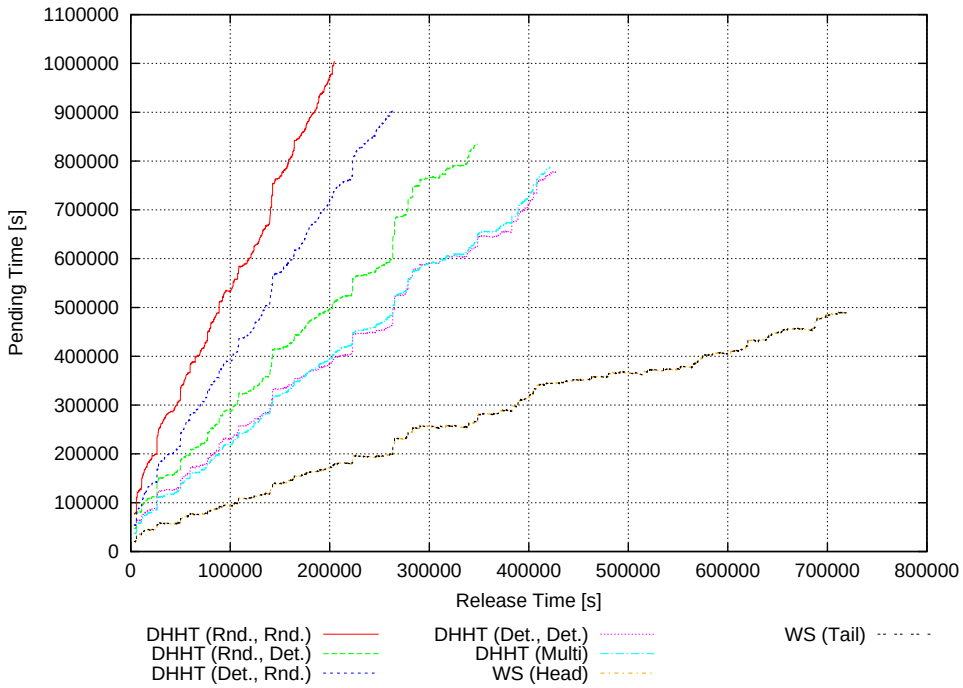

Figure A.84: *Pending time of jobs in experiment 32.*

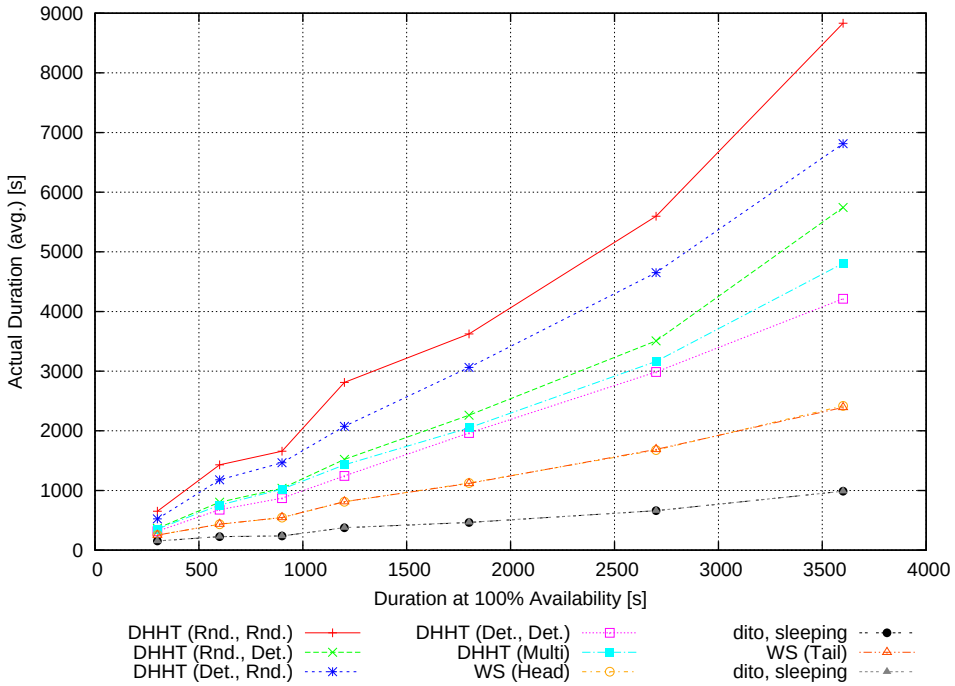

Figure A.85: *Actual average duration of processes in experiment 33.*

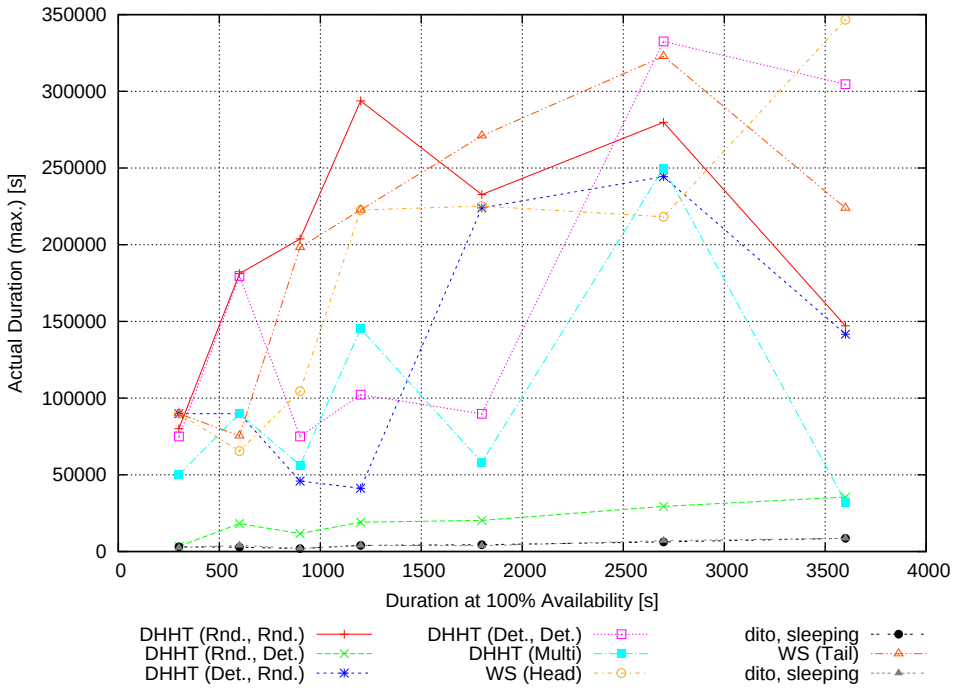

Figure A.86: *Actual maximum duration of processes in experiment 33.*

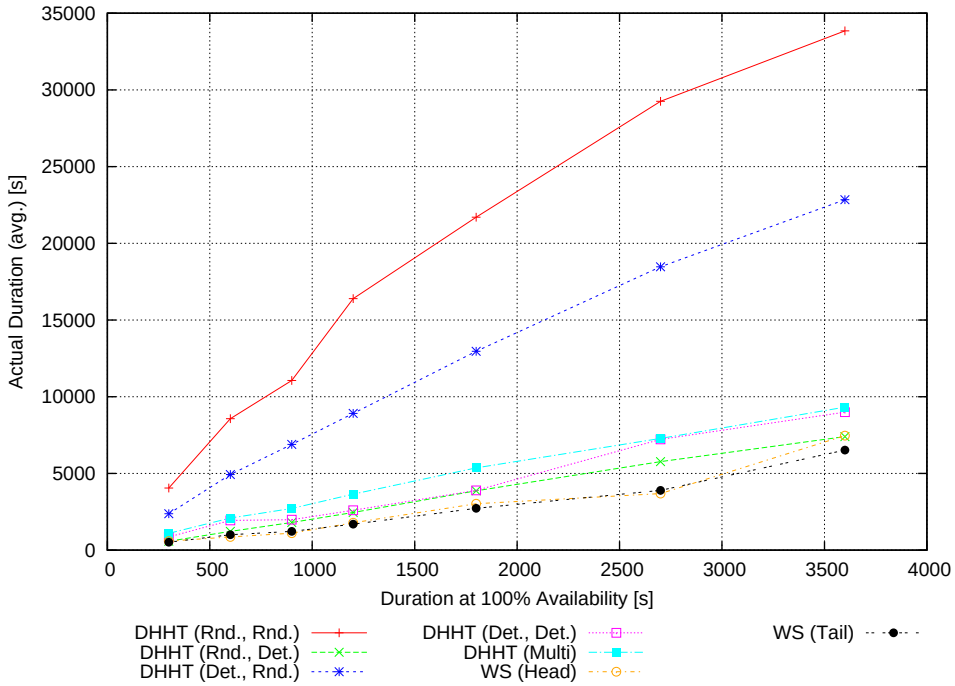

Figure A.87: *Actual average duration of jobs in experiment 33.*

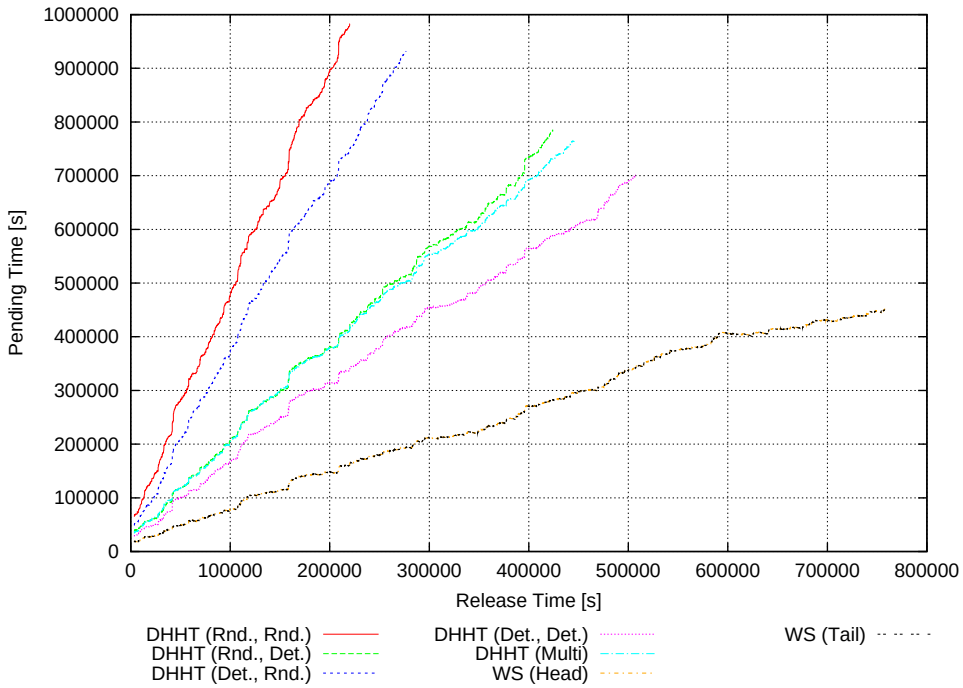

Figure A.88: *Pending time of jobs in experiment 33.*

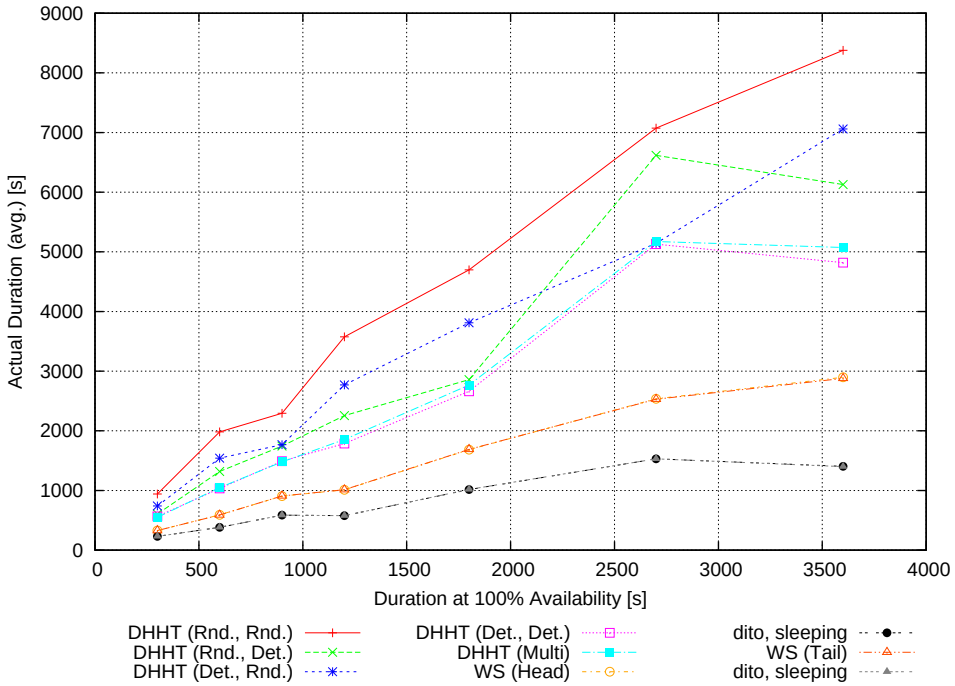

Figure A.89: *Actual average duration of processes in experiment 34.*

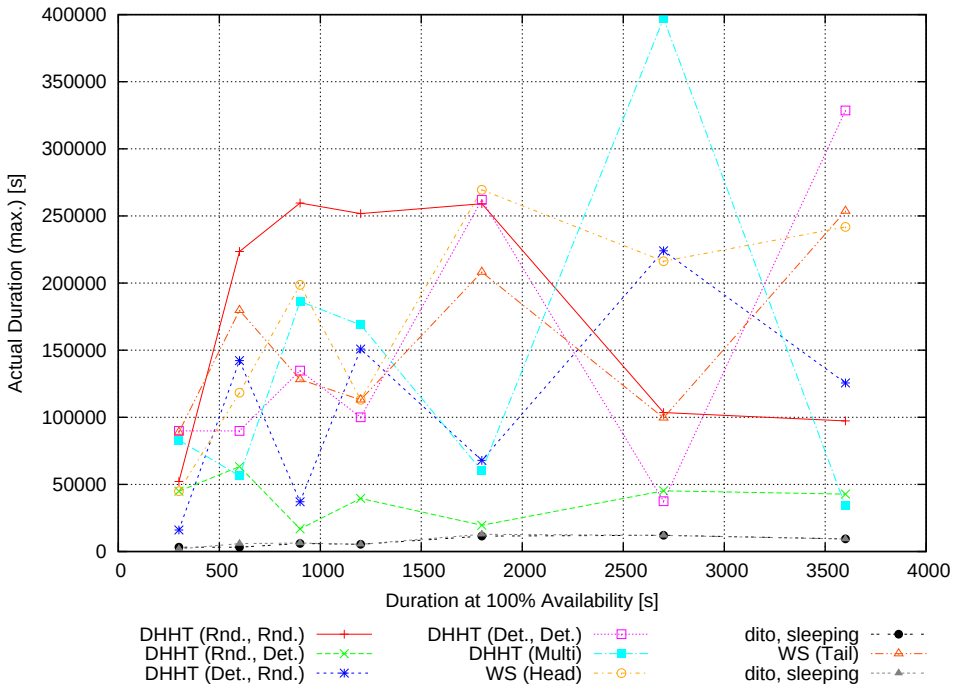

Figure A.90: *Actual maximum duration of processes in experiment 34.*

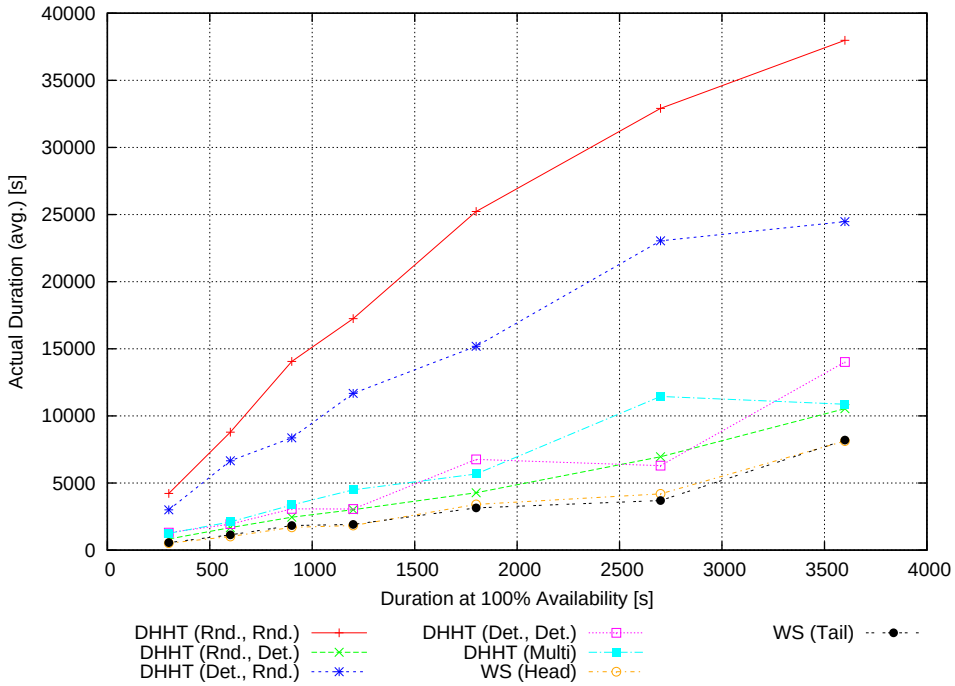

Figure A.91: *Actual average duration of jobs in experiment 34.*

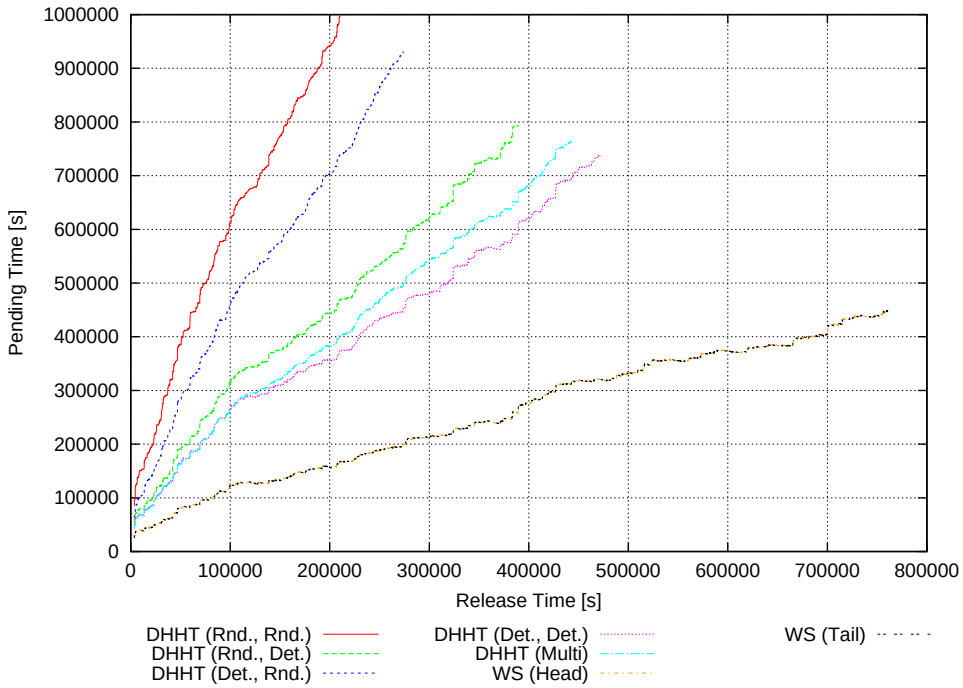

Figure A.92: *Pending time of jobs in experiment 34.*

A.4 Type C, Overload

In this section, we present the plots for the experiments of type C with an overload of factor 4.

A.4.1 Notebooks

This subsection contains the plots for the input profiles HPC2N (figures A.93 – A.96), LLNL Atlas (figures A.97 – A.100), LLNL Thunder (figures A.101 – A.104), SDSC BLUE (figures A.105 – A.108), and SDSC DataStar (figures A.109 – A.112) for the experiments of type C with an overload of factor 4 and using the notebook load profile.

