Atomic Quake – Use Case of Transactional Memory in an Interactive Multiplayer Game Server

Ferad Zyulkyarov

BSC-Microsoft Research Center

17.10.2008

Workshop on Language and Runtime Support for Concurrent Programming
Cambridge UK

BSC Microsoft Research Centre

Outline

- Quake Overview
- Using Transactions for Synchronization
- Runtime Characteristics

Quake

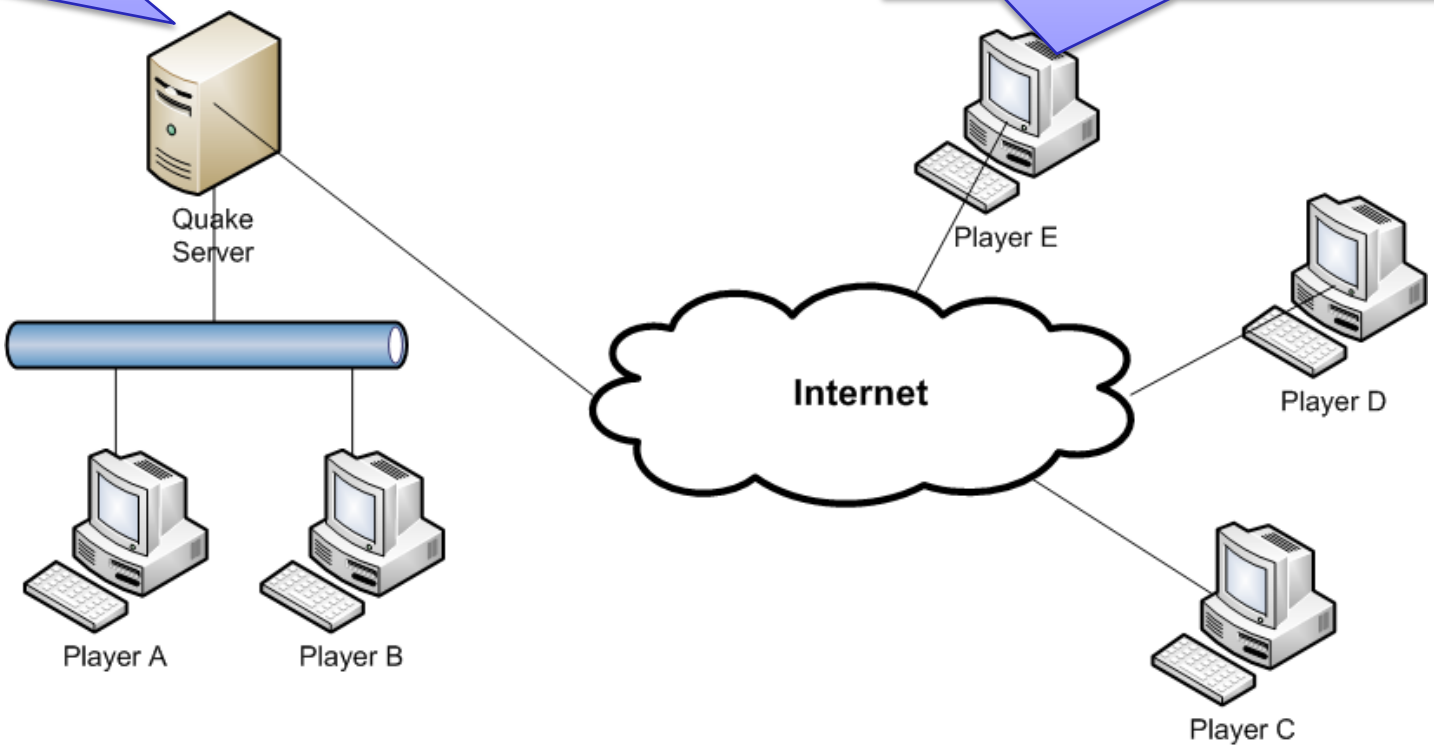
- Interactive first-person shooter game
- Very popular among gamers

- Challenge for the modern multi-core processors

Quake Server

The Server
-Where computations are done
-Maintains the game plot
-Handles player interactions
-Preserves game consistency

The Clients
-Send requests to convey their actions
-Render the graphics and sound

