OSS User's Guide

Michael Schöttner, Marc-Florian Müller, Kim-Thomas Rehmann
(Universität Düsseldorf)

September 30, 2010

1 What is OSS

The Object Sharing Service (OSS) implements distributed objects for nodes participating in an interactive multi-user grid application. OSS runs on each client machine to enable sharing of objects residing in volatile memory. An object in this context is a replicated volatile memory region, dynamically allocated by an application or mapped into memory from a file.

Objects may contain scalars, references, and code. Therefore, OSS handles concurrent read and write access to objects and maintains the consistency of replicated objects. Persistence and security for objects stored in files are provided by XtreemFS. Fault tolerance is provided by the grid checkpointing mechanisms developed in WP3.3. OSS is being developed for Linux on IA32 or AMD64/Intel64 compatible processors.

Please see our publications [MMSS08, MMSS09, RMS10] for further details on design, implementation and evaluation of OSS.

2 Installation of OSS

You can install OSS either using the prebuild distribution packages which are available on the XtreemOS release media, or you can build and install OSS from source code. We suggest using the first method mentioned, unless you wish to configure special build-time settings for OSS.

2.1 Installing OSS Using the Distribution Packages

The XtreemOS release contains the OSS library, as well as a raytracing demo application to demonstrate object sharing. During the XtreemOS installation procedure, simply select the checkbox *Object Sharing Service release* to install

the packaged version of OSS (Library and applications). If you have XtreemOS already installed and wish to install OSS, select it in the package management dialog, or run the following commands as root:

```
$> urpmi liboss0
$> urpmi oss
```

The first command installs the OSS library, the second command installs some example applications.

Application development based on OSS need the following additional packages:

```
$> urpmi liboss0-devel
$> urpmi liboss0-static-devel
```

2.2 Building and Installing OSS from Source

By building and installing OSS from source, you have full control over the installation process. You can configure how OSS is installed, and fine-tune all OSS features. The OSS sources can be installed from XtreemOS source repository:

```
$> urpmi oss-0.6-1xos2.0.src    (OSS sources)
```

2.2.1 Prerequisites

If you wish to build and install OSS from the source code, you need to have some additional development packages installed on your build system. The names of these packages depend on which Linux distribution you are using. Although the OSS build process includes diverse checks for these libraries, we cannot anticipate the requirements for building OSS on all Linux distributions available. If in doubt, please consult the package search of your distribution.

Under Mandriva or XtreemOS, the following packages are needed to build OSS and the included demo applications:

- gcc $\geq$ 4.3
- binutils $\geq$ 2.18
- make
- glibc-devel
- libglib2.0-devel $\geq$ 2.18
- libreadline5-devel

Under Debian GNU/Linux, the following packages are needed to build OSS and the included demo applications:

- gcc $\geq$ 4.3
- binutils $\geq$ 2.18
- make
- libc6-dev
- libglib2.0-dev $\geq$ 2.18
- libreadline5-dev
- libstdc++6-4.3-dev

In case your distribution does not include a recent GLib, the OSS installation process helps you download and install GLib from source. Simply run the following command, entering the root password when asked for:

```
$> make build/glib
```

For 32-bit OSS under Linux x86_64 you also need

- ia32-libs
- ia32-libs-gtk
- libc6-dev-i386
- lib32readline5

To compile 32-bit OSS under Linux x86_64, some distributions lack symbolic links to 32-bit libraries (libgthread-2.0.so, libglib-2.0.so, libstdc++.so). When encountering error messages telling that libgthread-2.0, libglib-2.0 or libstdc++ cannot be found, log in as user root and create the missing symlinks:

```
$> ln -s libgthread-2.0.so.0 /emul/ia32-linux/usr/lib/libgthread-2.0.so
$> ln -s libglib-2.0.so.0 /emul/ia32-linux/usr/lib/libglib-2.0.so
$> ln -s libstdc++.so.6 /emul/ia32-linux/usr/lib/libstdc++.so
```

The following packages are useful to generate documentation:

- doxygen
- graphviz
- texlive

Doxygen generates source code documentation, whereas graphviz and texlive enable dependency graph and PDF file output respectively.