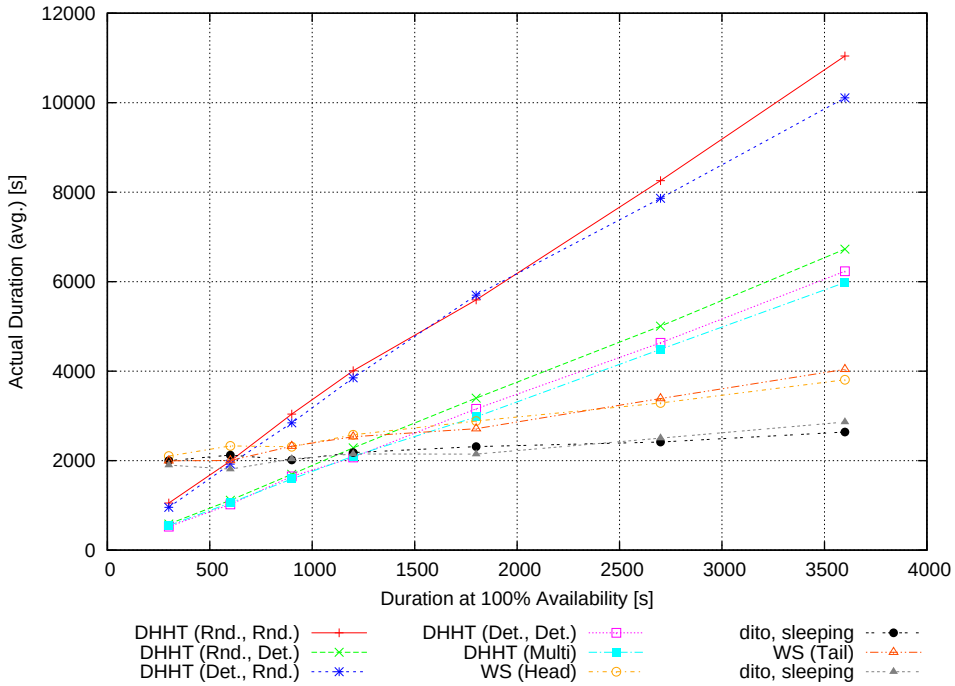

Figure A.93: *Actual average duration of processes in experiment 35.*

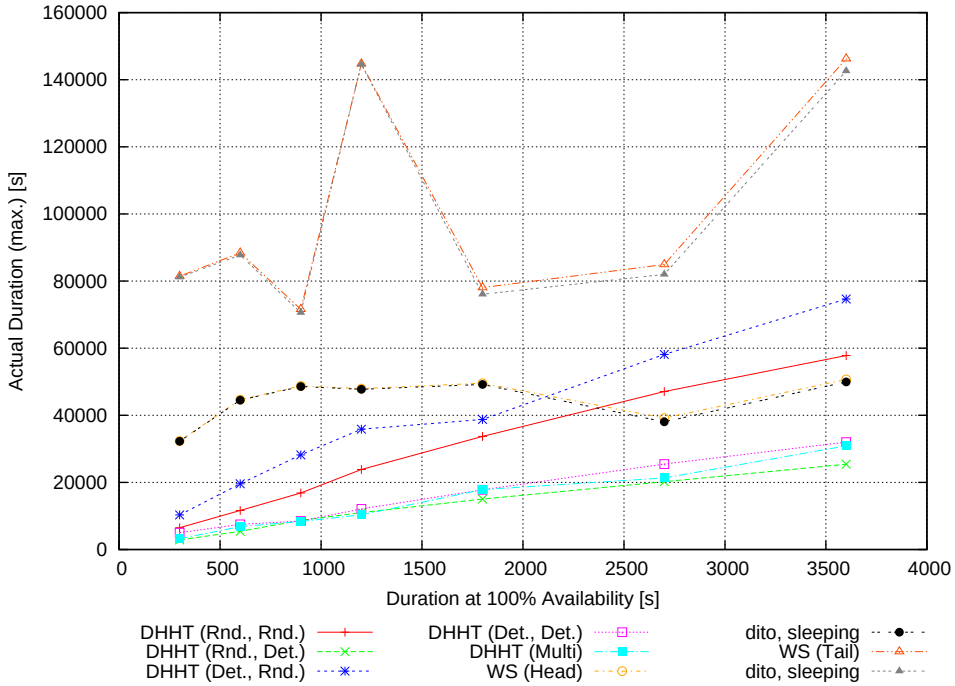

Figure A.94: *Actual maximum duration of processes in experiment 35.*

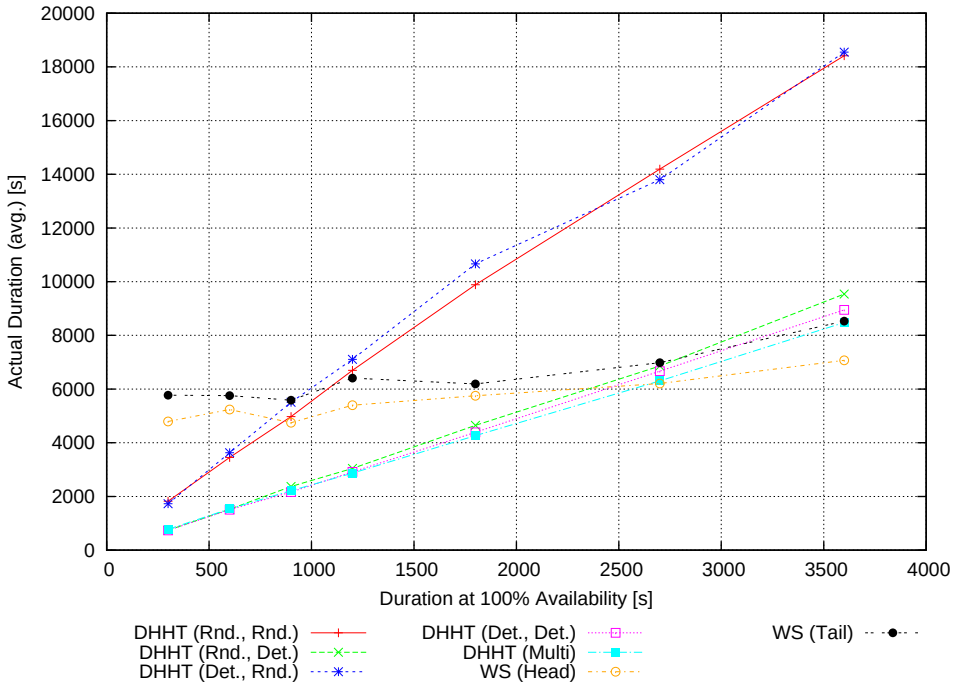

Figure A.95: *Actual average duration of jobs in experiment 35.*

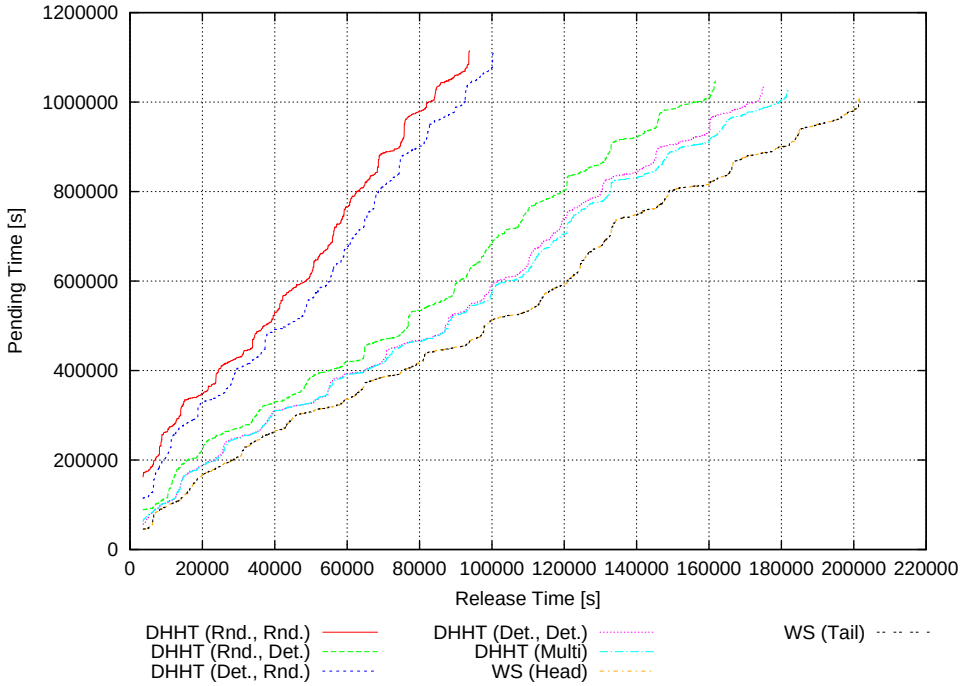

Figure A.96: *Pending time of jobs in experiment 35.*

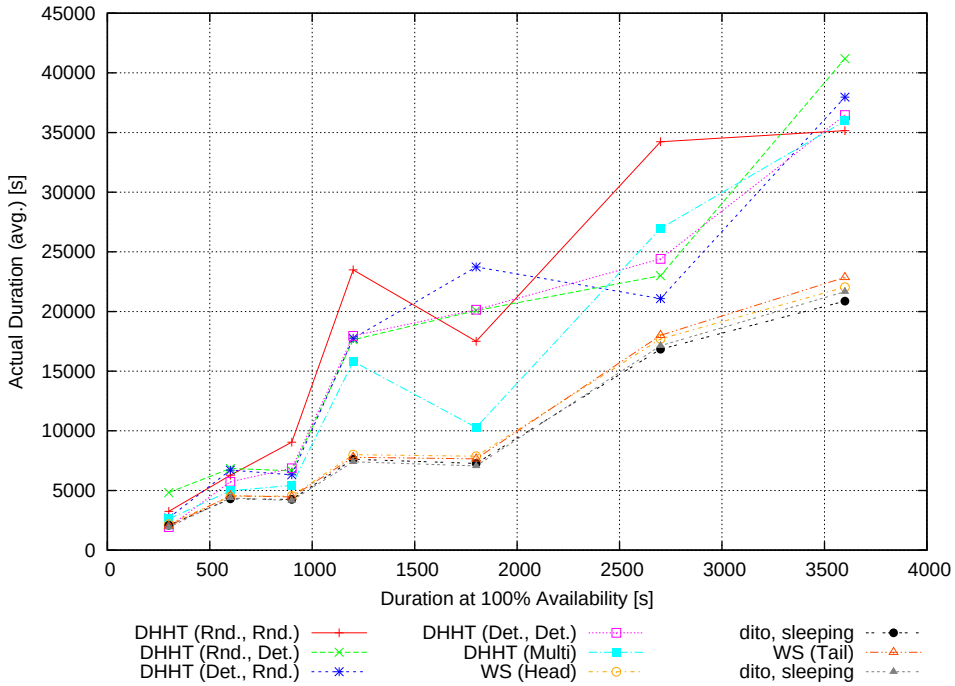

Figure A.97: *Actual average duration of processes in experiment 36.*

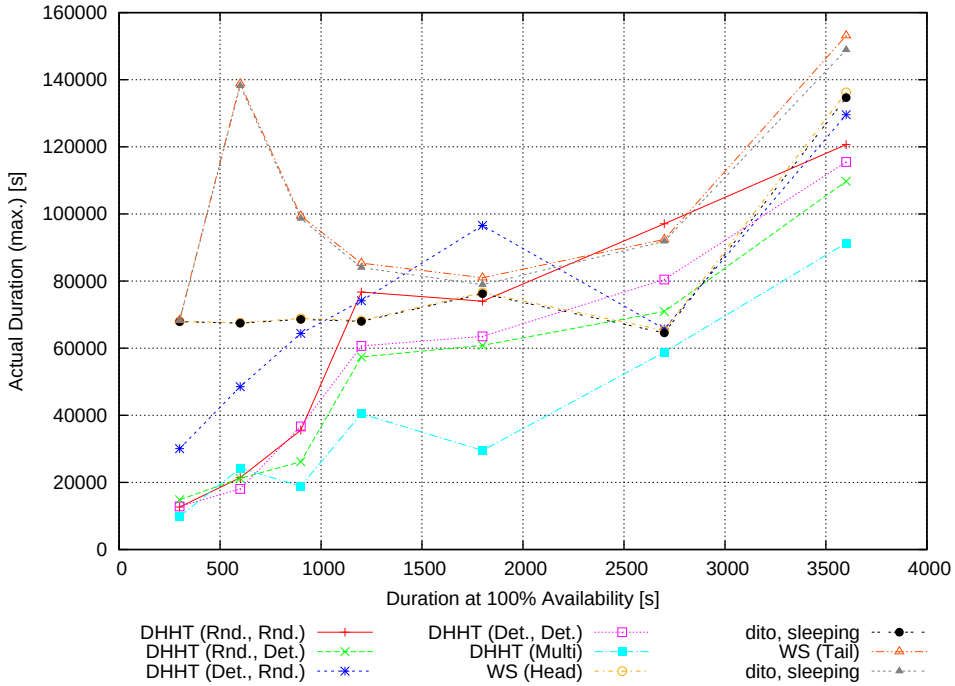

Figure A.98: *Actual maximum duration of processes in experiment 36.*

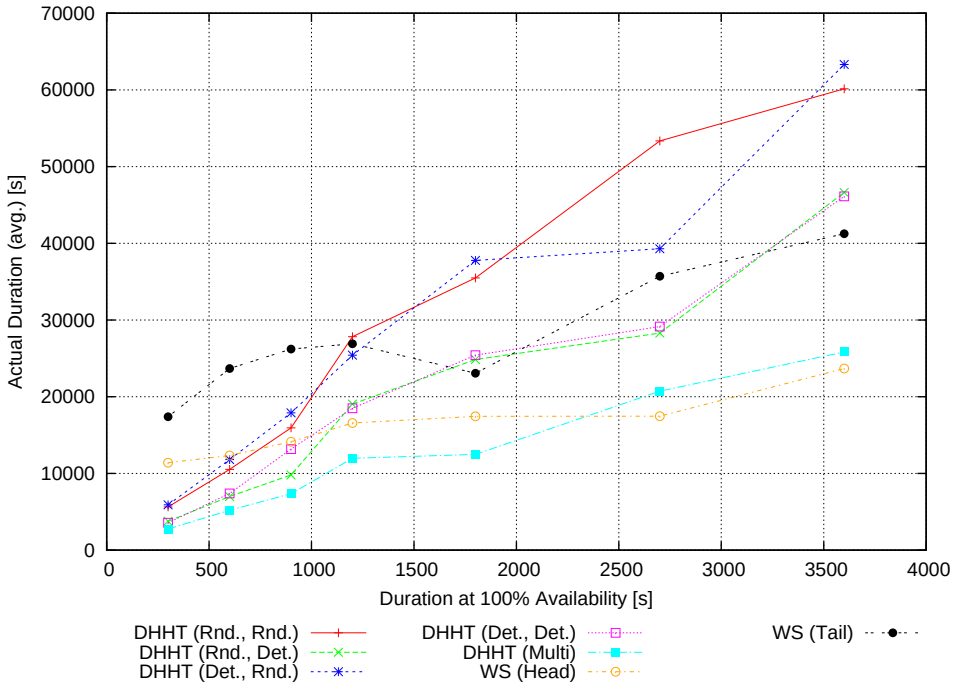

Figure A.99: *Actual average duration of jobs in experiment 36.*

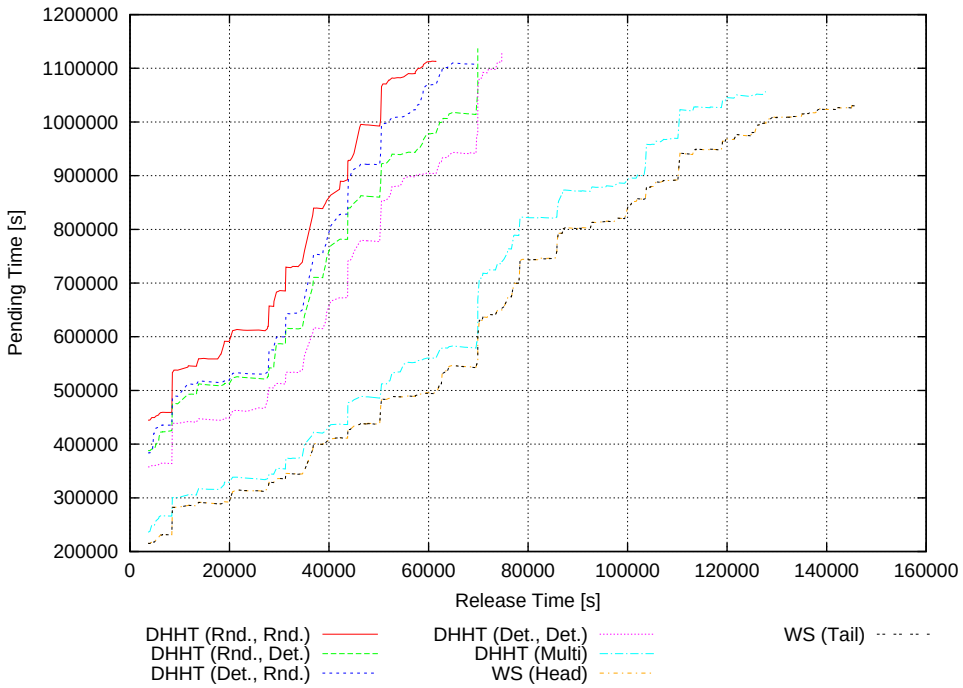

Figure A.100: *Pending time of jobs in experiment 36.*

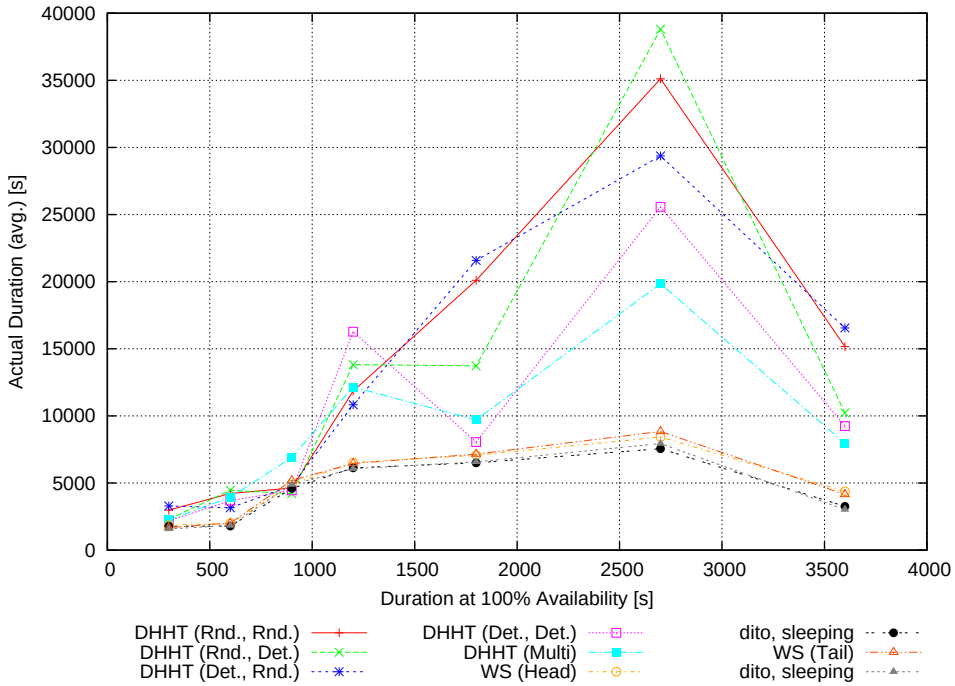

Figure A.101: *Actual average duration of processes in experiment 37.*

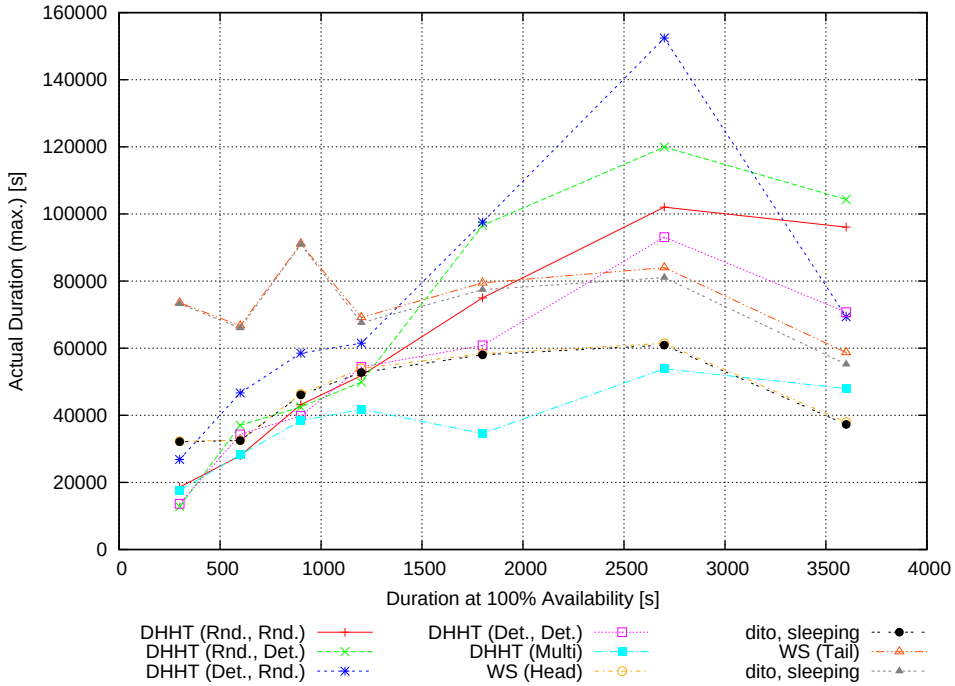

Figure A.102: *Actual maximum duration of processes in experiment 37.*

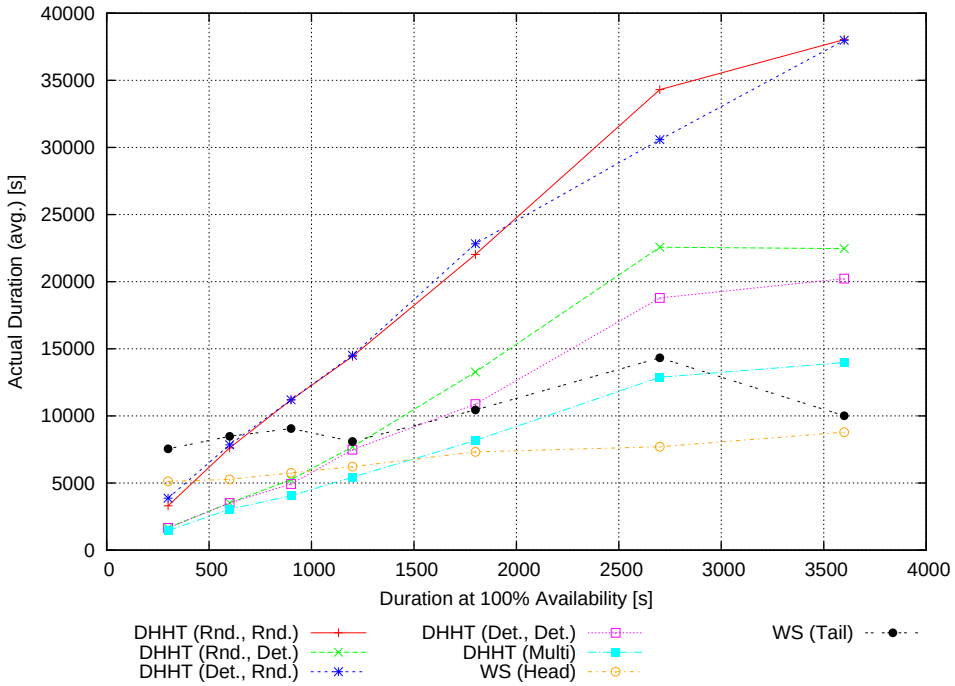

Figure A.103: *Actual average duration of jobs in experiment 37.*

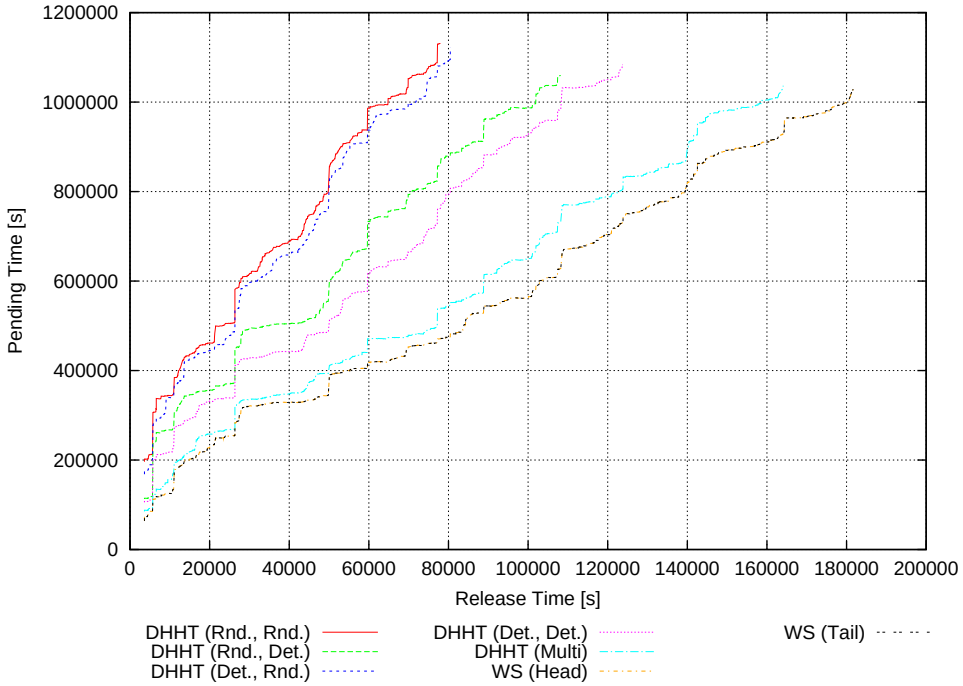

