School of Computer Science

3rd year project report

Application-specific Real-Time garbage collection for neural simulations on SpiNNaker

Mantas Mikaitis

BSc Computer Science supervised by
Dr. David Lester

May 1, 2016

Application-specific Real-Time garbage collection for neural simulations on SpiNNaker

Mantas Mikaitis, BSc
supervised by Dr David Lester
University of Manchester

***Abstract*—The limitations of internal processor memory space of the first generation SpiNNaker computer have always proven to be a big consideration when designing memory-dependant applications. In simulations of brain activity, many biologically plausible learning rules require history traces of each neuron's activity to be stored. The history traces rapidly fill the internal memory space, therefore we must introduce a memory management routine working in the background, which must additionally respect the biological timing constraints of the SpiNNaker simulations. Real-time garbage collection is an automatic memory management technique that can satisfy these requirements. This study presents the first ever implementation of real-time garbage collector for SpiNNaker architecture and evaluates the performance, carefully considering the biological real-time constraints of the system.**

***Index Terms*—garbage collection, automatic memory management, hard real-time systems**

CONTENTS

The source code of this project is available in the official SpiNNaker repository [20].

I. INTRODUCTION

Garbage collection is the art of automatic memory management as subtly stated on the cover of the garbage collection handbook [18]. The simplest task of garbage collection is memory compaction (Fig 1). To demonstrate, we allocate three objects, A, B and C on the memory heap using e.g. *malloc()* in C programming language. If object B eventually becomes inactive, i.e. the variable that used to point to it is assigned a different address, the memory that B occupies can be used for other purposes. The garbage collector's task is to enter while the main application is idle and do the following: Remove B object from memory and re-manage the locations of A and C in order to reclaim free space between them. After these steps, the space that B occupied is now residing in the whole block of free space at the end of the heap, where it can be re-used for new object allocation.

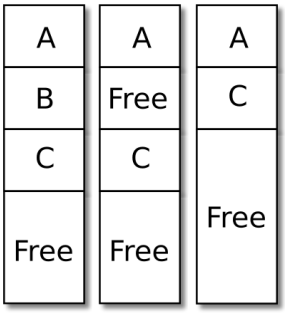

Fig. 1. Basic garbage collector operation: Reclaiming memory space occupied by a dead object B and compacting live objects A and C

It is estimated that 2^{13} events occur on a single core in SpiNNaker per each mili-second when simulation is run with biological time constraints [22]. Each of these events leave traces in the memory that are used in further simulation activities. If memory becomes full, some of the traces will be dropped out of memory unconditionally, even if they are still useful for simulation results. Therefore, the research into the automatic memory management on SpiNNaker is interesting because the limitations of the internal memory space of the SpiNNaker cores have many implications on a rapid simulation activities mentioned above. This study will provide useful insights into how memory can be managed on SpiNNaker, and where specifically automatic management could be useful for such architecture.

A. Goals of the project

The following points highlight the main goals of this study:

- Gain essential knowledge about SpiNNaker computer and neural simulations in order to develop and test applications. The background information containing the resulting findings about SpiNNaker is provided in Section II.
- Research two classical garbage collection algorithms: copying collector (Subsection III-A) and generational collector (Subsection III-B).
- Implement two aforementioned garbage collection algorithms on SpiNNaker. The most important aspects of the implementation phase are documented in Section IV.
- Investigate the most efficient copying functions that are available on SpiNNaker. The fastest copying operations were investigated in Subsection V-B and used in garbage collection operations that are most dependant on copying efficiency.
- Evaluate garbage collection on SpiNNaker, given the memory limitations described in Section II-B2, and biological real-time constraints of learning mechanisms introduced in Sections III-C and III-D. The analysis is given in Section V.
- Summarise the results of the study (Section VII) and provide pointers for further research (Section VI).

II. Background

The SpiNNaker project aims to create a massively parallel million-core computer, specifically constructed for large-scale neural network simulations, such as mammalian brain [1]. This chapter will introduce SpiNNaker in detail and identify the main applications that it targets. Additionally, the principles of garbage collection and the purposes of implementing it on SpiNNaker will be discussed.

A. Neuroscience

SpiNNaker is mainly targeted to Neuroscience research. For neuroscientists, SpiNNaker provides a framework to simulate neurons by providing a capability to describe them and their connections in programming languages, as well as observe the state of the neural network at any point of the running simulation [6].

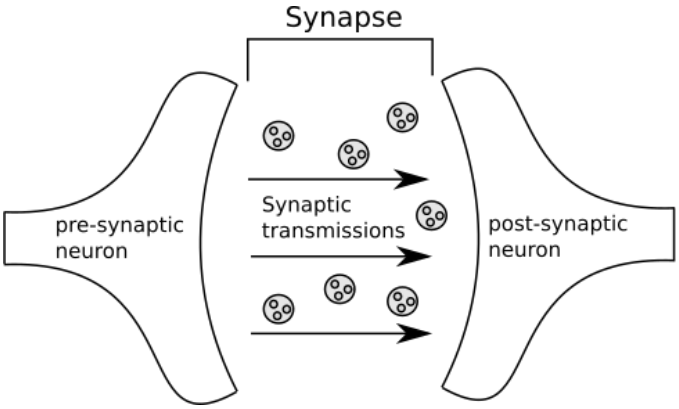

Fig. 2. A synaptic structure between neurons

One of the main applications that this project closely relates to is in Neuroscience category. Simulating the communications between neurons requires modelling synaptic plasticity. Biological neurons are connected together with chemical connections known as synapses. The strength of synapses changes overtime due to synaptic transmissions, the phenomena which is called synaptic plasticity. This scheme is demonstrated in Fig. 2, where channel labelled ”Synapse” is a synaptic structure between neurons that allows transmissions to take place (More specific type of plasticity is described in Sec. III-C1). In order to keep track of the progress of the simulation, SpiNNaker implementation stores information about each transition, and this information is further called history traces or post event traces of the neuron. Due to limited space on SpiNNaker hardware, we come to the memory management requirement: more neurons mean more history traces and therefore the higher storage requirements. In order to utilise the limited memory space most efficiently, we must manage the memory heap while the memory dependant application is running. At this point we start observing the need for an efficient garbage collector.

B. SpiNNaker Computer

In this subsection, the main details of the SpiNNaker chip will be briefly covered in order to demonstrate the limitations of the computer. The most important parts for this study are different memory types that are available on the chips.

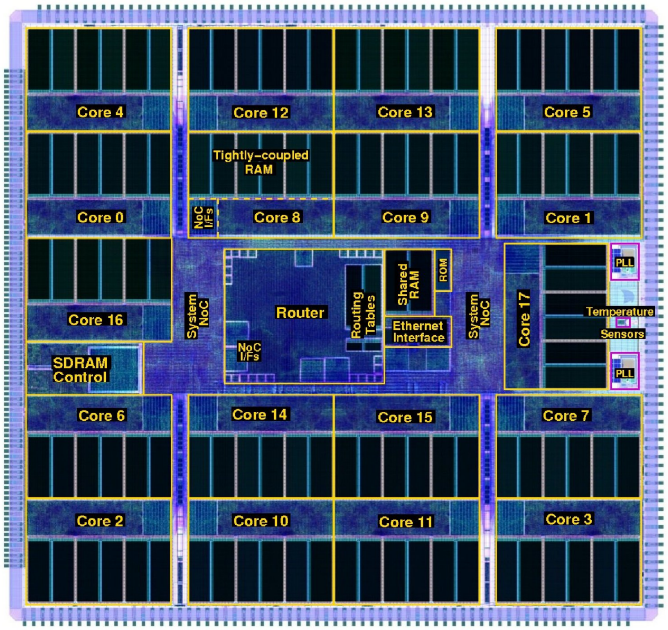

Fig. 3. SpiNNaker Chip

1) SpiNNaker Chip: SpiNNaker chip is made out of 18 ARM968 processors as well as a block of shared SDRAM of 128Mbytes. One of the cores is occupied with the SpiNNaker operating software and another one is used to improve the manufacturing yield. Therefore, 16 out of 18 cores are available for user applications. The software, that is used to implement and run simulations, allocates the neurons and other significant bits of the simulation to the 16 remaining cores. The allocation choice is made by considering the type of the simulation, e.g. size of the data structures that it requires. Typically each ARM968 in the SpiNNaker chip will be allocated up to 255 neurons.

