University of Delaware
Department of Electrical and Computer Engineering
Computer Architecture and Parallel Systems Laboratory

Design Manual for the Fresh Breeze Simulator

Xiao X. Meng[1], *Tom St. John*[1], *Jack B. Dennis*[2],
Guang R. Gao[1]

CAPSL Technical Memo 115
April 25, 2012

[1] University of Delaware and
[2] Massachusetts Institute of Technology

University of Delaware • 140 Evans Hall • Newark, Delaware 19716 • USA
http://www.capsl.udel.edu • ftp://ftp.capsl.udel.edu • capsladm@capsl.udel.edu

Contents

List of Figures

Abstract

This manual is intended for readers who want to understand the internal design and implementation of the Fresh Breeze simulator on the Cyclops64 architecture and also for those who want to develop new applications using the simulator.

The Fresh Breeze simulator was developed to verify the effectiveness and generality of the Fresh Breeze Memory model. The Fresh Breeze memory model was proposed by Professor Jack B. Dennis from MIT to support extremely fine grain management of storage resources across the entire memory hierarchy. In the envisioned Fresh Breeze system, the entire memory hierarchy is managed by the hardware, without assistance from the operating system. Furthermore, the data management granularity at every memory level is uniformly set to a data chunk whose size is as small as 128 bytes. Each chunk can contain up to 16 elements, with each element being either a scalar 64-bit data value or a pointer to another data chunk. Therefore, any complex data structure will be represented by an arbitrary tree of chunks under the Fresh Breeze memory model. For example, a linear array with 4096 elements can be represented as a three-level chunk tree. The chunks at the first two levels each contain pointers to chunks at the next lower level. The chunks at the leaf level will contain the data values. By using the fine-grain storage resource management, the Fresh Breeze memory model can increase the memory access concurrency for future many-core processors.

1 Introduction

The Fresh Breeze[5] simulator was developed to verify the effectiveness and generality of the Fresh Breeze memory model. The Fresh Breeze memory model[4] was proposed by Professor Jack B. Dennis from MIT to support extremely fine-grain management of storage resources across the entire memory hierarchy. In the envisioned Fresh Breeze system[3], the entire memory hierarchy is managed by the hardware, without assistance from the operating system. Furthermore, the data management granularity at every memory level is uniformly set to a data chunk whose size is as small as 128 bytes. Each chunk can contain up to 16 elements, with each element being either a 64-bit data value or a pointer to another data chunk. Therefore, any complex data structure will be represented by an arbitrary tree of chunks under the Fresh Breeze memory model. For example, a linear array with 4096 elements can be represented as a three-level chunk tree. The chunks at the first two levels each contain pointers to chunks at the next lower level. The chunks at the leaf level will contain the data values. By using fine-grain storage resource management, the Fresh Breeze memory model can increase the memory access concurrency for future many-core processors.

Another important feature of the Fresh Breeze memory model is that it enforces a write-once property on memory operations. The write-once property requires that, once a chunk has been sealed in the storage system, the chunk can no longer be modified. In other words, a chunk can be shared among tasks only after it has been rendered read-only by its producer task. The write-once property is very useful in the cache system design, since it circumvents the annoying cache coherency problem seen in many-core systems. Furthermore, the write-once property naturally leads to a functional view of the memory system.

A computation typically consists of three steps. The first step is get the input data value from existing data chunks. The second step is to perform the computation. The last step is to save the computation result into freshly created chunks. The functional memory system is a prerequisite to achieve composability in parallel programs, which is highly desirable in large-scale parallel program development.

Besides the unique memory model, the simulator also simulates a many-core processor chip which is closely designed for the Fresh Breeze memory model. In the following sections, we will provide a detailed description of the internal design and implementation of the Fresh Breeze simulator on the Cyclops64 architecture.

2 Overall Design of the Fresh Breeze Simulator on Cyclops64

This section describes the overall design scheme of the Fresh Breeze simulator on the IBM Cyclops64 architecture[1]. Each Cyclops64 chip has 80 processors, with 2 independent thread units per processor. Therefore, we have a total of 160 thread units, or computation cores, on a single Cyclops64 chip. Each processor has one floating point unit and every five processors share a 32-byte instruction cache. Each thread unit has its own 64-byte scratchpad memory which is a high-speed software managed cache. When the chip boots up, the 64-byte scratchpad memory of each thread unit can be uniformly configured to be divided into two sections. The first section is private to its own thread unit, called private scratchpad memory. The second section is shared by all the 160 thread units and is addressed in an interleaving manner, called global interleaved memory. The Cyclops64 architecture allows thread units to access each other's private scratchpad memory directly. However, the access time to local private scratchpad memory is much lower than the non-local global interleaved memory as the local access does not need to go across the on-chip inter-connection network. In addition to the on-chip memory, a Cyclops64 memory hierarchy consists of three levels. The private scratchpad memory is the first level, which is equivalent to the L1 cache level in the i386 architecture and is private to each thread unit. The global interleaved memory comprises the second level, which is equivalent to the on-chip L2 cache level in the i386 architecture and is shared by all thread units. The external DRAM is the third level.

The most interesting feature of the Cyclops64 memory hierarchy is that it is explicitly exposed to the application or system programmers. There is no cache coherence protocol in the hardware design, which makes it very simple and efficient. ETI Inc. has produced a lightweight runtime system on top of the Cyclops64 architecture, called the TNT-C kernel, which exports a non-preemptive multithreading programming interface. The application programmers can easily build their own parallel programs using normal C language plus an extended multithreading API. There are two important features of the TNT-C runtime kernel[2]. The first feature is that it supports a lightweight non-preemptive thread execution model, which means that once a thread has been dispatched to a thread unit, it will hold the thread unit until it terminates without preemption. The other feature is that the Cyclops64 memory hierarchy is explicitly exposed to applications running on top of the TNT-C kernel. The TNT-C kernel does not employ

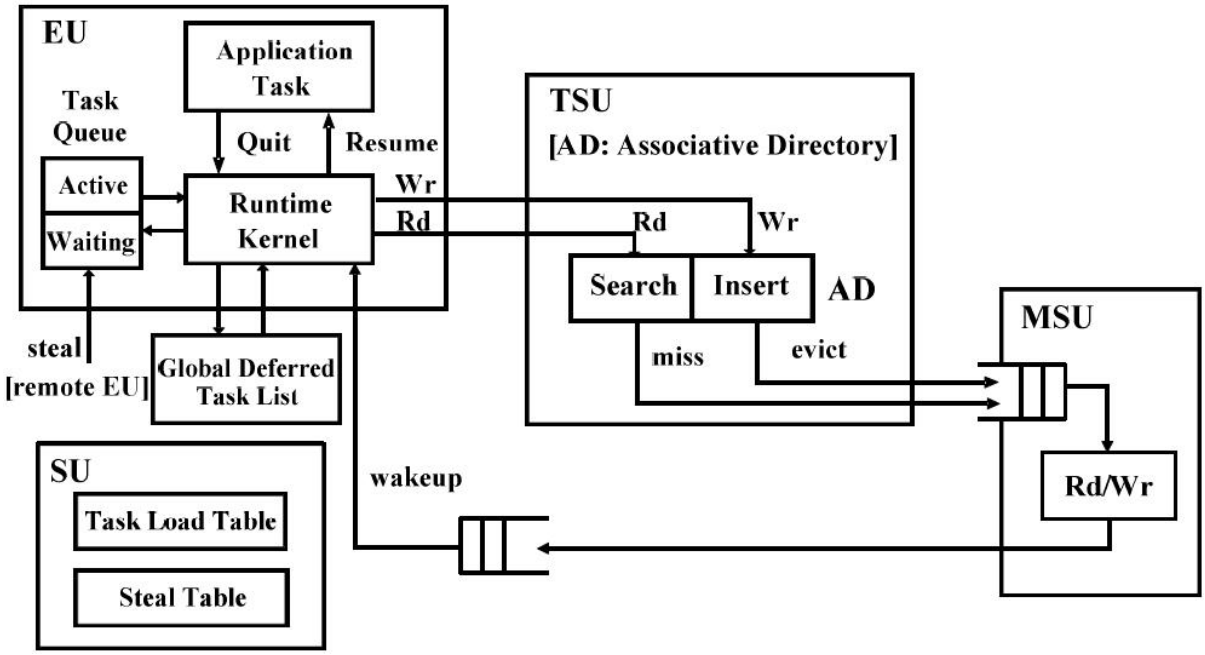

Figure 1: Overall design scheme of Fresh Breeze simulator

complex virtual memory techniques like a conventional OS and it only provides a simple dynamic heap memory allocator for global interleaved memory and external DRAM. In addition, the TNT-C compiler provides the static memory allocation for private scratchpad memory.

Our simulator is built on top of the TNT-C kernel as an application program. In order to utilize the features of the Cyclops64 architecture and TNT-C kernel, and also to accelerate the simulator development, we do not follow the standard discrete event based implementation approach. Instead, we directly map each functional unit in our envisioned Fresh Breeze system to one thread unit on the Cyclops64 chip. The simulation code running on each thread unit is used to simulate the function of its mapped unit in the Fresh Breeze system. In order to achieve time-accurate simulator, we use action padding mechanisms to uniformly balance the simulation cycles used by each critical system action based on our estimated real system cycles that should be used. For example, assume that the ratio between one real system cycle to one simulation cycle is R. Then, if action $A(X)$ in our target Fresh Breeze system uses $T(X)$ cycles to complete while the measured simulation cycles used in our non-padding simulation is $S(X)$, then the padding cycles for $A(X)$ should be $P(X) = T(X) - S(X) * R$. We will carefully choose a proper value of R to guarantee that for any $A(X_i)$, its padding cycles $P(X_i)$ is no less than 0. This is obviously required as we can only intentionally increase the simulation time of an event by letting the thread unit execute a number of idle instructions but we have no way, or at least no easy way, to reduce the simulation time of an event. To find out the proper value for R, we first need to calculate the ratio $R(X_i)$ between non-padding simulation cycles and the real system cycles for each of our interesting critical system actions (X_i). Then we set R to be the maximum ($R = MAX[R(X_i)]$).

Figure 1 gives a general picture of the overall design scheme of our simulator on the Cyclops64 architecture. There are four different types of functional units in the simulation system.

Execution Unit (EU): Each EU is simulated by a separate thread unit on the Cyclops64 chip. It represents a processing element in our envisioned Fresh Breeze system. We propose that each target Fresh Breeze chip will contain 40 independent processing elements, so there are 40 EUs in our simulator. The corresponding TNT thread, also called EU thread in this case, running on the thread unit mapped to the EU will simulate the EU's function, which mainly includes two tasks. The first task is to provide the application programmers with the Fresh Breeze runtime environment, that is, a data-flow oriented multithreading interface whose semantics are quite similar to the multithreading model of Cilk[6], a parallel programming runtime library. From a system software point of view, the first task is more like a runtime kernel or an extremely simplified operating system. However, we should keep in mind that in the real Fresh Breeze system, the runtime kernel is implemented entirely in hardware and the exporting interface is at the instruction level, but not the system call as in the conventional OS. The second task is to execute the application codes which are running on top of the Fresh Breeze runtime kernel.

The Fresh Breeze runtime kernel provides two sets of interfaces for the application programs. The first set is a spawn-join based multithreading API which enables the application programmers to build parallel programs on the Fresh Breeze simulator. The second set is a storage access API. The storage API provides all the necessary data chunk operations as required by the Fresh Breeze memory model. As we saw in Figure 1, the chunk access operations on TSUs are handled by the EU runtime kernel itself, while the operations on MSUs are handled by the simulation codes that run on the thread units for MSU. For example, if the accessed chunk is cached in the TSU, then the EU runtime kernel will fetch the chunk data from the TSU directly. However, if the chunk is not cached in the TSU, then the EU runtime kernel will issue a chunk access request to the MSU, and wait for the MSU to complete the requests. We will describe the detailed implementation of the EU runtime kernel in Section 3.2.

Scheduler Unit (SU): The Fresh Breeze architecture supports a fine-grain dataflow oriented parallel task execution model. At any time during the program execution, there will be a large number of tasks ready to be executed. This would require the Fresh Breeze chip to have some mechanism to evenly distribute those tasks across the 40 EUs at runtime, so as to better utilize the available processor resources. In order to achieve that goal, we add a hardware unit called a Scheduler Unit, on each target Fresh Breeze chip. The SU will help dynamically balance the tasks across the 40 EUs by utilizing a type of work-stealing policy which will be described in detail in Section 3. Similar to the EU implementation, the SU is also implemented by a separate thread unit. The SU thread running on it will execute the simulation code to simulate the function of the SU in the envisioned Fresh Breeze system. We will describe how the SU works in Section 3.3.

TopStor Unit (TSU): Each TSU is implemented by a separate thread unit. It represents the private L1 cache for each EU in the envisioned Fresh Breeze system, so there are 40 TSUs in the simulator. As we mentioned previously, the EU runtime kernel will directly handle the

chunk operations on TSUs, so the thread unit that simulates the TSU is only used to provide the scratchpad memory to store cached data chunks, which means that there is no execution activity on that thread unit during the simulation. When the simulator starts up, each TSU thread will initialize its private scratchpad memory layout, and then just go to sleep to wait for the termination of the simulator. We will describe how the TSU works in Section 4.

