Summer Research Report
Benchmarks on the VSoC Simulator

August, 2013

Advised by: Tali Moreshet

Callen Rain & Peng Zhao

Contents

1 VSoC **3**

1.1 Installation . . . 3

1.2 Structure . . . 5

1.3 Increase TCDM Size . . . 5

2 Application Support **7**

2.1 Shared Memory Allocation . . . 7

2.2 Initialization Flags . . . 8

2.3 Global Pointers . . . 8

2.4 Barriers . . . 9

2.5 Locks and Transactions . . . 9

3 Benchmarks **10**

3.1 Compilation . . . 10

3.2 Execution . . . 11

3.3 Summaries . . . 11

3.3.1 Hello World . . . 11

3.3.2 Count . . . 11

3.3.3 Matrix Multiplication . . . 11

3.3.4 C5 . . . 12

3.3.5 Patricia . . . 12

3.3.6 Skiplist . . . 13

3.3.7 Redblack . . . 13

3.3.8 K-Means . . . 14

3.3.9 Vacation . . . 14

3.3.10 Genome . . . 15

3.3.11 Labyrinth . . . 16

3.3.12 ScalParC . . . 17

1 VSoC

VSoC (Virtual System on Chip) is a virtual platform developed by our collaborators at University of Bologna. It is capable of simulating a cluster-based many-core architecture at a cycle-accurate level. We have been working on a beta version called vsoc-beta.

1.1 Installation

Step 1: Download the beta version of VSoC from the following website:

http://www-micrel.deis.unibo.it/virtualsoc/release/vsoc-beta/

Step 2: Extracting the archive in your home directory:

```
$ tar zxvf vsoc-beta.tgz
```

will create the vsoc-beta folder. We recommend using a Ubuntu virtual machine, because the simulator has not been tested on other platforms.

Step 3: *Install SystemC 2.2.0*

You can skip this step if you are using Dimitra Papagiannopoulou's virtual machine, because SystemC has already been installed on it. Otherwise, SystemC 2.2 source code can be downloaded from:

http://www.accellera.org/downloads/standards/systemc/

Here are the commands needed to install SystemC. You should replace the `$VSOC_ROOT_DIR` environment variable in line 3 with your vsoc-beta directory.

```
$ tar xzf systemc-2.2.0.tgz
$ cd systemc-2.2.0
$ patch -p1 < $VSOC_ROOT_DIR/systemc-2.2.0.patch
$ mkdir objdir
$ cd objdir
$ cd ../configure
$ make
$ make install
```

Step 4: *Install TLM 2.0.1*

You can skip this step if you are using Dimitra Papagiannopoulou's virtual machine, because SystemC has already been installed on it. Otherwise, SystemC 2.2 source code can be downloaded from:

http://www.accellera.org/downloads/standards/systemc

After extracting the `TLM-2.0.1.tgz` archive no installation is required.

Step 5: *Environment variables*

Environment variables are variables that we defined in shell. We can refer to them later by adding a `$` before the variable's name. VSoC uses a couple of environment variables, so we need to set them up before the actuall building process.

These environment variables are already defined in `vsoc-beta/SOURCE`, except two of them: the path to SystemC, and the path to TLM. Open `vsoc-beta/SOURCEME`, and set the following lines to the directories where you just installed SystemC and TLM:

- `SYSTEMC=/path/to/systemc-2.2.0`
- `TLM=/path/to/TLM`

Spaces are not allowed before and after '='. After doing this, run the following command to define all environmetn variables:

```
$ source SOURCEME
```

However, these values will be lost once we close the terminal. So we will need to run `source SOURCE` every time we open a new terminal. Is there a way to do it once and for all? The answer is yes. In Linux, there is a hidden file called `.bashrc` in the home directory. To view its content, type:

```
$ gedit ~/.bashrc
```

The `.bashrc` script is executed every time we open a new terminal. So if the environment variables are defined here, they are defined for ever. See Section 3.1 for more details.

Step 6: *The Actual Building Process*

NOTE: The build process described here has been successfully tested on Ubuntu Linux 10.04 (32 and 64 bit) and on CentOS 6.3 (32 and 64 bit) with `gcc 4.4.3`.