2) ARM968 Core: ARM968 contains 64Kbytes of data storage memory, DTCM[1], and also 32Kbytes of instruction memory, named ITCM [9]. The compiled binary is downloaded onto ITCM and any data structures that are used while user's application is running, are stored in DTCM. DTCM is the main area of interest in this study as this is the memory space where allocable heap resides, and where hundreds of history traces are recorded while simulation is running. The available memory for a simulation is smaller than 64Kbytes as some space is reserved for a processor stack along with static uninitialised variables.

3) SpiNNaker Toolchain: SpiNNaker Application Run-Time Kernel(SARK) [9] is SpiNNaker's operating system that is loaded into ITCM together with user's application. It contains various libraries from standard C, that are optimised for SpiNNaker, as well as special functions to assist with event driven programming model[2].

[1]TCM - Tightly Coupled Memory. TCM is very close to CPU as it can be accessed on every cycle. Contrary to the ordinary memory, there are no caches involved when accessing TCM thus avoiding any indeterminacy associated with caches [21]

[2]In the event driven programming model, the flow of application is driven by interrupts that are caused by certain events, e.g. a clock tick or a receipt of data packet.

In order to run application on SpiNNaker, the source code written in C is cross-compiled[3] into SpiNNaker's binary file of format .aplx [12]. The compilation process is done using GCC or ARMCC compilers [11]. It is then linked together with SARK and other libraries from SpiNNaker's code base.

4) Development infrastructure: In order to test applications, the development board was accessible (Fig. 4). It contains 4 SpiNNaker chips that together comprise 72 ARM968 cores [10]. The board is connected to a host computer using Ethernet interface and host-side applications exist to communicate to the board from a machine running Linux, OSX or Windows operating systems.

There are various developer applications created to communicate with the board, that allow developer to load the application and investigate it's activities while it is running on SpiNNaker. Ybug [13] is an application written in Perl that allows to observe status of a loaded application, or download memory information from the board, given a core address and a memory location. For this project, Ybug was mainly used to observe how memory contents are changing while garbage collection is running on a single chip. Other higher level interfaces exist that can be used to conveniently load applications on hundreds of cores at once, by projecting C binaries into data structures resembling maps or networks.

C. Fundamentals of Garbage Collection

Garbage collection is a technique of optimising memory without interference of the programmer. One of the first examples of garbage collection can be recognised in the early

[3]Cross compilation refers to compiling an application on one architecture with the intention of running it on a different architecture. In our case, an application is compiled on x86 host machine while targeting ARM968 on SpiNNaker.

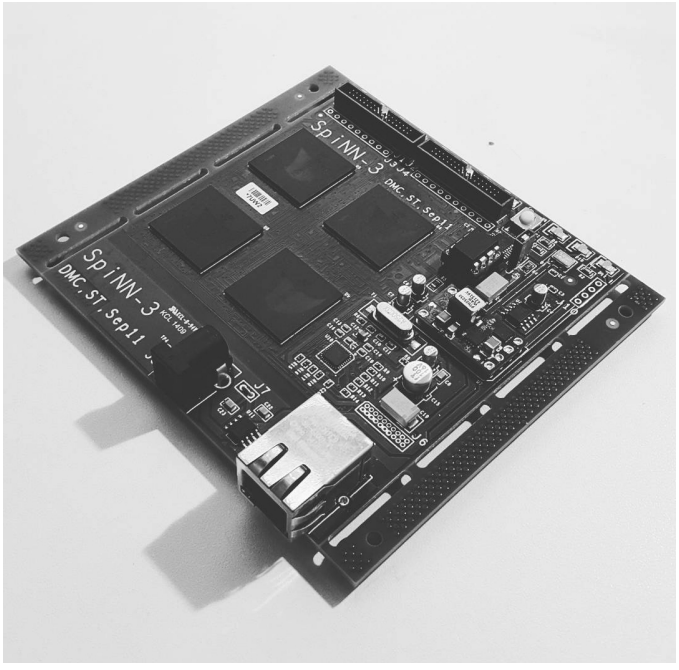

Fig. 4. 4-Node SpiNNaker development board

implementation of LISP programming language. The basic principle was detecting the non-free registers that are not referenced from anywhere in the application. Such a register may be considered abandoned by the program, because its contents can no longer be found by any process on the machine; hence we would like to reclaim the space, that the unused register occupies and recycle it for further use [2]. It is also common to demonstrate the principles of garbage collection by giving examples of well known programming languages. For example, we usually state that C is non-garbage collected language, i.e. programmer reclaims unused memory by using a function *free()*. On the other hand, Java has a large set of garbage collection techniques, and by default, garbage collection occurs as soon as programmer changes the only reference to a specific object to point to a different object.

In the late 1990s and early 2000s, a commercial adoption of programming languages allowed Real-Time garbage collection to be invented. Real-Time garbage collection is required when the specific system is considered to be a real-time system: object creation and access times of those objects are in the specific time boundaries [3]. A common example is a control of an aeroplane, where the delay between pilot's command and the occurrence of the action must be as minimal as possible. Because of this, in any given time window, garbage collection has to occur in an organised manner so that it would not violate the running periods of the mutator[4]. Conversely, the classical garbage collection is often called stop-the-world, since it pauses the execution of the mutator for the entirety of garbage collection duration before allowing it to continue execution.

D. Real-Time collection

As SpiNNaker has no global means of synchronisation, simulations are required to run in real-time. [1]. Therefore, SpiNNaker can be described as a hard real-time system, i.e. a system that upon receiving input events (e.g. neuron action descriptions) must make appropriate actions in a timely manner (e.g. send out all the packets to post event neurons before another time tick interrupts) [4].

Figure 5 demonstrates the mutator utilisation boundaries in real-time system and stop-the-world system. It can be observed that mutator stops at t_0 and must start at t_1. In real-time garbage collection, the given time window is respected as opposed to stop-the-world garbage collector, which stops the mutator for as long as it needs. Therefore, the following non-functional requirements can be established: real-time garbage collector must be optimised in such a manner, that would allow it to finish any collection processes before mutator operations take place at specific times; following from that, real time garbage collection process has to run more often, but respect the idle times of the mutator operations, as only at those idle times it can enter to collect garbage.

[4]An application that has allocated a set of objects and is periodically reading and writing them is called mutator. Mutator and collector are normally separated into two different entities as first demonstrated by Dijkstra et al [19] where two different processors were used for executing both programs.