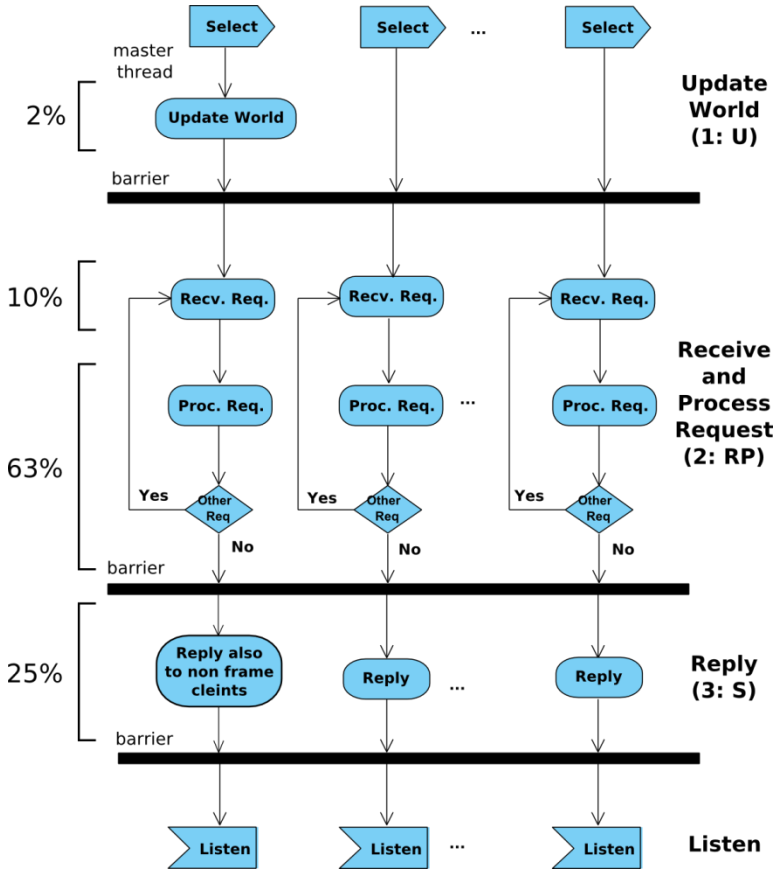


BSC~ Microsoft Research Centre

Parallelization Methodology

[Abdelkhalek, IPDPS'2004]

- Execution divided in 3 phases:
  1. Physics Update
  2. Read client requests & process
  3. Send replies
- Important is processing requests
- Processing requests prepares the next frame



BSC~ Microsoft Research Centre

Processing Requests - The Move Operation

- Motion Indicators
  - angles of the player's view
  - forward, sideways, and upwards motion indicators
  - flags for other actions the player may be able to initiate (e.g. jumping)
  - the amount of time the command is to be applied in milliseconds.
- Execution
  - simulate the motion of the player's figure in the game world in the direction and for the duration specified
  - determine and protect the game objects close to the player that it may interact with
    - Compute bounding box
    - Add all the objects inside the bounding box to list
  - execute actions the client might initiate (shoot, weapon exchange)

Synchronization

Shared Data Structure

- Per player buffer (arrays)
- Global state buffer (array)
- Game objects (binary tree)

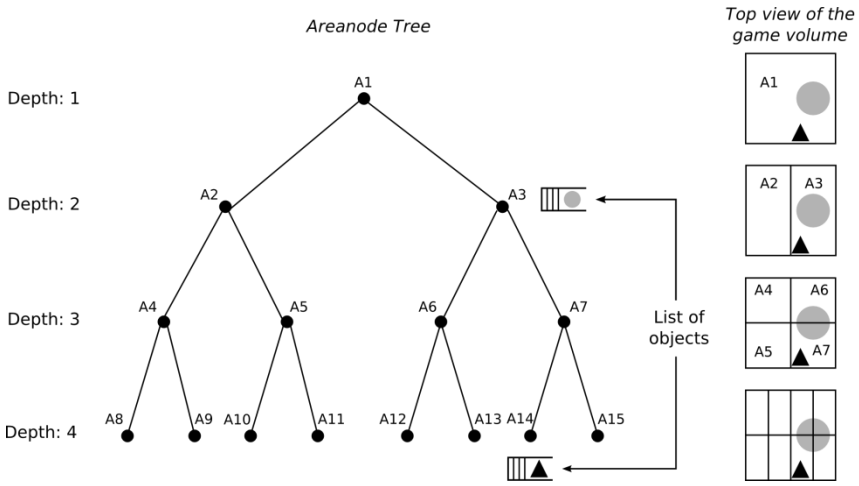
22% overhead
8 threads and 176 clients
[Abdelkhalek, IPDPS'2004]

Synchronization

- Each with a single lock
- Single lock
- Fine grain locking
  - Locks leaves corresponding to the computed bounding box
  - If objects is on the parent lock the object only, but NOT the parent node.

Areanode Tree

- Maps the location of each object inside the virtual world to a fast access binary tree *areanode* tree.
- Children nodes split the region represented by the parent in two halves.