MainStor Unit (MSU): Each MSU is implemented by a separate thread unit. As our current simulation scheme only has two memory levels, the MSUs represent the permanent storage units which comprise the second memory level. During the simulation, the MSU thread will loop over the chunk requests dispatched from the EUs. As previously stated, each TSU, as a private L1 cache, is bound to serve a specific EU, so their relationship is a one to one correspondence. However, the relationship between MSU and EU has no such restriction, so any MSU can receive chunk requests from any EU. To dispatch a request to the second level, the EU will address the target MSU based on the chunk UID. There are a total of 64 MSUs in our current simulator, and the storage space of the second level is equally divided across the 64 MSUs in a linear manner. For example, if the size of the entire storage space is S, then the i^{th} MSU will store the chunks whose UIDs fall inside the range of $[S * (i-1)/64, S * i/64]$. Furthermore, for the purpose of performance analysis, our current simulator has two configuration settings according to the access latency to the second memory level. In the first setting, we assume that the second memory level represents an on-chip L2 cache which has very low access latency, so the EU runtime kernel will run in blocking mode. The blocking mode means that after dispatching the chunk request to the MSU, the EU runtime kernel will be busy-waiting until the completion of the request. However, in the second setting, we assume that the second memory level represents external DRAM which has comparatively high memory access latency. Therefore, the EU runtime kernel will run in non-blocking mode, which means that after dispatching a request to the MSU, the EU runtime kernel will not wait for the request completion but instead switch out the currently running application task and pick up another task which is ready to execute. This manual will only discuss the implementation of the EU non-blocking mode which is more complex than the blocking-mode implementation. We will describe the internal design and implementation of the MSU in Section 5.

Profiling Unit (PU): Besides the previous four functional units, there is one other type of functional unit, the Profiling Unit, which is included in our current simulator for performance profiling. This is not necessary in the real Fresh Breeze system and is not shown in Figure 1. The PU is also implemented by a separate thread unit and there is only one PU in the simulator. Its job is to collect the performance data at runtime and output the collect data to the local disk files at the end of the simulation. Its implementation is quite simple and will not be discussed in this manual.

In summary, there is a total of 146 thread units being used in the Fresh Breeze simulation: 40 thread units serve as EUs, 40 thread units serve as TSUs, 64 thread units serve as MSUs, one thread unit serves as an SU and another one serves as a PU. Except for chunk data and chunk meta-data, all other data structures used by the simulator are stored in scratchpad memory. The chunk data and chunk meta-data are stored in external DRAM. The storage space of the

first level in our simulator is set to 20480 chunks, which is equivalent to 2.5 MB. The storage space of the second level is set to 2^{22} chunks, which is equivalent to 512 MB.

3 The Work Stealing Based Parallel Task Execution

This section describes the parallel task execution model of the Fresh Breeze architecture and how it is implemented in the simulator. Section 3.1 gives a high-level description of the work-stealing scheme, which is the policy used to balance the work across the processors at runtime. Section 3.2 and Section 3.3 describe the detailed implementations of the two types of the functional units, EU and SU, which work together in the Fresh Breeze system to realize the work-stealing based parallel task execution.

3.1 Work Stealing Scheme on the Fresh Breeze Chip

The work-stealing policy has proven to be a very efficient task scheduling policy for parallel computing systems like the many-core chip modeled by our simulator. Here, a task is equivalent to a thread in the conventional operating system concept. Under the Fresh Breeze execution model, a task is a sequence of instructions which has no data dependences among other tasks in the program during its execution. The basic idea of a work-stealing policy is that when a parallel program is running on multiple independent processors and any one of the processors has exhausted its local task queue, the exhausted processor will attempt to steal a task from one of the other processors which has a non-empty task queue.

In our simulator, the work-stealing scheme is used for a many-core chip and will be implemented by hardware in a real system. Based on that premise, we use a dedicated on-chip hardware unit to facilitate task scheduling across the 40 EUs. The SU functions as a supporting hardware unit whose role is to keep track of the work load for each of the 40 EUs, as well as to help the EUs determinte from which EU to steal a task when they do not have any local tasks ready to execute. There are three important data structures used in this scheme.

Pending Task List: Each EU has its own pending task list which contains the new tasks which have been spawned by its locally running application tasks. The difference between the pending task list and the active task queue, as in conventional operating systems, is that the tasks on the pending list have not yet been executed, so they do not have any execution status. In order to maximize performance, it would be very efficient to move such a task from a heavily loaded EU to a lightly loaded EU to provide load balancing. Therefore, the tasks in the pending list are the tasks that are available for stealing since the cost of task migration to a different EU is relatively low. For example, when an EU has no work to do, it will find a victim EU and try to steal a new task from its pending task list. It is obvious that the number of tasks on the pending task list can be used as a hint for the workload of an EU.

Task Load Table: The task load table keeps track of the work load of all EUs on a single Fresh Breeze chip. It is implemented as a bitmap with one bit for each EU. If the bit

is set, it means that the corresponding EU is overloaded or has high work load. Otherwise, the corresponding EU has low work load. Consequently, the EUs which have high work load should be candidates for work stealing. In the current simulator implementation, we pre-set a threshold (TASK_LOAD_THRES = 2) to decide whether an EU is overloaded or not. The task load table is updated by EUs when they spawn a new task to their local pending task list. The SU will then utilize the work load status recorded inthe task load table to update the task steal table. The latter will in turn be queried by the EUs to search for the victim EU when they perform task stealing, which we will discuss below. In Section 3.2.3, we will further describe how the task load table is actually implemented in the simulator.

Task Steal Table: As we mentioned previously, the task steal table is used to help the EUs find a victim EU when they perform task stealing. The victim EU is the EU that provides the new task from its local pending task list to the stealing EU. An EU will perform task stealing only when they have no executable active tasks and their pending task list is empty. The task steal table has one entry for each EU and the entry only records the ID of the victim EU. For example, when the i^{th} EU performs task stealing, it will query the i^{th} entry of the task steal table to find out the victim EU ID. Suppose that the current entry value is j. Then the i^{th} EU will try to steal a task from the pending task list of the j^{th} EU. Regardless of whether the steal operation is successful or not, the i^{th} EU will always clear the i^{th} entry in the steal table after the query operation. The SU will continue updating the task steal table based on the up-to-date work load distribution on the Fresh Breeze chip. Such handling is designed to prevent an EU from repeatedly stealing tasks from the same EU, whose work load may change over time.

3.2 Internal Design of EU

The EU represents a CPU on the target Fresh Breeze chip. It has two tasks in the simulation. The first task is to provide the runtime kernel, and the other task is to execute the application code on top of the runtime kernel. The runtime kernel works like an extremely simplified operating system kernel. It provides task management and storage access service to the application through the well-defined Fresh Breeze API calls which will be described in Section 6 and Section 7. Just like a conventional operating system kernel design, there are also two execution contexts in the EU. The first is the context of the runtime kernel, and the other is the context of the application task. As we know from general operating system knowledge, the execution context generally consists of hardware context and software context. In our simulator design, the hardware context includes the saved CPU register file, while the software context includes the saved stack space snapshot. We will describe the data structures which are used in our simulator to represent the application task and kernel execution contexts. Before that, we first describe the private scratchpad memory layout of the EU that stores the execution status of the EU, which includes the execution context of both running application tasks and runtime kernel.

As shown in Figure 2, the private scratchpad memory of the EU has been divided into three

sections.

Protected TNT Runtime Kernel: This section contains the private data of the TNT runtime kernel and it is a privilaged data area that is automatically protected by the Cyclops64 hardware mechanism. This section is transparent to users and application programmers and is not accessible by application code. The rest of the scratchpad memory is left available for application programs to use. Actually, the application programmer needs to specify in their program approximately how many bytes of the private scratchpad memory they intend to use. When the application begins executing on the Cyclops64 chip, the bootstrap procedure of the TNT kernel will dynamically configure the size of the TNT runtime kernel data section as required by the application. For example, if the total size of the private scratchpad memory is T bytes and the application program wants to use A bytes, then the kernel size is K = T - A bytes. The configuration of the kernel data section is done through setting the special boundary protection related registers. As the Fresh Breeze simulator itself is an application program that runs on top of the TNT kernel, we reserve the private scratchpad memory in the simulator by the following macros.

```
#define KERNEL_SIZE (3*(1<<12)) //12 KB
#define SRAM_SIZE (64*(1<<10)) //64 KB
#define MEM_SIZE (SRAM_SIZE-KERNEL_SIZE) //52 KB
#pragma spm reserve_mem
uint64 reserve_mem[MEM_SIZE/sizeof(uint64)];
```

The compilation directive **#pragma spm** is used to tell the compiler to place the variable *reserve_mem* in the private scratchpad memory area of each thread unit on the Cyclops64 chip. As it is the only variable in the simulator that needs to be placed in the private scratchpad memory area, *reserve_mem* will be placed right after the TNT kernel data section. As we can see, *reserve_mem* is defined as a linear byte array and its size is set to 52 KB. The macro *SRAM_SIZE* defines the scratchpad memory size of each thread unit on the Cyclops64 chip, which is 64 KB as specified by the Cyclops64 architecture. Since we do not use the global interleaved memory in our simulator at all, we then set the size of the global interleaved memory section to be zero and use the entire scratchpad memory as private memory. The above *KERNEL_SIZE* macro actually defines the size of the TNT kernel data section.

The resulting effect of the above macro definitions is that the data area that is occupied by *reserve_mem* is under the control of our simulator program. We can reorganize the data area however we want. In the case of the EU, we further divide this area into two sections. The first section is the task slot array and the other is a data structure with the type of *execution_unit* which is the EU descriptor. We will now describe the two sections.

Application Task Slot Array: There are a total of five task slots in the array. Each task slot is used to store the execution context of an application task. As shown in Figure 2, a task slot is defined as a *union task_union*. It has two members, *task* and *stack*. The *task* member has the type of *task_struct* which records the hardware and system contexts. The *stack* member is defined as a linear byte array with a size of 4 KB. It stores the software stack of the application

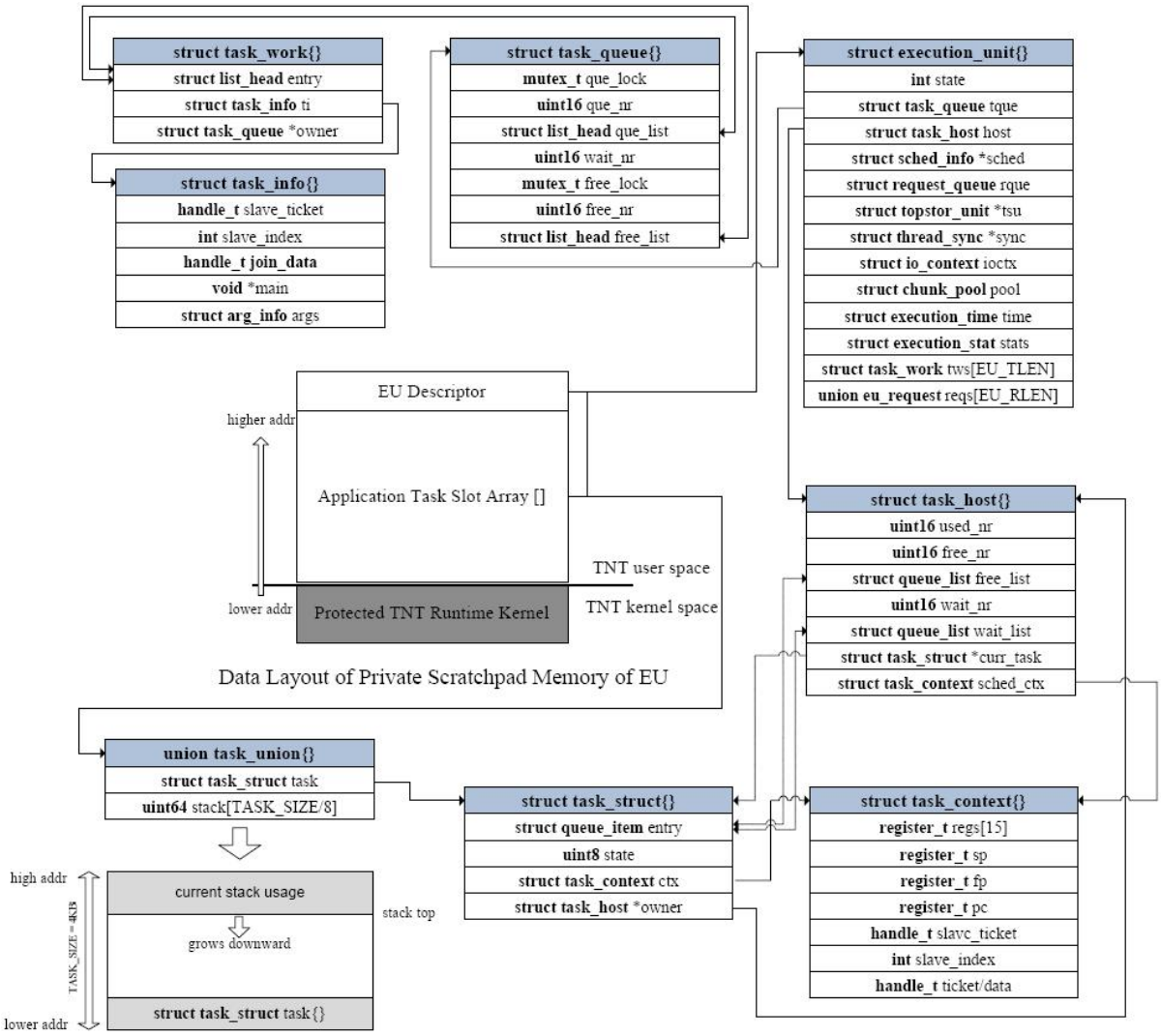

Figure 2: Data layout of the private scratchpad of an EU

task. From Figure 2, we can see that one task slot has the size of its maximum sized member, that is 4 KB. The *task* is placed at the bottom of the area while the *stack* starts from the top of the area and grows downstairs. It should be noted that the choice of the number of task slots and the size of each task slot is limited by the size of usable scratchpad memory on the Cyclops64 chip.