Figure A.104: *Pending time of jobs in experiment 37.*

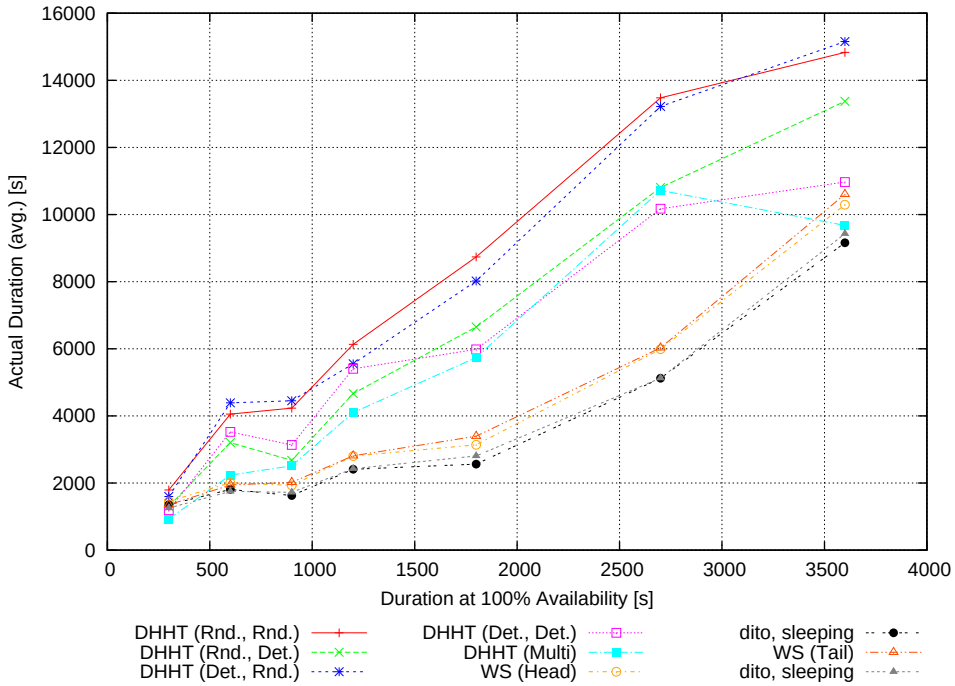

Figure A.105: *Actual average duration of processes in experiment 38.*

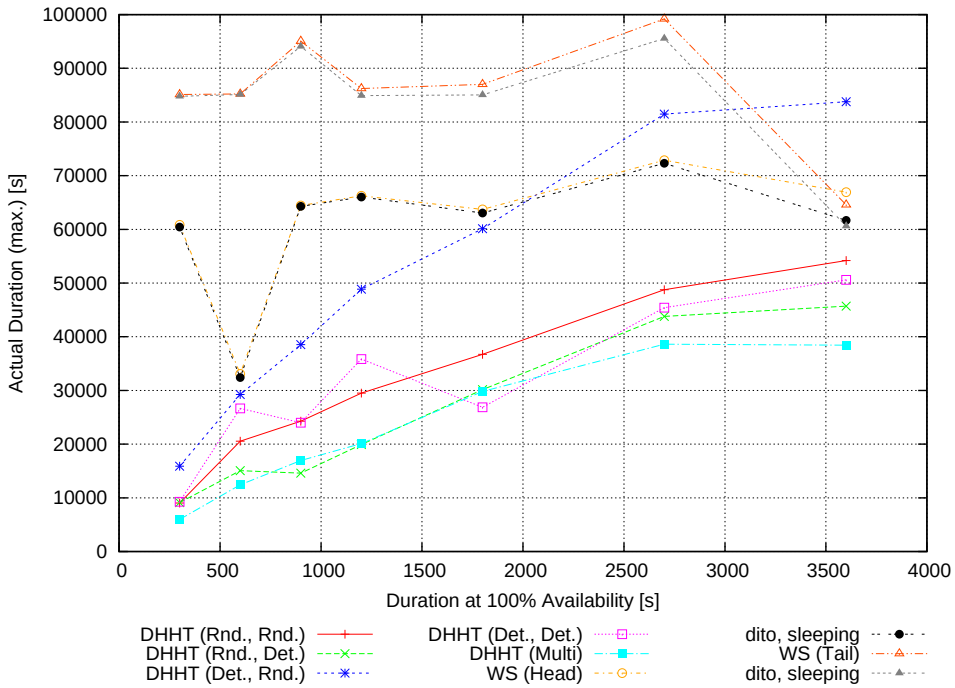

Figure A.106: *Actual maximum duration of processes in experiment 38.*

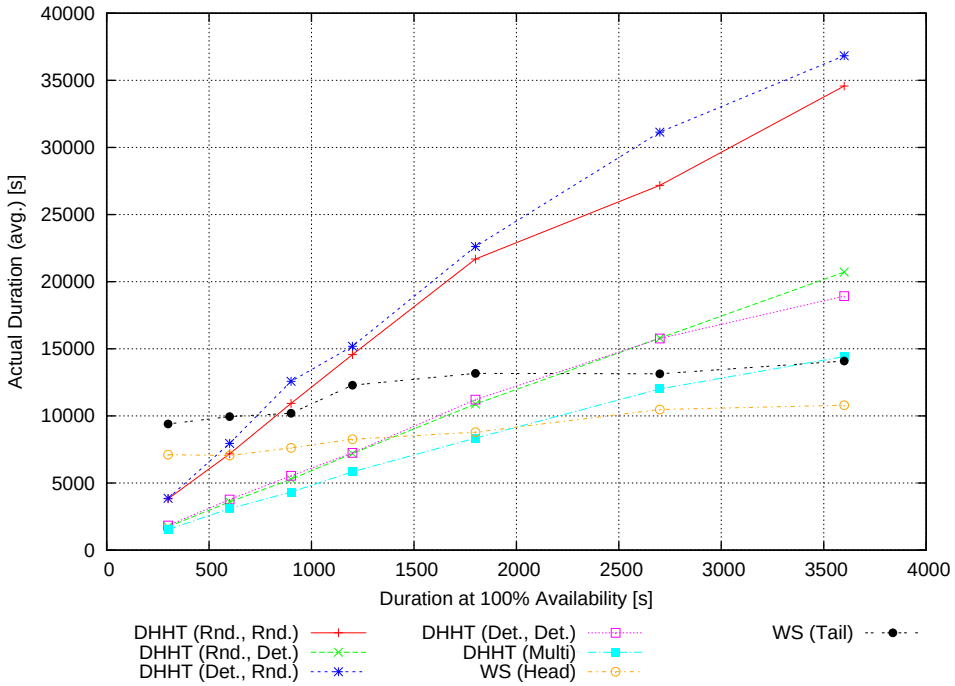

Figure A.107: *Actual average duration of jobs in experiment 38.*

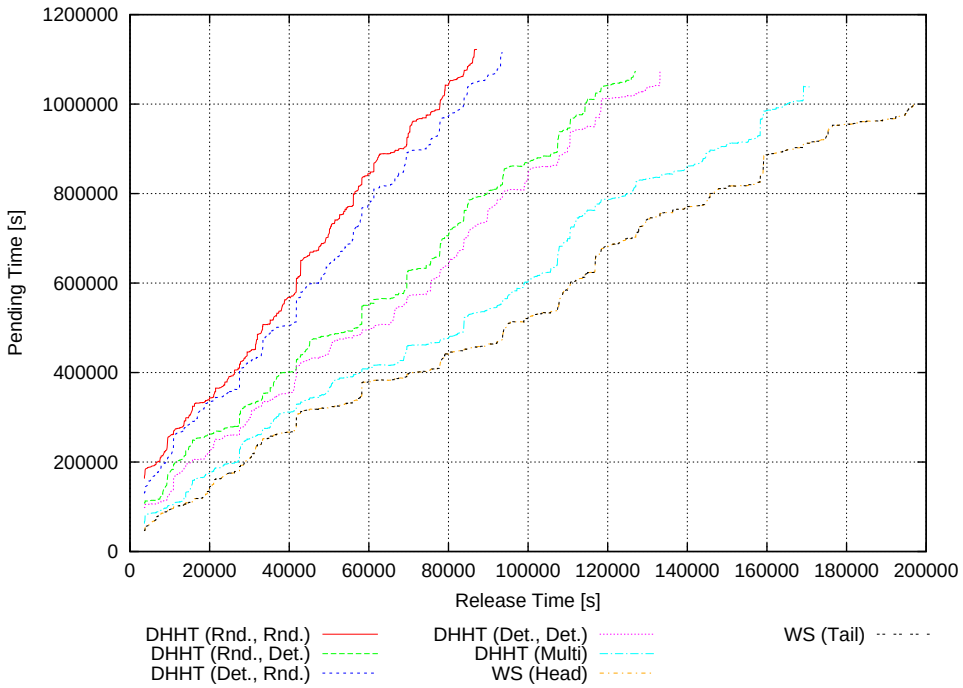

Figure A.108: *Pending time of jobs in experiment 38.*

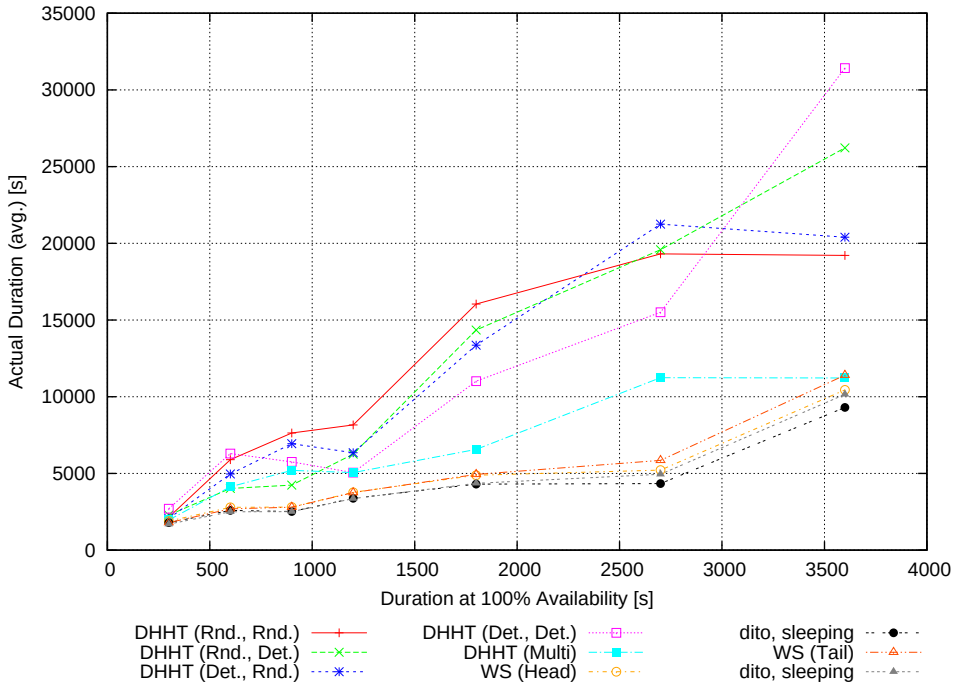

Figure A.109: *Actual average duration of processes in experiment 39.*

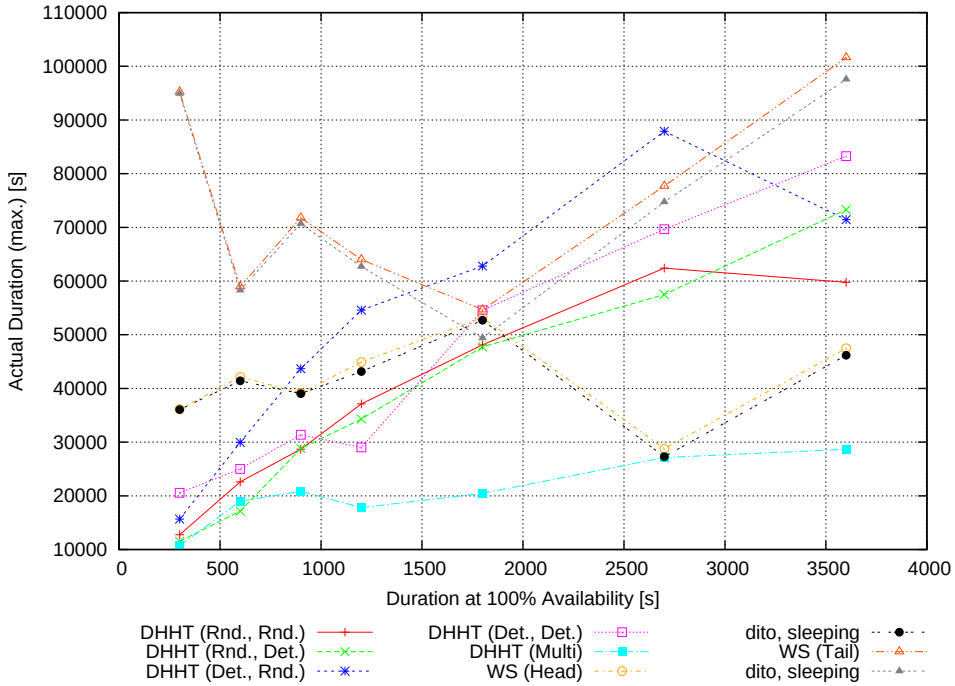

Figure A.110: *Actual maximum duration of processes in experiment 39.*

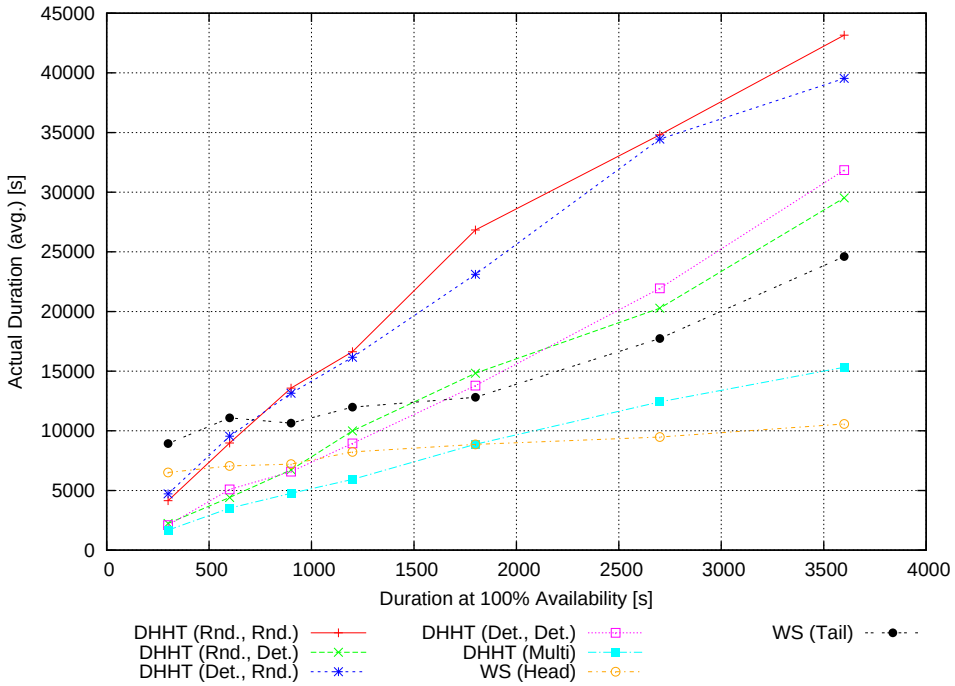

Figure A.111: *Actual average duration of jobs in experiment 39.*

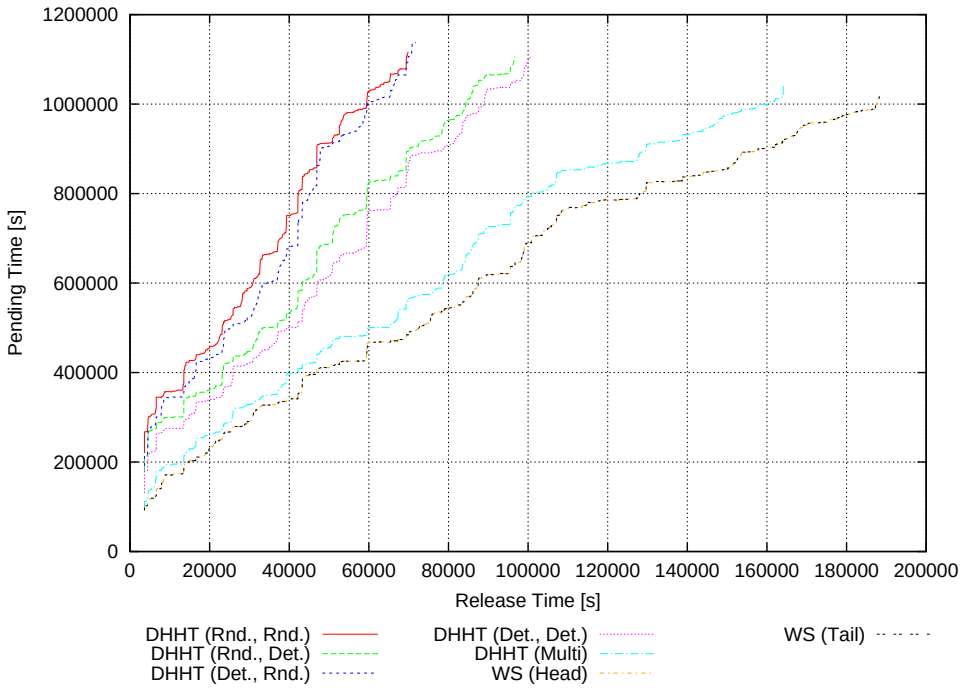

Figure A.112: *Pending time of jobs in experiment 39.*

A.4.2 Office PCs

This subsection contains the plots for the input profiles HPC2N (figures A.113 – A.116), LLNL Atlas (figures A.117 – A.120), LLNL Thunder (figures A.121 – A.124), SDSC BLUE (figures A.125 – A.128), and SDSC DataStar (figures A.129 – A.132) for the experiments of type C with an overload of factor 4 and using the office PC load profile.