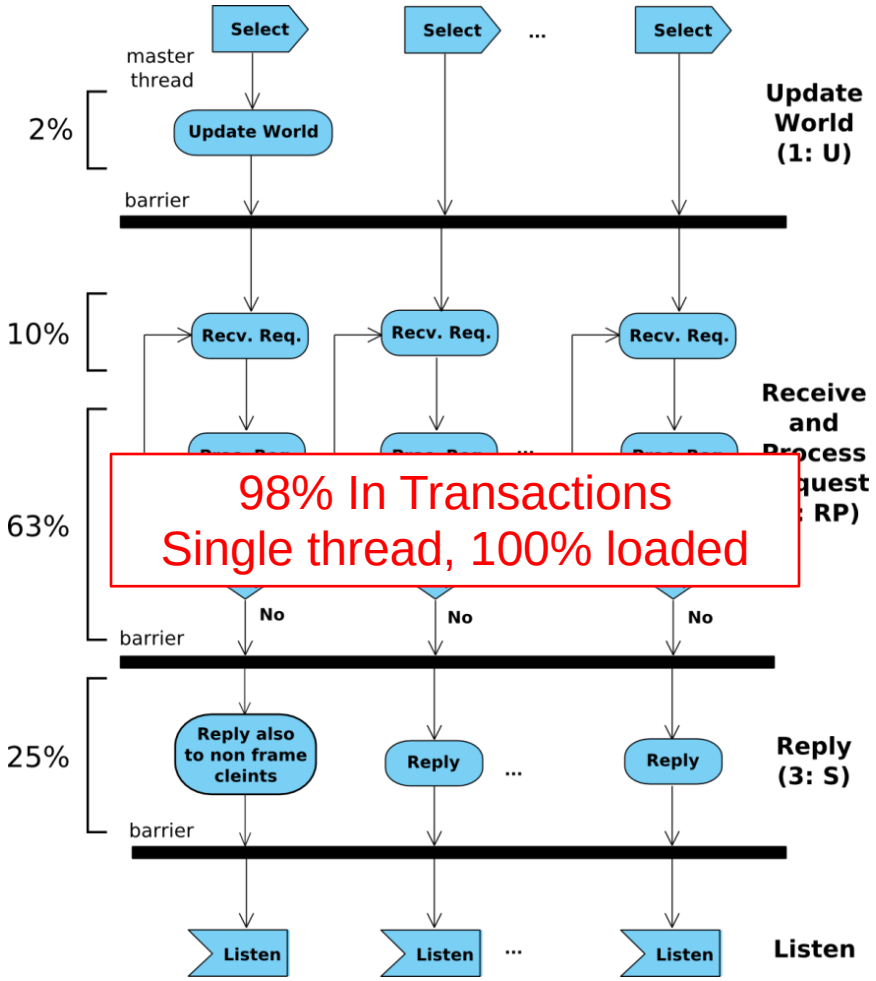
BSC~ Microsoft Research Centre

Using Transactions

- General Overview
- Challenges
- I/O
- Where Transactions Fit
- Error Handling Inside Transactions
- Failure Atomicity

Atomic Quake – General Overview

- 27,400 Lines of C code
- 56 files
- 63 atomic blocks
- Irregular parallelism
  - Requests are dispatched to their handler functions.