The build process of VirtualSoC is completely automated, it also builds Sim- SoC and DRAMSim. The SimSoC library requires the MPFR library (a C library for multiple-precision floating-point computations with correct round- ing). For Debian based systems, run the following command to install the MPFR library:

```
$ sudo apt-get install libmpfr-dev
```

In case you are using a different Linux distribution, refer to you distributions application manager to install the correct package.

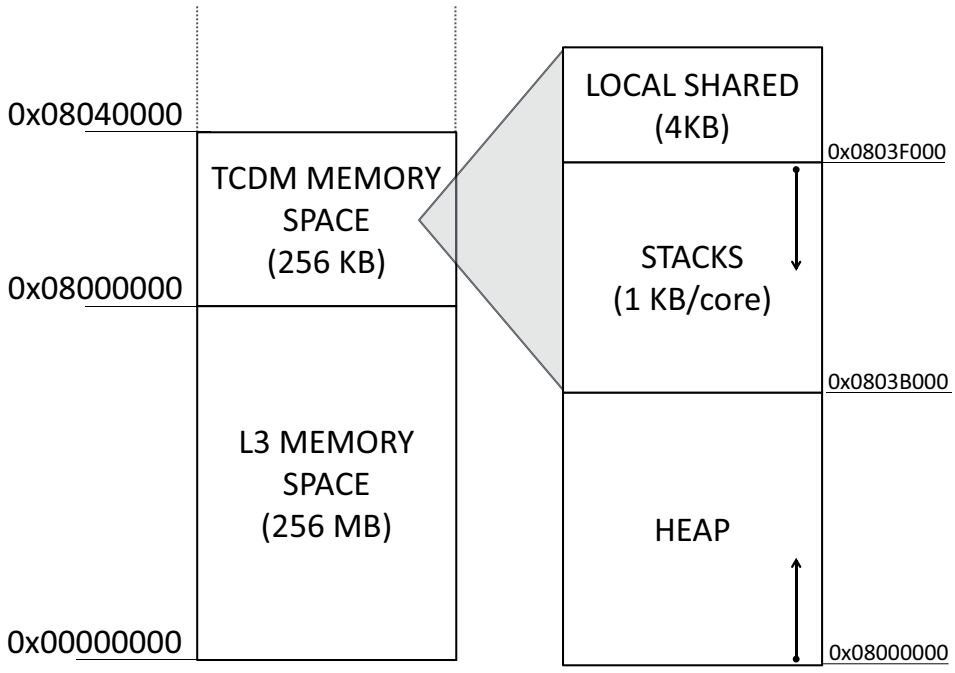
Once all requirements are satisfied, go to the `vsoc-beta/scripts` directory and run:

```
$ vsoc_build -a
```

NOTE: the vsoc build script with the option -a (all) builds the Virtu- alSoC Virtual Platform as well as the 3rd party libraries. If invoked without any option only VirtualSoC is built.

1.2 Structure

VSoC implements a clustered, network-based, many-core architecture. The documentation vsoc.pdf [1] explains its structure in full detail. Here we will only talk about the memory architecture of VSoC, since it's what we've been working on.



The picture above shows the memory architecture of VSoC. The lower end is filled by L3 memory (not supported yet), and TCDM comes right after it. TCDM is short for Tightly Coupled Data Memory. It's similar to a shared L1 cache, but with many banks. Each bank by itself supports memory and write, which means TCDM allows many memory accesses at the same time, as long as they belong to different banks. See vsoc.pdf [1] for more details.

The TCDM memory space can be further divided into the stacks, the heap, and the local shared region. The heap is where we dynamically allocate memories, and the stack keeps track of local variables and function calls. It important to make sure that the heap and stack don't overlap. In other words, we should never allocate memory in the stack, otherwise we may get some weird errors.

1.3 Increase TCDM Size

In VSoC, the default TCDM size is defined to be 0x40,000 Bytes (256 KB). Unfortunately, there are some particularly "hungary" applications that require more memory to run. If we run these applications with the default TCDM size, VSoC will run out of shared memory and crash.

To solve this problem, we will need to increase the size of TCDM. Here is a step-by-step procedure:

1). Open vsoc-beta/src/core/config.h , and change CL_TCDM_SIZE (around line 60) to the desired value.

- Skiplist on a 8-core system needs 0x00080000 Bytes (512KB) to run.