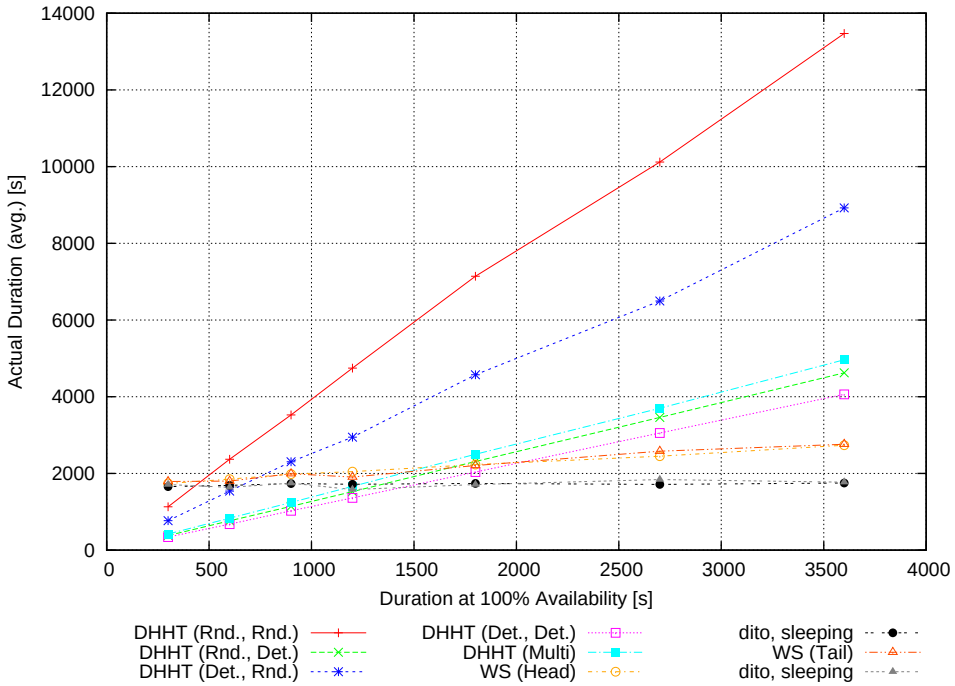

Figure A.113: *Actual average duration of processes in experiment 40.*

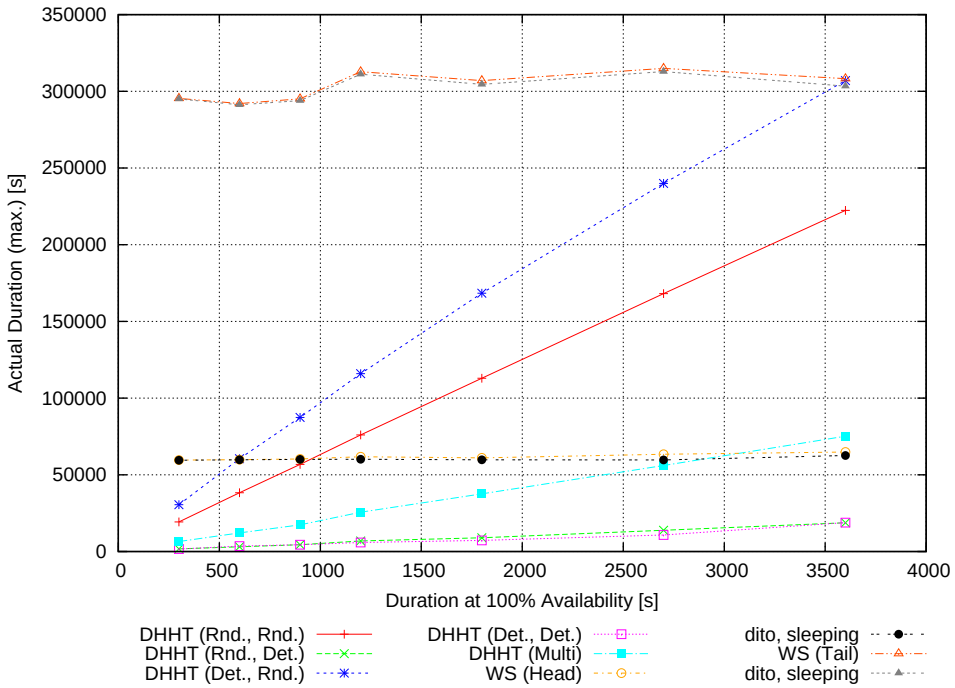

Figure A.114: *Actual maximum duration of processes in experiment 40.*

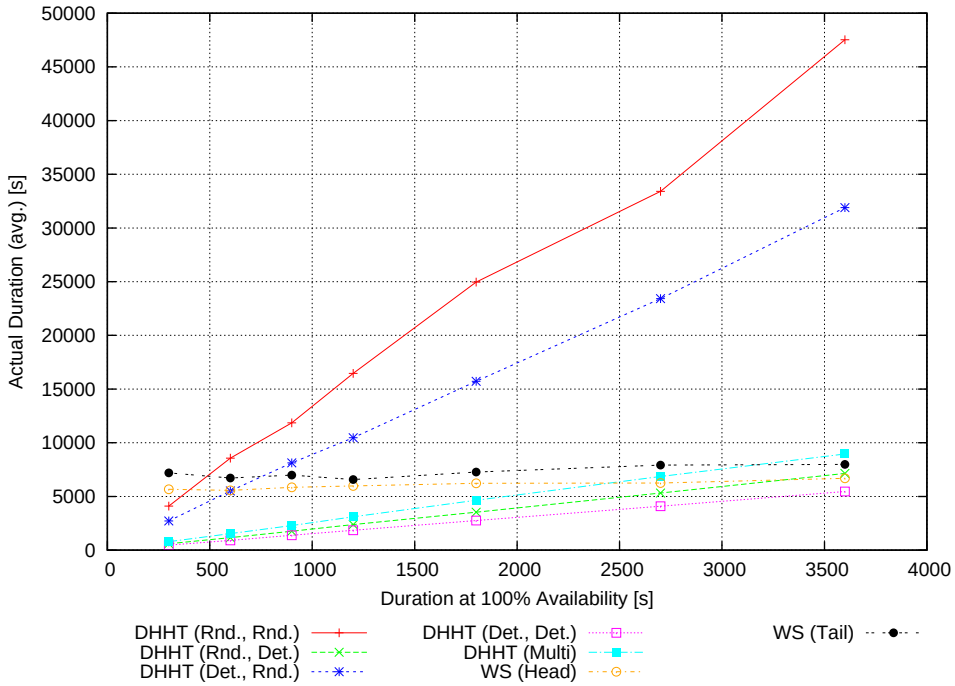

Figure A.115: *Actual average duration of jobs in experiment 40.*

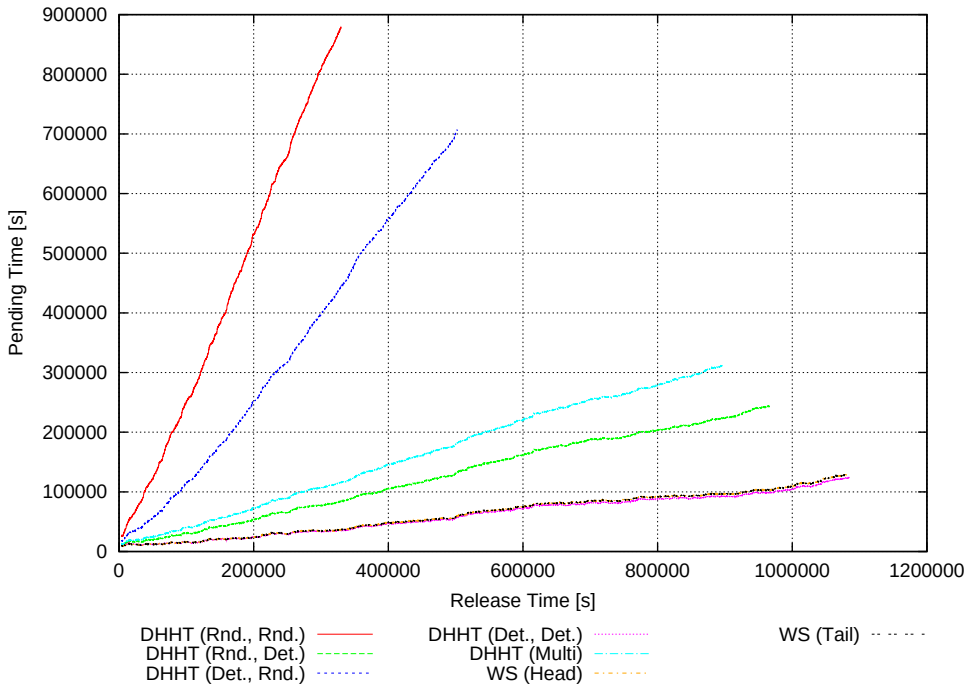

Figure A.116: *Pending time of jobs in experiment 40.*

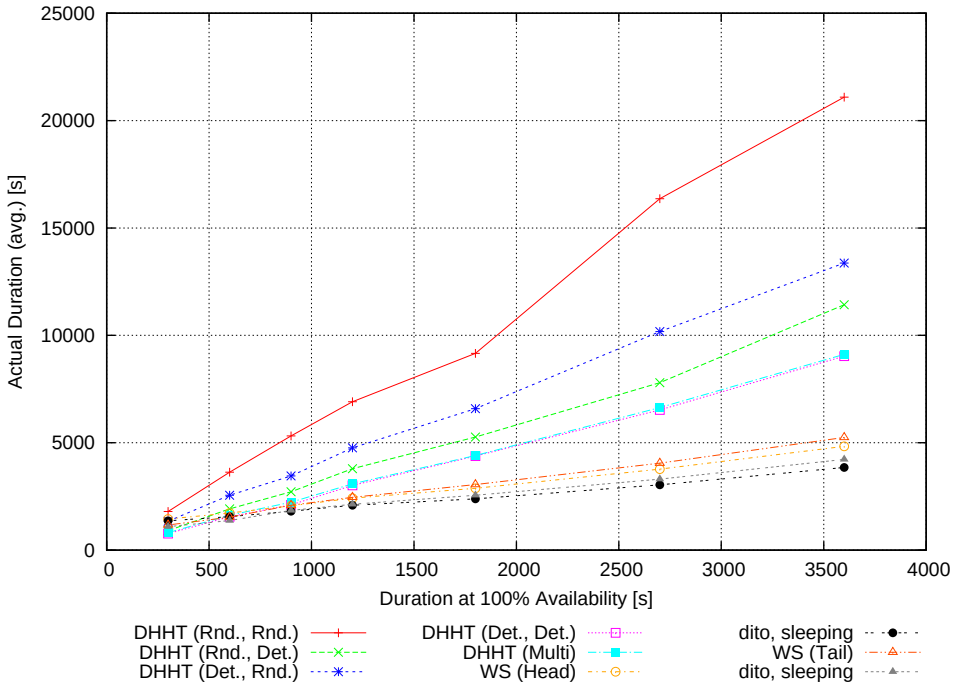

Figure A.117: *Actual average duration of processes in experiment 41.*

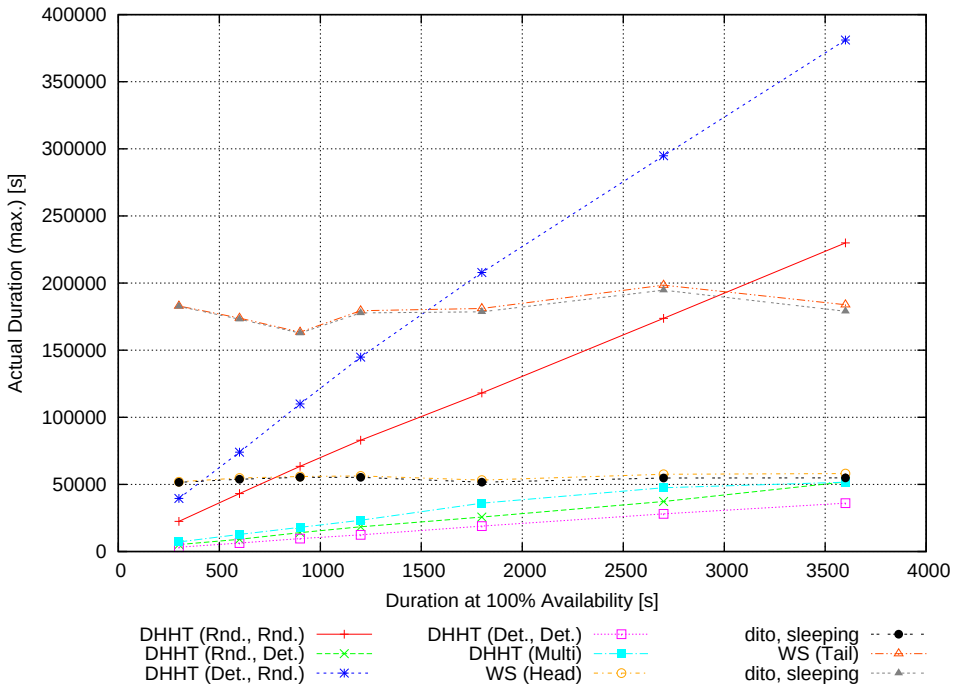

Figure A.118: *Actual maximum duration of processes in experiment 41.*

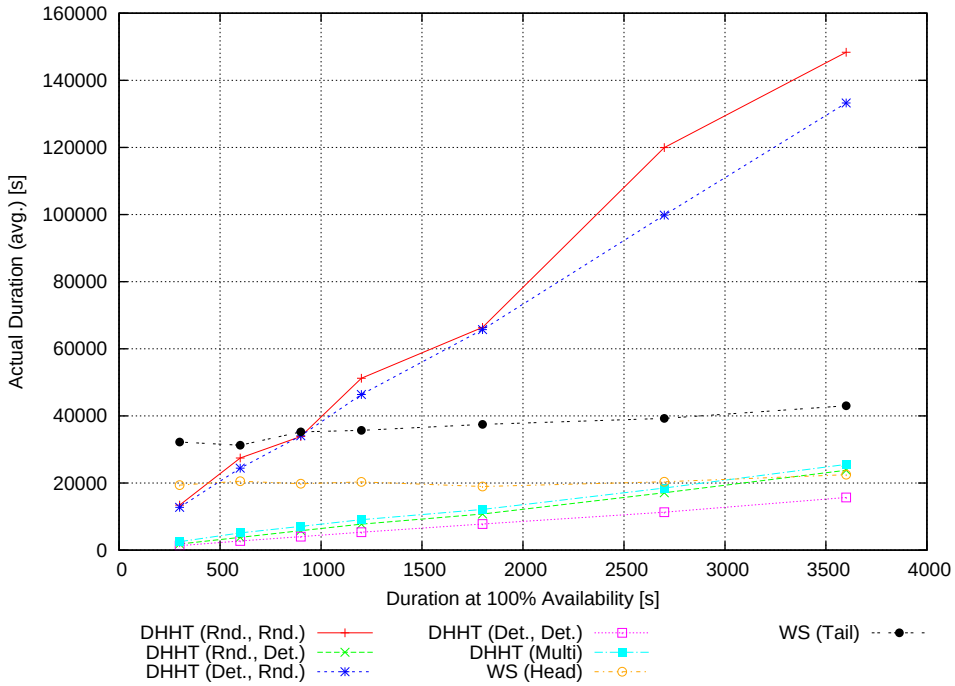

Figure A.119: *Actual average duration of jobs in experiment 41.*

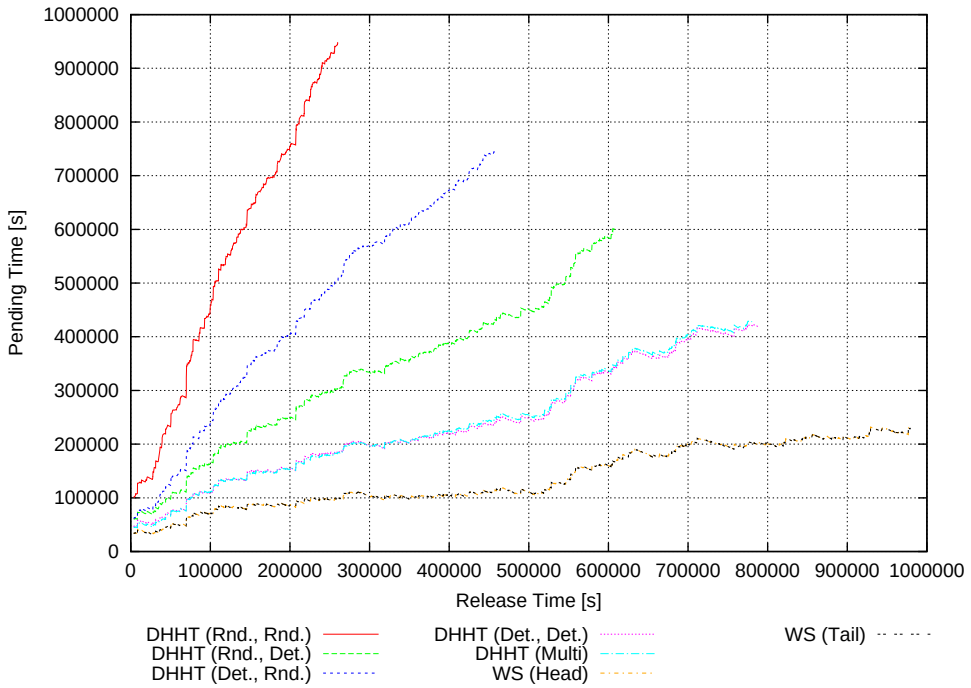

Figure A.120: *Pending time of jobs in experiment 41.*

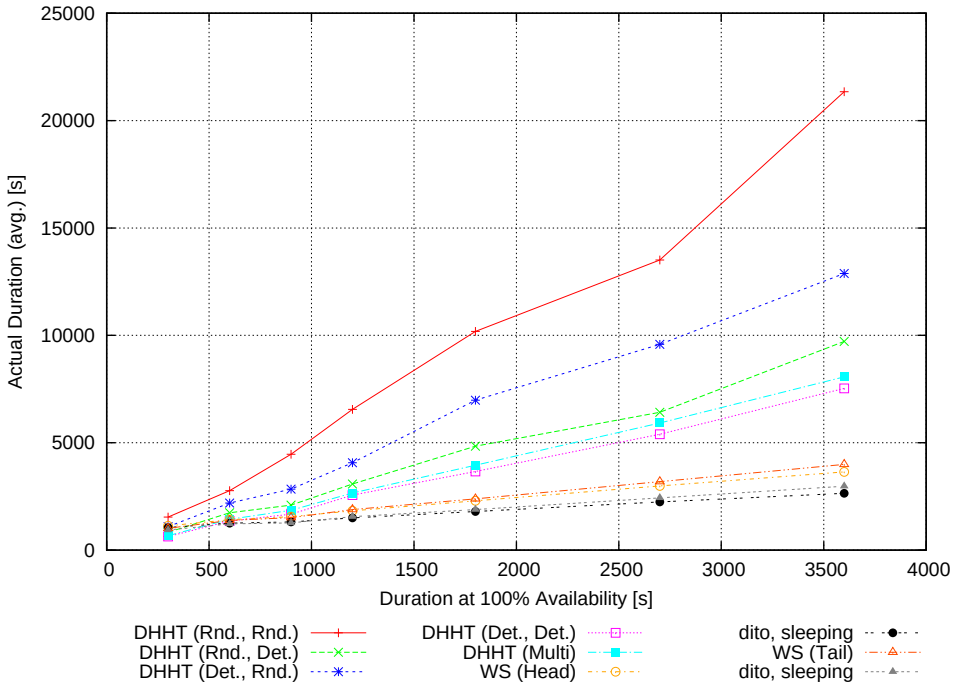

Figure A.121: *Actual average duration of processes in experiment 42.*

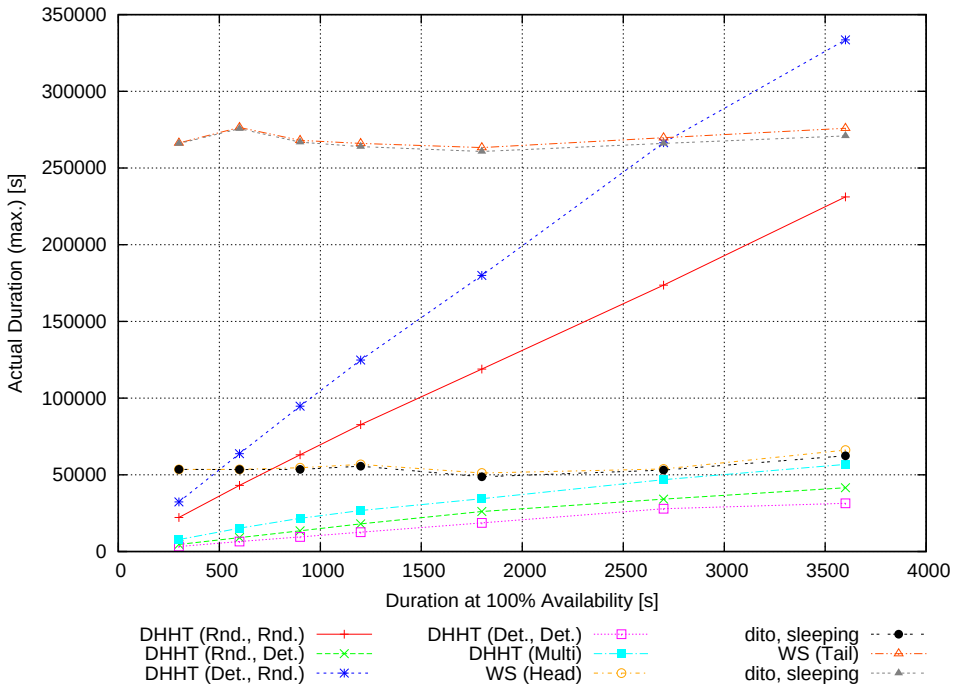

Figure A.122: *Actual maximum duration of processes in experiment 42.*

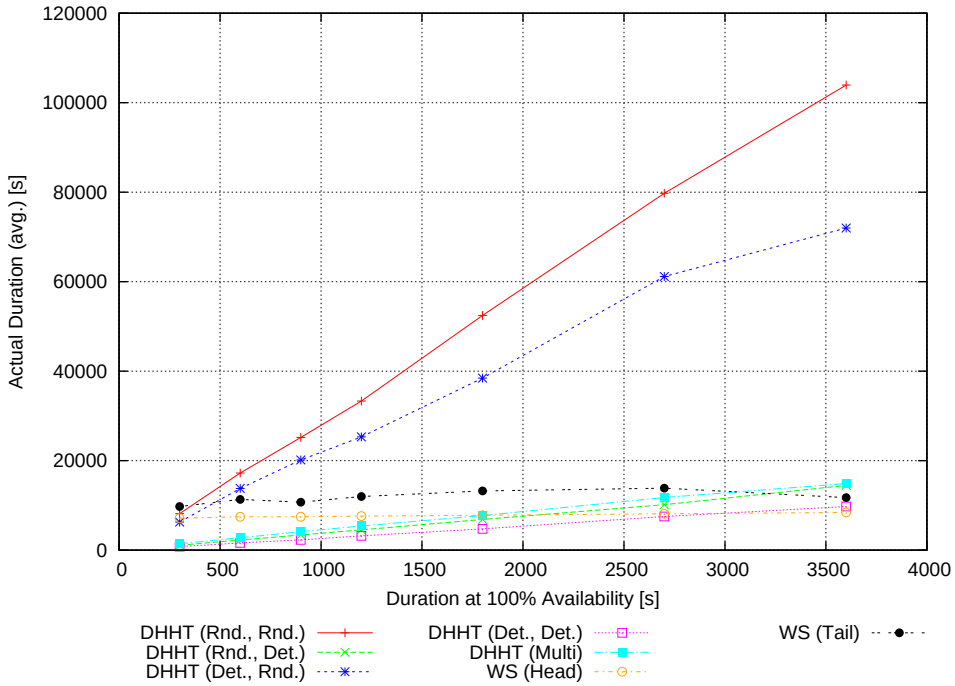

Figure A.123: *Actual average duration of jobs in experiment 42.*

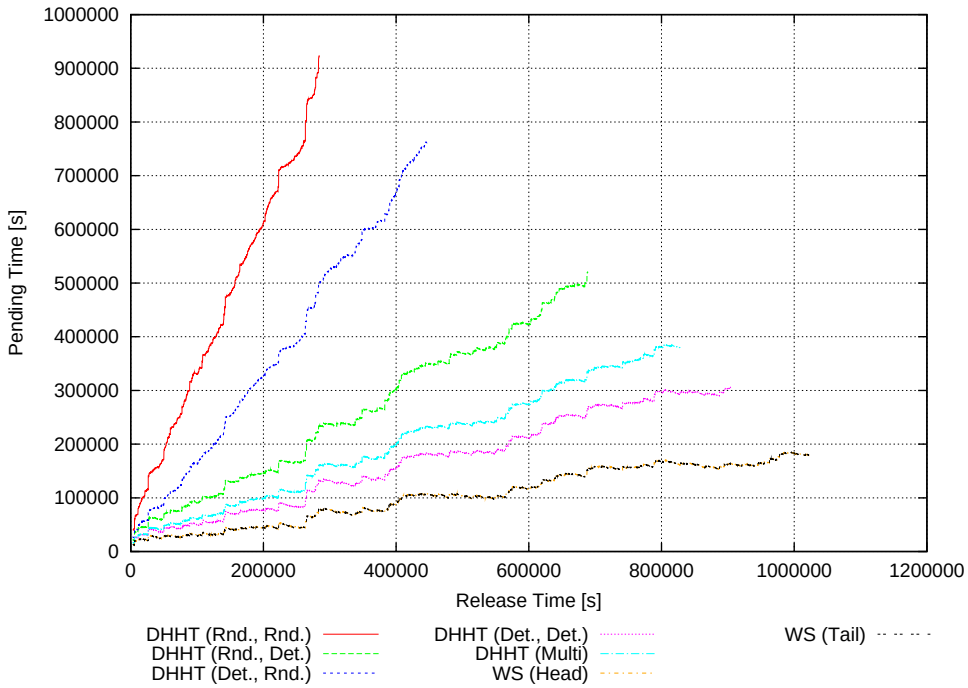