2.2.2 Compilation

Unpack the OSS source code archive, change to the base directory that just has been created. The following step allows altering the default configuration of OSS (hardware architecture, features, ...) if desired. If this step is omitted, OSS will be built in its default configuration. A description of the configuration option can be found in Appendix 4.

```
$> make menuconfig
```

The following command builds the OSS library:

```
$> make
```

The make system autodetects most tools used for building OSS. If you encounter any errors, please ensure you have a recent compiler and linker installed, and that all developer packages mentioned above are installed correctly.

If you wish to pass configuration parameters via command line or to enable non-standard features, you can directly supply the corresponding parameters to make. For example, run `make -B ARCH=I686` to build OSS for 32-Bit x86 machines and `make -B ARCH=X86_64` to build OSS for 64-Bit x86 machines respectively.

2.2.3 Installation

The following command installs the OSS library on the system, by default in the `/usr/local` hierarchy. For write access to system directories, you need root privileges.

```
$> make install
```

You can change the default installation hierarchy by specifying `prefix=<pathname>`, e.g. to install OSS below `/usr`, run the command

```
$> make install prefix=/usr
```

Software distributors can specify an additional prefix for the actual installation directory by defining `DESTDIR=<additional-prefix>` on the make command line.

2.3 Testing the OSS Installation

The OSS make system includes a command to verify that OSS has been installed correctly, and that everything needed for running a program that uses OSS is set up correctly:

```
$> make verify-install
```

The program should output the version and build information of the OSS library found according to the example below:

```
Object Sharing Service version 0.6 architecture I686
   subversion revision 5844 (2010-03-05 14:51:38)
build 1
Object Sharing Service has been installed correctly.
```

2.3.1 Simple test of Object Sharing

The simple test application (*oss_simple*) starts two instances of a program. The first process creates a shared object and writes the string *hello* to it. The second process waits until the object has been created, and as soon as it reads the expected string, it overwrites it with the string *world*.

2.3.2 The Raytracer Application

The raytracer is based on a application developed for a course at the MIT and has been ported to OSS with the focus on testing and demonstrating transactional shared memory. All graphical objects and the image file are allocated in transactional shared memory. Start the first node with

```
$> oss_raytracer --address <IP1>
```

and the subsequent nodes with

```
$> oss_raytracer --address <IPn> --bootstrap <IP1>
```

where `IP1` is the IP address for the first node, and `IPn` is replaced by the IP address of the respective node. To configure the tracing progress, the first node will ask some parameters:

1. Consistency model: 't' for transactional consistency, 's' for strong consistency
2. Number of Nodes
3. Number of accesses (applies to transactional consistency only): number of accesses between transaction boundaries
4. Pattern: specify one of 'l', 'c', 'p', 'x' or 'm'. 'l' for line by line, 'c' for column by column, 'p' for x partitions, 'x' for every Xth dot, 'm' for matching pages
5. Scene: 1, 2, or 3
6. Columns (e. g. 640)

7. Rows (e. g. 480)

After rendering is done, you can give new parameters and render another scene. There are three predefined scenes in this project. *Scene1* is very simple with one sphere in the center, a few bowls around and only a few lights. *Scene2* is very complex with some arrangements of bowls and reflecting walls. *Scene3* displays the letters "'OSS"' consisting of bowls. You can write own scenes as *C* files analoguous to *SceneDemo1.c*.

3 Developing Applications using OSS

The internal interface of the OSS library is implementation-dependent and may be extended in the future, based on insights gained during the development of OSS-based applications. In contrast, WP3.1 has defined an XOSAGA interface for object sharing that represents the OSS interface in a portable way.

3.1 Internal Interface of the OSS Library

The interface of the OSS library is declared in the header file `oss.h`. The following command generates an interface documentation in HTML and (if latex is available) in PDF format:

```
$> make interface-doc
```

The documentation is stored in the `build/doc/` subdirectory.