- Skiplist on a 16-core system needs 0x000b0000 Bytes (768KB).
- RedBlack on a 16-core system needs 0x00080000 Bytes (512KB).
- Full version of Patricia needs 0x00400000 Bytes (4MB).

See Sections 3.3.5, 3.3.6, and 3.3.7 for more details.

2). Open vsoc-beta/apps/support/simulator/vsoc.ld

Change the value of STACK_TOP (line 6) to 0x08000000 + CL_TCDM_SIZE - 0x1004.

Change the value of ORIGIN (line 10) to 0x08000000 + CL_TCDM_SIZE - 0x1000.

For example, if CL_TCDM_SIZE is set to be 0x00080000, then STACK_TOP should be 0x0807effc, and ORIGIN should be 0x0807f000.

The variables STACK_TOP and ORIGIN define the start of the stack. They need to change together with TCDM size to make sure that the stack is always located at the end of shared memory. See Section 2.1 for more details.

3). Go to the vsoc-beta directory, and run the following command:

```
$ source SOURCEME
```

4). Go to the vosc-beta/scripts directory, and run:

```
$ vsoc_clean
$ vsoc_build -a
```

5). Clean and recompile the benchmark (to make changes in vsoc.ld effective), and run it on the simulator.

2 Application Support

The application support section details the features that we added to VSoC to support running the benchmarks the we ported from MPARM. The features below will allow you to allocate shared memory and manage other details of parallelization between cores.

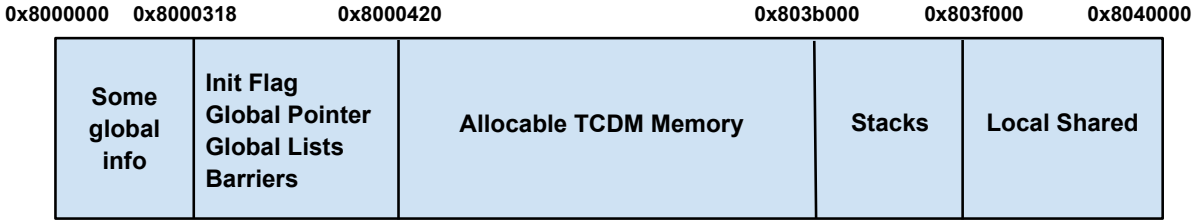
2.1 Shared Memory Allocation

The benchmarks we ported from MPARM require shared memory in order to execute parallel code. We built a shared memory allocation system (shmalloc) that handles these operations. The shmalloc code is located in `shmalloc.c` in the application support directory.