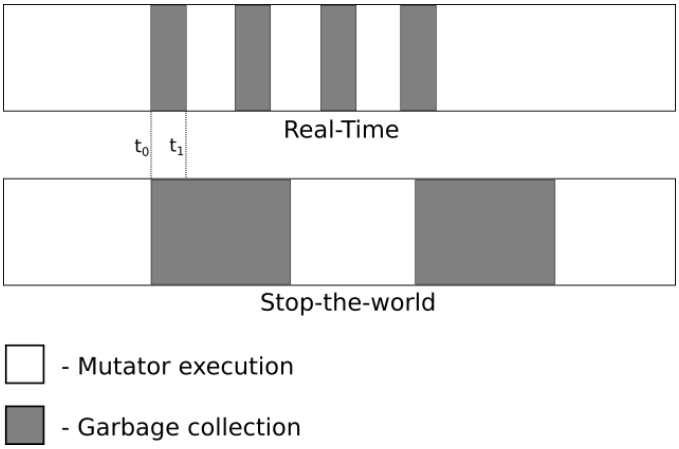

Fig. 5. Garbage Collection: Real-Time versus stop-the-world. Illustration inspired by [15].

III. Research

In this section the foundations, upon which SpiNNaker garbage collection is built, are laid out. Two classical garbage collection algorithms are of interest in this study: incremental copying collector and an optimisation of it - generational garbage collector. This section also contains information about synaptic plasticity, the main simulation application for which garbage collection will be applied.

A. Copying garbage collector

One of the first ideas of real-time garbage collection was studied and presented by H. Baker [5] with useful notes on it written by Lieberman et al [3]. Baker proposes the available memory space to be divided into two parts, called *fromspace* and *tospace* (Figure 6). The memory allocation (e.g. using malloc() in C or CONS in Lisp) is allowed only in *tospace* and the garbage collection process traces the accessible objects in *fromspace*, incrementally moving them to *tospace*. When no objects remain in *fromspace*, it can be used for allocating new objects. The operation called *flip* occurs which interchanges the roles of two spaces. Thus, after the flip, *tospace* becomes a free memory block that was previously labelled *fromspace*.

When an object is moved from *fromspace* to *tospace*, an invisible pointer is left in *fromspace* that will direct any access to *tospace* where object now resides. After any such access, the reference must be updated to point directly to the new location of the object in *tospace*.

When an object is evacuated to *tospace*, some of the components in the object might be pointing back to *fromspace*. Such pointers cannot persist as we need to recycle the whole space that *fromspace* occupies. The operation called *scavenging* is undertaken to remove such pointers. Scavenging copies all objects that are referenced from the particular object in *tospace*, to *tospace*, and updates the references. To achieve this, *tospace* is divided into two parts: creation and evacuation areas. Scavenging process linearly scans the evacuation area of *tospace*, looking for the back-pointers to *fromspace* and moves any objects from there to the same evacuation area

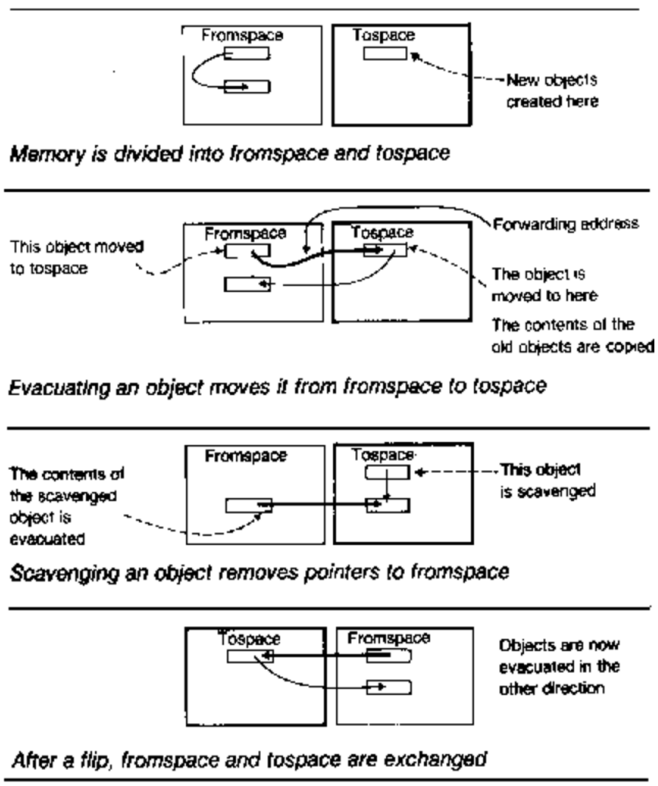

Fig. 6. Henry Baker's Real-Time Garbage Collector. Illustration from Lieberman's original paper [3].

that the parent object resides in. Creation area is only used for allocating new objects and thus does not need to be scavenged.

Baker's algorithm is a simple, yet efficient solution for garbage collection, therefore has been chosen as the foundation theory for this SpiNNaker project.

B. Generational garbage collector

Lieberman and Hewitt [3] have demonstrated an improvement over Baker's copying garbage collector, by introducing heuristic techniques to differentiate the rates of garbage collection of different memory regions. Their main idea is to implement a garbage collector that is based on the lifetimes of objects, i.e. if a particular group of objects is predicted to live in the system longer, we do not need to check whether they are dead as often as the ones that are more temporary.

The main principles of Lieberman's collector is to fragment the memory into small regions as opposed to Baker's division into two parts. Memory regions contain two values that allow us to control the rate of garbage collecting them: *generation number* and *version number*. When the region becomes used for storing objects, its generation number gets assigned a current generation number. The current generation number is then incremented. The garbage collection process is very similar to Baker's: copy all accessible objects from a region to a new space, scavenge all back-pointers and recycle the old space. When such region is garbage collected, its version number (now in a new location) is incremented. As a result of all above, the generation and version numbers tell us how old the region is and how much it was garbage collected. These two numbers allow to predict how relatively temporary or permanent the data on the specific region is, and therefore, enables control over how often we should put our processing power into scanning the objects in it.

The generational garbage collector is an appropriate solution for SpiNNaker, specifically, for synaptic plasticity memory management for two reasons: sequential arrival of neuron's spikes that allow straightforward modelling of generations, and memory regions being split into relatively small buffers, that are equivalent to Lieberman's regions.

C. Learning mechanisms: Synaptic plasticity

In order to demonstrate the activities that SpiNNaker undertakes and the amounts of data generated while simulations run, this section will briefly explain more about synaptic plasticity and high level design principles of garbage collection for plasticity.

1) Spike Timing Dependant Plasticity: Spike-Timing-Dependant Plasticity(STDP) [27] is a commonly used model of synaptic plasticity. At it's core, STDP draws a simple idea: a neuron is capable of receiving input spikes through channels, called synapses, that connect it to other neurons. Then, the input arrival a few mili-seconds before neuron fires, leads to strengthening of that channel, whereas input arrival a few mili-seconds after neuron fires, leads to weakening of the channel. This change in the channel strength between neurons is important because synaptic plasticity, in general, is believed to be one of the main phenomena driving learning and memory activities in the brain [26]. Storing history traces of neurons is the main activity in plasticity simulations.

2) Garbage collection for plasticity: In a general garbage collector, the references to the objects can be traced and the decision made whether it is garbage or not depending on the count of the references. In our application-specific collector, this decision is made in a different manner, by considering the specific data structures used in synaptic plasticity. The history trace of synaptic event is dead when it has served its purpose for the currently running simulation on the SpiNNaker. For this project, we will assume that the history trace can be considered outdated after it has been in the system for 500ms (A consequence of time constants used by Morrison et al [24]). Noteworthy, the difference between general garbage collector and application specific collector that is being implemented, is that we do not consider references to the history traces but their lifetime in the system. History traces are small values, stored into a buffer which is allocated once, and a reference to the buffer lives throughout the whole simulation period.