Next, we will take a look inside the *task_struct* structure to see how it defines the hardware and system contexts for an application task.

state: This defines the running state of an application task. The two important states are TASK_RUN and TASK_BLOCKED. The former state indicates that the task is currently running on the EU, and the latter state indicates that the task is switched out of the EU and is waiting for the completion of some events, such as the completion of a storage request or the

release of a free request slot. We will discuss the task state change in Section 3.2.2.

task_context: This embedded data structure defines the hardware context and spawn-join related system context. We will describe the latter in Section 7. Here we first look at the members used to define the hardware context: *regs[15]* saves R47 through R61 which are specified in the Cyclops64 ABI as the callee save registers; *sp* saves R3, which is specified as the stack pointer; *fp* saves R62, which is specified as the stack frame pointer; *pc* saves the program counter. The usage of these members will be discussed in Section 3.2.3 to illustrate the software implementation approach for context switching that is used in the Fresh Breeze simulator.

EU Descriptor: As shown in Figure 2, the EU descriptor with the type of *execution_unit* contains or links to all information that is needed to record the execution status of the EU during simulation.

tuid: This is a system-wide identifier for the thread unit that serves as an EU. The field is a 16-bit word and consists of two parts, with the four high-order bits being used to denote the type of thread unit. From Section 2, we know that there are five different types of thread units. For the EU, the type value is set to 0. The remaining 12 bits are used to record the index of the thread unit with the same type.

tque: This embedded data structure defines the pending task list of the EU. It should be noted that any task on the pending task list has not yet been executed, so it does not have any execution status except for some initial startup information. Therefore, each new task on the pending task list is defined as a *task_work* structure, which we will describe in the next section. The *que_xxx* fields define the in-use pending task list, while *free_xxx* fields define the free new task slot list. Due to the limited size of scratchpad memory, the pending task list cannot grow infinitely without a boundary. In our current implementation, the maximum size of the pending task list is set to 64. The embedded *tws[]* in the *execution_unit* structure stores the new task slots array. At the beginning of the simulation, all new task slots are put in the free list. When the EU creates a new task, it gets a free new task slot from the free list (*alloc_task()* in *task.c* file). In addition, when the EU starts a new task on the pending task list, it will put the used new task slot back to the free list (*free_task()* in *task.c* file).

host: This embedded data structure defines the active task list or the current usage of the task slot array. Here it is worth noting that the task slot array is different from the new task slot array. The former records the execution context of an active task that has been executed but hasn't yet finished. The latter records the initial context of a new task that has not yet executed. Obviously, the storage size of the latter is much smaller than the former. *user_nr* and *free_nr* record the number of used task slots and the number of unused task slots respectively, while *free_list* links all unused task slots. As we can see from Figure 2, it implements the linkage through the *item* field in the *task_struct* structure which is a single-linked list; *wait_nr* records the number of active tasks which are waiting for the free usable storage access requests and *wait_list* links all such waiting active tasks together; *curr_task* points to the *task_struct* structure of the currently running task; *sched_ctx* is an embedded *task_struct* structure which is used to save the execution of the EU runtime kernel, but not the application task. The difference between these two types of context is that the EU runtime kernel doesn't need space to store

the software stack which is stored at the default TNT thread stack space that lies inside the TNT kernel data section.

rque: This embedded data structure defines the storage request queue. The *req_xxx* field defines the pending request queue and the *free_xxx* field defines the free request list. Similar to the pending task list, the storage request queue cannot grow indefinitely. The total number of available request slots is four times the total number of available task slots, and they are stored in the *reqs[]* array of *execution_unit*. The reason for this limit is that storage requests are usually issued by application tasks, so the number of active requests is related to the number of active tasks. Depending on the request status, each active request either represents a storage access request issued from an EU to the MSU or represents the completion status of a finished storage request. It is easy to see that if the request is in pending status, it should be placed in the pending request queue of the MSU that waits to serve the request. However, if the request is a completion notification, it should be placed in the pending request queue of the EU that waits for post-processing. It should be noted that both the EU and MSU share the same *request_queue* data structure to describe the request queue.

tsu: This points to the TSU descriptor of the type *topstor_unit*. It describes the status and information of the TSU which is associated with the EU and serves as its private L1 cache unit. This pointer allows the EU runtime kernel code to more easily access its private TSU.

3.2.1 Pending Task List

As we discussed in the previous section, the pending task list contains the newly created tasks which have not yet begun execution. It will be accessed in the following scenario.

Create a New Task: When the application task requests to spawn a new task (*spawn_one()* in Section 7) through the Fresh Breeze API, the EU runtime kernel will invoke the *task_create()* function to serve the request. The *task_create()* function is implemented as follows (see *task.c* file):

Step 1: Invoke *task_alloc()* to get a new task slot (*tw*) from the free new task list.

Step 2: Invoke *init_task_info()* (in *task.c* file) to fill in the *tw* with task startup information. The task startup information consists of two parts. The first part consists of application task code which contains the initial instruction pointer and input arguments. The second part contains spawn-join related task hierarchical information which we will revisit in Section 7.2.

Step 3: Invoke *push_task()* (in *task.c* file) to put the new task at the head of the pending task list. After inserting the new task, the *push_task()* function will also check to see whether the number of pending tasks exceeds the preset overloading threshold. If it does, then it will invoke *task_load_set()* (in *sched.h* file) to set the corresponding overload bit stored in the task load table which is stored in the private scratchpad memory of SU, as we will see in Section 3.3.

Start a New Task: When there is an available task slot and the pending task list is non-empty, the EU runtime kernel will invoke *pop_task()* to get a new task from the head of the

pending task list. As opposed to the *push_task()* function, if the pending task list is currently in the overloaded status, which means that the corresponding overload bit is set in the task load table, then *pop_task()* needs to check whether the pending task list is still overloaded. If it is not, then it will invoke *task_load_clear()* (in *sched.h* file) to clear the overload bit in the task load table.

Steal a New Task: When there is an available task slot and the pending task list is empty, the EU runtime kernel will invoke *steal_task()* (in *sched.c* file) to try to steal a new task from any of the other EUs. The *steal_task()* function is defined as follows:

Step 1: Use the *atomic_set_and_return()* routine to get the current victim EU ID from the task steal table for this EU. The *atomic_set_and_return()* routine will finish the following two steps in atomicity. The first step is to fetch the value of the victim EU ID and the second step is to reset the corresponding task steal table entry to a special value of *STEAL_BUSY*, which means that the task steal entry is stale and is waiting for the SU to update, as we will see in its usage in the next step.

Step 2: Check the retrieved victim EU ID. If it is set to a normal EU ID, then go to Step 3. If it is set to *STEAL_BUSY*, then it means that the task steal table entry is stale and the SU has not had a chance to update it yet. As we know from Step 1, each time that the EU gets the victim EU ID from the task steal table, it will also reset the entry to *STEAL_BUSY*. Once this happens, the EU will invoke *relax()* (in *lib.h* file) to take a short nap to busy wait for the update by the SU and then return to Step 1 to retry. If the victim EU ID is set to *STEAL_WAIT*, it means that there are no available EUs that have non-empty pending task lists, so the SU cannot find a victim EU. Once that happens, *steal_task()* directly returns *NULL* to notify of failure since there is no possibility that the SU can find a victim EU in a short time.

Step 3: Invoke *__steal_task()* (in *sched.c* file) to attempt to steal a new task from the specified victim EU. Unlike *pop_task()*, *__steal_task()* will attempt to fetch a new task from the tail, but not the head of the pending task list.

The rationale behind such a design is that we assume the tasks at the end of the tail of the pending task list are normally at a higher layer of the task hierarchy tree than the tasks at the head of the pending task list, as the former tasks are created earlier than the latter ones. Furthermore, the tasks at higher layers will most likely be the start of a larger sub-computation. Based on that information, it is beneficial to quickly balance the work load across multiple EUs by stealing tasks from the tail of the pending task list rather than the head.

If *__steal_task()* fails to steal a new task from the specified victim EU, it may be due to the possibility that the work load of the victim EU has changed and the SU has not yet updated. If that happens, it will return to Step 1 to try again. Otherwise the stolen task is returned to the EU runtime kernel.

In summary, the pending task list is manipulated in the following three ways: (1) a new task will be inserted into the head of the list; (2) a new task will be removed by the local EU from the head of its list; (3) a new task will be stolen by a remote EU from the tail of the list.

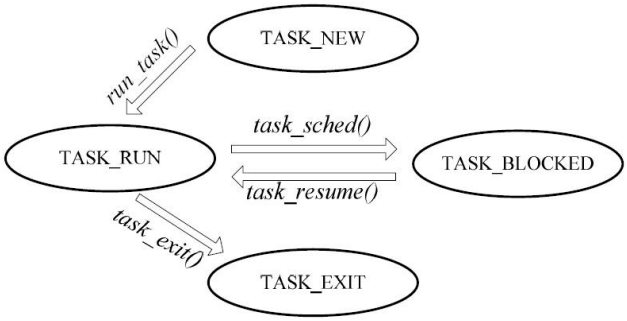

Figure 3: Task state change diagram

3.2.2 Task State Change

An application task in the Fresh Breeze execution model has the following five states (as defined in the *task.h* file).

TASK_NEW: This state indicates that the application task is a brand new task which has not yet begun execution. The task in this state should be placed in the pending task list.

TASK_RUN: This state indicates that the application task is currently running on the EU.

TASK_BLOCKED: This state indicates that the application task is blocked and waiting for the completion of a storage request.

TASK_EXIT: This state indicates that the application task has finished execution and its task slot will be reclaimed by the EU runtime kernel once it has been switched out of the EU processor.

Figure 3 shows the task state change diagram under the Fresh Breeze task execution model. As we can see from the figure, during the lifetime of an application task, it will experience the following state change sequence.

(1) **TASK_NEW** to **TASK_RUN:** When starting a new task on the EU, the runtime kernel invokes *run_task()* (in *task.c* file) to set up the initial execution context for the new task, and then goes to the first instruction of the application code. During this process, the task state will undergo a state change from TASK_NEW to TASK_RUN. In Section 3.2.4, we will describe the implementation of *run_task()*.

(2) **TASK_RUN** to **TASK_BLOCKED:** If the application code issues a synchronous storage request like a chunk read or write operation, which we will describe in Section 6, the EU runtime kernel will switch out the currently running task and pick up another ready task so as to keep the EU busy with useful computation. The low speed storage access time will be overlapped with the computation time of the other tasks.

(3) **TASK_BLOCKED** to **TASK_RUN:** When the storage request has been completed, the MSU will put the request back into the pending request queue of the initiating EU.

Some time later, the EU runtime kernel will get the completed request from its pending request queue (*get_request()* in *req.c* file). The I/O context information recorded in the request will tell the runtime kernel to resume the execution of the application task which issued the request. During this process, the task state will undergo a state change from TASK_BLOCKED to TASK_RUN.

The task resume operation is implemented by the *execute_resume()* (in *eu.c* file) function which consists of two steps.

Step 1: Retrieve the I/O context information from different types of storage requests. As we will see in Section 6 and Section 7, there are three types of requests which will incur task blocking. These requests are the chunk read operation, chunk write operation and join ticket create operation.

Step 2: Invoke the *task_resume()* (in *task.c* file) function to wake up the blocked application task. The *task_resume()* invocation will first set the task state to TASK_RUN, then call *task_sched()* (in *task.c* file) to resume its execution.

(4) **TASK_RUN** to **TASK_EXIT:** When an application task is done, it will call *task_exit()* (in *task.c* file) to terminate its execution. The function will first change the task state from TASK_RUN to TASK_EXIT, and then call *task_sched()* to relinquish the EU. As we will see in Section 3.2.3, *task_sched()* will reclaim the task slot of a switched out task if its state is set to TASK_EXIT.

It is worth noting that if an application task issues multiple storage requests, it may go back and forth between case (2) and case (3) several times during its lifetime.

3.2.3 The Handling of Task Context Switching

This section describes how to implement task context switching in the Fresh Breeze simulator. As we have seen in previous sections, task context switching will occur either when an application task relinquishes its execution or when the EU runtime kernel resumes the execution of a blocking task when the waiting request has been completed. The job of context switching is actually done by the *task_sched()* function. It is easy to see that *task_sched()* will handle two kinds of context switching. The first case switches from the context of an application task to the context of the runtime kernel and the other case switches back. In the former case, the input argument to *task_sched()* is set to *NULL*, while in the latter case, the input argument is set to the execution context descriptor (*task_struct*) of the application task. We will examine each of these two cases separately.

Case 1: Context Switch from Application Task to Runtime Kernel

Step 1: Get the hardware context descriptor of the currently running task from the active task list descriptor (*prev_ctx = host->curr_task->ctx*) and then set the *curr_task* field to *NULL*, as the EU will execute runtime kernel code next.

Step 2: Get the hardware context of the runtime kernel from the active task list descriptor (*next_ctx* = &*host->sched_ctx*).