Figure A.124: *Pending time of jobs in experiment 42.*

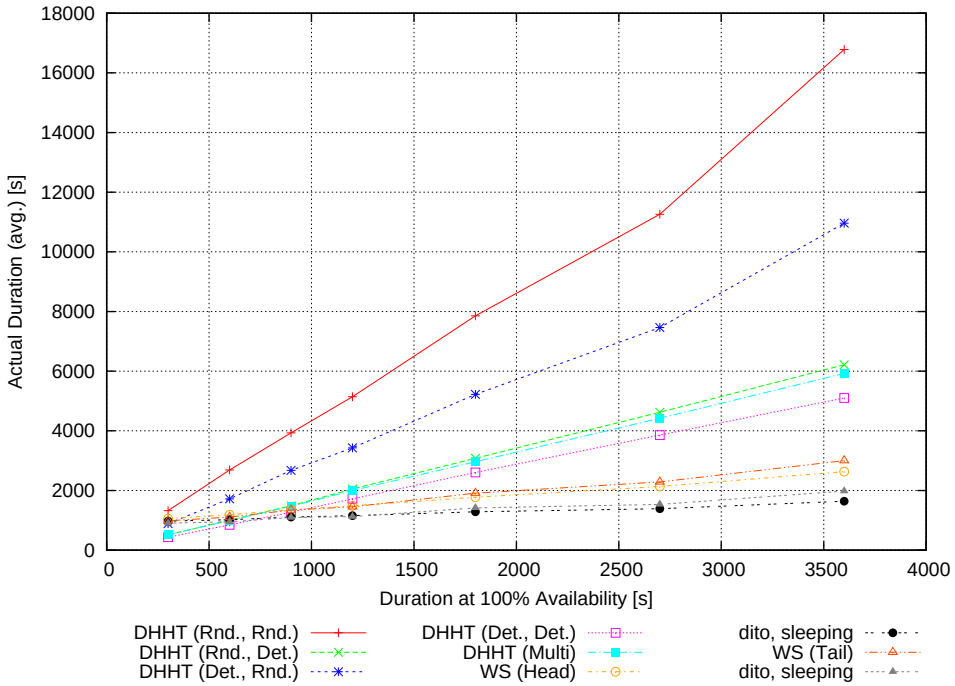

Figure A.125: *Actual average duration of processes in experiment 43.*

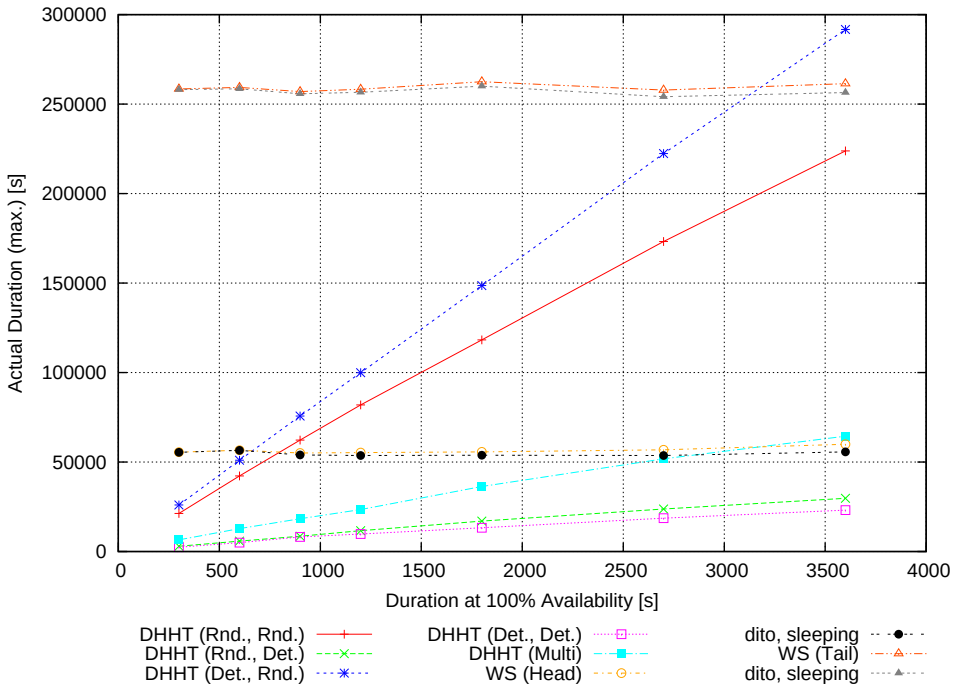

Figure A.126: *Actual maximum duration of processes in experiment 43.*

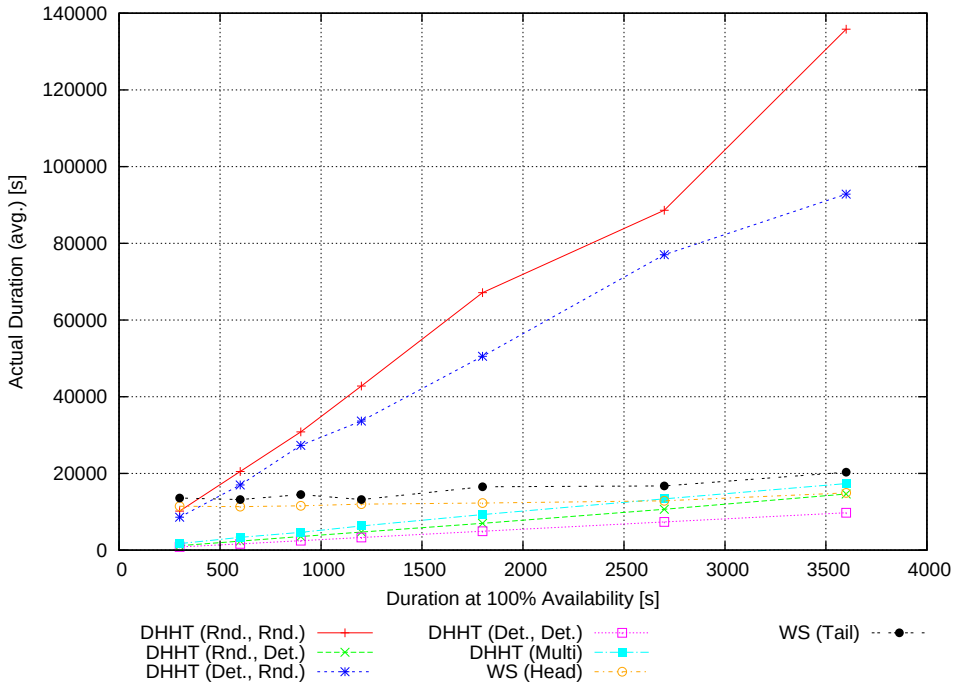

Figure A.127: *Actual average duration of jobs in experiment 43.*

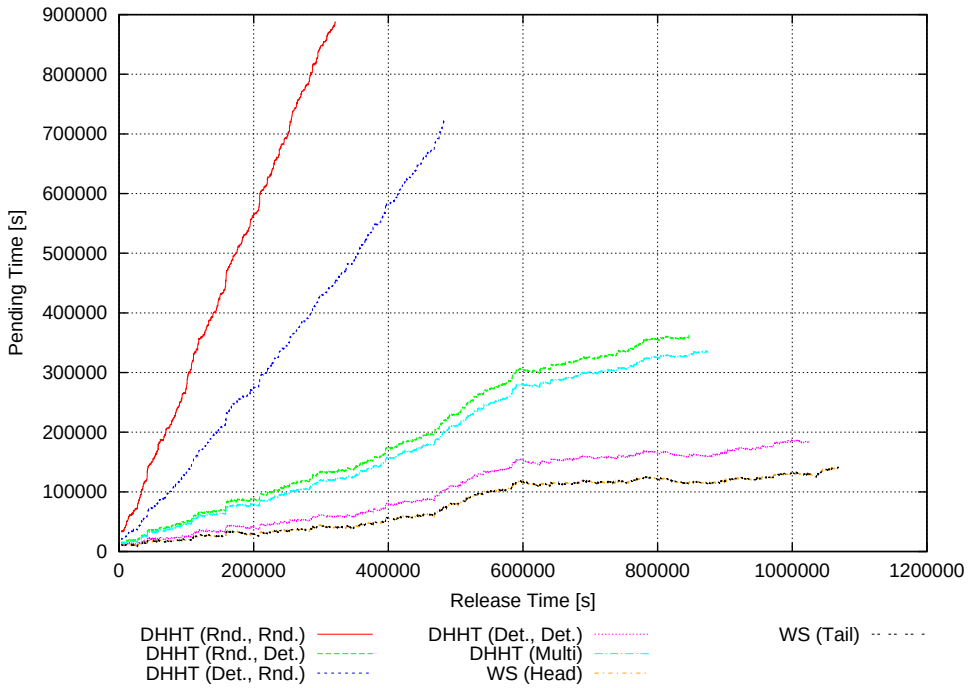

Figure A.128: *Pending time of jobs in experiment 43.*

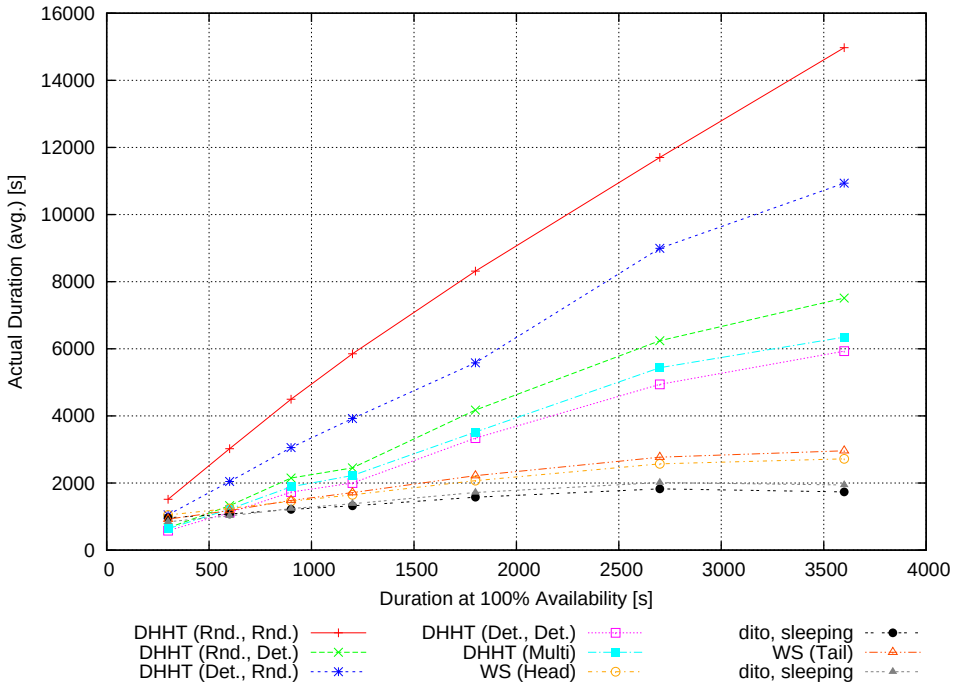

Figure A.129: *Actual average duration of processes in experiment 44.*

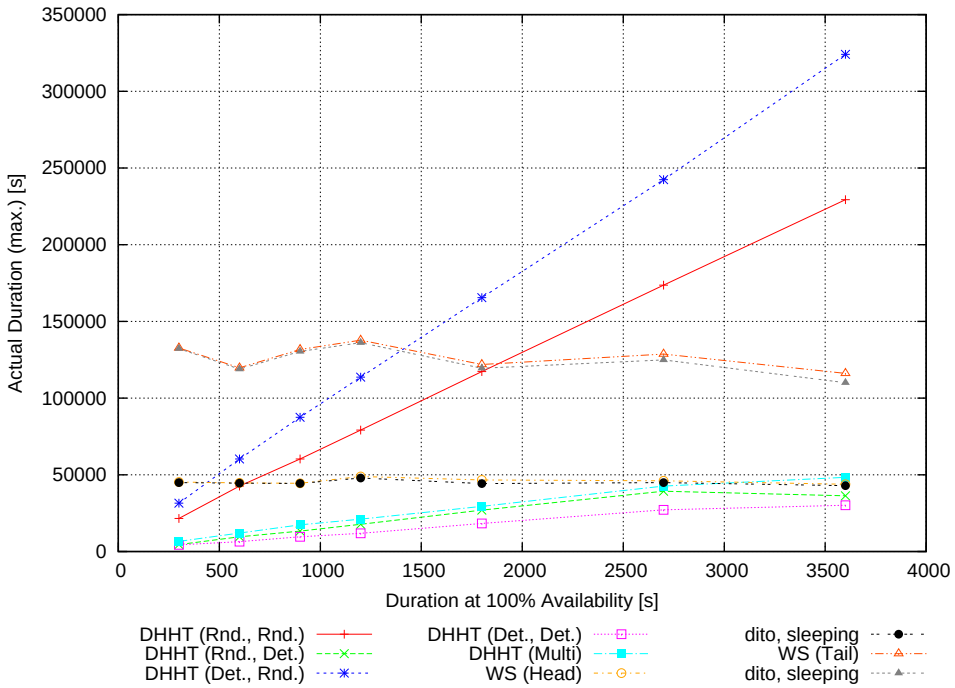

Figure A.130: *Actual maximum duration of processes in experiment 44.*

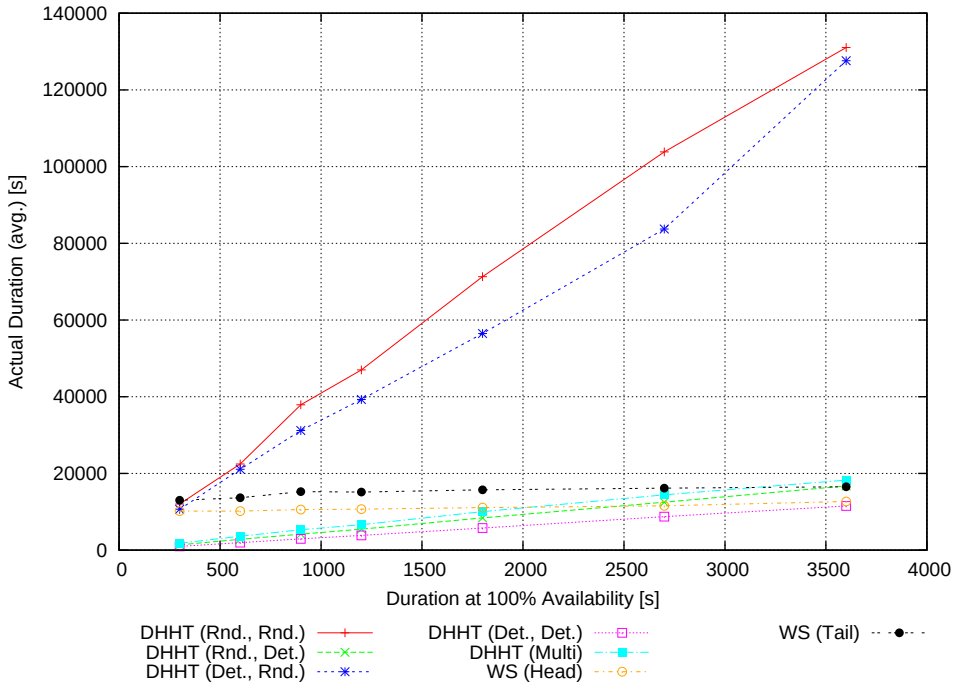

Figure A.131: *Actual average duration of jobs in experiment 44.*

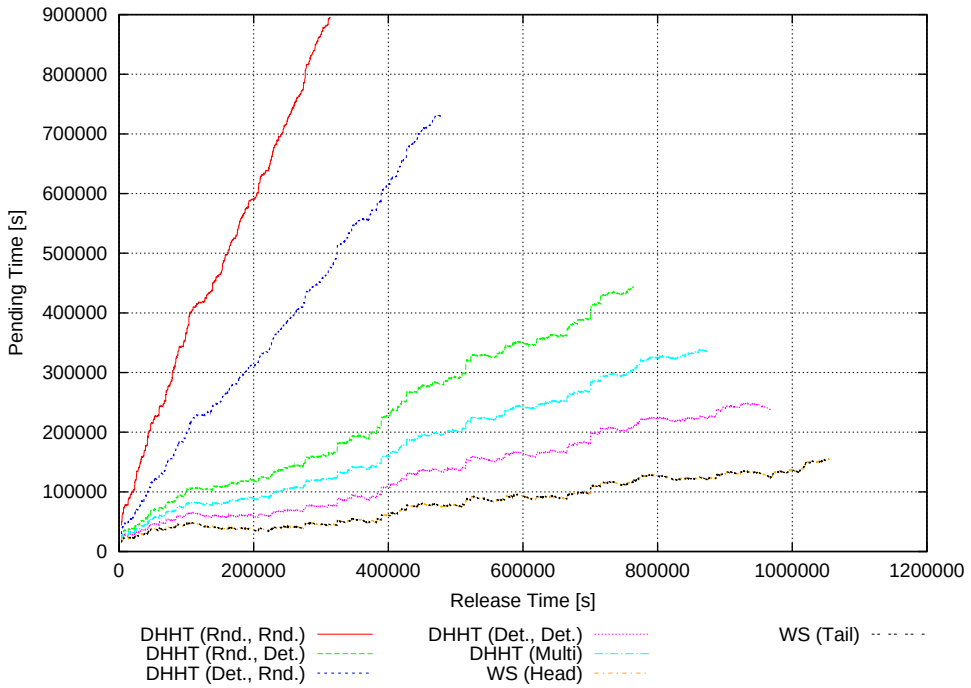

Figure A.132: *Pending time of jobs in experiment 44.*

A.4.3 Pool PCs

This subsection contains the plots for the input profiles HPC2N (figures A.133 – A.136), LLNL Atlas (figures A.137 – A.140), LLNL Thunder (figures A.141 – A.144), SDSC BLUE (figures A.145 – A.148), and SDSC DataStar (figures A.149 – A.152) for the experiments of type C with an overload of factor 4 and using the pool PC load profile.