D. SpiNNaker Events

A major goal of the SpiNNaker architecture is to be able to run brain simulations in real time [1]. SpiNNaker has a timer which is used to manage most of the events on the system. The main event that happens periodically is a timer interrupt, on which neuron states are evaluated and spike transmissions are performed [8]. The period of timer interrupt can be chosen by the user, but most commonly it is set to 1ms. Any code

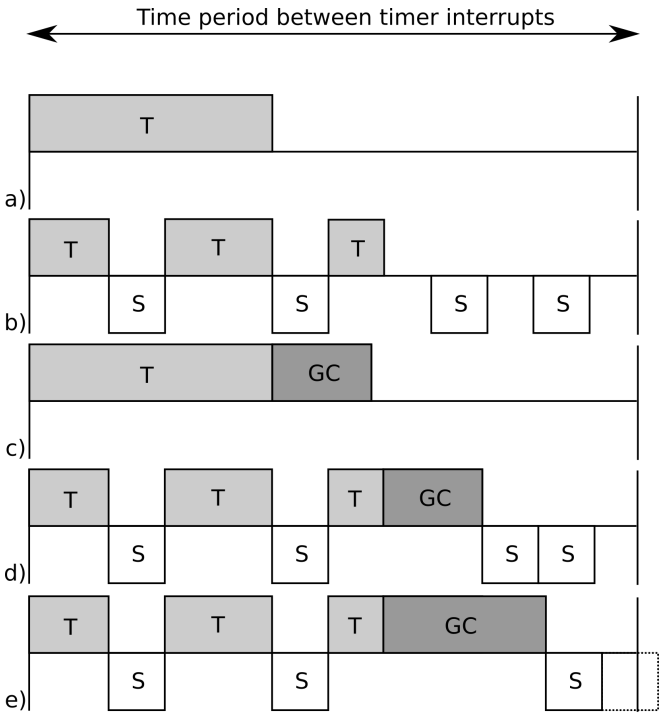

Fig. 7. Simulation activities that are taking place on each timer interrupt. T - Timer, GC - Garbage Collection, S - Spike processing. a) Activities that are marked T run for some fraction of time per each timer interrupt. b) Spike processing events have higher priority, therefore timer interrupt activities get spread out. c) Garbage collection is added to each timer interrupt. d) Garbage collection cannot be interrupted, it locks CPU, therefore some spikes get deferred. e) Real-time violation occurs if garbage collection takes too long and spike processing cannot finish on this timer interrupt.

that runs as part of timer interrupt must finish before another interrupt arrives, including new garbage collection routines introduced in this project.

When neuron potentials are evaluated on timer interrupts, some neurons fire spikes and therefore another event on the system, caused by arriving neuronal spikes, is spike processing. Spike processing makes efferent neuron processors read new information from SDRAM and store it in the internal memory. This interrupt has a higher priority than timer interrupt and thus it will be executed instantly, this way, pausing timer interrupt activities.

Figure 7 demonstrates the activities that happen between two timer interrupts. Spikes interfering the timer interrupt will spread it out across the given period of time. If timer does too much work while high spiking rate occurs, timer interrupt process will not be able to finish in a given period and synchronisation will be violated. Additionally, garbage collection operation is atomic, because it modifies the whole heap space, and due to this, it can defer some spikes.

I hypothesise that if garbage collection operations can be made as efficient as possible in order not to add significant amount of work per each timer interrupt, then the real time requirements of the SpiNNaker system will be met. If there is a need to defer spike processing by some fraction of a mili-second, it is allowed to do that as long as it will be processed in the same mili-second in order not to overload the event queue. At the end of the mili-second, another timer interrupt will occur and before that all of the queued events must finish processing.

IV. Development

This section covers the development of the main tools for automatic memory management as well as documents the essential changes to current state of the SpiNNaker toolchain. The development has been done in C programming language because most of the SpiNNaker toolchain, including low level operating system, is written in C. The development consisted of 5 main software components that are documented below.

A. Synaptic event history trace buffers

The history traces of synaptic events are stored on the DTCM heap in order to have the best update speed when neurons fire. In this subsection the implementation of buffers, that are used to store traces, is documented. Additionally, the changes to buffer mechanism are proposed that will enable garbage collector routines to relocate the buffers to different memory parts.

1) Existing buffer implementation: Current SpiNNaker implementation of the buffer to store history traces works as follows: the maximum number of history traces that each neuron can generate and store is controlled by a macro called *MAX_POST_SYNAPTIC_EVENTS* and is currently set to 16. These blocks of 16 history traces are then allocated to a certain number of neurons and are accessed for updating as a standard C array structure. Each buffer contains two arrays called times[] and traces[], which are of fixed size, and can store number of elements controlled by the macro *MAX_POST_SYNAPTIC_EVENTS*. Additionally, each buffer also contains an integer number *counter*, which indicates how many *times* and *traces* are currently stored in the buffer.

2) Variable sized buffers: In order to do garbage collection we must be able to relocate the objects in the memory heap. However, with the implementation of a standard C array the

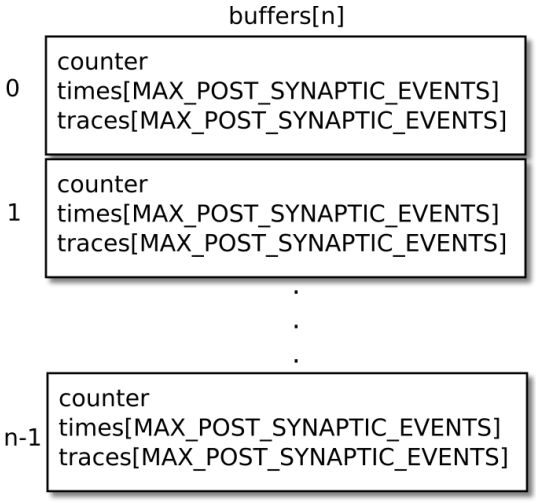

Fig. 8. Implementation of post event history trace buffers

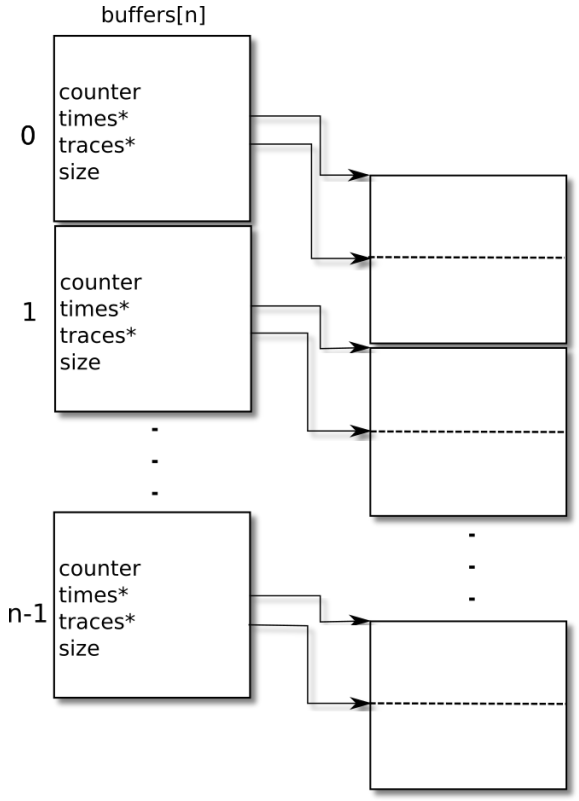

Fig. 9. Experimental synaptic history trace buffer structure

base address of the array is constant and therefore cannot be updated [17]. The solution to this is to allocate memory in DTCM heap using sark_malloc() from SARK library and keep a record of the pointer to it, in order to access the elements. In this way we are able to update the pointer to point to a new location. If memory is copied correctly from one location to another, and the reference is updated, the higher level application will be able to access data without any noticeable change. This proposal is demonstrated in Figure 9. Each buffer now contains *times** and *traces** pointers instead of fixed size arrays as shown in the previous section. Additionally, new variable *size* is introduced in order to find the end address of each buffer and allow variable sized buffers.