Let us quickly walk through the basic functionality of the OSS library. For a more detailed and precise discussion of the internal library interface, please see the Doxygen documentation generated directly from the source code. To get a deeper understanding of how to design applications that access shared objects, we suggest looking at the source code examples in the `src/apps/` subdirectory.

```
int
oss_startup(
     const char *addr,
     const char *listen_port,
     const char *bootstrap_addr,
     const char *bootstrap_port
     );
```

The `oss_startup` call starts the OSS system by joining a bootstrap peer. The `addr` and `listen_port` parameters allow to bind the OSS instance to a specific interface. If no bootstrap peer is specified (i. e. a NULL pointer is passed), a new distributed object storage is created. A return value of zero indicates successful startup.

```
void *
oss_alloc(
     size_t size,
     oss_consistency_model_t consistency_model,
     oss_alloc_attributes_t *attributes
     );
```

The `oss_alloc` call creates a shared object of specified size, initializes it with a consistency model and further attributes (defined by the consistency model), and returns an identifier for the object.

```
void
oss_free(
     void *ptr
     );
```

The `oss_free` call frees some memory which has previously been dynamically allocated using `oss_alloc`.

```
oss_transaction_id_t
oss_bot(
     oss_transaction_priority_t priority,
     oss_transaction_attributes_t *attributes
     );
```

The `oss_bot` call marks the begin of a transaction with given priority and attributes. OSS guarantees that all accesses to distributed objects between `oss_bot` and `oss_eot` perform atomically, consistent, isolated, and durable. The return value references the transaction that has been started, or equals `oss_undefined_transaction_id` which indicates that the transaction failed to start.

```
int
oss_eot(
     oss_transaction_id_t taid
     );
```

The `oss_eot` call denotes the end of the supplied transaction.

```
int
oss_abort(
     oss_transaction_id_t taid
     );
oss_permit_abort(
     oss_transaction_id_t taid
     );
```

Both calls handle voluntarily aborting a transaction. An application that somehow finds out that it cannot commit, or that committing will have adverse effect, may call `oss_abort` to unconditionally abort the supplied transaction. Depending on the transaction attributes used, the transaction will restart or simply fail. An application may optionally call `oss_permit_abort` to mark locations in the code where it is safe to abort a transaction. If the transaction is already known to fail on commit, OSS can restart the transaction and need not delay restarting the transaction until `oss_eot`. If the success of the transaction is not yet determined, the call to `oss_permit_abort` will simply appear as a void statement.

```
void *
oss_nameservice_get(
     const char *id
     );

void
oss_nameservice_set(
     const char *id,
     void *val
     );
```

OSS contains a simple name service, which applications can use to store and retrieve object IDs. The name service has a tree structure, with slashes (/) separating directory levels. Each entry begins with a slash. An application or OSS module can set a value for a name by calling `oss_nameservice_set` and retrieve a value by calling `oss_nameservice_get`. A value that has not yet been set is treated as object ID `NULL`.

```
void
oss_wait(
     void *addr,
     unsigned char value
     );
```

The barriers implementation allows applications to wait until the character object pointed to by `addr` contains a target value. The character object may be subject to transactional or strong consistency.

3.2 Linking against the OSS Library

The OSS library is built as a static shared library (`liboss.a`) and as a dynamic shared library (`liboss.so`). Simply specify the option `-loss` to the compiler

driver or linker, which will link against the appropriate static or dynamic library. If you did not install the library into a well-known location such as `/usr/lib`, you will need to specify the path to the library via the option `-L<path>`.

4 Performance Measurements

The Figure 1 shows performance measurements of the transactional memory provided by OSS. The results do not illustrate the overall performance of OSS rather the conflict and network related penalties for a best and worst case scenario. Therefore, measurements have been done with all optimizations (local commits, linked transactions, consistency domains etc.) turned off. In the worst case scenario all peers concurrently increment a common variable stored in the transactional memory, in the best case scenario each peer increments its own variable. The best case scenario (private variable) solely shows the network overhead produced by the commit protocol. The worst case additionally causes penalties due to a high conflict rate which results in transaction aborts and restarts.