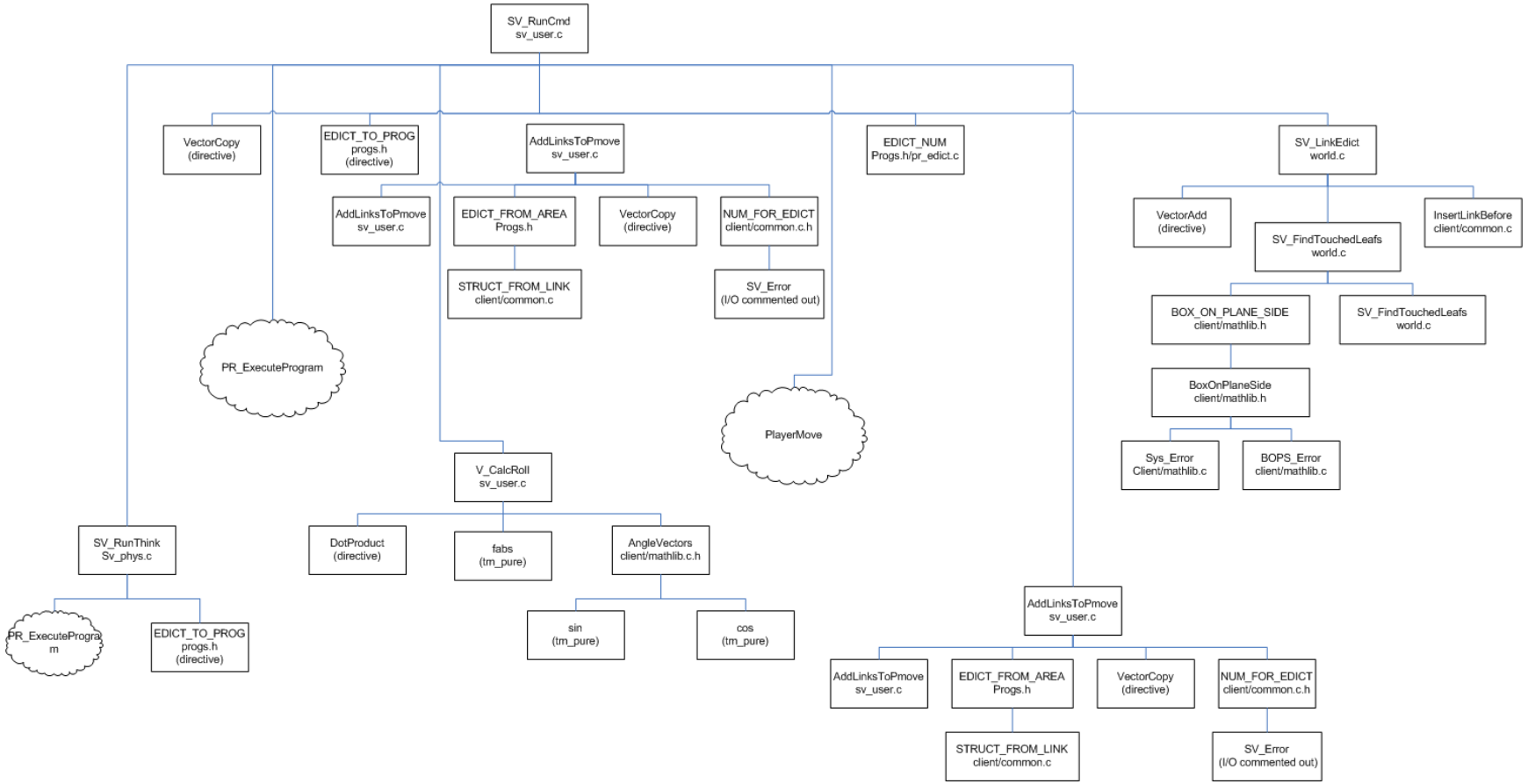
BSC~ Microsoft Research Centre

Complex Atomic Block Structure

- Calls to internal functions
- Calls to external libraries
- Nesting up to 9 levels
- I/O and System calls (memory allocation)
- Error Handling
- Privatization

Example Callgraph Inside Atomic Block

SV_RunCmd is the function that dispatches the execution to the appropriate request handler function.



Nodes drawn with clouds represent calls to functions with call graph as complicated as this one.

Challenge – Unstructured Use of Locks

Locks

```
1 for (i=0; i<sv_tot_num_players/sv_nproc; i++){
2   <statements1>
3   LOCK(cl_msg_lock[c - svs.clients]);
4   <statemnts2>
5   if (!c->send_message) {
6     <statements3>
7     UNLOCK(cl_msg_lock[c - svs.clients]);
8     <statements4>
9     continue;
10  }
11  <stamemnts5>
12  if (!sv.paused && !Netchan_CanPacket (&c->netchan)) {
13    <statmenets6>
14    UNLOCK(cl_msg_lock[c - svs.clients]);
15    <statements7>
16    continue;
17  }
18  <statements8>
19  if (c->state == cs_spawned) {
20    if (frame_threads_num > 1) LOCK(par_runcmd_lock);
21    <statements9>
22    if (frame_thread_num > 1) UNLOCK(par_runcmd_lock);
23  }
24  UNLOCK(cl_msg_lock[c - svs.clients]);
25  <statements10>
26 }
```

Atomic Block

```
1 bool first_if = false;
2 bool second_if = false;
3 for (i=0; i<sv_tot_num_players/sv_nproc; i++){
4   <statements1>
5   atomic {
6     <statemnts2>
7     if (!c->send_message) {
8       <statements3>
9       first_if = true;
10    } else {
11      <stamemnts5>
12      if (!sv.paused && !Netchan_CanPacket(&c->netchan)){
13        <statmenets6>
14        second_if = true;
15      } else {
                                    d) {
                                  > 1) {
21          }
22        } else {
23          <statements9>;
24        }
25      }
26    }
27  }
28  }
29  if (first_if) {
30    <statements4>;
31    first_if = false;
32    continue;
33  }
34  if (second_if) {
35    <statements7>;
36    second_if = false;
37    continue;
38  }
39  <statements10>
40 }
```

Solution
explicit "commit"

Extra code

Complicated
Conditional
Logic

BSC~ Microsoft Research Centre

Challenges – Thread Local Memory

Locks

```
1  void foo1() {
2     atomic {
3        foo2();
4     }
5  }
6
7  __attribute__((tm_callable))
8  void foo2() {
9     int thread_id = pthread_getspecific(THREAD_KEY);
10    /* Continue based on the value of thread_id */
11    return;
12 }
```

The call to the pthread library serializes the transaction.