B. Memory compactor

One of the most basic techniques to manage memory is memory compaction. Memory compactor processes memory heap at certain periods and it has a capability of moving objects from their original locations. There are 2 main requirements for an effective and secure memory compactor, one functional and one non-functional: 1) After compactor finishes, all objects on the heap must be in a single consecutive block followed by a free block of memory (if any left) and 2) On any successful move operation of the memory block, compactor must update all references to this block. This must be done independently from the user's application that will be using the reference to access the relocated block.

The limitations of DTCM are influencing the design of compactor significantly. Due to highly limited memory space of 64KB, the DTCM heap is almost fully filled with post trace buffers as well as other essential data required for synapse dynamics and neuron implementation. Therefore, there is no working space left for the intermediate working storage that memory compaction requires. The solution would be to shift all objects down in a sequence, but that would require a sorted list of object references, and sorting operation on each cycle would be too expensive. The decision was made to experiment whether it is possible to do it using shared SDRAM, where enough memory space is available for the compaction.

The memory compactor works as follows: each neuron's buffer is copied to a pre-allocated space on SDRAM. Any free holes in the memory are not taken by the copying operation and the result is a single consecutive block of buffers in SDRAM. Then, the whole consecutive block of buffers is copied back to DTCM, starting at address that was originally allocated for the purpose of storing post trace buffers. The references to each neuron's buffer are then updated and therefore the mutator application can continue adding traces to the buffers. As a result, all of the previous data with memory holes in it is overwritten and thus recycled.

Figure 10 demonstrates the high level detail of the memory compaction operation. The illustration shows the first state of DTCM heap containing memory blocks scattered around the heap which is transformed into a single consecutive block after using compaction.

The specific implementation of compactor in SpiNNaker considers post-event history trace buffers that were presented in section IV-A, as atomic objects. In order to understand the start and end addresses of the buffer, compactor refers to the special variables stored in the buffer, i.e. *start address* and *size* of the buffer. Therefore, a non-functional requirement can be identified: the *size* variable of each buffer must be strictly managed by other parts of garbage collection in order for the compaction operation to recognise correct data regions for copying. As a result, the data will neither be violated by the compaction operation nor will any extra, unused data will be preserved.

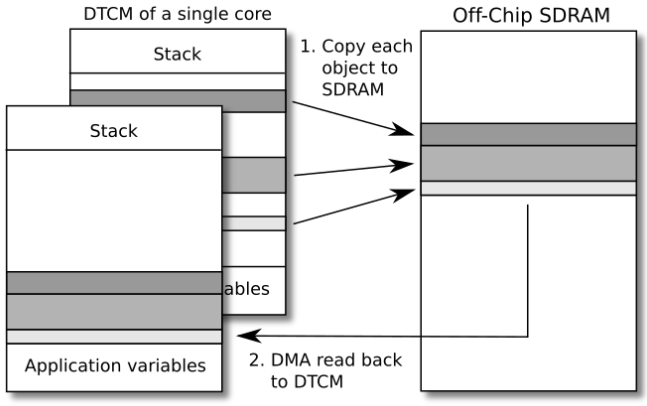

Fig. 10. A visualisation of the compactor

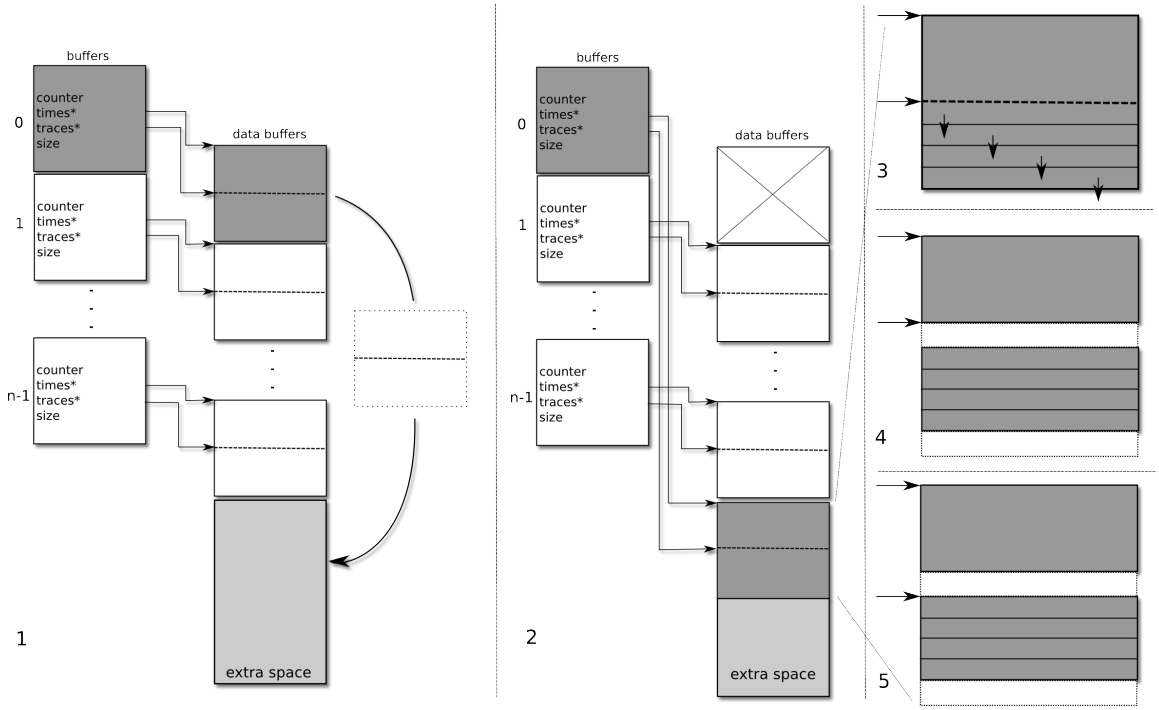

Fig. 11. Steps for dynamic buffer extension

C. Buffer extender

In order to use the space of the buffers, that are not filled completely, more efficiently, the proposal is to initially set the initial number of events to a small number and extend each buffer that needs more space, dynamically. If there is a buffer that needs to add a trace, but is already full, copy the buffer to the end of the data structure of all buffers, into the allocated extra space, and update the reference of it. Relocation of buffers requires to update their base addresses. This is a direct application of the experimental buffer structure proposal that was introduced in Section IV-A2.

Figure 11 demonstrates the steps of buffer extension in detail. In step 1 we move the buffer to the end of the data structure, where arbitrary size of extra storage is allocated specifically for buffer extension. In this case, the variable *size* has a specific use: as the sizes of objects in this data structure are varied, we must keep track of them to decide how much bytes of each buffer to copy when required. After step 1, the previously used space is now marked X and therefore should be recycled by garbage collection. In step 2, the pointers *times** and *traces** of the relocated buffer, are updated to point to the new location of the buffer. At this point the buffer extension can be done in order to make space for a new history trace. Therefore, in step 3, all traces are shifted down to make space for an entry in *times**. In step 4, new space is introduced from the preceding steps for *times** as well as new space at the end, for *traces**. In step 5, the pointer *traces** is moved down to point to the new location of the shifted data structure.