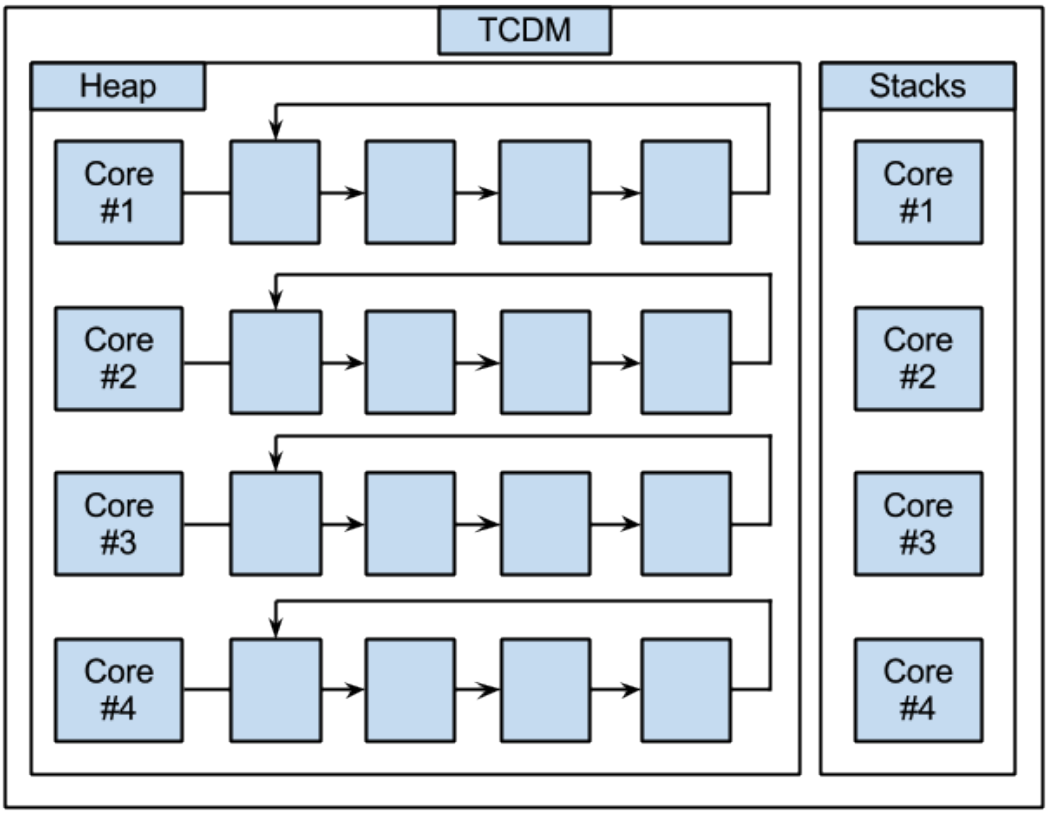
Our original design for shmalloc was to have a single pointer that would be initialized at the beginning of the benchmark. The pointer would start at the beginning of allocable shared memory. An application could request memory of a certain size, and the shmalloc function would increment the memory pointer and return a pointer to the allocated memory. There was no opportunity to free memory. Because this system introduced some possible conflict among cores in accessing the shared TCDM, we had to use a lock on the memory. However this approach does not work when transactions are used instead of locks. There would be no way to roll back changes.

Later we were finding that many of the benchmarks were running out of memory quickly because the TCDM would fill up and we had to disable all of the freeing functions in the benchmarks. We decided to implement a more elaborate system that would be capable of freeing memory. This system operates on a large circular linked list of the available free memory. It initializes the free list to be the size of the entire allocable heap in initialization. A set of counters keeps track of how many free memory chunks of certain size ranges are available for allocation. When a request comes through for a certain memory size, the system can either scan the list for the best fit, or take the first fit that is large enough and split it up. Splitting memory excessively increases fragmentation and is generally more inefficient, but scanning through a large free list is also inefficient. Our system checks the counters to see if a small best fit even exists and then will look for it if it does exist. Otherwise, it will split a larger chunk of memory.

The graphic below details where the allocable shared memory in the heap starts and ends, and where the per-core stacks are located.



We looked into the version of shmalloc that was implemented for MPARM because it worked for the previous transactional memory research that was done and operated without locks. We found that each of the cores had a free list for its own allocation. This avoids allocation conflicts so that no locks are needed. After implementing a similar system, we found that many of the benchmarks would run out of memory because the single core doing the initialization and more of the allocations wouldn't have enough space. We ended up giving core 0 most of the memory expecting that it will be doing the initial allocations. A diagram of this system is provided below.



2.2 Initialization Flags

For many of the benchmarks, certain operations need to be executed by one core in the beginning of the program. Often this core needs to allocate memory for shared data structures and initialize other application support features. An initialization stage can be written into a benchmark by checking the processor id and only letting one processor run the section of code.

To avoid letting the other cores start work on the main portion of the benchmark, the function `WAIT_FOR_INITIALIZATION()` can be used. This function watches a specific portion of memory and returns when it equals a predetermined flag value. When the initializing processor finishes the initialization stage, it will set the flag on that piece of memory and all the processors can move forward with valid data structures. To set the flag, the function `INITIALIZATION_DONE()` is used. More details about the implementation of the initialization phase can be found in `app_support.c`.

2.3 Global Pointers

Since most of the cores are simply waiting while one core carries out the steps of initialization, functions through the system are needed to provide the other cores access to the shared global data structures. There is a fixed memory location that contains this pointer. When the initializing processor would like to make a global pointer out of a newly allocated piece of memory, it must call `make_global_pointer()` this will set the global memory location to point to this memory.

When the associated processors need to retrieve the global pointer after initialization is complete, they

must call `get_global_pointer()`. This function will return the contents of the global pointer memory location. All cores will now have access to the same global pointer.