Step 3: Call *do_task_switch(prev_ctx, next_ctx)* to perform the hardware context switch between *prev_ctx* (application task) and *next_ctx* (runtime kernel). The return value of *do_task_switch()* is set to the hardware context descriptor of the switched out application task, and is saved in local variable *prev_ctx*. It is necessary when we switch to the context of the runtime kernel that the old value of *prev_ctx* is set to itself. We will later discuss *do_task_switch()* (in *task.c* file) since the function deals with the particular features of the Cyclops64 architecture.

Step 4: Now we are in the execution context of the EU runtime kernel. As we got the hardware context descriptor of the switched out application task in the previous step, we now call *finish_task_switch()* (in *task.c* file) to handle the post-processing based on the current status of the application task. If the current task state is TASK_RUN, then the runtime kernel has nothing to do. Otherwise, the task state should be TASK_EXIT. In this case, the runtime kernel needs to recycle its occupied task slot by calling *task_delete()* (in *task.c* file).

Case 2: Context Switch from Runtime Kernel to Application Task

Step 1: Get the hardware context descriptor of the next running application task from the input argument (*next_ctx* = &*next_ts->ctx*).

Step 2: Get the hardware descriptor of the runtime kernel from the active task list descriptor (*prev_ctx* = &*host->sched_ctx*).

Step 3: If the next running task is a brand new task, then change its state from TASK_NEW to TASK_RUN.

Step 4: Set the EU's current running task to be the next running task (*host->curr_task* = *next_ts*).

Step 5: Call *do_task_switch(prev_ctx, next_ctx)* to perform the hardware context switch. Once the function returns, the EU will be in the execution context of the next running application task. It is worth noting that the EU either starts to execute the first instruction of a brand new task, or execute the next instruction of a previously interrupted task. We will see the difference in the following discussion.

Next we will take a look at how we handle the particular features of the Cyclops64 architecture to implement the hardware context switch by using a software approach. The Cyclops64 architecture itself doesn't provide a dedicated instruction to support the context switch. The following description provides the actual code implementation of *do_task_switch()*. It is implemented using inline assembly C.

```
#define switch_to(prev_ctx, next_ctx) do {
1.  asm volatile( "ldd r43, %1\n\t" /* load prev_ctx pointer to temp register R43 */
2.                "ldd r44, %2\n\t" /* load next_ctx pointer to temp register R44 */
3.                "stm r47, r43, r61\n\t" /* save callee save registers */
4.                "std r62, 128(r43)\n\t" /* save frame pointer(r62)*/
5.                "std r2, 120(r43)\n\t" /* save stack pointer(r3) */
```

```
6.                "ldd r3, 120(r44)\n\t" /* restore stack pointer(r3) */
7.                "bal r45, 4\n\t" /* get next program counter */
8.                "addi r46, r45, 16\n\t" /* set return program counter of prev */
9.                "std r46, 136(r43)\n\t" /* save return program counter of prev */
10.               "ldd r46, 136(r44)\n\t" /* get return program counter of next */
11.               "br r46\n\t" /* restore program counter of next */
                  /* from now on, next task/sched context */
12.               "ldd r62, 128(r44)\n\t" /* restore frame pointer(r62) */
13.               "std r43, %0\n\t" /* set prev_ctx pointer */
14.               "ldm r47, r44, r61\n\t" /* restore callee save registers */
                  :"=m" (prev_ctx)
                  :"m" (prev_ctx), "m" (next_ctx)
                  :"r63" /* save & restore return addr automatic by GCC */
                      );
}while(0)
```

Lines 1-2 save the values of *prev_ctx* and *next_ctx* to unused registers R43 and R44. These two registers have already been automatically saved by caller so that we can freely use them during the context switch process;

Line 3 saves the 15 callee saved registers (R47 to R61) in *prev_ctx->regs[]*;

Line 4 saves the stack frame register (R62) in *prev_ctx->fp*;

Line 5 saves the stack pointer (R3) in *prev_ctx->sp*;

Line 6 restores the stack pointer from the next hardware context descriptor *next_ctx->sp*;

Lines 7-8 get the instruction pointer to be saved;

Line 9 saves the isntruction pointer in *prev_ctx->pc*;

Line 10 gets the restore instruction pointer of the next context *next_ctx->pc*;

Line 11 jumps to restore insruction pointer of the next context;

Line 12 restores the frame pointer from the next context *next_ctx->fp*;

Line 13 sets *prev_ctx* from saved value in R43;

Line 14 restores the callee saved registers from the next context *next_ctx->regs*;

It is worth noting that since a brand new task has not yet begun execution, it does not have hardware context to restore as we do in Line 12 through Line 14. To avoid that, when the EU starts to run a new task, the *set_task_unit()* function will initialize the task initial program counter to *task_helper* (in *task.c* file) but not the first instruction of application code as specified in the Fresh Breeze API (*spawn_one()* in *fb_exm.c* file). In addition, the *set_task_unit()* function will also manipulate the initial software stack by setting the stack pointer properly. The following is the actual code implementation of the *task_helper* routine. It is also implemented using inline assembly C.

```
static void task_helper(void) {
     asm volatile(
1.       "ldm r8, r44, r15\n\t" /* load input parameters */
2.       "ldd r47, 64(r44)\n\t" /* set task entry function */
3.       "br r47" /* jmp to the task entry function */
     );
}
```

Line 1 loads task input arguments into registers that are specified by the Cyclops64 ABI to receive the function input arguments. The task input arguments are saved in *ctx->regs[]* which are initialized by *set_task_unit()* accordingly;

Line 2 loads the first instruction of the application code to R47;

Line 3 jumps to the first instruction of the application code;

3.2.4 Put it All Together: How EU Works?

Figure 4 gives a working diagram of the EU. It basically consists of a main loop which is implemented by the *eu_main()* function (in *eu.c* file).

Step 1: Check whether the EU is still running. This is done using the macro *IS_EU_RUN()*. The macro checks the *state* field in the EU descriptor. If it is not set to THD_RUN, then jump to Step 9 to terminate the EU thread. Otherwise, go to the next step as the EU is still running.

Step 2: Check if the pending request queue is empty or not. If it is empty, then *get_request()* will return *NULL* and jump to Step 4. Otherwise, *get_request()* will return the pending request at the head of the request queue and go to the next step to handle the request.

Step 3: The pending request represents the completed request that is replied from the MSU. It contains the processing result that needs the EU for post-processing, in order to retrieve data from the information recorded in the request, or resume the execution of a task that is being blocked while waiting for request completion. The EU runtime kernel will invoke *eu_execute()* (in *eu.c* file) to handle different types of requests. The function dispatches different request handlers according to the specific request type.

Step 4: Check if there are any available task slots. This check is performed by *has_free_task_unit()* (in *task.h* file) which checks *host->free_nr*. If there is no empty task slot, then jump to Step 7. Otherwise, proceed to the next step.

Step 5: As we know from Step 4, there are still available task slots to use for the new task. The runtime kernel tries to find a new task to run. This is done by *grab_task()*. The function first tries to get a task from the head of the local pending task list by *pop_task()*. If it fails, then the function tries to steal a task from any of the other EUs using *steal_task()*. If they both fail, *grab_task()* returns *NULL*, and the runtime kernel jumps to Step 7. Otherwise, it will return the new task descriptor.

Step 6: Run the newly grabbed task by calling *run_task()* (in *task.c* file). The function will first allocate a task slot from the active task list by calling *alloc_task_unit()* (in *task.c* file).

Then it will call *set_task_unit()* as we previously discussed to setup the execution context for a new task by initializing the newly allocated task slot properly. Once *run_task()* returns, either the newly started task has exited, or it has blocked out of the EU to wait for the completion of a storage request.

Step 7: Check whether the EU has done any useful work in the above steps. The criterion is that either the EU has handled a pending request (*did_req != 0*) or the EU has started a new task (*did_tsk != 0*). If the EU has done nothing, it means that the EU has either finished all of its work, or all its active tasks are blocked and waiting for the storage request completion. Therefore, instead of rushing into the next iteration immediately, it is wise to go to Step 8 to take a nap to wait for some progress from either the storage system or the other EUs which will produce more new tasks that are available for stealing.

Step 8: Invoke *relax()* to let the EU remain idle for several cycles.

Step 9: The EU has been flagged (*IS_EU_RUN() == FALSE*) to stop and then quit the main loop of the EU thread to terminate. This condition only occurs when the entire application simulation has completed. It is flagged by the *task_exit()* invocation of the last task in the application program. As we will see in Section 7.2, *task_exit()* takes an input value *done*. If *done* equals 2, it means that the entire application simulation has completed.

3.3 Internal Design of SU

As shown in Figure 5, the scheduler information is the only data structure that is placed in the scratchpad memory. However, it contains a number of embedded data structures that will be used to implement two major functions in the Fresh Breeze simulation. One function is to facilitate work-stealing based task scheduling on the Fresh Breeze chip. The other function is to implement task defer and restore operations for excess new tasks that cannot be accommodated by the limited size of the EU pending task lists. We will not cover the task defer and restore operations in this manual. This subsection will focus on how the SU works to help the work-stealing based task scheduling.

From the discussion in Section 2, we know that there are two work-stealing related data structures in Figure 5. One data structure is *task_load_table*, which stores the task load table, and the other is *task_steal_table*. As we can see from the figure, both of these two data structures are very simple. The former is implemented by a bitmap (defined in *bitops.c* file). The latter is implemented by an integer array. We have already discussed enough about how the EUs update the task load table. Here, we will talk about how the SU thread updates the task steal table. The main loop of the SU thread is implemented by *su_main()* (in *sched.c* file). Its implementation is very simple and the loop iteration just calls *su_update()* (in *sched.c* file) to update the task steal entry for each of the 40 EUs. For each EU steal entry, *su_update()* will call the routine *task_steal_set()* (in *sched.c* file) to search the task load table from a random offset.

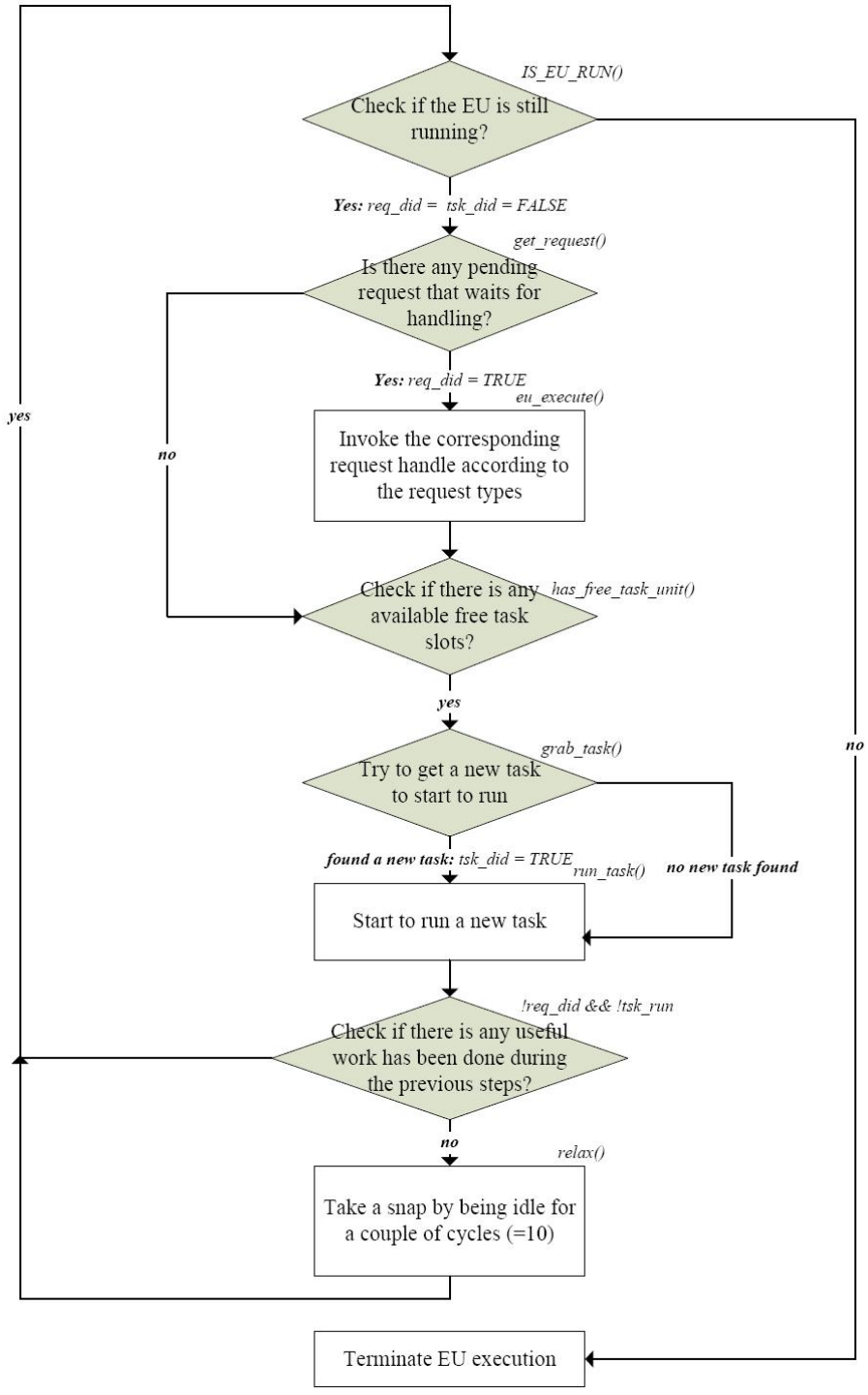

Figure 4: The EU working diagram

4 The First Storage Level: the Private L1 Cache

This section describes the internal design of the first storage level. The first storage level in our simulator represents the on-chip L1 cache in the target envisioned Fresh Breeze system. As