When the extension is finished, mutator detects that there is till more space to add the history trace and it is done in the standard way:

```
times[counter++] = new_time;
traces[counter++] = new_trace;
```

The allocation of extra space at the end and gaps in the memory left from relocating the buffers, poses new research questions. If we allocate a bigger chunk of extra space the rate of compaction operation can be lowered because memory gaps do not need to be re-used as early. On the other hand, if there is not much space available for allocation of extra space, we must compact more often in order to re-use the gaps left from the buffer extension operations. Additionally, allocation of a bigger chunk of extra space means that less space is available for initial neuron data structures, and therefore, it will mean that a number of neurons per core will have to be decreased. To sum everything up, the relationship between R - Rate of compaction, N - Number of neurons and E - Extra space allocated is indirectly proportional:

$$E \propto (R \times N)^{-1} \tag{1}$$

It is most likely that these settings will need to be different for the different simulation environments. To identify a functional requirement, the users must be able to tune all garbage collection parameters identified above.

D. Scanner

Scanning is the main operation that finds garbage on the heap and recovers the memory space that the outdated objects occupy. Scanning operation is demonstrated in Fig. 13.

In step 1, the buffer that is being scanned is coloured. Every trace is checked from oldest to newest or from index 0 to *counter* respectively. In step 2, the outdated trace at index 0 is identified and therefore must be recycled. First, the

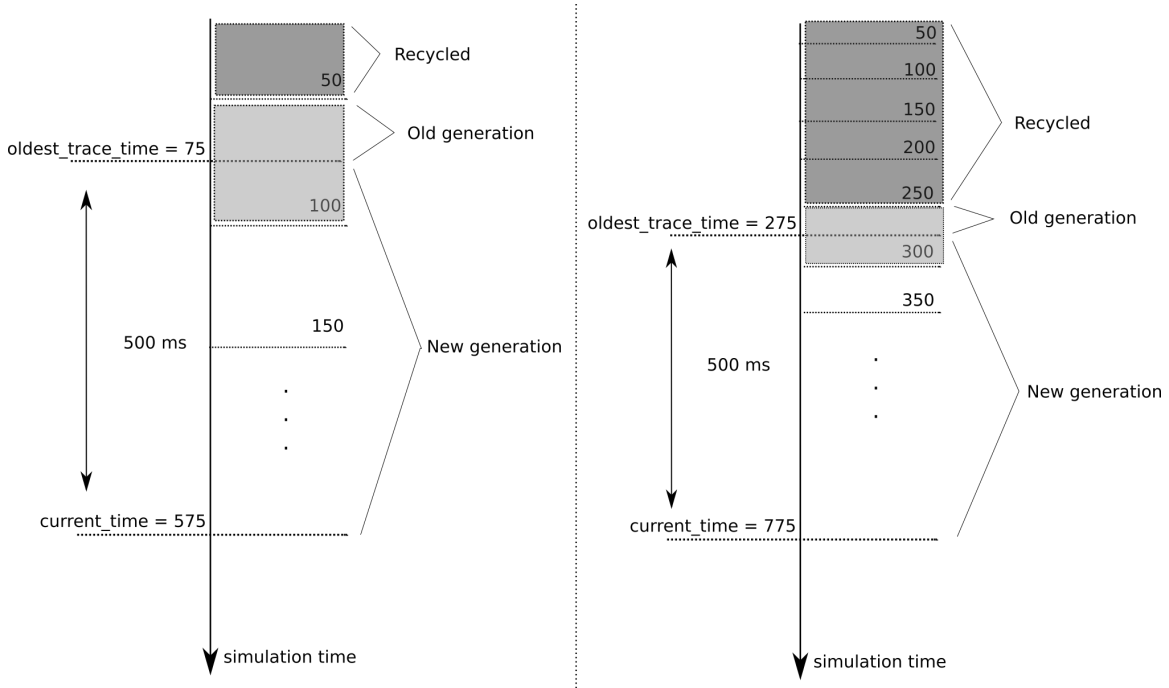

Fig. 12. Generational garbage collection at two specific time instants. The oldest trace time was set to $current\_time - 500ms$. The generation step is 50ms. All traces are additionally categorised into two higher level generations, *old generation* and *new generation* similarly as demonstrated in Garbage Collection Handbook, Chapter 9 [18]. At any point in time, the current buffer (marked light colour) that is garbage collected, may contain both new and old generation traces, but only old traces are removed from the system. The generations that are marked darker colour are already collected and will not be checked again as all newer traces will only be added to newer generations as dictated by the increasing simulation clock.

pointer *times** is advanced in order to remove the outdated entry from the range of the buffer. At this point, the trace entry must be removed from the *traces** buffer. In step 3, this is demonstrated as shift up of all traces from index *1*. This way the trace at *index 0* is overwritten and therefore garbage collected.

After undertaking all above steps the scanner leaves the buffer in a state that is demonstrated in step 4, Fig. 13. Two blocks are now marked "Memory gap": One at the start of the buffer, reclaimed by garbage collecting the array *times** and one at the end from garbage collecting the array *traces**. These memory gaps are expected to be made re-usable by the memory compactor. However, in order to achieve that we must, as an addition to advancing the pointer *times**, reduce the overall *size* of the buffer by decreasing sizes of the two memory gaps from the variable *size*. This way, the compactor will detect only those traces that are in the range *times** + *sizeof(buffer)* so that memory gaps would be overwritten with useful data from the surrounding buffers.

E. Baker's garbage collector

The compactor and scanner in subsections IV-B and IV-D respectively form a variant of Baker's real-time garbage collector described in subsection III-A. The only difference between original H. Baker's proposal and ours is that we do not split the memory heap into two in order to achieve seamless switch between two spaces when one gets filled. Instead,

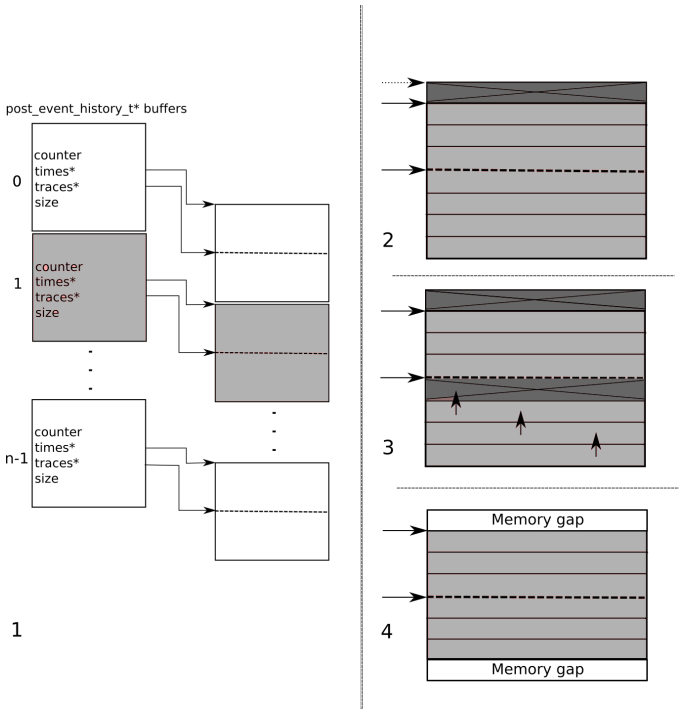

Fig. 13. Garbage collection: Scanning history traces. The objects marked dark grey colour and X are outdated objects that will be garbage collected.

our compaction operation, equivalent to Baker's copying from one space to another, is copying to and from SDRAM at once so there is no need for a switch. The performance of this garbage collector is provided in Sec.V-F.