Hoisted the library call.

Atomic

```
1  void foo1() {
2     int thread_id = pthread_getspecific(THREAD_KEY);
3     atomic {
4        foo2(thread_id);
5     }
6  }
7
8  __attribute__((tm_callable))
9  void foo2(int thread_id) {
10    /* Continue based on the value of thread_id */
11    return;
12 }
```

!!! Thread private data should be considered in TM implementation and language extension. !!!

Challenges - Conditional Synchronization

Locks

```
1 pthread_mutex_lock(mutex);
2 <statements1>
3 if (!condition)
4     pthread_cond_wait(cond, mutex);
5 <statements2>
6 pthreda_mutex_unlock(mutex);
```

Retry [Harris et al. PPoPP'2005]

```
1 atomic {
2    <statements1>
3    if (!condition)
4       retry;
5    <statements2>
6 }
```

Retry not implemented by Intel C compiler.
Left as is in the Quake code.

I/O in Transactions

```
1 void SV_NewClient_f(client_t cl) {
2    <statements1>
3    Con_Printf("Client %s joined the game", cl->name);
4    <statements2>
5 }
```

Solvable with ad hoc.

```
__attribute__((tm_pure)) Con_Printf(char*);
```

```
1 void SV_NewClient_f(client_t cl) {
2    <statements1>
3    Con_Printf("Client %s joined the game", cl->name);
4    <statements2>
5 }
```

Why not *tm_escape*?

```
1 void SV_NewClient_f(client_t cl) {
2    <statements1>
3    tm_escape {
4        Con_Printf("Client %s joined the game", cl->name);
5    }
6    <statements2>
7 }
```

- I/O used to print information messages only to the server console
  - Client connected
  - Client killed
- Commented all the I/O code out.

Where Transactions Fit?

Guarding different types of objects with separate locks.

```
1  switch(object->type) { /* Lock phase */
2      KEY: lock(key_mutex); break;
3      LIFE: lock(life_mutex); break;
4      WEAPON: lock(weapon_mutex); break;
5      ARMOR: lock(armor_mutex); break
6  };
7
8  pick_up_object(object);
9
10  switch(object->type) { /* Unlock phase */
11      KEY: unlock(key_mutex); break;
12      LIFE: unlock(life_mutex); break;
13      WEAPON: unlock(weapon_mutex); break;
14      ARMOR: unlock(armor_mutex); break
15  };
```

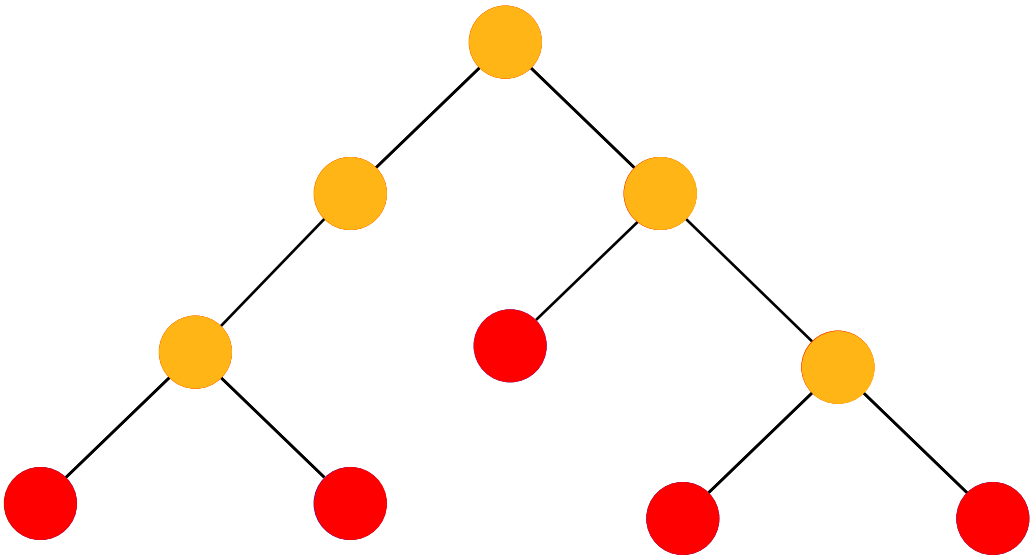
Lock phase. → atomic {

pick_up_object(object); → pick_up_object(object);

} ← Unlock phase.

Algorithm for Locking Leafs of a Tree

- More complicated example of fine grain locking: applied in region-based locking in Quake.