2.4 Barriers

Barriers are checkpoints in the benchmarks that force all the cores to all reach a point in the code before they can continue execution. This feature was already implemented in the semaphore memory in VSoC, but we simply added similar functions to those in MPARM that allow for the use of barriers without large changes to the benchmark code from MPARM.

Barriers must be initialized with a call to `BAR_INIT()` by one core in initialization. This will create the counters that accommodate the underlying functionality of the barriers. When the programmer would like to insert a barrier in the code, they must call `BARRIER()` with the number of cores as an argument.

2.5 Locks and Transactions

The final piece of parallel application support is the functionality of the atomic operations protected with locks and transactions. There are two ways to include these in application code. They can be specified manually with `WAIT()` and `SIGNAL()` for locks and `START_TRANSACTION()` and `END_TRANSACTION()` for transactions.

There is also a switch in `appsupport.c` that can be turned on and off easily to simply the debugging of code that requires transactions. Using this switch, the programmer should insert functions `BEGIN_ATOMIC()` and `END_ATOMIC()` and set the flag to indicate if transactions or locks are being used.

3 Benchmarks

3.1 Compilation

In order to compile each of the benchmarks, configure the Makefile to fit the system that you are running on. Check the environment variables and the compiler. If the compilation of the benchmark fails because a particular environment is not set, you can either add lines to the **.bashrc** file which will load the variables when a particular bash shell is initialized. A sample section from our **.bashrc** file is shown below. Then run "make". This will create an executable binary file called **app.exe** in **o-optimize/** which is then made into 4 TargetMem.mem files in the main application directory. These are fed into the simulator when the benchmark is run.

Listing 1: Sample .bashrc file for environment variables

```
#####################################################################
#Following is for VSoC

#root directory of the Vsoc package
export VSOC_ROOT_DIR=/home/student/vsoc-beta

#SystemC
export SYSTEMC=/home/student/systemc-2.2.0
export SYSTEMC_HOME=${SYSTEMC}
export SYSC_TARGET_ARCH=linux
#echo $SYSC_TARGET_ARCH
export SC_SIGNAL_WRITE_CHECK=DISABLE

#TLM
export TLM=/home/student/TLM
export TLM_HOME=${TLM}
export TARGET_ARCHI=linux

# --- VirtualSoC Virtual Platform ---
export VSOC_SRC_DIR=${VSOC_ROOT_DIR}/src
export VSOC_BUILD_DIR=${VSOC_ROOT_DIR}/build
export VSOC_BIN_DIR=${VSOC_ROOT_DIR}/bin
export VSOC_SCRIPTS_DIR=${VSOC_ROOT_DIR}/scripts
export VSOC_APP_DIR=${VSOC_ROOT_DIR}/apps
export VSOC_DOC_DIR=${VSOC_ROOT_DIR}/doc
export PATH=${PATH}:${VSOC_SCRIPTS_DIR}

#Vsoc Binaries:
export PATH="${VSOC_BIN_DIR}:$PATH"

#Third-party software
export MICSIM_TRD_PARTY_DIR=${MICSIM_ROOT_DIR}/3rd_party
export LIB_SIMSOC=${MICSIM_TRD_PARTY_DIR}/src/LIBSIMSOC
export LD_LIBRARY_PATH=${LD_LIBRARY_PATH}:${MICSIM_TRD_PARTY_DIR}/lib

#UniBo ARM-ELF-GCC
export PATH=/home/student/arm-elf-gcc/install32/bin/:${PATH}

#libGOMP
```

```
40 export LIBGOMP_PATH=${VSOC_APP_DIR}/libgomp

   #ARM RTEMS
   export PATH=/home/student/TMSIM/rtems-cross/bin/:${PATH}
```

3.2 Execution

To run the benchmark on the simulator, the simulator first needs to know which benchmark you want to run. Move into the v**soc-beta/bin/** directory and run **vsoc_set_app**, which will provide a dialog in which you can choose a benchmark as the application that you want to run. Make sure to choose 1 binary for each cluster you would like to run (these benchmarks have only been tested with 1 cluster and 16 cores). When this script finishes, you can run the main VSoC simulator with a few runtime options. More information on these options can be found in the documentation for VSoC, found in **doc/**. A sample execution command would be:

```
./vsoc.x -c16 --intc=c
```

which would run a benchmark with 16 cores and 1 cluster.