For the measurements we have used our P2P commit protocol with two different token mechanism for transaction serialization. The token was passed among the peers either by a dedicated coordinator or P2P based approach. Furthermore, we have expanded the transaction duration and pause between to successive transactions to simulate real live applications. The red and green lines show the overall transaction thoughput by using the coordinated and p2p based token passing mechanism. The black line shows the maximum theoretical throughput based on the transaction time and pause, presumed no conflicts occur.

The diagrams in the left column show the negative impact of network communication, but nevertheless growing of overall transaction throughput. To improve OSS' performance the implemented optimizations aim at exploiting locality to prevent unnecessary network communication (e. g. local commits) and linked transactions to hide the network latency by acquiring the token in the background while starting the next transaction. The right column shows, that the programmer must be aware transactional conflicts and performance issues. So he has to optimize its program to get a low conflict rate.

OSS Appendix

A Support

Please visit the OSS website at the University of Duesseldorf for contact to the developers and further information on OSS. The XtreemOS bugtracker is available at SourceForge.

B OSS Configuration Options

This chapter describes all configuration options of OSS, which affect compilation of OSS. The configuration dialog is accessible via

```
$> make menuconfig
```

The configuration dialog is modelled after the Linux kernel configuration dialog. Press the *enter* key to select an item and press the *space* key to toggle a selection. Use the cursor keys to navigate between items, to exit from a menu or to display a help text for the selected item.

B.1 Debugging

Library developers can configure a number of debugging options.

B.1.1 debug level for whole build process

Selects a global level for debug output unless this value is overridden by a *per-file debug level*.

B.1.2 debug glib

Enables debug output for glib related operations.

B.1.3 debug networking

Enables debug output for network related operations.

B.1.4 Per-file debug levels

Allows a fine granular debug level selection for specific source files.

B.2 Code generation

The binary code of the OSS library can be compiled for different processor architectures.

B.2.1 Processor Architecture

Defines the processor architecture for which OSS is compiled. OSS supports the following architectures:

- AMD64/Intel64 architecture (64-bit operating system provided)

- I686 architecture (32-bit or 64-bit supported)

The usage of a 32-bit OSS version in an 64-bit XtreemOS system requires the installation of a 32-bit compatibility layer (32-bit libraries).

B.3 Library Interface

In addition to the base functions which the OSS library always exports, a number of functions are tagged as optional or experimental.

B.3.1 oss_mmap

Exports the command `oss_mmap` which creates an object from the content of a file, and the commands `oss_munmap` and `oss_msync`, which will unmap and synchronize object and file in a future version of OSS.

B.3.2 oss_sync/oss_push/oss_pull

Provides three additional calls for explicit synchronization of weakly consistent objects. These calls have not yet been implemented in the current OSS release.

B.3.3 oss_nameservice_get/oss_nameservice_set

Exports the functions of the nameservice to the API. This allows applications to use the internal nameservice of OSS.

B.3.4 nameservice consistency

Selects the consistency model of nameservice entries. Some internally defined entries are always handled according to strong consistency.

B.3.5 miscellaneous debug functions

Exports further debugging functions to the API *(see oss.h)*.

B.3.6 unstable library interface

Exports functions for retrieving the own node id, the number of nodes, and setting the number of nodes participating on transactional consistency to the API *(see oss.h)*. These functions are used for debugging purposes only. *(Without claim to be still available in future versions of OSS)*.

B.3.7 oss_wait

Enables distributed barriers for strong and transactional consistency.

B.3.8 hashmap

Exports the functions for managing a hashmap of shared objects. This allows applications to use the internal hashmap implementation of OSS.

B.4 Communication

The OSS library interface deliberately does not specify how nodes are interconnected. Internal to the OSS library, node interconnection can be implemented in several ways.