1: Lock Parent

2: Lock Children

3: Unlock Parent

4: If children has children go to 2

RESULT

Code for Lock Leafs Phase

```
while (!stack.is_empty())
{
parent = stack.pop();

if (parent.has_children())
{
        for (child = parent.first_child(); child != NULL; child.next_sibling())
        {       lock(child);
                stack.push(child);
        }
        unlock(parent);
}
// else this is leaf and leaf it locked.
}
```

<UPDATE LEAVES CODE>

/* Follows code for releasing locks as complicate as for acquiring */

Equivalent Synchronization with TM

```
atomic
{
UPDATE LEAVES CODE
}
```

Error Handling Inside Transactions

Approach: When error happens commit the transaction and handle the error outside the atomic block.

Locks

```
1 void Z_CheckHeap (void)
 2 {
 3   memblock_t *block;
 4   LOCK;
 5   for (block=mainzone->blocklist.nex
 6   if (block->next == &mainzone->blo
 7   break; // all blocks have been hit
 8   if ( (byte *)block + block->size != (b
 9       Sys_Error("Block size does not
10   if ( block->next->prev != block)
11       Sys_Error("Next block doesn't have proper back link");
12   if (!block->tag && !block->next->tag)
13       Sys_Error("Two consecutive free blocks");
14   }
15   UNLOCK;
16 }
```

Commit and Abort Handlers

Transactions

```
1 void Z_CheckHeap (void) {
 2 memblock_t *block;
 3 int error_no = 0;
 4 atomic{
 5    for (block=mainzone->blocklist.next;;block=block->next){
 6       if (block->next == &mainzone->blocklist)
 7          break; // all blocks have been hit
 8       if ((byte *)block + block->size !=
 9          (byte *)block->next; {
10           error_no = 1;
11           break; /* makes the transactions commit */
                                    ->prev != block) {
                                    ;
                                    && !block->next->tag) {
18           error_no = 3;
19           break;
20       }
21    }
22 }
23 if (error_no == 1)
24    Sys_Error ("Block size does not touch the next block");
25 if (error_no == 2)
26    Sys_Error ("Next block doesn't have proper back link");
27 if (error_no == 3)
28    Sys_Error ("Two consecutive free blocks");
29 }
```

Failure Atomicity

Original

```
1 void PR_ExecuteProgram (func_t fnum, int tId){
 2   f = &pr_functions_array[tId][fnum];
 4   pr_trace_array[tId] = false;
 5   exitdepth = pr_depth_array[tId];
 6   s = PR_EnterFunction (f, tId);
 7   while (1){
 8     s++;    // next statement
 9     st = &pr_statements_array[tId][s];
10     a = (eval_t *)&pr_globals_array[tId][st->a];
11     b = (eval_t *)&pr_globals_array[tId][st->b];
12     c = (eval_t *)&pr_globals_array[tId][st->c];
13     st = &pr_statements[s];
14     a = (eval_t *)&pr_globals[st->a];
15     b = (eval_t *)&pr_globals[st->b];
16     c = (eval_t *)&pr_globals[st->c];
17     if (--runaway == 0)
18       PR_RunError ("runaway loop error");
19     pr_xfunction_array[tId]->profile++;
20     pr_xstatement_array[tId] = s;
21     if (pr_trace_array[tId])
22       PR_PrintStatement (st);
23   }
24   if (ed==(edict_t*)sv.edicts && sv.state==ss_active)
25     PR_RunError("assignment to world entity");
26   }
27 }
```

With failure atomicity

```
1 void PR_ExecuteProgram (func_t fnum, int tId){
 2   f = &pr_functions_array[tId][fnum];
 4   pr_trace_array[tId] = false;
 5   exitdepth = pr_depth_array[tId];
 6   s = PR_EnterFunction (f, tId);
 7   while (1){
 8     s++;    // next statement
 9     st = &pr_statements_array[tId][s];
10     a = (eval_t *)&pr_globals_array[tId][st->a];
11     b = (eval_t *)&pr_globals_array[tId][st->b];
12     c = (eval_t *)&pr_globals_array[tId][st->c];
13     st = &pr_statements[s];
14     a = (eval_t *)&pr_globals[st->a];
15     b = (eval_t *)&pr_globals[st->b];
16     c = (eval_t *)&pr_globals[st->c];
17     if (--runaway == 0)
18       abort;
19     pr_xfunction_array[tId]->profile++;
20     pr_xstatement_array[tId] = s;
21     if (pr_trace_array[tId])
22       PR_PrintStatement (st);
23   }
24   if (ed==(edict_t*)sv.edicts && sv.state==ss_active)
25     abort;
26   }
27 }
```

The Benefit of Failure Atomicity

Original

- PR_RunError dumps the stack trace and terminates the server.