3.3 Summaries

3.3.1 Hello World

This benchmark simply prints out a "Hello World" statement to the screen. It was included in the version of VSoC that we downloaded and consisted of one line of code that printed a line of code to the screen.

In the summer of 2013, Peng and Callen (Swarthmore College) made a few changes to the benchmark to test two of the application support functions that we made.

3.3.2 Count

Count is a very basic benchmark that was obtained from Iris Bahar's Low Power VLSI System Design class at Brown University. The benchmark is included as an example in a shared memory synchronization lab for the class. It originally ran on the MPARM hardware simulator. Students are shown the unparallelized and the parallelized versions of the benchmark and then asked to perform similar modifications to another application.

In the summer of 2013, we decided to port the benchmark from this lab application on MPARM to the VSoC system we were working on. It was useful as a simple test of the locks implemented in VSoC (it turned out that these locks were not completely functional).

The functionality of the benchmark is simple. The cores take turns acquiring a lock for a shared counter. They increase it by one and release the lock. The final value of the counter is correct if none of the cores have interfered with each other's atomic increases.

3.3.3 Matrix Multiplication

Along with hello world, matrix mult is a benchmark that came with the VSoC system from University of Bologna.

The operation of the benchmark is the multiplication of two arrays. These arrays are defined in matrix.h. Originally they were generated with MatLab and are sized 16x16.

Matrix multiplication isn't terribly useful for synchronization tests because the application can simply break up the matrix into pieces and assign each core to have their own chunk. They can store a local copy of the arrays and only operate on the part they are assigned to. Since this arrangement doesn't use locks, the matrix multiplication benchmark is more useful as a test of the VSoC installation but not in the testing of transactional memory.

3.3.4 C5

The C benchmark was developed as a microbenchmark was developed by the team from Brown University, Swarthmore College, and University of Bologna in transactional memory research. It utilizes basic synchronization on a scale much smaller than that of the STAMP benchmarks or even a benchmark like patricia.

The benchmark functions by performing operations on shared and local arrays. These operations are simple integer manipulation, and have no practical relevance.

These operations are split into 4 parts in the C benchmark. Each core has a local array and there is also an array shared by all the cores. The local array just simulates work that doesn't require synchronization functionality. The shared array contains several vectors that overlap on each other. Constants such as the size of the vectors, number of overlap elements, and number of iterations performed are defined at the beginning of the testbench file.

The locks required for the C benchmark are specifically assigned to certain areas of the array. The layout for the lock setup is shown in the main source file.

3.3.5 Patricia

The original patricia benchmark was written by Matt Smart from The University of Michigan. His description of the functionality of the program is provided below:

> This code is an example of how to use the Patricia trie library for doing longest-prefix matching. We begin by adding a default route/default node as the head of the Patricia trie. This will become an initialization functin (pat_init) in the future. We then read in a set of IPv4 addresses and network masks from "pat_test.txt" and insert them into the Patricia trie. I haven't yet added example of searching and removing nodes.

This version of Patricia was then ported to the MPARM system simulator used at the Univeristy of Bologna in research of Transactional Memory Systems. This benchmark is useful because it contains several critical sections of code. Researchers can test different memory configurations in muti-core and many-core systems using the patricia benchmark because all the cores will have to share a single data structure.

Finally, in the summer of 2013, Peng and Callen (Swarthmore College) ported thisbenchmark to the Virtual System on Chip (VSoC) simulator used by Brown University and Swarthmore College in transactional memory research for many-core clustered systems.

**Important:*

We reduced the input buffer size in this version of Patricia, because VSoC's default TCDM size doesn't support large input buffers.

The input buffer size in Patricia is defined by two variables: INITSIZE, which is the number of nodes we pre-populate in the patricia trie, and BUFSIZE, the number of IP addresses we lookup and insert in the tree. Larger input buffer leads to a larger Patricia trie, which in turn needs more memory to allocate. The Patricia benchmark in MPARM had INITISIZE = 1024, and BUFSIZE = 4096. However, this specification needs a 4MB TCDM to run, while the default TCDM size is only 256KB.