F. Generational garbage collector

Baker's garbage collector suffers from the need to linearly scan the whole memory space to find garbage. Similarly, our variant of garbage collector looks for outdated traces in every buffer. I.e. If simulation has 255 neurons per core, 255 buffers will be checked on every scanner iteration. That results into 255 accesses to the stack to retrieve the addresses to the buffers.

Instead of scanning all the buffers, they can categorised into specific groups, further called generations (Fig.12). Which buffer is put into which generation depends on how likely they can contain outdated traces: older generations contain buffers with high probability of having garbage in them. We will make this decision according to the simulation clock and the time entry of the oldest trace in a particular buffer. We start by setting how much generations will be used in a macro GENERATIONS_TO_USE. Then the generation step can be calculated in a following way, given the duration of the simulation: *generation_step = simulation_clock_ticks / GENERATIONS_TO_USE*. The generation step will be used to create different generations as the simulation time progresses. At the time of adding the first trace into the buffer, we get current simulation clock tick and calculate the current generation that that time lies in: generation_to_add_to = ceiling(current_time / generation_step) and therefore add the buffer to this generation. Additionally, we keep track of the variable *oldest_trace_time* at all points in time. This variable defines the oldest trace time that is still considered useful for the simulation purposes. Any trace that is older than that can be recycled.

The scanner then works in the following way: it calculates the current generation that the oldest trace variable lies in by again using the ceiling function defined above. It then loops through all the buffers that are in that generation looking for garbage. Any other buffers that are in younger generations are not looked at as we can be sure that they do not contain traces older than all the traces in the currently garbage collected generation. When the buffer is found to have garbage in it, the usual scanner operation is undertaken to remove outdated history traces. However, in this garbage collector we additionally need to remove the buffer from the generation and add it to the younger generation. This is done by considering new oldest trace that garbage collected buffer contains after the garbage was removed from it.

V. Analysis

This section analyses the implementation of garbage collection on SpiNNaker. The investigation in the performance of copying methods and run times of different parts of garbage collection are provided.

All analysis in this section was done by running a specific simulation from SpiNNaker's code base. It was set to contain 2500 neurons with a timer interrupt period of 1ms. Such a simulation uses a single SpiNNaker chip on the developer board and occupies 10 ARM968 cores in that chip resulting in 10 copies of the garbage collector running at once. All the performance statistics were done using the simulation data from all 10 cores. Noteworthy, the simulation has random input delays, therefore every run can demonstrate different behaviour, which is especially visible in the plotted graphs. The running times were evaluated using a profiler that is demonstrated in Appendix A and also used in [23].

A. Probability of memory overflow

There is a consideration to be made about the predicted improvement that dynamic buffer extension(IV-C) introduces into the system. Let us consider what is the likelihood that at some point the simulation will run out of memory. The probability that this will happen is equivalent to the probability that one of the buffers will reach it's maximum limit. Therefore, for a simulation with *n* neurons and a probability distribution of a discrete random variable (number of traces in the buffer at an arbitrary point in time) *p*:

$$P_{overflow} = \sum_{i=1}^{n} p_i \tag{2}$$

Assuming that simulation is set up to contain 10 neurons and a maximum of 16 traces in each buffer, the probability that at any particular time of observing the history trace counter *x*, we will have 16 traces in it is *1/16*. Therefore using equation 2 we get:

$$P_{overflow} = \sum_{i=1}^{10} \frac{1}{16} = \frac{10}{16} = 62.5\% \tag{3}$$

with the expected value of the number of traces in the buffer:

$$\mu_x = x_1 \times p_1 + x_2 \times p_2 ... \times x_{16} \times p_{16} = 8,5 \tag{4}$$

where p is a probability distribution and $x_1 ... x_{16}$ are all possible values of *x*. The variance is then:

$$Var(x) = \frac{1}{16} \times \sum_{i=1}^{16} (x_i - \mu_x)^2 = 21,25 \tag{5}$$

We may now calculate the same values for the new implementation with dynamic buffer extension. The probability of overflowing the memory is now the probability that the overall number of traces in the heap will reach the maximum. Therefore, for *n* neurons and *k* maximum traces (including any extra space allocated) we get:

$$P_{overflow} = \frac{1}{n \times k} \tag{6}$$

Again taking an example of n=10 and k=16 as before, we get:

$$P_{overflow} = \frac{1}{160} = 0.625\% \tag{7}$$

where the expected value of the overall number of traces *X* on a core, is the sum of expected values of all buffers:

$$\mu_X = \sum_{i=1}^{10} \mu_i = 10 \times \mu_x = 85 \quad (8)$$

and the variance of the overall number of traces *X* at an arbitrarily chosen point in time is the sum of variances of the number of traces in each buffer:

$$Var(x_1 + ... + x_{10}) = \sum_{i=1}^{10} Var(i) = 10 \times Var(x) = 215.5 \quad (9)$$

B. Data copying methods

In this subsection, various methods of copying data between ARM cores and SDRAM are analysed. The expected performance analysis of various copying methods in terms of numbers of machine instructions is provided.

1) sark_mem_cpy() and standard memcpy(): Various implementations of memory copying functions exist, both from standard C libraries, like memcpy() as well as SpiNNaker implementations sark_mem_cpy() and spin1_memcpy(). Both sark_mem_cpy() and spin1_memcpy() copy memory byte by byte, either by treating memory region as char* array, which is byte size or using ARM instructions LDRB and STRB that stand for loading and storing a byte. It is also essential to note that memcpy() compiles to LDR and STR instruction loops, that load and store whole words of 4 bytes on each iteration. Because SpiNNaker implementations copy byte by byte, it takes approximately *4n* instructions to copy a block of *n* bytes.

2) Direct Memory Access: Each ARM core on the SpiNNaker chip contains a DMA(Direct Memory Access) controller [7]. DMA controller is used to move data between I/DTCM and SDRAM without requiring CPU. The current implementation of DMA preparation function in SpiNNaker code base, takes 78 ARM instructions to prepare a transfer. The speed of memory transfer using DMA controller is estimated to be around 15 ns/word [14].

3) ARM block copy: An implementation of ARM block copy [25] is provided as part of this project. It achieves more effective copying operation for block sizes that are multiple of 4 words. Each iteration of ARM block copy routine copies 4 words or 16 bytes with a single instruction. If the number of words in a given block is not a multiple of 4, the remaining words are copied one at a time. Using this method, the number of instructions to copy the block of n bytes is denoted by the following equation:

$$4(\lfloor \frac{n}{16} \rfloor + \lceil \frac{n \pmod{16}}{4} \rceil) + 9 \quad (10)$$

Taking, for example, a block of size n=68, we will be doing 4 iterations of quad-copy (copy four words, or 16 bytes at once) and a single iteration of word-copy to copy remaining 4 bytes. Using equation 10, the number of instructions it would take to copy 68 bytes of memory is:

$$4(\lfloor \frac{68}{16} \rfloor + \lceil \frac{68 \pmod{4}}{4} \rceil) + 9 = 4(4+1) + 9 = 29 \quad (11)$$

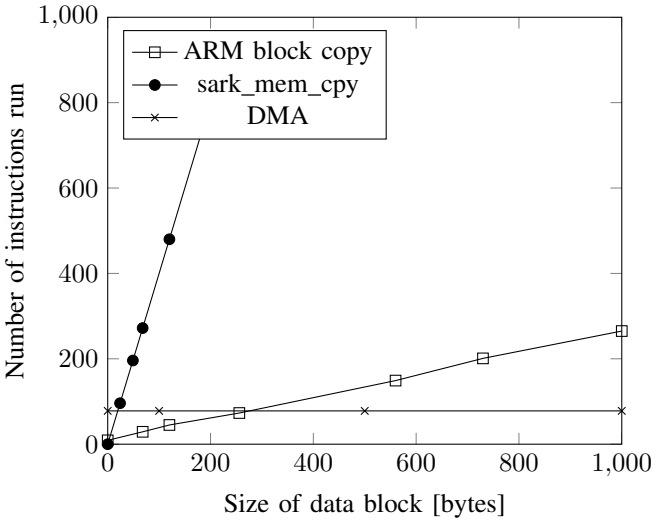

Fig. 14. Growth of the number of instructions required to copy data. For comparison, single byte, multiple word and DMA growths are provided.