Using failure atomicity

- Abort reverts the updates to the global variables
- The effect is as if the client packet was lost
- Server continues to run

Privatization Example

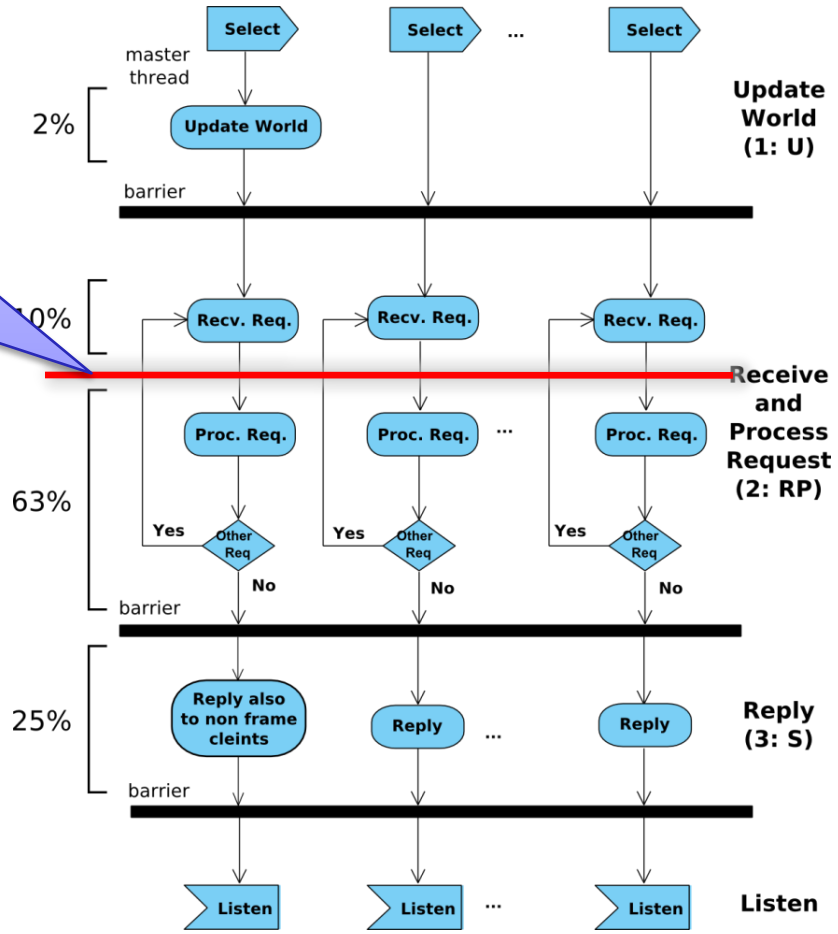
```
1 void* buffer;
2 atomic {
3    buffer = Z_TagMalloc(size, 1);
4 }
5 if (!buffer)
6     Sys_Error("Runtime Error: Not enough
memory.");
7 else
8    memset(buf, 0, size);
```

- The code assumes that after the memory block is returned, it will not be returned again until it is freed.
- Then the memory block is modified (zeroed).

Experimental Methodology

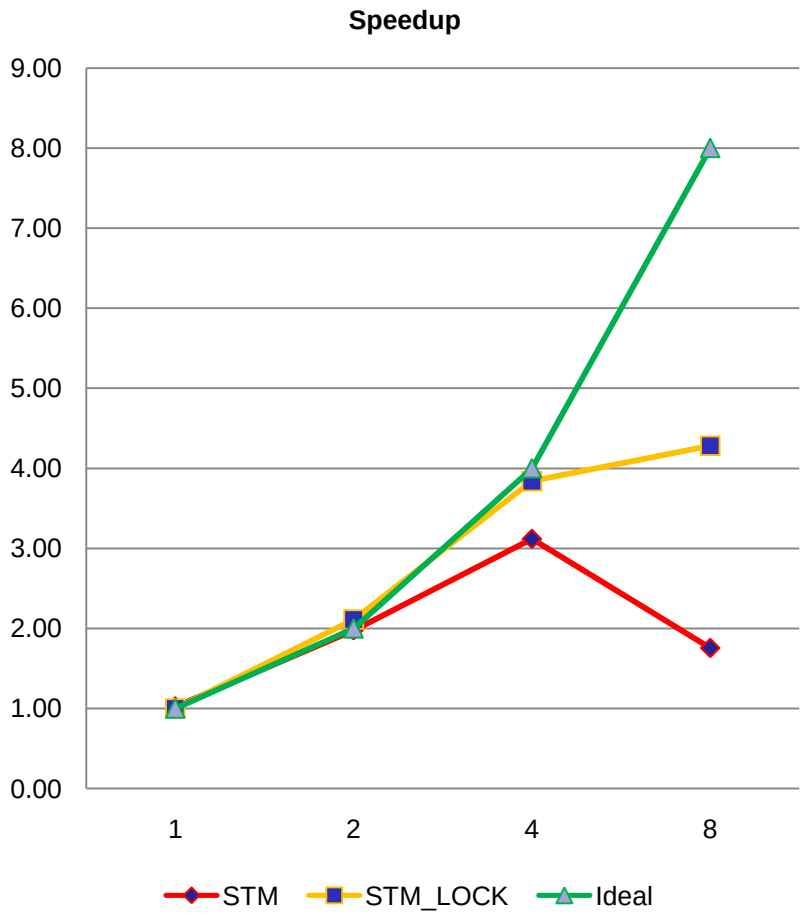
Added extra synchronization so that all the threads perform request processing at the same time.
• Equal to 100% loaded server