There are two solutions to this problem. We could reduce the input buffer size. If we decrease INITSIZE to 128, and BUFSIZE to 512, then Patricia runs well on VSoC. We could also increase the size of TCDM. See section 1.5 for more details.

3.3.6 Skiplist

The original skiplist benchmark was published on http://epaperpress.com as a sorting and searching example. The descriptions can be found here:

http://epaperpress.com/sortsearch/skl.html

This version of skiplist was then ported to the MPARM system simulator used at the Univeristy of Bologna in research of Transactional Memory Systems. This benchmark is useful because it contains several critical sections of code. Researchers can test different memory configurations in muti-core and many-core systems using the patricia benchmark because all the cores will have to share a single data structure.

Finally, in the summer of 2013, Peng and Callen (Swarthmore College) ported thisbenchmark to the Virtual System on Chip (VSoC) simulator used by Brown University and Swarthmore College in transactional memory research for many-core clustered systems.

**Important:*

In this benchmark, every core has a private skiplist, and together they have a shared skiplist. Every skiplist needs 0x9010 bytes of shared memory, which means the 8-core version needs 0x9010 * (8+1) = 0x51090 bytes, and the 16-core version needs 0x9010 * (16+1) = 0x99110 bytes. However, the default TCDM size in VSoC is only 0x40000 bytes. So we need to increase the TCDM size in order to run skiplist on 8 or 16 cores. See section 1.5 for more details.

There are also some parameters need to be set inside the benchmark, including PERCENT_LOOKUP, PERCENT_INSERT, and PERCENT_DELETE. They represent the percentage of lookup, insert, and delete operations to perform. The default values are lookup = 90%, insert = 9%, and delete = 1%. They can be found and set at the beginning of testbench.c, and their values will affect the size of critical sections.

3.3.7 Redblack

The original redblack benchmark was published on http://epaperpress.com as a sorting and searching example. The descriptions can be found here:

http://epaperpress.com/sortsearch/skl.html

This version of redblack was then ported to the MPARM system simulator used at the Univeristy of Bologna in research of Transactional Memory Systems. This benchmark is useful because it contains several critical sections of code. Researchers can test different memory configurations in muti-core and many-core systems using the patricia benchmark because all the cores will have to share a single data structure.

**Important:*

In this benchmark, every core has a privite redblack tree, and together they have a shared redblack tree. Every redblack tree needs 0x5010 bytes of shared memory, which means the 16-core version needs 0x5010 * (16+1) = 0x55110 bytes. However, the default TCDM size is only 0x40000 bytes. So we need to increase the TCDM size in order to run redblack on 16 cores. See section 1.5 for more details.

There are also some parameters need to be set inside the benchmark, including PERCENT_LOOKUP, PERCENT_INSERT, and PERCENT_DELETE. They represent the percentage of lookup, insert, and delete operations to perform. The default values are lookup = 90%, insert = 9%, and delete = 1%. They can be found and set at the beginning of testbench.c, and their values will affect the size of critical sections.

3.3.8 K-Means

K-means was originally included in Stanford's STAMP benchmark suite for multiprocessor systems. It was then ported to the MPARM simulator, and this file documents the changes necessary to run it on the VSoC simulator developed by researchers from the University of Bologna.

K-means is a clustering algorithms that can cluster a set of vectors into a certain number of groups. For example, the main input to the benchmark contains 64 vectors with 8 elements each. The system clusters them into 8 groups. The test input, found in "goldinput.h", uses 12 objects each with 2 attributes, and clusters them into 2 groups.

The algorithm works in two stages. First, random vectors are chosen from the set as the initial cluster centroids. Then, each vector is assigned to the centroid that is closest to it. The centroid is then recomputed as the point which minimized the sum of squares distance between the centroid and each of the vectors. Then the vectors are reassigned again. These two steps are iterated until no vectors switch centroids and the distance each centroid moves when it is recalculated is reduced to zero.

3.3.9 Vacation

Vacation was originally released as part of the STAMP benchmark suite. It was ported to MPARM in the summer of 2008 by Trilok Acharya.