B.4.1 Overlay Routing

Allows configuration of overlay network related options. Unless selected, the node network is fully meshed; however, connections are established on demand.

B.4.2 Superpeer Network

Enables routing of OSS messages in the overlay network **[Experimental]**.

B.5 Monitoring

For performance measurements as well as for automatic reconfiguration during runtime, the library contains a monitoring subsystem. The subsystem allows the library developer to intersperse monitoring events in the source code. Different handlers can be attached to monitoring events by specifying their names in the configuration dialog. The default no-op handler is called `null`. The `count` handler simply counts the number of events. The `printf` handler prints the events seen immediately, including the source code location and a custom pointer value. The `latency` handler measures the duration of events, whereas the `slist` handler accumulates the pointer values of all events seen in a singly-linked list.

B.5.1 monitoring

Enables monitoring of several OSS internal operations for statistics and dynamic reconfiguration.

B.5.2 log monitor data to file

Enables logging the monitoring data to a file. Unless selected, the monitors print statistics to standard output.

B.5.3 periodic dump

Time interval in seconds of periodic monitoring data dump.

B.5.4 short log

Reduces verbosity of logging information output.

B.5.5 clock

Selects the time source for the monitoring subsystem. Use `clock_gettime` to measure elapsed time in micro-seconds, use `rdtsc` to measure CPU clock cycles, or use `gettimeofday` to measure elapsed time in micro-seconds using a POSIX-compliant call.

B.5.6 object_mmap

Monitors object mappings.

B.5.7 object_alloc

Monitors object allocations.

B.5.8 object_free

Monitors object deallocations.

B.5.9 read_fault

Monitors detected read accesses.

B.5.10 write_fault

Monitors detected write accesses.

B.5.11 read_access

Monitors read accesses evoked by a test application that has been prepared to announce read accesses.

B.5.12 write_access

Monitors write accesses evoked by a test application that has been prepared to announce write accesses.

B.6 Memory allocator

OSS supports different memory allocators.

B.6.1 mspace allocation from dlmalloc

Enables the mspaces memory allocator. The mspace allocator is a general-purpose allocator, which is very reliable and versatile.

B.6.2 millipage implementation

Enables the millipage memory allocator. This allocator concentrates multiple objects allocated on different memory pages on one physical page frame. The millipage allocator is well suited for allocations of small objects if another allocator might induce false sharing.

B.6.3 simple list allocator

Enables the simple first-fit memory allocator. The simple list allocator is very fast for allocations, but frequent deallocations may induce external fragmentation.

B.6.4 replica management

Currently, replication is handled using invalidations and requests for invalid objects. A future release of the library will include a full-featured replica management that handles a combination of object invalidations and updates.

B.6.5 diff computation and transfer

Diff computation and transfer will speed up object accesses, but it is still under development and not included in the current release.

B.7 Applications

B.7.1 build raytracer

Builds the raytracer application, shipped with OSS.

B.7.2 build wissenheim

Builds the wissenheim application out of OSS. This option is only intended for debugging purposes regarding Wissenheim over OSS. Wissenheim on XtreemOS comes with its own build system.

B.8 Remote installation

UDUS infrastructure specific options *(not for public usage)*.

References

[MMSS08] Marc-Florian Müller, Kim-Thomas Möller, Michael Sonnenfroh, and Michael Schöttner. Transactional data sharing in grids. In *PDCS 2008: International Conference on Parallel and Distributed Computing and Systems 2008*, 2008.

[MMSS09] Kim-Thomas Möller, Marc-Florian Müller, Michael Sonnenfroh, and Michael Schöttner. A software transactional memory service for grids. In *ICA3PP 2009: International Conference on Algorithms and Architectures for Parallel Processing*, 2009.

[RMS10] Kim-Thomas Rehmann, Marc-Florian Müller, and Michael Schöttner. Adaptive conflict unit size for distributed optimistic synchronization. In *The Sixteenth International Conference on Parallel Computing (Euro-Par 2010)*, Ischia, Naples, Italy, 8 2010.