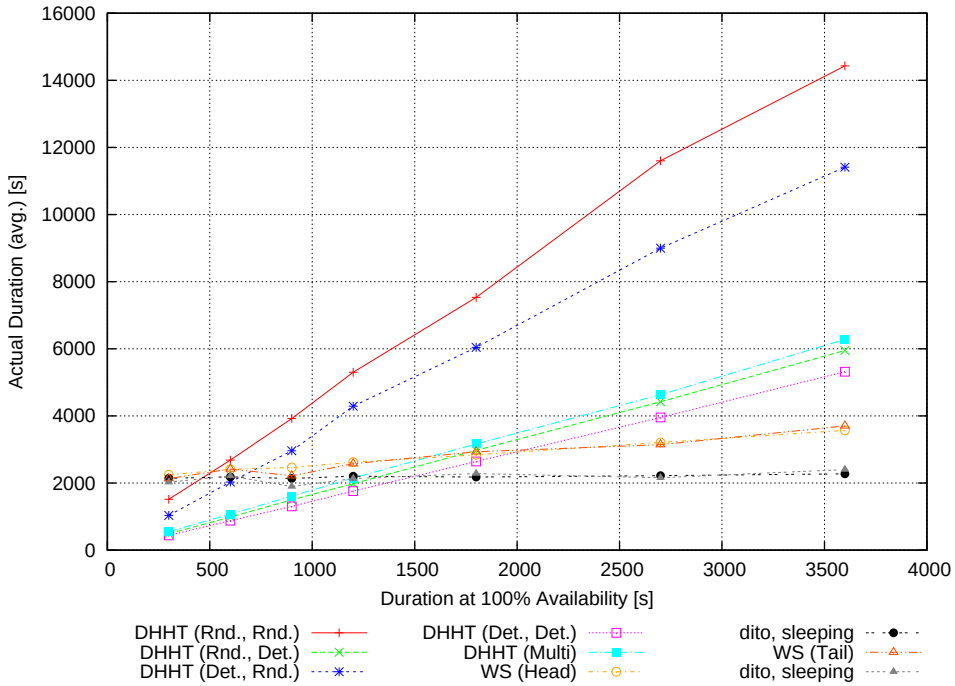

Figure A.133: *Actual average duration of processes in experiment 45.*

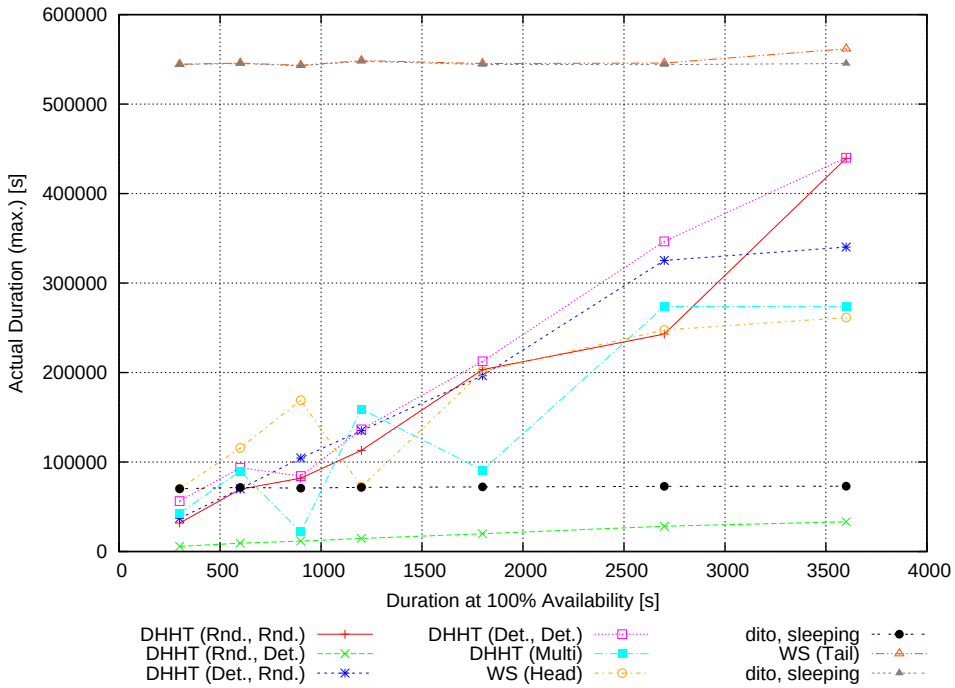

Figure A.134: *Actual maximum duration of processes in experiment 45.*

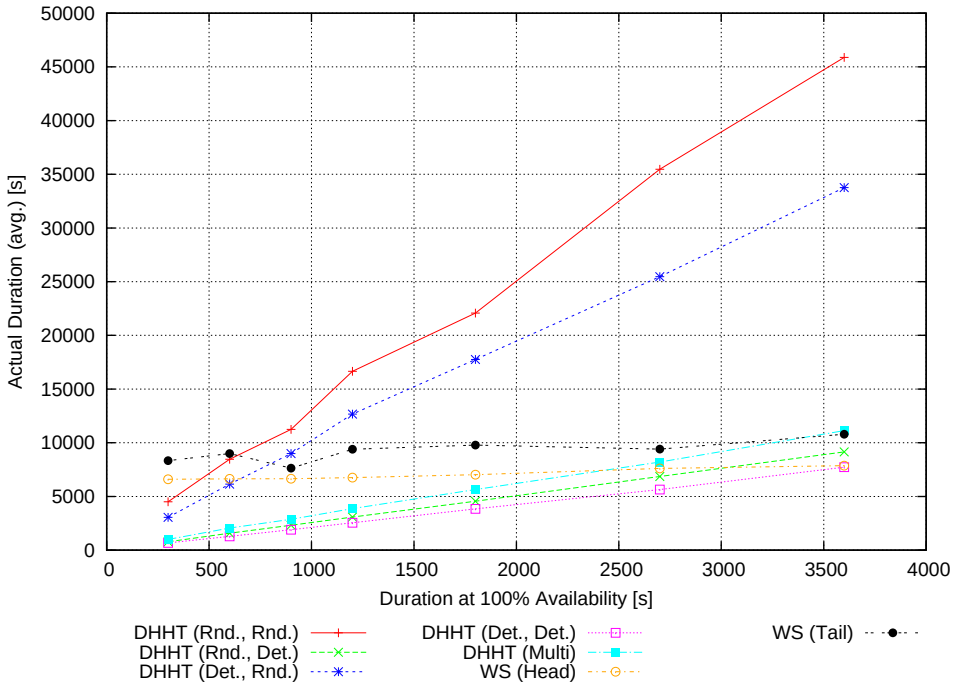

Figure A.135: *Actual average duration of jobs in experiment 45.*

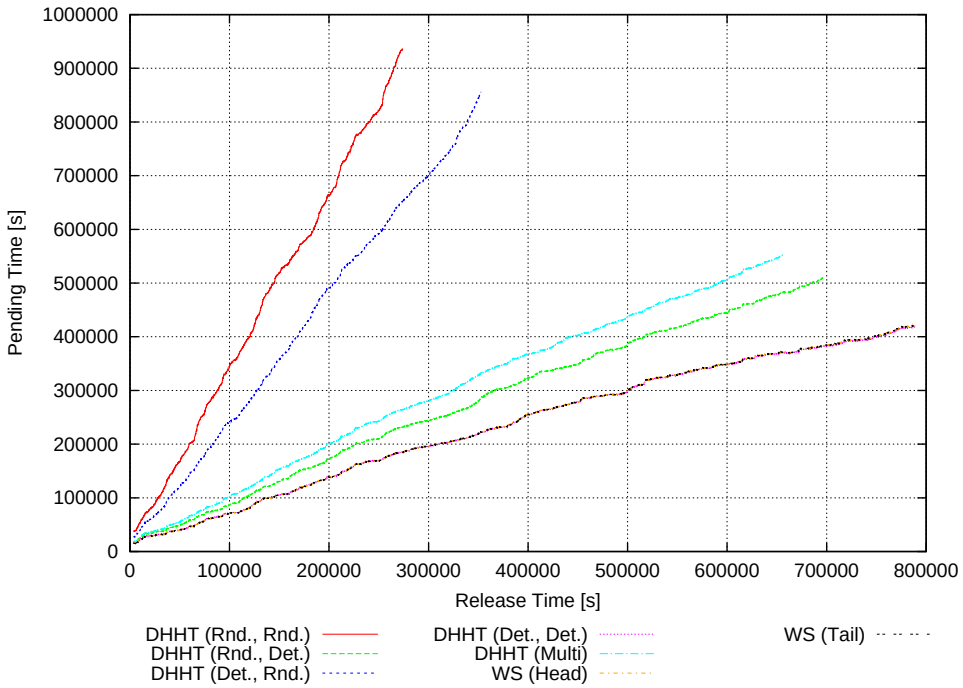

Figure A.136: *Pending time of jobs in experiment 45.*

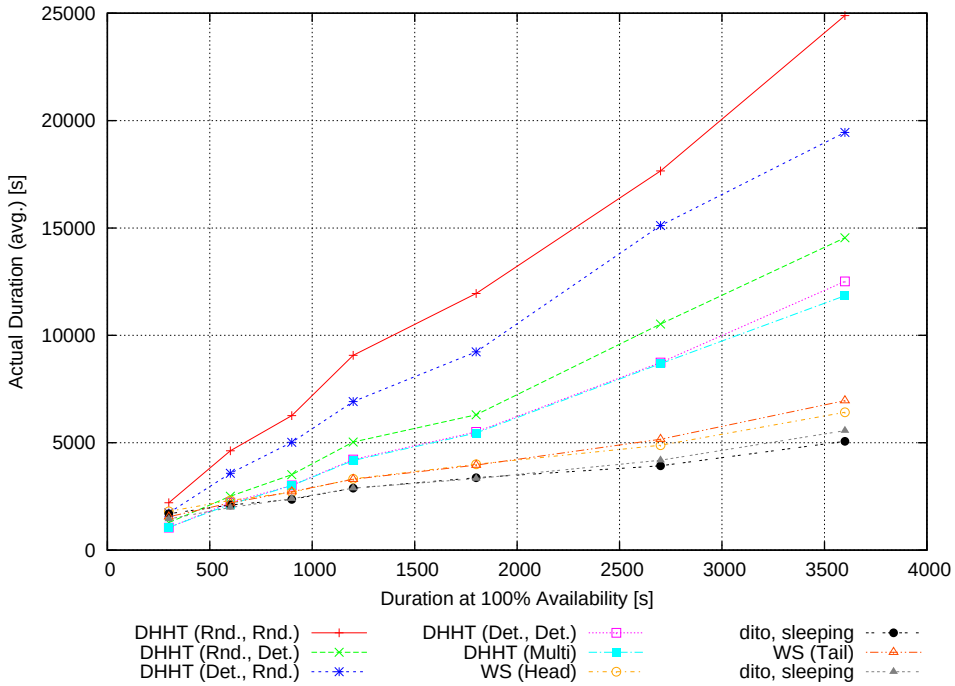

Figure A.137: *Actual average duration of processes in experiment 46.*

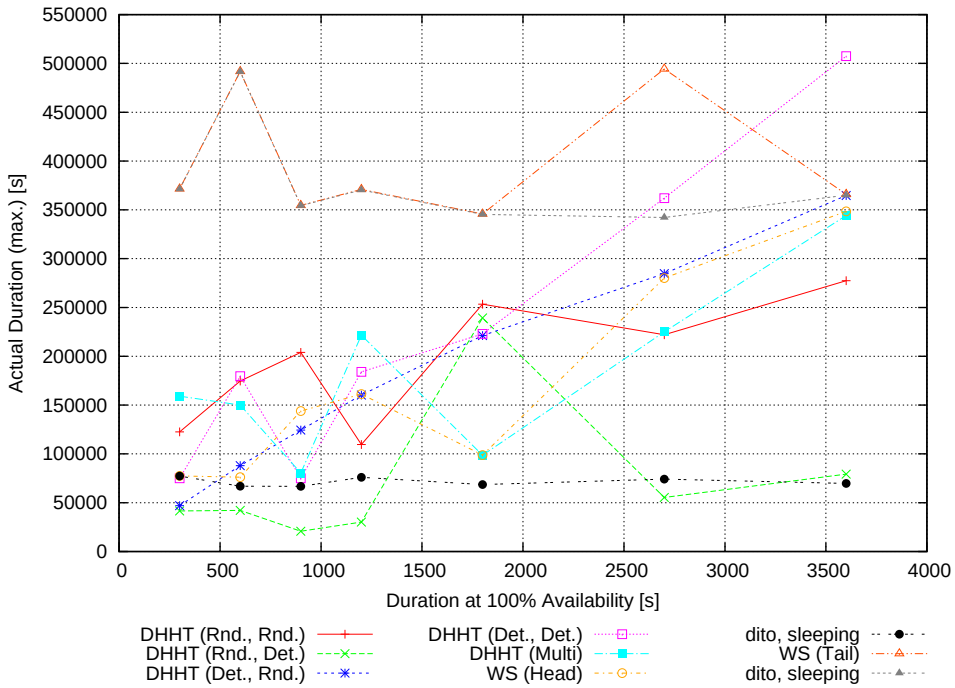

Figure A.138: *Actual maximum duration of processes in experiment 46.*

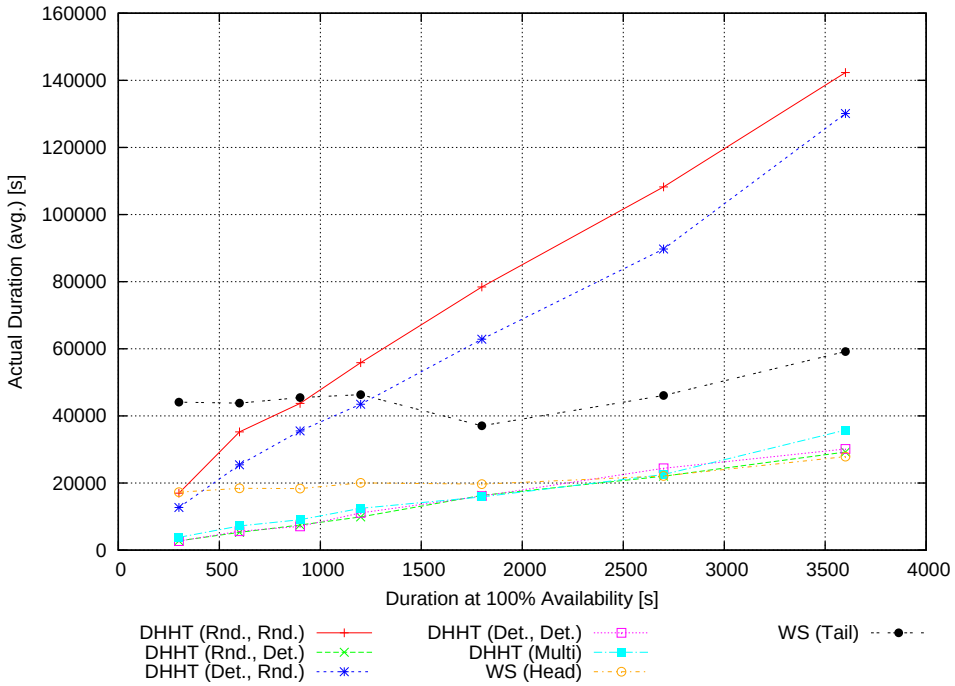

Figure A.139: *Actual average duration of jobs in experiment 46.*

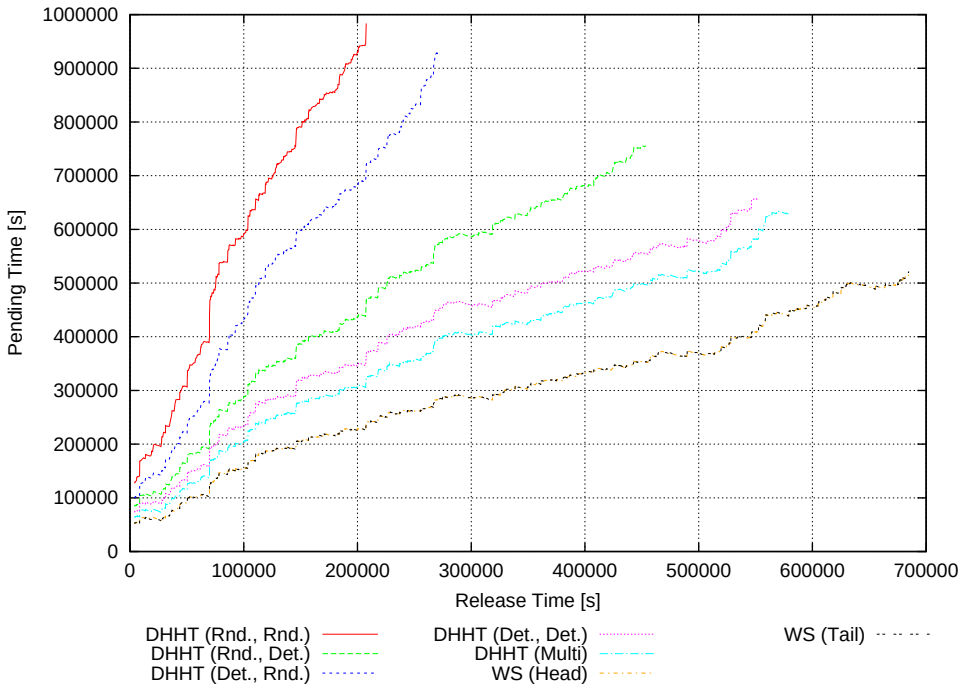

Figure A.140: *Pending time of jobs in experiment 46.*

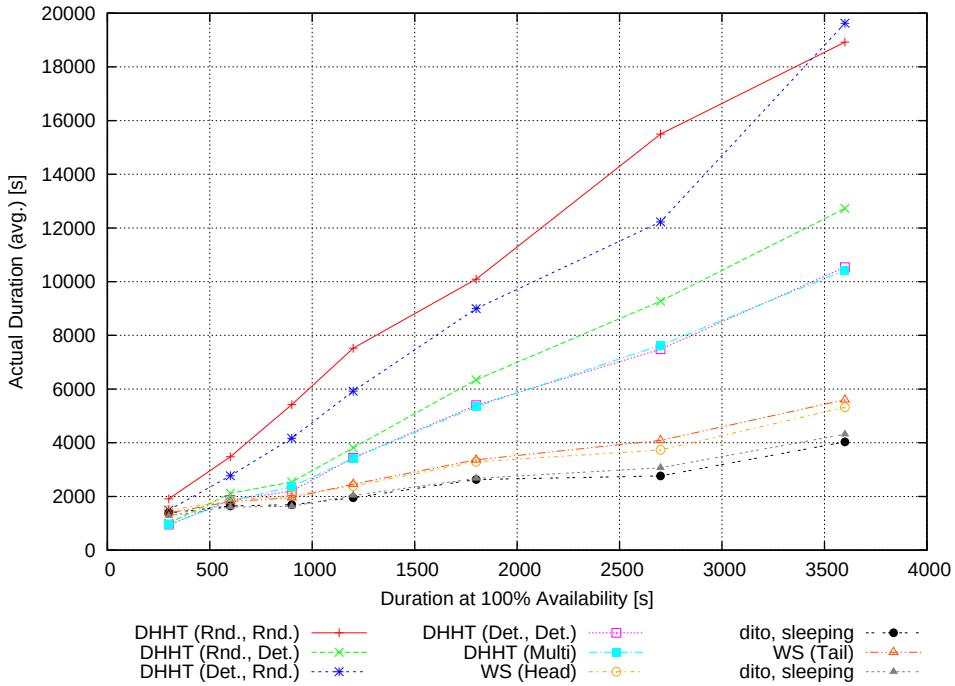

Figure A.141: *Actual average duration of processes in experiment 47.*

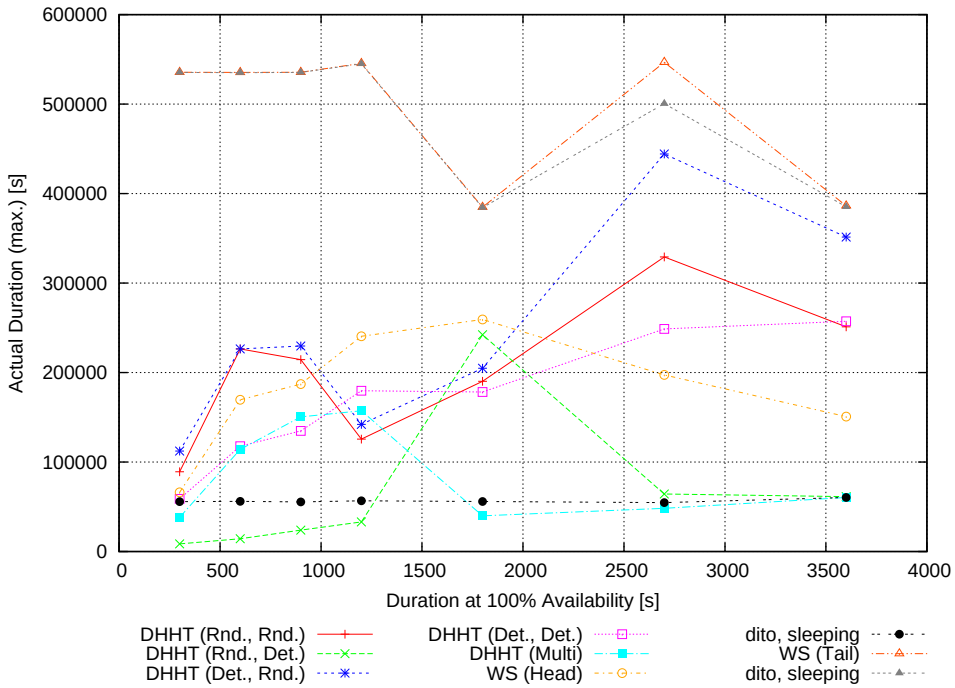

Figure A.142: *Actual maximum duration of processes in experiment 47.*

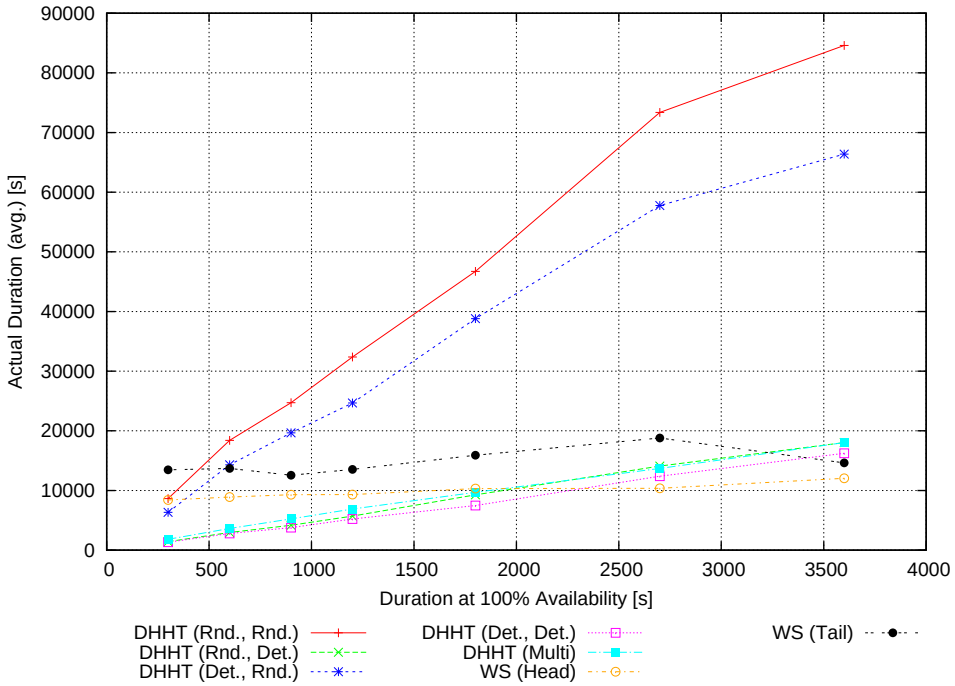

Figure A.143: *Actual average duration of jobs in experiment 47.*

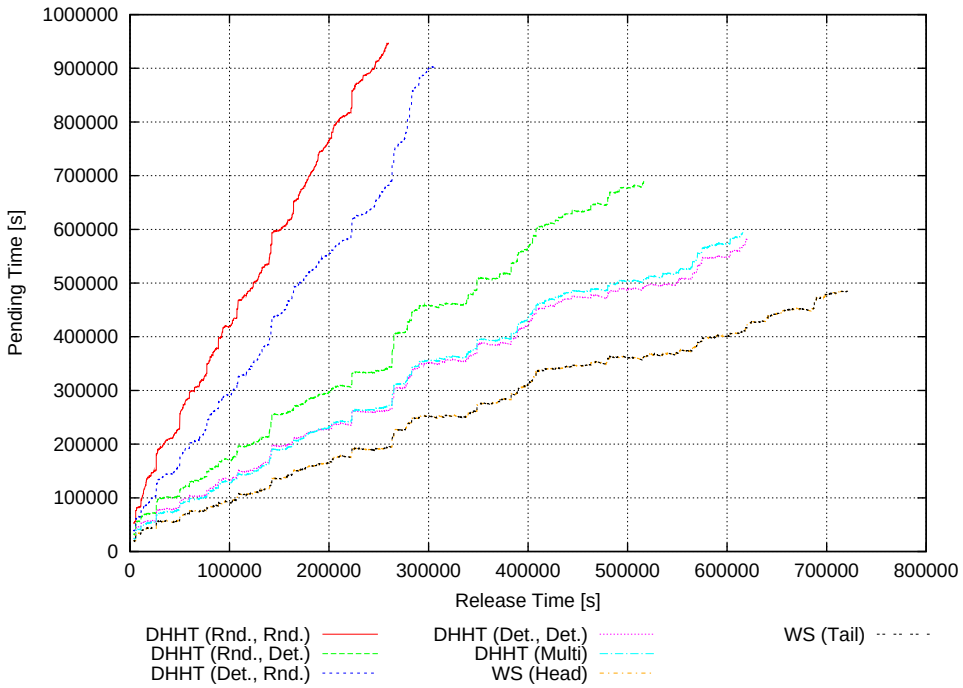

Figure A.144: *Pending time of jobs in experiment 47.*

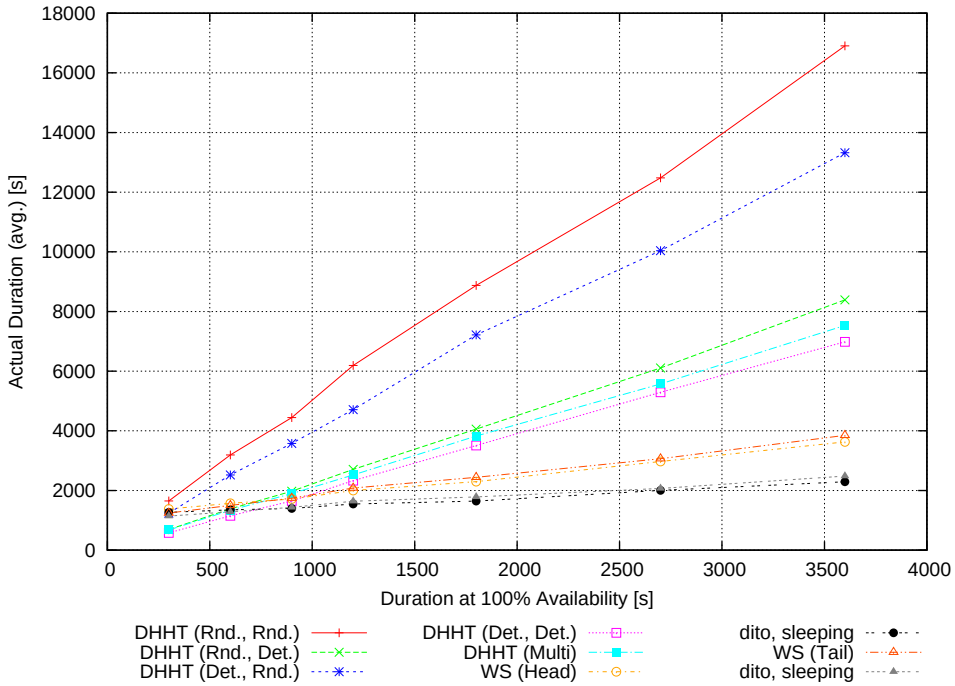

Figure A.145: *Actual average duration of processes in experiment 48.*

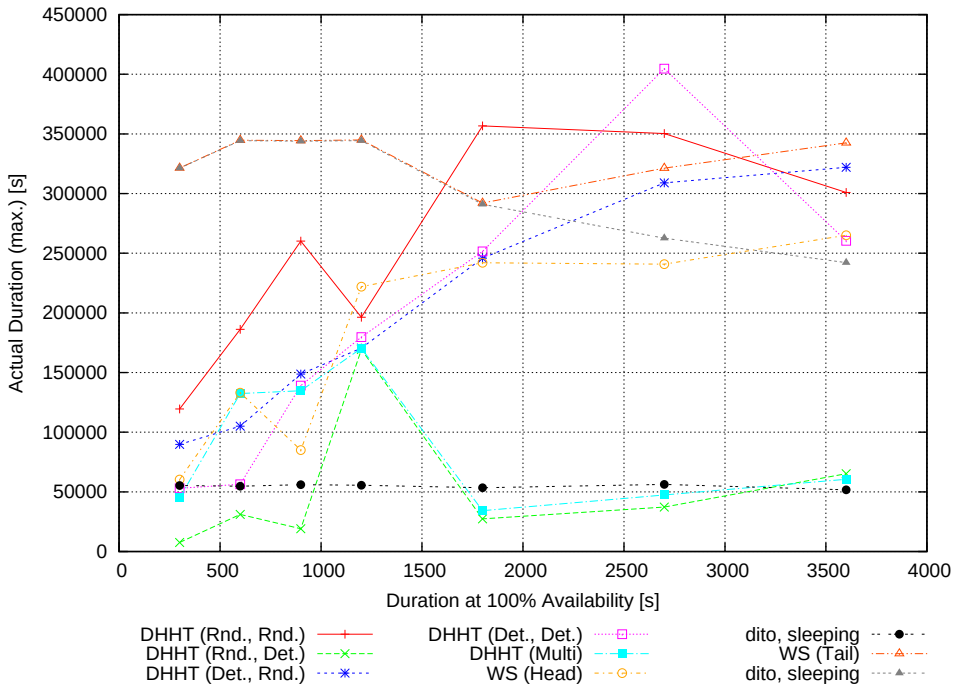

Figure A.146: *Actual maximum duration of processes in experiment 48.*

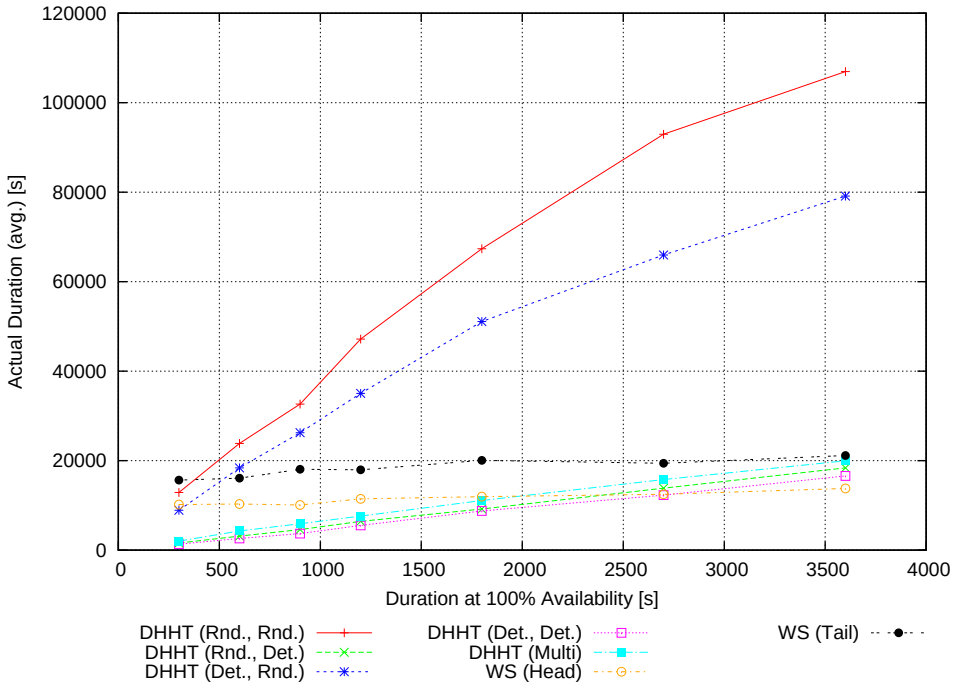

Figure A.147: *Actual average duration of jobs in experiment 48.*

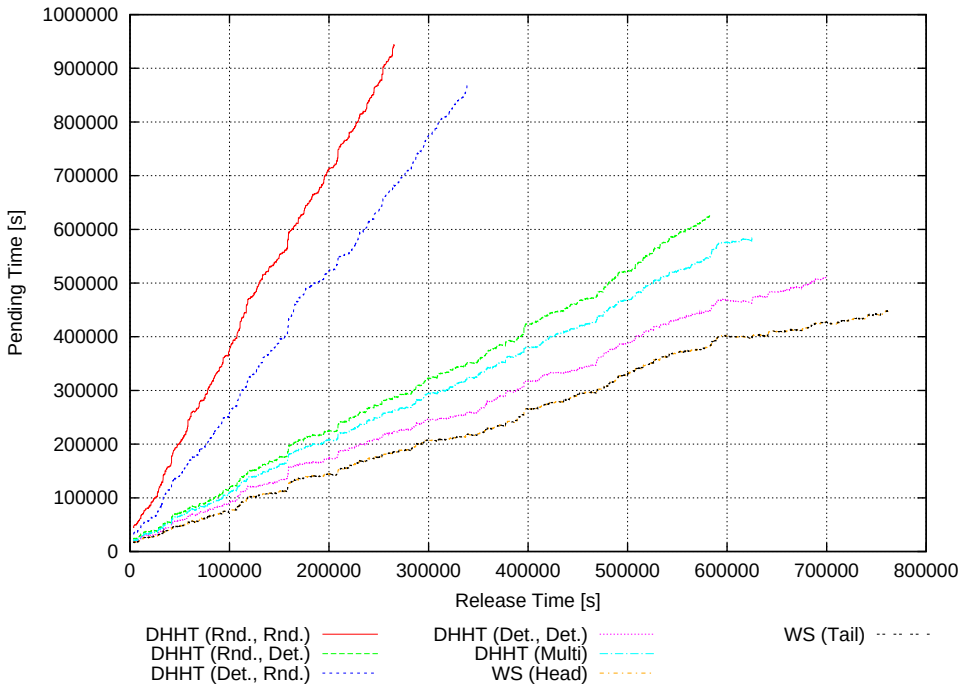

Figure A.148: *Pending time of jobs in experiment 48.*

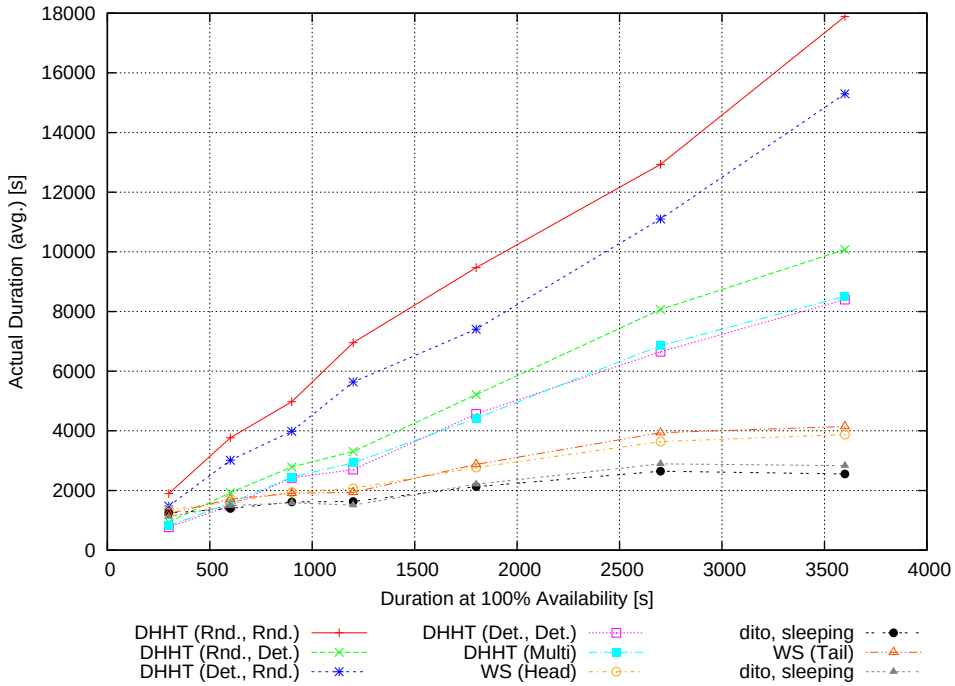

Figure A.149: *Actual average duration of processes in experiment 49.*

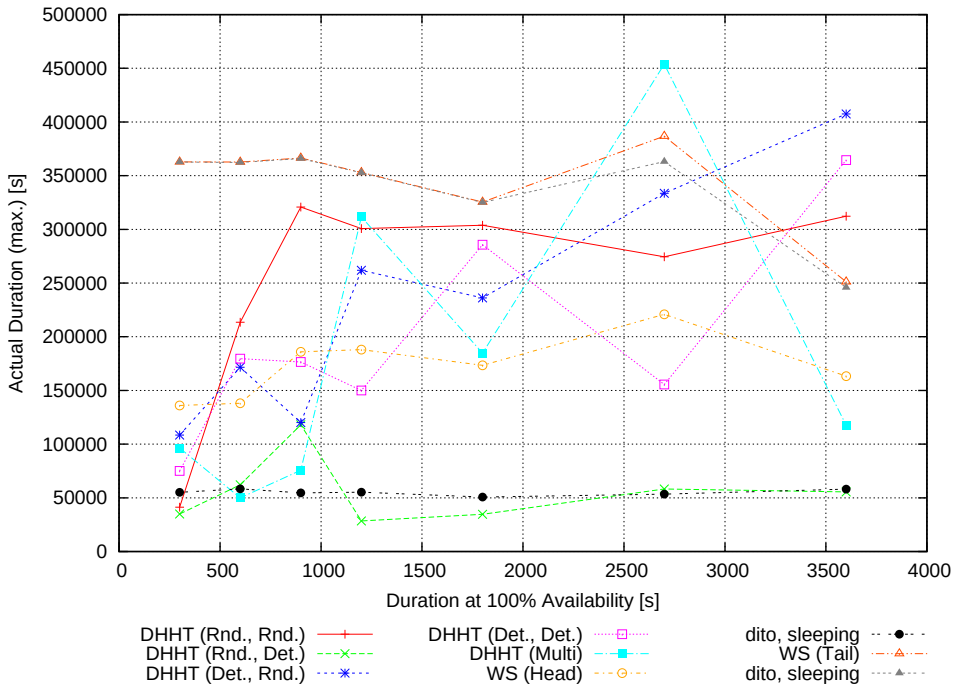

Figure A.150: *Actual maximum duration of processes in experiment 49.*

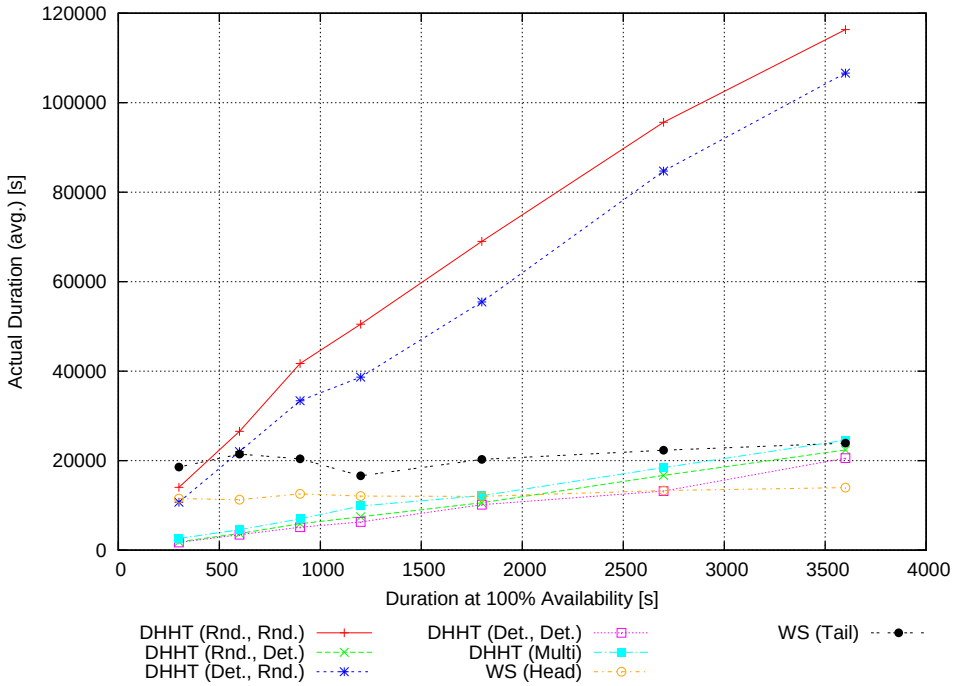

Figure A.151: *Actual average duration of jobs in experiment 49.*

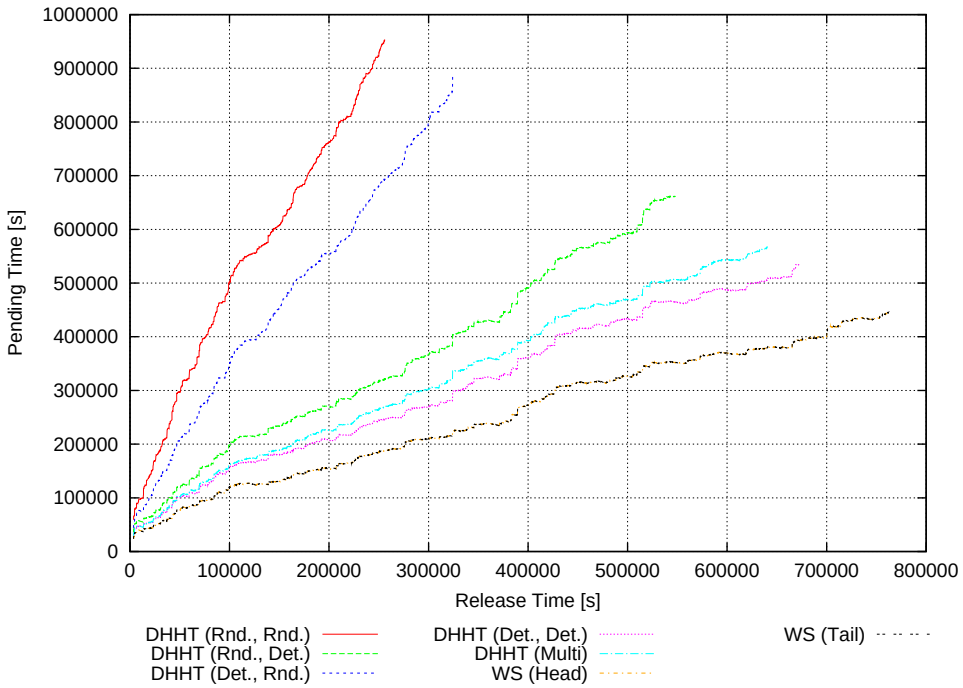

Figure A.152: *Pending time of jobs in experiment 49.*

Bibliography

[ACK+02] David P. Anderson, Jeff Cobb, Eric Korpela, Matt Lebofsky, and Dan Werthimer. SETI@home: An experiment in public-resource computing. *Communications of the ACM*, 45(11):56–61, 2002.

[ALW+03] Khaled Alzoubi, Xiang-Yang Li, Yu Wang, Peng-Jun Wan, and Ophir Frieder. Geometric spanners for wireless ad hoc networks. *IEEE Transactions on Parallel Distributed Systems*, 14(4):408–421, 2003.

[And04] David P. Anderson. Boinc: A system for public-resource computing and storage. In *GRID '04: Proceedings of the 5th IEEE/ACM International Workshop on Grid Computing*, pages 4–10, Washington, DC, USA, 2004.

[BDG+08] Ulrik Brandes, Daniel Delling, Marco Gaertler, Robert Görke, Martin Hoefer, Zoran Nikoloski, and Dorothea Wagner. On modularity clustering. *IEEE Transactions on Knowledge and Data Engineering*, 20(2):172–188, 2008.

[BGM05a] Olaf Bonorden, Joachim Gehweiler, and Friedhelm Meyer auf der Heide. Load balancing strategies in a web computing environment. In *Proceeedings of International Conference on Parallel Processing and Applied Mathematics (PPAM)*, pages 839–846, Poznan, Poland, 2005.

[BGM05b] Olaf Bonorden, Joachim Gehweiler, and Friedhelm Meyer auf der Heide. A web computing environment for parallel algorithms in Java. In *Proceeedings of International Conference on Parallel Processing and Applied Mathematics (PPAM)*, pages 801–808, Poznan, Poland, 2005.

[BGM06] Olaf Bonorden, Joachim Gehweiler, and Friedhelm Meyer auf der Heide. A web computing environment for parallel algorithms in Java. *Scalable Computing: Practice and Experience*, 7(2):1–14, 2006.

[Bis04] Rob H. Bisseling. *Parallel Scientific Computation: A Structured Approach using BSP and MPI*. Oxford University Press, 2004.

[BJvOR03] Olaf Bonorden, Ben Juurlink, Ingo von Otte, and Ingo Rieping. The Paderborn University BSP (PUB) library. *Parallel Computing*, 29(2):187–207, 2003.