From the original README,

```
This benchmark implements a travel reservation system powered by a
non-distributed database. The workload consists of several client threads
interacting with the database via the system's transaction manager.

The database is consists of four tables: cars, rooms, flights, and
customers. The first three have relations with fields representing a
unique ID number, reserved quantity, total available quantity, and price.
The table of customers tracks the reservations made by each customer and
the total price of the reservations they made. The tables are implemented
as Red-Black trees."
```

It would be wise to read Trilok's README, found in the app directory, for details on how he ported the original STAMP benchmark to MPARM. In this README, I will only discuss changes that were made by Peng and I to get the benchmark to run on VSoC. The initial port to MPARM was a much, much larger project.

3.3.10 Genome

The original genome benchmark was written by Chi Cao Minh from Stanford University. His description of the functionality of the program is provided below.

```
This benchmark implements a gene sequencing program that reconstructs the
gene sequence from segments of a larger gene.

For example, given the segments TCGG, GCAG, ATCG, CAGC, and GATC, the
program will try to construct the shortest gene that can be made from them.

For example, if we slide around the above segments we can get:

         TCGG
    GCAG
        ATCG
  CAGC
       GATC
 =============
  CAGCAGATCGG

This gives a final sequence of length 11. Another possible solution is:

    TCGG
       GCAG
   ATCG
        CAGC
  GATC
 =============
  GATCGGCAGC

This solution has length 10. Both are consistent with the segments provided,
but the second is the optimal solution since it is shorter.

The algorithm used to sequence the gene has three phases:

    1) Remove duplicate segments by using hash-set
    2) Match segments using Rabin-Karp string search algorithm [3]
        - Cycles are prevented by tracking starts/ends of matched chains
    3) Build sequence

The first two steps make up the bulk of the execution time and are
parallelized.


References
----------

[1] C. Cao Minh, J. Chung, C. Kozyrakis, and K. Olukotun. STAMP: Stanford
```

Transactional Applications for Multi-processing. In IISWC '08: Proceedings of The IEEE International Symposium on Workload Characterization, September 2008.

[2] C. Cao Minh, M. Trautmann, J. Chung, A. McDonald, N. Bronson, J. Casper, C. Kozyrakis, and K. Olukotun. An Effective Hybrid Transactional Memory System with Strong Isolation Guarantees. In Proceedings of the 34th Annual International Symposium on Computer Architecture, 2007.

[3] R. M. Karp and M. O. Rabin. Efficient randomized pattern-matching algorithms. IBM Journal of Research and Development, 1987.

3.3.11 Labyrinth

The original labyrinth benchmark was written by Chi Cao Minh from Stanford University. His description of the functionality of the program is provided below.

```
Given a maze, this benchmark finds the shortest-distance paths between
pairs of starting and ending points. The routing algorithm used is Lee's
algorithm [2].

In this algorithm, the maze is represented as a grid, where each grid
point can contain connections to adjacent, non-diagonal grid points.
The algorithm searches for a shortest path between the start and end
points of a connection by performing a breadth-first search and labeling
each grid point with its distance from the start. This expansion phase
will eventually reach the end point if a connection is possible. A second
traceback phase then forms the connection by following any path with a
decreasing distance. This algorithm is guaranteed to find the shortest
path between a start and end point; however, when multiple paths are made,
one path may block another.

When creating the transactional version of this program, the techniques
described in [3] were used. When using this benchmark, please cite [1].


References
----------

[1] C. Cao Minh, J. Chung, C. Kozyrakis, and K. Olukotun. STAMP:
    Stanford Transactional Applications for Multi-processing. In IISWC
    '08: Proceedings of The IEEE International Symposium on Workload
    Characterization, September 2008.

[2] C. Lee. An algorithm for path connections and its applications. IRE
    Trans. On Electronic Computers, 1961.

[3] I. Watson, C. Kirkham, and M. Lujan. A Study of a Transactional
```

Parallel Routing Algorithm. Proceedings of the 16th International Conference on Parallel Architectures and Compilation Techniques, 2007.

3.3.12 ScalParC

References

[1] Daniele Bortolotti *vsoc-beta/doc/vsoc.pdf*, VSoC beta release.

[2] MicrelLab - DEI *vsoc-beta/doc/appsupport.pdf*, VSoC beta release.

[3] MicrelLab - DEI *vsoc-beta/doc/setup.pdf*, VSoC beta release.

[4] MicrelLab - DEI *vsoc-beta/doc/simulator.pdf*, VSoC beta release.