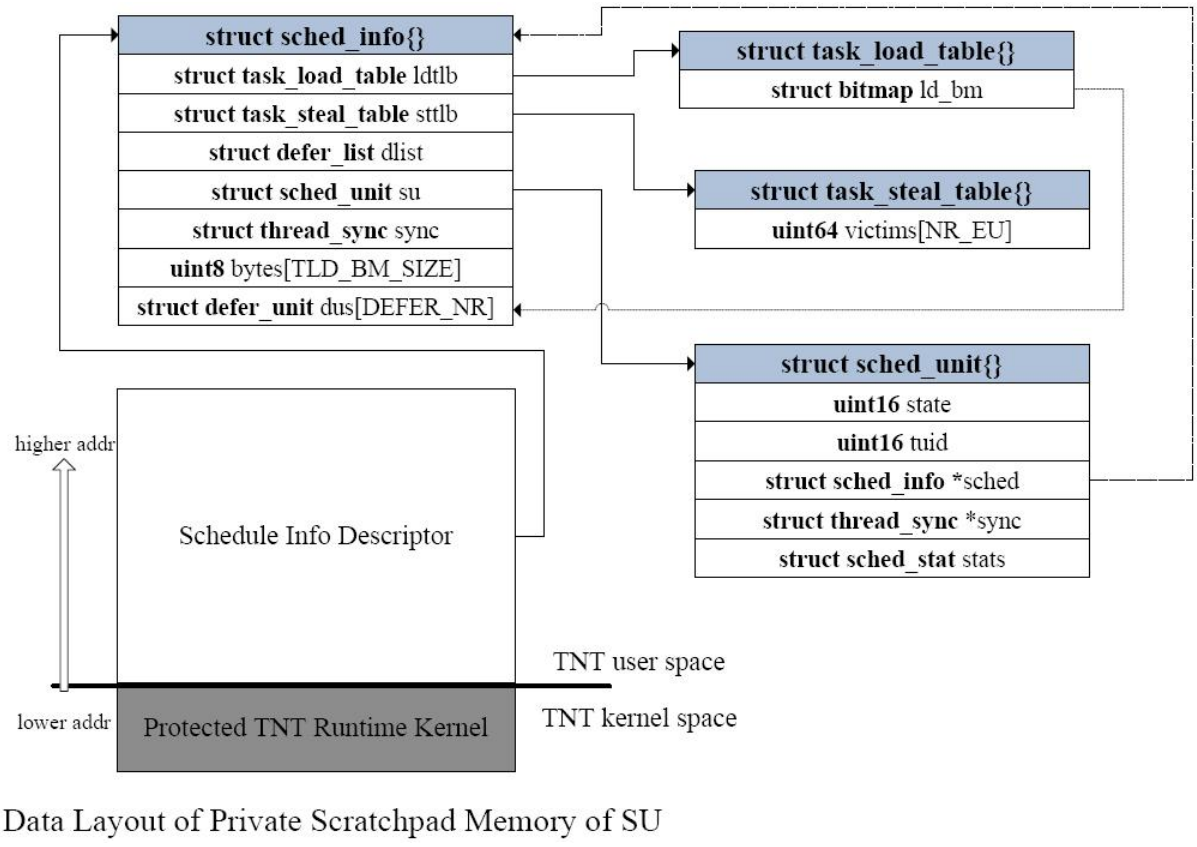

Figure 5: Data layout of the private scratchpad memory of the SU

we mentioned in Section 2, there is one private L1 cache unit for each EU, and the functional unit which represents an L1 cache unit is called a TSU. The TSU does nothing during the simulation, but simply provides its private scratchpad memory to store the L1 cache data. Due to the size limitation of the scratchpad memory, we cannot store the chunk data on the TSU's scratchpad memory area, but only the cache index data structure which is used to realize the mapping between the chunk UID and its location in the L1 cache unit. The cached chunk data and its meta-data are stored in the external DRAM. Furthermore, in order to minimize the usage of external DRAM, as we only have 1 GB of DRAM available on an individual Cyclops64 chip, the cached chunk shares the same physical data storage with the second storage level. This will not have any impact on the correctness of our simulation.

The write-once property of the Fresh Breeze memory model makes the cache system design in the Fresh Breeze system very easy. The most important benefit is that we can ignore the cache coherence protocol in our design. The resulting L1 cache organization in our simulator is as follows:

Cache Replacement Policy: We use an LRU (least recently used) cache replacement algorithm. Under an LRU policy, the most recently accessed chunks are placed towards the MRU (most recently used) end of the list. For example, if a chunk is acessed and hit in the

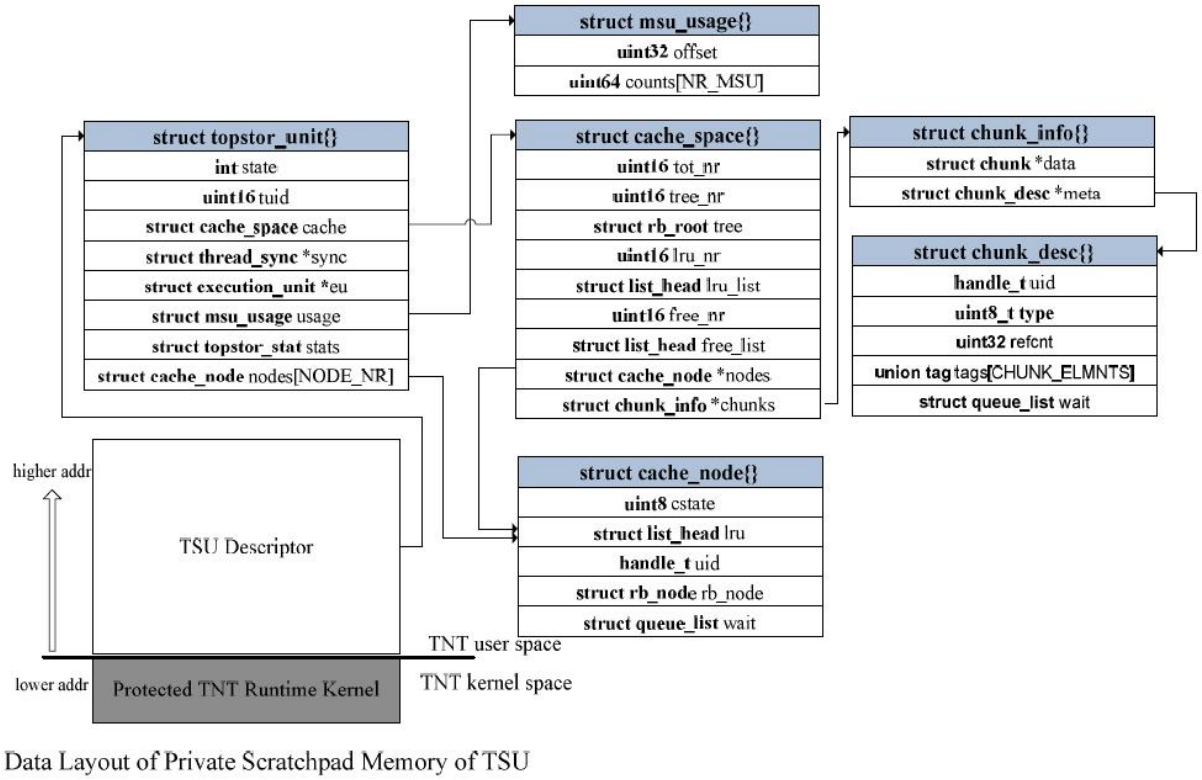

Figure 6: The data layout of the scratchpad memory of a TSU

cache, then the chunk will be moved to the MRU end of the list. If a chunk is accessed but misses in the cache and there is no free chunk slot, then the replacement handler will evict out the at the LRU end of the list to make to to store the newly accessed chunk.

Cache Index Structure: The cache index structure is used to map the UID of a chunk back to its location in the L1 cache unit. To speed up the simulation, we choose the RED-BLACK tree as the cache index data structure. A RED-BLACK tree is a balanced binary search tree. The average search time of a RED-BLACK tree is much better than an arbitrarily built binary search tree, and its rebalancing overhead is acceptable. Since the cache node insertion and deletion operations are used in the cache miss handling, their rebalancing overhead is negligible as compared with the overall cache miss handling.

Cache Storage Capacity: The cache capacity of each TSU is set to 512 (defined by *CACHE_NODE_NR* in *tsu.h* file) chunks, which is equivalent to 64 KB. Therefore, total cache capacity at the L1 cache level is 2.5 MB.

Figure 6 shows the data layout of the scratchpad memory of a TSU. As we can see, the TSU descriptor is the only data structure stored in the scratchpad memory. It is a big data structure and contains all the necessary information for the EU to access its L1 cache. We will not describe the definition of the TSU descriptor and its embedded data structures.

tuid: The same as the field in the EU descriptor. It is a system-wide ID that identifies the

thread unit that serves as the TSU.

eu: This field points back to the associated EU. It is used to ease the simulation code implementation.

cache: This embedded data structure is of type *cache_space*. It describes the current usage status of the L1 cache. It is the key to the implementation of the L1 cache, and we will describe the data structure in further detail.

usage: This embedded data strucure is of type *msu_usage*. It is used to keep track of the storage usage of the MSUs at the second level. Having the knowledge of storage usage distribution at the second level, a cache access operation which needs to allocate a new chunk from the second level will direct the request to the MSU which currently has the lowest storage usage. The benefit is that, in the long run, storage usage and access traffic will be balanced across all storage units at the second level, thus avoiding access hot spot issues. As we can see in the figure, its *counts* field is an integer array with each entry recording the number of allocated chunks in the corresponding MSU.

nodes[]: This is an array of type *cache_node*. Each element in the array represents one cache entry which contains all information about a cached chunk copy. As we can see in Figure 6, *cstate* records the status of a cache entry. The status of a cache entry can be described by four flags. CFREE indicates that the cache entry has not yet been allocated. CLRU indicates that the cache entry is on the LRU list. CCACHE indicates that the cache entry is linked in the cache search tree. CVALID indicates that the chunk data represented by this chunk entry is valid. The *lru* field is the entry that links this cache entry into the LRU list, *uid* is the UID of the cached chunk and *rb_node* is the entry that links this cache entry into the cache search tree.

We now examine the definition of *cache_space*:

tot_nr: This field stores the total number of cache entries found on a single TSU. As previously stated, this number is set to 512.

tree_nr: This field records the total number of cache entries that are linked in the cache search tree.

tree: This field represents the root node of the cache search tree. Its definition follows the standard RED-BLACK tree algorithm with a small extension to facilitate cache operations.

lru_nr and *lru_list*: These two fields enable LRU list manipulations.

free_nr and *free_list*: These two fields enable the free list manipulations. At the beginning of the simulation, all cache entries are unused and they are collected in the free list. As the simulation progresses, cache access operations will allocate cache entries from the free list for the missed chunks.

nodes: This field points to the cache node entry array that is stored in the scratchpad memory of the TSU. This field will ease the code implementation.

chunks: This field points to the L1 cached chunk information array. Each entry in this array is of type *chunk_info*. As we can see from Figure 6, *data* points to the location of the chunk

data that is stored on the DRAM and *meta* points to the location of meta-data that is stored on the DRAM. We should notice here that as we mentioned at the beginning of this section, in our simulator design, both of the two storage levels share the same physical chunk storage copy. Therefore, the locations recorded by the *chunk_info* structure directly refer to the chunk's permanent storage locations at the second level.

As mentioned in Section 2, there is no activity running on the TSU during simulation. The EU runtime kernel code will directly access the L1 cache. In Section 4.1, we will provide a description of the implementation of the cache access operations that are used by the high level chunk operations at the EU side to manipulate the L1 cache.

4.1 Cache Access Operations

In this section, we give the implementation of the two most important cache access operations. The first operation is *topstor_read()* (in *tsu.c* file) and the other operation is *topstor_write()* (in *tsu.c* file). They are the two key routines that are used to serve chunk read and chunk write operations that are requested by the EU.

***topstor_read* operation:**

Step 1: Check whether the requested chunk is already in the cache. If it is already in the cache, go to the next step for cache hit processing. Otherwise, go to Step 5 for cache miss handling. The check operation is completed by *cache_search()* (in *cache.c* file).

Step 2: Since the chunk is already in the cache, we further check its status by macro *TSU_VALID()* (in *tsu.h* file). If it is valid, then go to the next step to get the chunk information from cache entry. Otherwise, go to Step 4 to wait for the current chunk read operation to complete. The concurrent chunk read issue is due to the fact that there are multiple active tasks running on the EU. Suppose that one of the tasks issues a read operation to a chunk and the chunk is missed in the L1 cache. Then the task needs to be blocked out to wait for the read operation to be completed by the MSU. While at the same time, another task that takes over control of the EU will issue a read request to the same chunk. Once this happens, the second task will see the cache entry in the L1 cache but its status is set to invalid since the MSU has not yet completed the request which was issued by the first task. As can be seen in Step 5, the cache miss handling will first create a new entry in the L1 cache before it sends the request to the MSU.