[BL94] Robert D. Blumofe and Charles E. Leiserson. Scheduling multithreaded computations by work stealing. In *Proceedings of the 35th Annual Symposium on Foundations of Computer Science*, pages 356–368, 1994.

[boi] Berkeley Open Infrastructure for Network Computing (BOINC). Website: `http://boinc.berkeley.edu/`.

[cam] The Caml language. Website: `http://caml.inria.fr/`.

[CCR+03] Brent Chun, David Culler, Timothy Roscoe, Andy Bavier, Larry Peterson, Mike Wawrzoniak, and Mic Bowman. PlanetLab: an overlay testbed for broad-coverage services. *SIGCOMM Computer Communication Review*, 33(3):3–12, 2003.

[CLQ06] Giacomo Cabri, Letizia Leonardi, and Raffaele Quitadamo. Enabling Java mobile computing on the IBM Jikes research virtual machine. In *PPPJ '06: Proceedings of the 4th International Symposium on Principles and Practice of Programming in Java*, pages 62–71, New York, NY, USA, 2006.

[Cyb89] G. Cybenko. Dynamic load balancing for distributed memory multiprocessors. *Parallel and Distributed Computing*, 7:279–301, 1989.

[dis] distributed.net — the Internet's first general-purpose distributed computing project. Website: `http://www.distributed.net/`.

[EvDO02] A. J. Enright, S. van Dongen, and C. A. Ouzounis. An efficient algorithm for large-scale detection of protein families. *Nucleic Acids Research*, 30(7):1575–1584, 2002.

[FGNC01] Gilles Fedak, Cecile Germain, Vincent Neri, and Franck Cappello. Xtremweb: A generic global computing system. *IEEE International Symposium on Cluster Computing and the Grid*, pages 582–587, 2001.

[Fün98] Stefan Fünfrocken. Transparent migration of Java-based mobile agents. In *Mobile Agents*, pages 26–37, 1998.

[Gen10] Craig Gentry. Computing arbitrary functions of encrypted data. *Communications of the ACM*, 53(3):97–105, 2010.

[gim] Great Internet Mersenne Prime Search (GIMPS). Website: `http://www.mersenne.org/`.

[GM10] Joachim Gehweiler and Henning Meyerhenke. A distributed diffusive heuristic for clustering a virtual P2P supercomputer. In *Proceeedings of 24th International Parallel and Distributed Processing Symposium (IPDPS, HPGC)*, Atlanta, USA, 2010.

[Gon01] Li Gong. JXTA: A network programming environment. *IEEE Internet Computing*, 5:88–95, 2001.

[GS06] Joachim Gehweiler and Gunnar Schomaker. Distributed load balancing in heterogeneous peer-to-peer networks for web computing libraries. In *Proceeedings of 10th IEEE/ACM International Symposium on Distributed Simulation and Real Time Applications (DS-RT)*, pages 51–58, Torremolinos, Malaga, Spain, 2006.

[GT10] Joachim Gehweiler and Michael Thies. Thread migration and checkpointing in Java. Technical Report tr-ri-10-315, Heinz-Nixdorf-Institute, June 2010.

[HMS+98] Jonathan M. D. Hill, Bill McColl, Dan C. Stefanescu, Mark W. Goudreau, Kevin Lang, Satish B. Rao, Torsten Suel, Thanasis Tsantilas, and Rob H. Bisseling. BSPlib: The BSP programming library. *Parallel Computing*, 24(14):1947–1980, 1998.

[jav] The JavaGo library. Website: `http://homepage.mac.com/t.sekiguchi/javago/`.

[jxt] The JXTA community. Website: `https://jxta.dev.java.net/`.

[KLL+97] David Karger, Eric Lehman, Tom Leighton, Rina Panigrahy, Matthew Levine, and Daniel Lewin. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the world wide web. In *STOC '97: Proceedings of the 29th Annual ACM Symposium on Theory of Computing*, pages 654–663, New York, NY, USA, 1997.

[Lov93] L. Lovász. Random walks on graphs: A survey. *Combinatorics, Paul Erdös is Eighty*, 2:1–46, 1993.

[MMS09] Henning Meyerhenke, Burkhard Monien, and Thomas Sauerwald. A new diffusion-based multilevel algorithm for computing graph partitions. *Journal on Parallel and Distributed Computing*, 69(9):750–761, 2009.

[MWL00] Matchy J. M. Ma, Cho-Li Wang, and Francis C. M. Lau. Delta execution: A preemptive Java thread migration mechanism. *Cluster Computing*, 3(2):83–94, 2000.

[NG04] M. E. J. Newman and M. Girvan. Finding and evaluating community structure in networks. *Physical Review E (Statistical, Nonlinear, and Soft Matter Physics)*, 69(2), 2004.