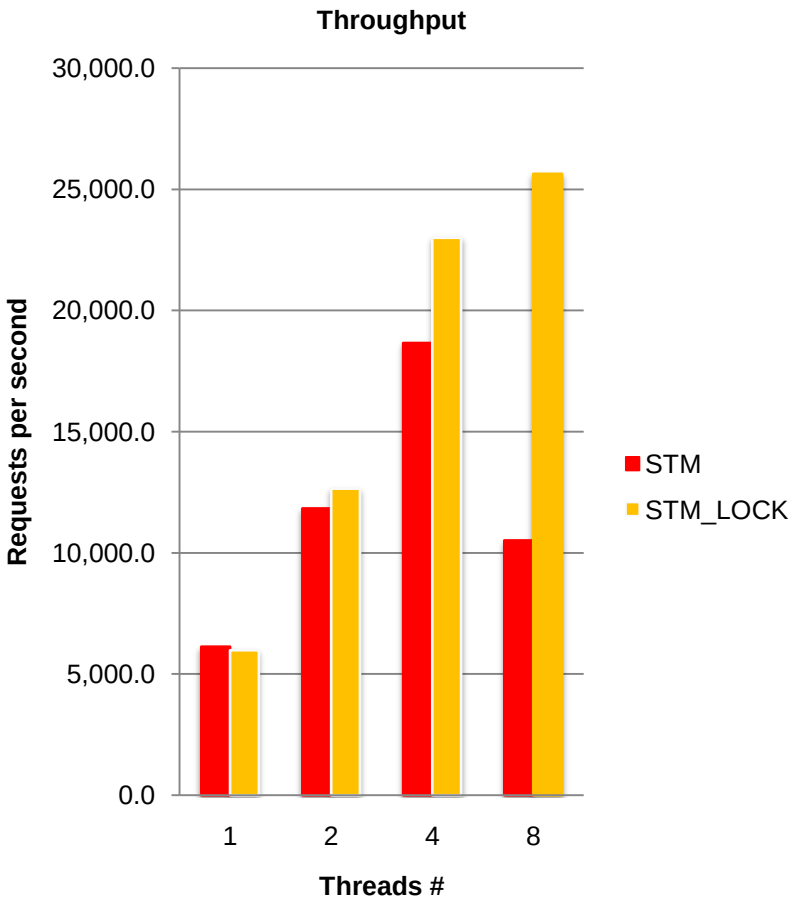
Used small map for 2 players representing high conflict scenario



BSC~ Microsoft Research Centre

General Performance





*In STM_LOCK atomic blocks are guarded by global reentrant lock. The rationale is to count the STM overhead in.

BSC~ Microsoft Research Centre

Overall Transactional Characteristics

Table1: Transactional characteristics.

Threads	Transactions	Aborts		Serialized Tx
		Num	%	
1	36,667	0	0.00%	17
2	75,824	241	0.42%	31
4	166,000	2,612	1,58 %	85
8	477,519	76,771	25.50%	237

Table 2: Read and writes set size (in bytes).

Threads	Read Set					Write Set				
	Min	Avg	Max	Total	Ratio	Min	Avg	Max	Total	Ratio
1	8	477	53,566	17,515,419	83%	0	95	11,161	3,492,370	17%
2	8	540	172,508	40,907,196	83%	0	115	47,784	8,737,623	18%
4	4	575	181,740	95,505,459	81%	0	131	52,032	21,737,915	19%
8	4	798	1,591,946	381,290,019	81%	0	183	352,640	87,837,969	19%

Conclusion

- TM is not mature enough.
- We need
  - Rich language extensions for TM.
  - Strict semantics for TM language primitives.
  - Stable toolset (compilers, profilers, debuggers).
  - Compatibility across tools and TM implementations.
  - Improved performance on single thread and graceful degradation on conflicts.
  - External library call, system calls, I/O.
  - Interoperations with locks.
- Atomic Quake is a rich transactional application to push the research in TM ahead.

Край

ferad.zyulkyarov@bsc.es