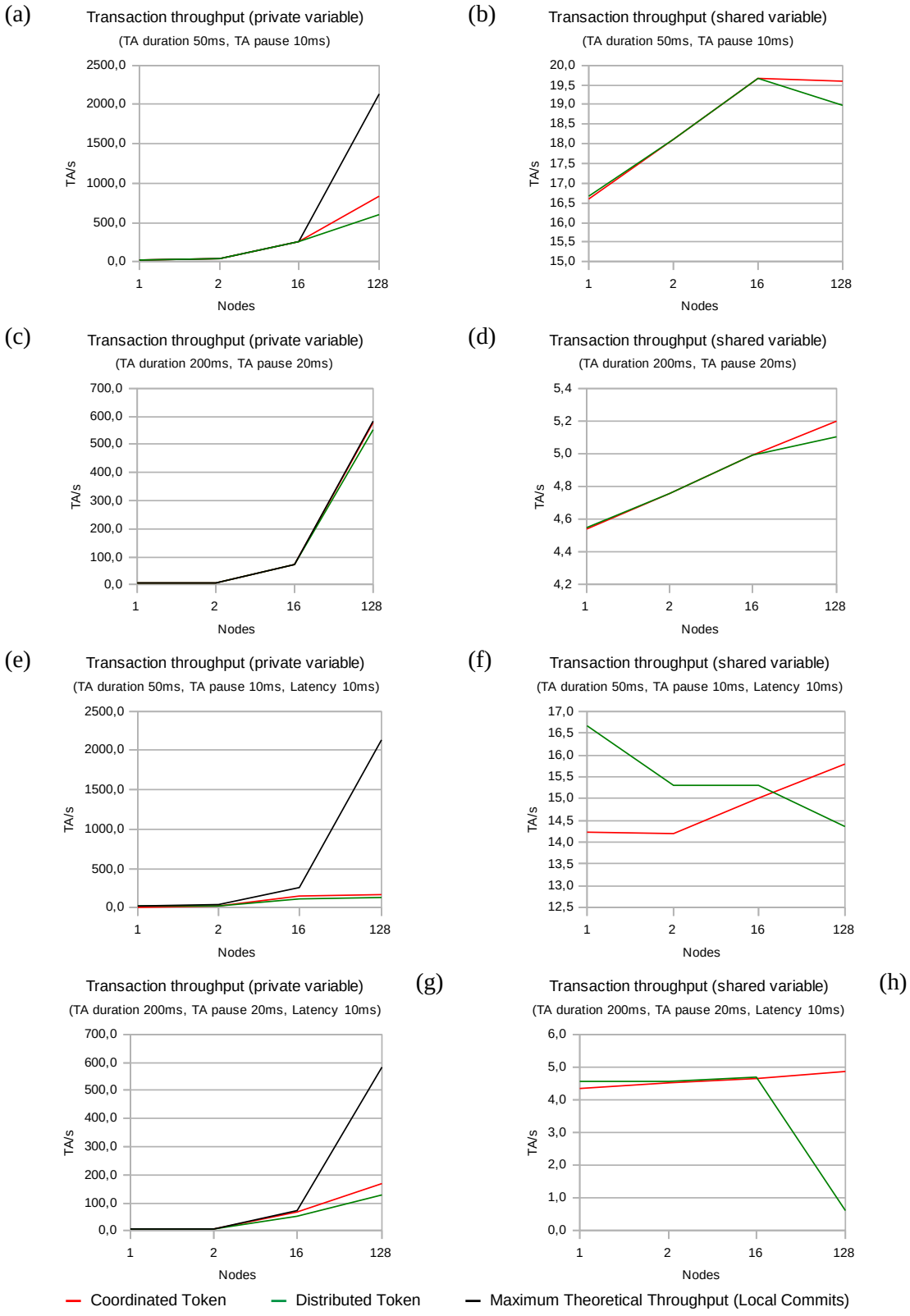

Figure 1: Performance measurements of conflicting and non-conflicting variable incrementations