Step 3: Get the chunk information from the cache node which is stored in the *chunk_info* structure. The chunk information needed to complete a chunk read operation at the EU side are the locations of chunk data and meta-data. This enables the EU runtime codes to copy the defined data elements from the DRAM to the application provided buffer. This copy operation is performed by *get_chunk_info()* (see *cache.h* file). After this step, the cache hit handling is completed, then jump to Step 10 to return.

Step 4: Invoke *tsu_request_wait()* (in *tsu.c* file) to wait for the completion of the ongoing chunk read operation. The function inserts the current application task slot into the wait quete

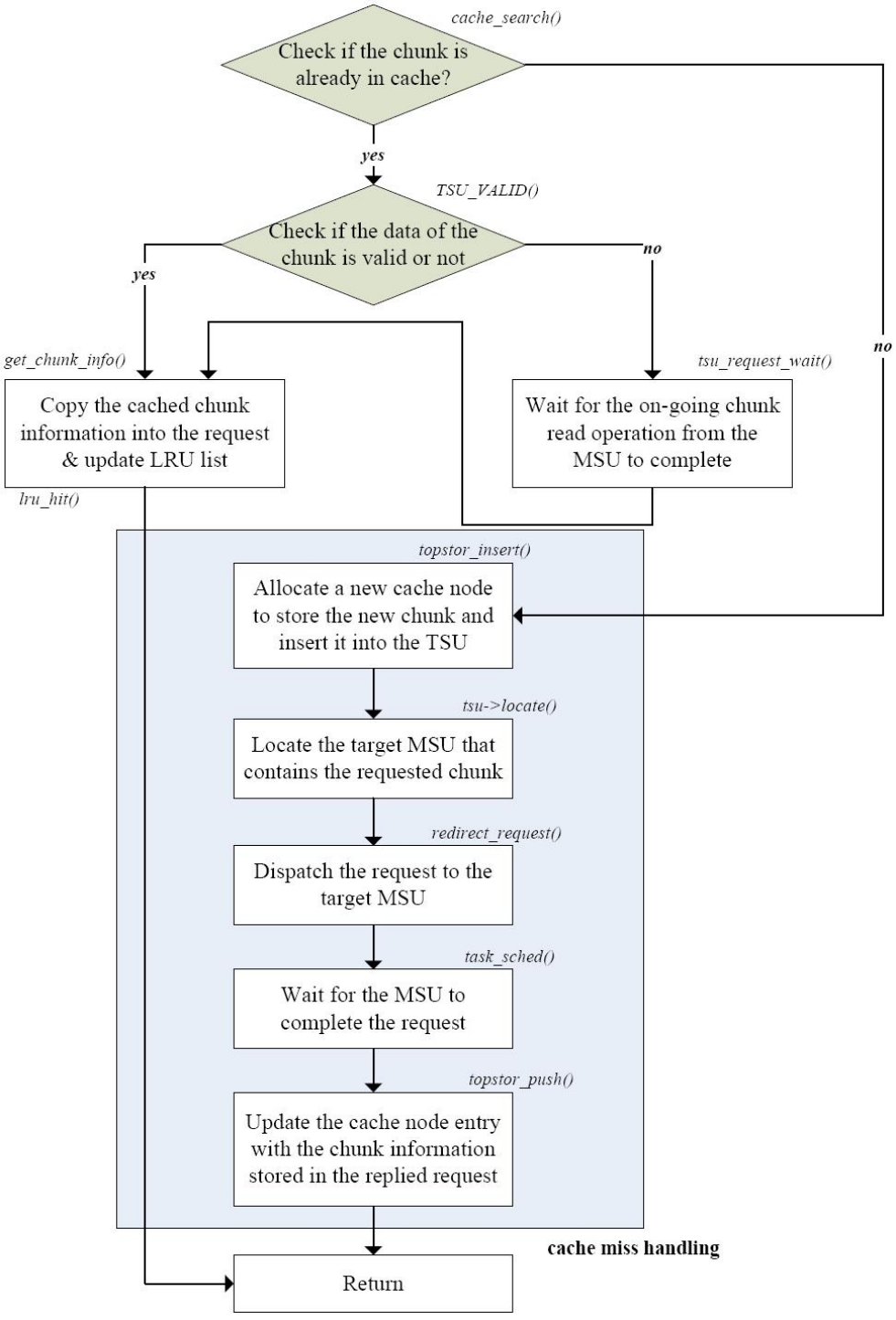

Figure 7: The diagram of *topstor_read()* operation

headed at the *wait* field of the cache entry, and then calls *task_sched()* to relinquish the EU. Once the function returns, the previous chunk read operation has been completed and jumps

back to Step 3 to complete the chunk read operation.

Step 5: Since the chunk is not in the cache, invoke *topstor_insert()* (in *tsu.c* file) to allocate a new cache node and also insert it into the L1 cache. The concrete operations include inserting the new cache node into the LRU list and the cache search tree. We should note that at this time, cache entry status is set to invalid.

Step 6: Based on the chunk UID, the *tsu->locate()* method calculates the target MSU that stores the chunk. As stated in Section 2, the chunk storage space is linearly divided among the 64 MSUs, so the calculation is a simple division. Once we locate the target MSU, we go to the next step to dispatch the request to the target MSU.

Step 7: Invoke *redirect_request()* (in *req.c* file) to insert the request to the head of the pending request queue of the MSU (see Section 5 for more details).

Step 8: Invoke *task_sched()* to relinquish the EU to wait for the request completion by the MSU. Once the function returns, the MSU has completed the read request.

Step 9: Invoke *tsu_push()* (in *tsu.c* file) to update the cache entry with the information recorded in the replied request as well as change the status of the cache entry to valid. At the same time, *tsu_push()* will call *tsu_request_wake()* to wake up any tasks that are blocked on this cache entry.

Step 10: All work is completed. Return to the caller.

***topstor_write* operation:**

Step 1: Invoke *tsu->select()* method to pick up an MSU to allocate a new chunk for this write operation. As stated earlier in this section, the selected MSU should be the one with the lowest chunk usage.

Step 2: Invoke *redirect_request()* to insert the request to the head of the pending request queue of the MSU.

Step 3: Invoke *task_sched()* to relinquish the EU to wait for the request completion by the MSU. Once the function returns, the MSU has completed the write request.

Step 4: Invoke *tsu_push()* to allocate a new cache entry and then fill in the cache entry with the chunk information recorded in the replied request. Next, insert the initialized cache entry into the cache system. It is worth noting that *tsu_push()* is used by both read and write processing. Its implementation is comparatively complex as it will first check whether the chunk is already in the cache. If it is already there, then the function only needs to update its content, which is the case for the chunk read operation. Otherwise, it will call *tsu_insert()* to perform the additional work to create a new cache entry for the chunk. That is the case for the chunk write operation.

Step 5: All work is completed. Return to the caller.

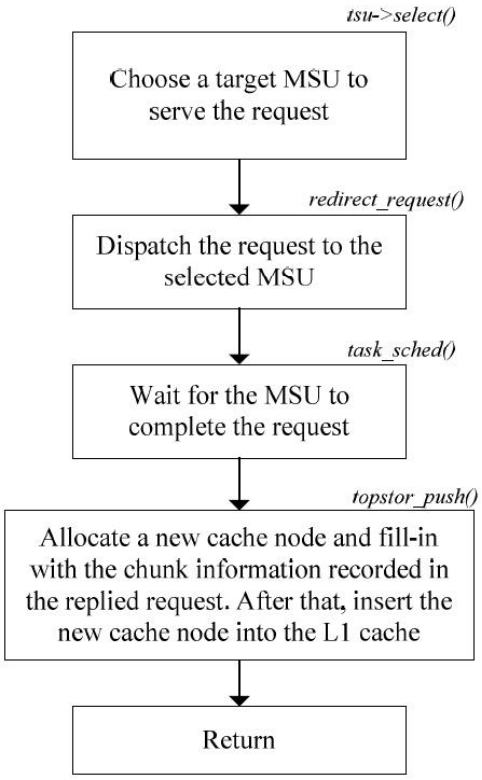

Figure 8: The diagram of *topstor_write()* operation

5 The Second Storage Level: the Main Memory

This section describes the internal design of the second storage level. The second storage level is used in our simulator to simulate the main memory in the envisioned Fresh Breeze system. As we mentioned in Section 2, there are 64 MSUs in the second level with each MSU containing about 8 MB of storage space. This yields a total storage capacity of 512 MB at the second level. In our current simulator design, the entire storage space is divided across the 64 MSUs in a linear manner for simplicity. Of course, any advanced non-linear space division approach is highly encouraged by the Fresh Breeze memory model.

Figure 9 gives the data layout of the scratchpad memory of the MSU. As shown in the figure, the MSU descriptor is the data structure that is stored in the scratchpad memory. It includes several embedded data structures to present the status of the MSU. The most important one is *storage_space* which keeps track of the chunk allocations for that MSU. The *res_bm* field is a bitmap that is used to record whether each chunk stored in the MSU has been allocated or not. Since the second level is the lowest storage level in our current simulator, we use the bitmap to keep track of chunk allocation. If the bit in the bitmap is set to one, it means that the corresponding chunk has already been allocated, otherwise it has not been allocated. The *offset* field denotes the starting chunk UID of the storage space that is stored on that MSU. The *chunks* field points to the data chunk array which is stored on the external DRAM. The *metas*

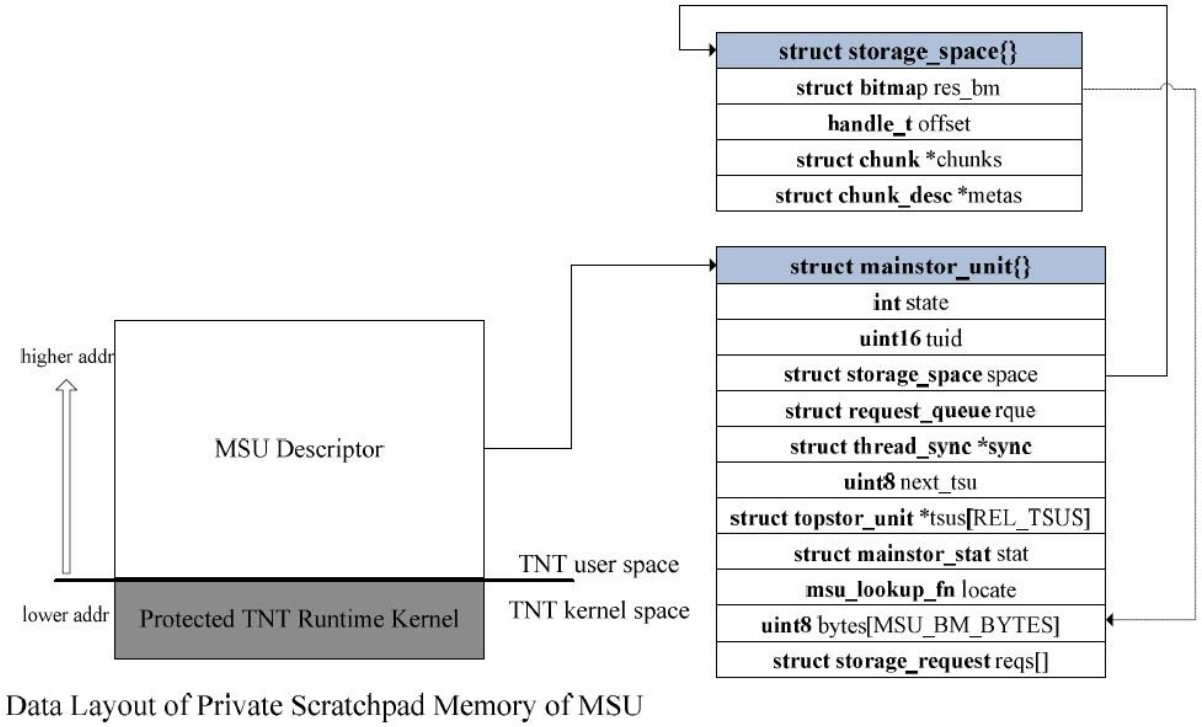

Figure 9: Data layout of scratchpad memory of an MSU

field points to the associated meta-data array which is also stored on the external DRAM.

The meta-data associated with each chunk is defined by the data structure *chunk_desc*. Its definition is shown in Figure 6. Before we explain the main memory access operations, we need to examine its definition.

uid: The chunk UID that uniquely identifies the chunk in the system. It is a 64 bit integer.

type: This field defines the type of the chunk. There are three types of chunks (defined in *chunk.h* file). CHK_DATA indicates the chunk is a normal chunk; CHK_JOIN1 indicates the chunk is a join ticket chunk; CHK_JOIN2 indicates the chunk is a join data chunk. We will describe the latter two special types of chunks in Section 7, which are utilized to support the task execution model.

refcnt: This is the usage reference count of a chunk. It consists of two subfields. The higher order 16 bits count the references held by the L1 cache copies and the lower order 16 bits count the references held by the application task. A chunk can only be recycled when both its user reference count and cache reference count go down to zero. Otherwise, the chunk cannot be recycled. The user reference count will be released by the Fresh Breeze API call *chunk_down()* (in *fb_exm.c* file). The cache reference count will be released when the chunk has been evicted out of an L1 cache.