[pada] The Paderborn Thread Migration and Checkpointing (PadMig) library. Website: `http://padmig.cs.uni-paderborn.de/`.

[padb] The Paderborn University Remote Method Invocation (PadRMI) library. Website: `http://padrmi.cs.uni-paderborn.de/`.

[Paw05] Renaud Pawlak. Spoon: annotation-driven program transformation — the AOP case. In *AOMD '05: Proceedings of the 1st Workshop on Aspect Oriented Middleware Development*, New York, NY, USA, 2005.

[PH99] Michael Philippsen and Bernhard Haumacher. More efficient object serialization. In *Proceedings of the 11 IPPS/SPDP '99 Workshops Held in Conjunction with the 13th International Parallel Processing Symposium and 10th Symposium on Parallel and Distributed Processing*, pages 718–732, London, UK, 1999.

[pla] PlanetLab. Website: `http://www.planet-lab.org/`.

[puba] The Paderborn University BSP (PUB) library. Website: `http://publibrary.sourceforge.net/`.

[pubb] The Paderborn University BSP-based Web Computing (PUB-Web) library. Website: `http://pubweb.cs.uni-paderborn.de/`.

[RSAU91] Larry Rudolph, Miriam Slivkin-Allalouf, and Eli Upfal. A simple load balancing scheme for task allocation in parallel machines. In *Proceedings of the 3rd Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 237–245, 1991.

[Sar99] Luis F. G. Sarmenta. An adaptive, fault-tolerant implementation of BSP for Java-based volunteer computing systems. In *Lecture Notes in Computer Science*, volume 1586, pages 763–780, 1999.

[set] Search for Extraterrestrial Intelligence (SETI@home). Website: `http://setiathome.berkeley.edu/`.

[SMY99] Tatsurou Sekiguchi, Hidehiko Masuhara, and Akinori Yonezawa. A simple extension of Java language for controllable transparent migration and its portable implementation. In *Coordination Models and Languages*, pages 211–226, 1999.

[spo] The Spoon framework. Website: `http://spoon.gforge.inria.fr/`.

[SS05] Christian Schindelhauer and Gunnar Schomaker. Weighted distributed hash tables. In *SPAA '05: Proceedings of the 17th Annual ACM symposium on Parallelism in Algorithms and Architectures*, pages 218–227, New York, NY, USA, 2005.

[ŠS06] J. Šíma and S. E. Schaeffer. On the NP-completeness of some graph cluster measures. In *32nd International Conference on Current Trends in Theory and Practice of Informatics (SOFSEM)*, volume 3831 of *LNCS*, pages 530–537, 2006.

[SSY00] Takahiro Sakamoto, Tatsurou Sekiguchi, and Akinori Yonezawa. Bytecode transformation for portable thread migration in Java. In *ASA/MA 2000: Proceedings of the 2nd International Symposium on Agent Systems and Applications and 4th International Symposium on Mobile Agents*, pages 16–28, 2000.

[Swi58] S. Swierczkowski. On successive settings of an arc on the circumference of a circle. *Fundamenta Mathematicae*, 46:187–189, 1958.

[Thi01] Michael Thies. *Combining Static Analysis of Java Libraries with Dynamic Optimization*. Shaker Verlag, 2001.

[TRV+00] Eddy Truyen, Bert Robben, Bart Vanhaute, Tim Coninx, Wouter Joosen, and Pierre Verbaeten. Portable support for transparent thread migration in Java. In *ASA/MA 2000: Proceedings of the 2nd International Symposium on Agent Systems and Applications and 4th International Symposium on Mobile Agents*, pages 29–43, London, UK, 2000.

[Val90] Leslie G. Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8):103–111, 1990.

[van00] Stijn van Dongen. *Graph Clustering by Flow Simulation*. PhD thesis, University of Utrecht, 2000.

[wor] The Parallel Workloads Archive. Website: `http://www.cs.huji.ac.il/labs/parallel/workload/`.

[xtr] XtremWeb: An open source platform for desktop grids. Website: `http://www.xtremweb.net/`.

[YKGF06] Haifeng Yu, Michael Kaminsky, Phillip B. Gibbons, and Abraham Flaxman. SybilGuard: Defending against sybil attacks via social networks. In *SIGCOMM '06: Proceedings of the 2006 Conference on Applications, Technologies, Architectures, and Protocols for Computer communications*, pages 267–278, New York, NY, USA, 2006.