There are two things worth noting here. The first thing is that once the user reference count reaches zero, the chunk can no longer be used since the application has already finished using

it. The other thing is that there can be such a situation where, although the application is no longer using the chunk, some TSUs still keep the unused chunk copy in their L1 cache. Once this happens, the chunk will stay in the memory until the last TSU has evicted the chunk out of its L1 cache.

tags: This is a byte array with 16 (where CHUNK_ELMNTS equals 16) elements. Each byte in the array defines the data semantics of the chunk data element. As each chunk has 128 bytes, there are 16 data elements in a single chunk. A data element in a chunk can be either a scalar data value (*DATA64*) or a pointer to another chunk (*HANDLE*). We also allow partially defined data chunks, which means that some of its data elements can be left undefined (*UNDEF*).

During the simulation, the MSU thread will loop over the storage requests sent from the EU or some other MSUs (implemented by *msu_main()* in *msu.c* file). The latter is for recursive garbage collection. The routine that is used to service the storage request is *msu_execute()* (in *msu.c* file). Its implementation is quite similar to *eu_execute()* as it will invoke a specific request handler according to the request type. In the following subsection, we will describe the implementation of two important main memory access operations: read and write.

5.1 Main Memory Access Operations

In this subsection, we describe the implementation of two major main memory access operations (read and write), which are implemented by the routines *mainstor_read()* and *mainstor_write()* respectively (in *msu.c* file).

mainstor_read operation

Step 1: Invoke *msu_find_chunk()* (in *msu.c* file) to find the locations of the requested chunk data and its meta-data in the external DRAM. Inside this function, it will perform a sanity check on the access, such as whether there is any user reference count held on this chunk. As we know, in the Fresh Breeze memory model, a chunk can receive a read operation only after it has been written and sealed in the storage system, which will hold an initial user reference count on this chunk. Based on that, we can infer that the system is incorrect if the MSU receives a read request to a chunk whose user reference count is zero.

Step 2: Increase the cache reference count on this chunk for this read operation. As we know, the EU will hold a copy for this chunk in its L1 cache. This is performed by *inc_chunk_copies()* (in *chunk.h* file).

Step 3: Save information related to the request, such as the chunk locations and the current chunk usage of this MSU, etc. This information will enable the EU to proceed with the read request and update its local second-level storage usage status.

Step 4: Reply back to the pending request queue of the EU. This is performed by *reply_request()* (in *req.c* file).

mainstor_write operation

Step 1: Invoke *msu_alloc_chunk()* (in *msu.c* file) to allocate a data chunk.

Step 2: Similar to Step 2 of *mainstor_read()*, we then increment the cache reference count of this newly allocated chunk for this initial write operation.

Step 3: Similar to Step 3 of *mainstor_read()*, we save information related to the request, such as the chunk location and the current chunk usage of this MSU, etc.

Step 4: Reply back to the pending request queue of the EU.

6 The Implementation of Chunk Operations

This section describes the implementation of the storage access interface. The storage access interface enables the application tasks to manipulate the data chunks and construct the data structures as needed by the program. This interface represents the storage part of the Fresh Breeze API. Section 6.1 describes the prototype syntax of the interface. Section 6.2 discusses its internal implementations.

6.1 Storage Access Interface

In our simulator, we support four types of chunk operations that are defined as part of the Fresh Breeze API. We will now discuss the programming interface of these operations.

int chunk_read(handle_t uid, struct chunk *buff, union tag *tags)

Description: This function is used to read a data chunk from the storage system into the application provided buffer. Along with the chunk data, this function will also return the tag information associated with this data chunk, enabling the application program to check the types of its data elements.

Parameters:

uid: The UID of the chunk to be read.

buff: The application provided buffer which is used to receive the chunk data from the storage system.

tags: The application provided buffer which is used to receive the chunk tags from the storage system.

int chunk_write(handle_t *uid, struct chunk *buff, union tag *tags)

Description: This function is used to write a chunk of data into the storage system. The application program not only needs to provide the buffer that contains chunk data which needs to be sealed, but also the buffer that stores the tag information associated with the chunk data. The latter information will be used by the storage system to define the types of data elements located in the chunk. In addition, the application programs needs to provide a 64 bit integer pointer that will be used by the storage system to save the UID of the newly allocated ata

chunk. The application program can later retrieve the previously written chunk by using the returned chunk UID.

Parameters:

uid: This points to a 64 bit integer that will be used to receive the UID of the newly sealed chunk.

buff: The application provided buffer which is used to pass the chunk data to the storage system.

tags: The application provided buffer which is used to pass the associated chunk tags to the storage system.

void chunk_up(handle_t uid)

Description: This function is used by the application to hold a reference count on a chunk. It should be called by the application program before a task starts to use a chunk.

Parameters:

uid: The UID of the requested chunk.

void chunk_down(handle_t uid)

Description: This function is used by the application to release a reference count on a chunk. It should be called by the application program once a task has finished its usage of a chunk.

Parameters:

uid: The UID of the requested chunk.

6.2 Internal Implementations

This subsection will use *chunk_read()* and *chunk_write()* as examples to illustrate how the simulator implements the storage access API.

chunk_read operation

Step 1: Invoke *task_alloc_request()* (in *req.c* file) to allocate a free request. The allocated request will be used to store the request information (see Step 2), and later will be used to save the request processing result from the MSU.

Step 2: Invoke *build_read_request()* (in *req.c* file) to set up the request accordingly. This will save the application provided buffer pointers into the request along with the requested chunk UID and also set the request type to *READ*.

Step 3: Invoke *topstor_execute()* (in *tsu.c* file) to access its private L1 cache to service the request. The *topstor_execute()* function is similar to *eu_execute()*. It will invoke a specific request handler based on the request type. In this case where it is a read request, then the *topstor_read()* handler will be invoked. As shown in Section 4.1, when this function returns,

the request has completed whether it is served from the L1 cache (cache hit) or from the main memory (cache miss).

Step 4: Invoke *complete_read_request()* (in *req.c* file) to copy the chunk data and its associated tag information from somewhere in the storage system (or specifically, the external DRAM) to the application provided buffer. The request will contain all necessary I/O information to enable the copy operations. The tag information can accelerate the copy operations as the undefined data elements will not need to be copied.

Step 5: Invoke *free_request()* (in *req.c* file) to free the request.

chunk_write operation

Step 1: Invoke *task_alloc_request()* to allocate a free request.

Step 2: Invoke *build_write_request()* (in *req.c* file) to set up the request accordingly.

Step 3: Invoke *topstor_execute()* to access its private L1 cache to service the request. The *topstor_write()* handler will be invoked by this function to handle the write request. The *topstor_write()* function has already been discussed in Section 4.1. Once this function returns, the request has been completed by the storage system.

Step 4: Invoke *complete_write_request()* (in *req.c* file) to store the application provided chunk data and its associated tag information into the locations specified by the storage system. The locations are where the newly allocated chunk is stored by the storage system. The function also needs to save the UID of the new chunk into the user provided buffer. It is worth noting that in our simulator design, we delegate all data copy operations to the EU side (the same for the chunk read case). This design choice is based on the features of the Cyclops64 architecture. The goal is to reduce the data transfer time between scratchpad memory and external DRAM in order to speed up the simulation.

Step 5: Invoke *free_request()* to free the request.

7 The Implementation of the Spawn-Join Mechanism

This section describes the spawn-join based parallel task execution model in the Fresh Breeze system. Section 7.1 discusses the principles of the spawn-join mechanism used in the Fresh Breeze system. Section 7.2 describes the syntax of the task management interface exported to the application program of the spawn-join mechanism. Section 7.3 describes the internal implementation of the interface.

7.1 The Principle of the Spawn-Join Mechanism

The spawn-join based parallel task execution model supported by the Fresh Breeze system requires the application programmer to split a large computation procedure into a number of phases. Each phase will consist of a group of small tasks. According to the spawn-join

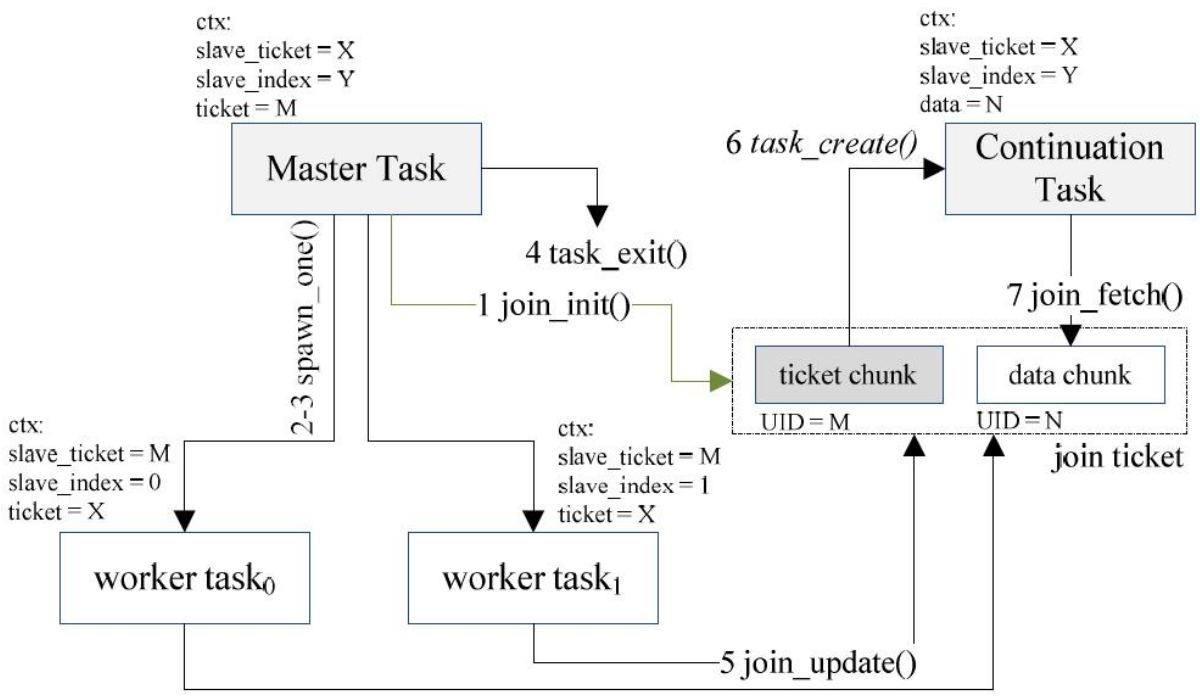

Figure 10: Join Ticket and its Relationship with Master Task, Continuation Task and Worker Tasks

mechanism, the roles of the tasks in the group are different. There are types of tasks: master task, continuation task and worker task. We will now discuss the functions of these three different types of tasks and how they cooperate to finish a computation job in parallel.

Master Task: This is the initial task in a task group. There is only one master task per group. The job of the master task is to initialize the computation and fork a group of worker tasks to perform the computation.

As shown in Figure 10, the master task will first invoke *join_init()* (in *fb_exm.c* file) to create a join ticket. As we saw in Figure 10, the join ticket consists of two chunks. One chunk is the join ticket chunk which controls the join operation involved with the spawned worker tasks. The other chunk is the join data chunk which is used to collect the results sent by the worker tasks. Next, it will create a number of worker tasks by invoking *spawn_one()* (in *fb_exm.c* file). Each invocation will create a new worker task. For each worker task, the master task needs to specify the task function, input arguments and join index. As we will see below, the join index specifies the element in the join data chunk that saves the result produced by this particular worker task. Different worker tasks must not share the same join index. In the example shown in Figure 10, the master task has created two worker tasks. The left worker task has an index value of 0, while the right worker task has an index value of 1. As we can see from the figure, the spawn-join related information of a task is stored in *task_context*. The three fields contained in this data structure are *slave_ticket*, *slave_index* and a union of *ticket* and *data*. As a worker task, *slave_ticket* is the UID of the join ticket created by its master task and *slave_index* is the

struct join_ticket{}
uint64 count
uint64 limit
uint64 mask
handle_t data
uint64 main
uint64 nr
uint64 args[MAX_ARGS]
handle_t slave_ticket
uint64 slave_index

Figure 11: The definition of the join ticket chunk

join index. Depending on the role of the task, the union field has different uses which will be explained in the remainder of this section.

Referring back to Figure 10, once the master task has spawned the worker tasks, it has completed its work. It will then invoke *task_exit()* to terminate. It should be noted that due to the limited size of a chunk, there can be at most 16 worker tasks participating in a join operation.

Worker Task: The worker tasks are the workhorse of the group. They are created by the master task. Once they have completed their computation, they send the computation result to the join ticket by invoking *join_update()* (in *fb_exm.c* file). As we can see from Figure 10, once all worker tasks have completed, the continuation task will be created automatically by the runtime kernel using *task_create()*. First, the task function and input arguments of the continuation task are specified by the master task in the join ticket chunk. When the last worker task reaches the join ticket, the storage system will notify the EU which hosts that last worker task about this event. The notification is piggybacked in the reply request to the join update operation along with the necessary startup information for the continuation task, which is previously stored in the join ticket chunk. Sometime later, when the EU runtime kernel checks the replied request, it will create the continuation task accordingly.

Continuation Task: The continuation task is designed to retrieve the results from the worker tasks, and after post-processing, it may also initiate the next phase of computation with the role of a master task. As we can see from Figure 10, the continuation task inherits the values of *slave_ticket* and *slave_index* from the master task. It is worth noting that for a master task, the union field of *task_context* is cast to *ticket* which is the UID of the created join ticket chunk. However, for a continuation task, the union is cast to *data* which is the UID of the created join data chunk. The continuation task utilizes the UID stored in *data* to fetch the computation result produced by the worker tasks. This is performed by invoking *join_fetch()* (in *fb_exm.c* file). Since the continuation task may initiate the next phase of computation, the union field then needs to be reused as a *ticket*. As a result, the implementation of *join_fetch()* will reset the *data* field to zero after loading the join data chunk.

Figure 11 provides the definition of the join ticket chunk, which plays a critical role in the implementation of the spawn-join based task management mechanism under the Fresh Breeze memory model. Its content is defined as follows:

count: The number of worker tasks that have completed their computation and arrived at the join point. As we know from the above discussion, once a worker task has finished its job, it will call *join_update()* to send its computation result to the join ticket. The MSU that stores the join ticket will then update the *count* field of the join ticket chunk and save the computation result into the element of the associated join data chunk which is specified by the worker task's slave index.

limit: The total number of worker tasks that are involved in the join operation. This parameter is set by the master task in the join init operation. As we previously mentioned, a data chunk can contain at most sixteen data elements, so the number of worker tasks that can be involved in a single join operation can also be at most sixteen.

mask: This is a sixteen bit vector that is used to mark the arrival of the worker tasks. For example, if the 7th bit in the mask vector is set, then it indicates that the worker task whose slave index is 7 has arrived at the join point.

data: This is the data chunk handle which points to the associated join data chunk.

main: This is the task function pointer of the continuation task of this join operation.

nr: This is the number of input arguments to the continuation task. It is worth noting that under the Fresh Breeze architecture, the number of input arguments to a task function can be at most eight. Such a limitation is due to the limited size of a join ticket chunk.

args: This is the input argument array of size 8 (when MAX_ARGS is equal to 8).

slave_ticket: This is the UID of the join ticket chunk that the master task joins as a worker task. As we know that the continuation task will inherit the system execution context from its predecessor - the master task, this value will be set for the continuation task before it begins to run on the EU.

slave_index: This is the slave index of the master task in the join operation which is identified by the *slave_ticket*. It is also part of the system execution context that the continuation task will inherit from its master task.

7.2 Task Management Interface

In our simulator, we have defined five task management operations. They represent the task management portion of the Fresh Breeze API. We will now discuss the programming interface of these five operations one-by-one.

int join_init(int limit, void *func, struct arg_info *args)

Description: This function is used by the master task to create a join ticket to start a new phase of computation.

Parameters:

limit: The number of worker tasks involved with this join operation.

func: The task function pointer of the continuation task.

arg: The input arguments to the continuation task.

int spawn_one(int index, void *func, struct arg_info *args)

Description: This function is used by the master task to create one worker task.

Parameters:

index: The join index of the worker task as specified by the master task.

func: The task function pointer of the worker task.

arg: The input arguments to the worker task.

void join_update(int type, data_t val)

Description: This function is used by the worker task to send a data value or record to the join ticket.

Parameters:

type: The type of the data element.

val: A simple scalar value or a record handle which is specified by the parameter *type*.

handle_t join_fetch(void)

Description: This function is used by the continuation task to retrieve the UID of the join data chunk which contains the computation results produced by the worker tasks.

Parameters: NULL

void task_exit(int done)

Description: This function is used by any application task to terminate execution explicitly. It is worth noting that the runtime kernel will implicitly terminate the task execution inside the *join_update()* function.

Parameters:

done: This value indicates the type of task termination. It can take any one of the following three values: 0 indicates that it is a normal task termination requested by the application task; 1 indicates that the task termination is requested by the runtime kernel but not the application task. In the implementation of *task_exit()*, if *done* is set to 1, then it does not need to invoke *task_sched()* to change execution context as we are already in the runtime kernel context; 2 indicates that the entire application is finished, so *task_exit()* needs to flag all functional units in the simulator to stop.

7.3 Internal Implementations

This subsection describes the implementation of the task management API calls.

join_init operation

The *join_init()* function accepts the number of worker tasks involved in this join operation, the task function pointer, and input arguments of the continuation task as its arguments. It will then invoke *create_join_ticket()* (in *fb_exm.c* file) to perform the following steps.

Step 1: Invoke *task_alloc_request()* to allocate a free request.

Step 2: Invoke *build_create_request()* (in *req.c* file) to set up the request accordingly.

Step 3: Invoke *topstor_execute()* to send the request to an MSU to allocate and initialize two special chunks for the join ticket. One chunk is the join ticket chunk (of type *CHK_JOIN1*) and the other is the join data chunk (of type *CHK_JOIN2*). The request handler installed at the TSU to handle the join init request is *topstor_create()* (in *tsu.c* file). The function will select the MSU which currently has the lowest chunk usage to service the request, and then put the request into the pending request queue of the selected MSU. Next, the function calls *task_sched()* to relinquish the EU to wait for completion from the selected MSU. Therefore, when *topstor_create()* returns, the join ticket has been created inside the storage system. We will now describe how the MSU services the join init request.

Step 4: Fetch the UID of the created join ticket chunk from the acknowledged request.

Step 5: Call *free_request()* to free the request.

Step 6: Return the UID of the join ticket chunk to its caller, the *join_init()* function. The latter function will save the UID in the ticket field of the current task's execution context structure (see Figure 10).

As we mentioned above, the creation of two special chunks is performed by the MSU. The function to handle the creation is *mainstor_create()*. Its implementation is straightforward.

Step 1: Call *msu_alloc_chunk()* to allocate a join ticket chunk of type *CHK_JOIN1*.

Step 2: Call *msu_alloc_chunk()* to allocate a join data chunk of type *CHK_JOIN2*.

Step 3: Call *init_join_ticket()* (in *chunk.c* file) to initialize the join ticket chunk. This consists of setting up the connection between the join ticket chunk and join data chunk by setting the *data* field to point to the join data chunk (see Figure 11). The join ticket is configured by setting the *limit*, *count* and *mask* fields. The task startup information of the continuation task is saved in the *nr*, *args* and *main* fields.

Step 4: Save the UID of the created join ticket chunk in the acknowledged request to notify the EU runtime kernel. The EU runtime kernel will save the UID in the *ticket* field of the execution context of the currently running task (which should be a master task). This value will be used by the following Fresh Breeze function call *spawn_one()* when it creates the worker task for this join operation.

Step 5: Call *reply_request()* to reply to the EU.

Figure 12 shows the implementation diagram of the join init operation in the Fresh Breeze simulator.

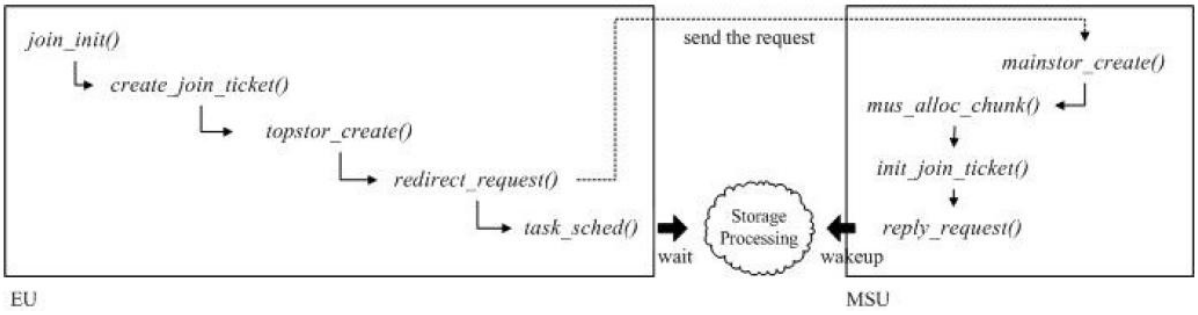

Figure 12: The diagram of a join init operation in the Fresh Breeze simulator

spawn_one operation

The *spawn_one()* function is called by the master task after *join_init()* to create a worker task for the join operation. Unlike *join_init()*, there are no storage operations involved in *spawn_one()*. Its work is performed entirely by the EU runtime kernel. The *spawn_one()* function is simply a wrapper function for *task_create()*, which we have already discussed in Section 3. It is worth noting that in *init_task_info()*, the runtime kernel will save the spawn-join related system context information in the new worker task slot (slave ticket and slave index). The slave index is specified by the input argument to *spawn_one()* and the slave ticket is set to the value of the *ticket* field from the current task's context, which is the UID of the join ticket chunk created by the preceding *join_init()* function.

join_update operation

The *join_update()* function is used by the worker task to send its own computation result to the join ticket. This function sends a join update request to the MSU that stores the join ticket chunk and then terminates the worker task. When the MSU completes a join update request from any non-final worker task, it will simply free the request after the update operation. However, if it is the update request from the last worker task, the MSU will reply to the request to tell the EU that executes the last worker task about the completion of the join operation. This is done by setting a flag (*STATUS_CONT* in *req.h* file) in the replied request. The MSU will also store the continuation task startup information inside the replied request to enable the EU to start it. The implementation of *join_update()* is shown as follows.

Step 1: Call *update_join_ticket()* (in *fb_exm.c* file) to send a join update request to the corresponding MSU. The join update request should include the slave index of the worker task and its computation result which can be either a scalar 64-bit value or a chunk pointer to a data record. It is worth noting that after dispatching the update request to the MSU, *update_join_ticket()* will return immediately without waiting for the completion of the request from the MSU, as the worker task has finished its work and can terminate immediately.

Step 2: Call *task_exit()* to terminate the task.

On the MSU side, the join update service routine is *mainstor_update()* (in *msu.c* file). Its implementation is shown as follows.

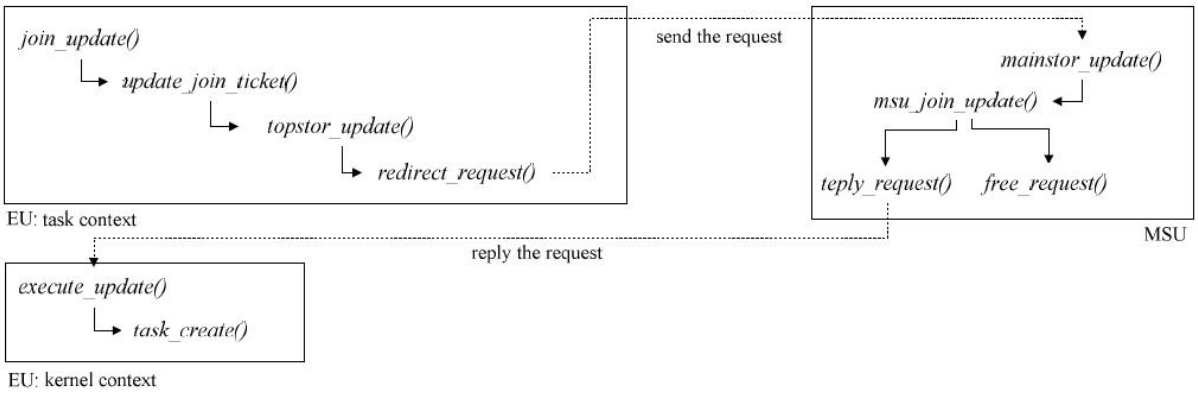

Figure 13: The diagram of a join update operation in the Fresh Breeze simulator

Step 1: Call *msu_join_update()* (in *msu.c* file) to update the status of the join ticket chunk, such as incrementing the *count* field. It also saves the computation result included in the update request into the corresponding data element of the join data chunk.

Step 2: Check whether the count value of the join ticket chunk equals its limit. If it is equal to the limit, then the MSU will return the request to the EU to notify it about the completion of the join operation and will also attach the necessary task startup information for the continuation task and the UID of the join data chunk into the returned request. Otherwise, the MSU will simply free the update request.

The service routine of the EU to service the replied update request is *execute_update()* (in *eu.c* file). This routine retrieves the continuation task information from the replied request and then creates a new task on its local pending task queue.

Figure 13 shows the implementation diagram of a join update operation in the Fresh Breeze simulator.

join_fetch operation

The *join_fetch()* function is used by the continuation task to retrieve the computation results of the worker tasks from the storage system. The function returns the UID of the join data chunk. Then the continuation task can use the UID to read the chunk data from the storage system. It is worth noting that the EU runtime kernel will automatically reset the *data* field of the continuation task's system context. This field will later be reused to store the UID of the join ticket chunk if the continuation task needs to start a new computation phase.

References

[1] J. del Cuvillo, W. Zhu, Z. Hu, and G. R. Gao. FAST: A functionally accurate simulation toolset for the Cyclops 64 cellular architecture. Technical report, University of Delaware, 2005.

[2] J. del Cuvillo, W. Zhu, Z. Hu, and G. R. Gao. Tiny threads: A thread virtual machine for the Cyclops 64 cellular architecture. In *Proceedings of the 19th IEEE International Parallel and Distributed Processing Symposium*. IEEE Computer Society, 2005.

[3] J. B. Dennis. Fresh Breeze: a multiprocessor chip architecture guided by modular programming principles. *ACM SIGARCH Computer Architecture News*, 31(1):7–15, 2003.

[4] J. B. Dennis. The Fresh Breeze model of thread execution. In *Workshop on Programming Models for Ubiquitous Parallelism*. IEEE, 2006. Published with PACT-2006.

[5] J. B. Dennis, G. R. Gao, and X. X. Meng. Experiments with the Fresh Breeze tree-based memory model. In *International Symposium on Supercomputing, Hamburg*, June 2011.

[6] M. Frigo, C. E. Leiserson, and K. H. Randall. The implementation of the Cilk-5 multithreaded language. *ACM SIGPLAN Notices*, 33:212–223, May 1998.