4) Statistical comparison of copying methods: Fig. 14 demonstrates the growth of the number of instructions run for different copying functions. From the diagram, the following analysis can be done: ARM block copy is more efficient than sark_mem_copy due to data parallelism; ARM block copy is the most efficient method of copying data of up to approximately 250 bytes; DMA is more efficient of copying data blocks bigger than 250 bytes.

Due to different speeds of the memories used, SDRAM(slow) and DTCM(fast), number of instructions run is not enough to indicate most efficient copying operations and in what scenarios to use which operation. It is estimated that data access from ARM core to SDRAM takes 100ns but varies according to the contention on the bus [14], [16]. We shall now proceed with the evaluation of the run time performance of functions that are using the aforementioned copying routines.

C. Memory compactor

Memory compactor was introduced in Sec. IV-B. In this subsection, the performance analysis of the compactor is provided.

1) Timing performance: Table I summarises the time it takes to compact the memory with differently sized history trace buffers used and different numbers of neurons per core. From the given run-times it is evident that the fastest copying routine is ARM block copy with standard C memcpy() showing slightly smaller performance. Sark_mem_copy is impractical in this case, due to byte by byte copying. Additionally, DMA is slower because our atomic data elements are of small size.

To sum up the results, for the compaction operation, ARM block copy is best suited to copy small data blocks, therefore it will be used for copying individual neuron buffers to SDRAM. On the other hand, DMA is best suited to copy

type of simulation	ARM Block Copy	memcpy	DMA	sark_mem_cpy
40/4/32b	0.039 ±0.0013	0.044 ±0.0005	0.064 ±0.0	0.146 ±0.004
255/4/16b	0.4 ±0.08	0.66 ±0.08	0.91 ±0.09	1.7 ±0.1
255/4/32b	0.95 ±0.043	1.06 ±0.13	1.07 ±0.13	3.87 ±0.09

TABLE I. Average running times of the compaction operation. The time is expressed in mili-seconds and standard error is also provided. The run-times were evaluated by running compactor more than 20 times. Each row represents the type of simulation, with the parameters, respectively: number of neurons per core, number of history traces in the buffer and the initial size of the buffer. It is worth to note that a lot of buffers are extended as simulation progresses, and therefore the final size can be larger than initial size. Each column represents the copying method used to copy each buffer to SDRAM. Reading the whole block back to SDRAM was done in a single DMA call in all of the cases.

big blocks (more than 250Bytes, as established above), thus it will be used to copy all blocks back to DTCM.

2) Improving compactor: The timing performance shows that the compaction operation, in the most common case of 255 neurons, runs for 400-900 micro-seconds. As this is close to 1ms, it would violate real-time requirements of the SpiNNaker system (III-D). Therefore, the compactor's work must be divided into smaller chunks, i.e. the working space will be divided into 4 or more regions and on each call of the compactor only one of the regions will be compacted, but different region each time. The downside of this approach is that a single compaction of the whole memory heap must be distributed across different timer interrupts.

The implementation of the fragmented compactor has been provided and in order to control the size of the memory work space, the value of the macro called *COMPACTOR_FRAGMENTATION_FACTOR* is manipulated. This macro defines the number of parts that the memory space will be divided into, and at the same, the number of times the compactor must be run in order to compact the whole heap.

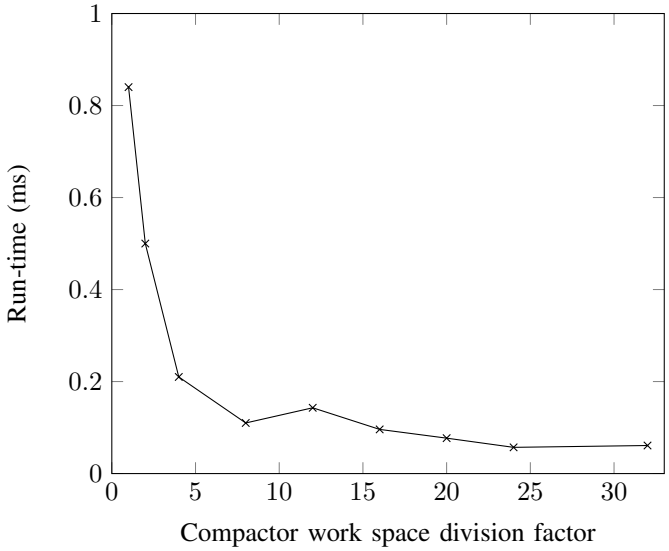

Fig. 15. Run times of the single invocation of the compactor as values of the compactor space division factor are increased.

Fig. 15 demonstrates that the most useful factors that critically lower the running time of the compactor are from 2 to 8. Further factors do not provide a significant run-time decrease as well as require many invocations to compact the whole memory space.

D. Buffer extender

The buffer extender (IV-C) operates by copying a single buffer and shifting elements down by a small number of bytes. Table II summarises the running times of extension operations with various, differently sized buffers.

type of simulation	ARM Block Copy	sark_mem_cpy
4/16b	1481 ±140	1831 ±188
4/32b	1598 ±178	1918 ±141
12/56b	2084 ±163	2518 ±236

TABLE II. Average running times of the buffer extension (ns) for different copying methods and different simulation types. Type of simulation parameters are, in order: initial number of traces and size of the buffer.

On average, the extender is called *3.5 times* per SpiNNaker time step thus it approximately occupies 1% of the time available in a single timer interrupt. Additionally, there is not much difference between ARM Block copy or sark_mem_copy() as both of them give similar results for the buffer extension operation. This is the result of extender doing only a single copy operation making other operations, like finding the addresses and shifting elements, more significant than copying.

E. Scanner

The scanner, that was introduced in section IV-D, is doing a significant amount of work, while linearly scanning all of the buffers and then shifting the array elements of the buffers that are found to contain garbage. Noteworthy, history traces are added into buffers in the ascending order according to their *time* entry. Due to this, if we find that a trace is not old enough to be garbage, we can safely stop there instead of scanning rest of the traces in the array, as all of the subsequent traces will be younger. As a result, our linear scanning operation is not fully scanning the whole heap most of the times. In comparison, general garbage collectors, that manage any kind of objects must scan every object on each invocation, due to objects not possessing any relationship or order. The run times of the scanner are summarised in Table III.

type of simulation	time (ns)
40/16/64b	3870 ±20
40/16/128b	3913 ±28
255/4/16b	36578 ±1410
255/4/32b	43171 ±1645

TABLE III. Running times of the scanning operation (ns). Type of simulation parameters are, in order: number of neurons, initial number of traces and size of the buffer.

We can observe that scanner takes more time to execute as the number of neurons increases. This is due to the

fact that more neurons introduce more data structures on the memory heap. Additionally, as simulation complexity is increased (size of buffer increases), element shifting is introduced which causes processor to spend additional few thousand nanoseconds for executing scanning operation.

F. Baker's garbage collector

With all the above tools developed, the variant of Baker's garbage collector, that was described in section IV-E, can be evaluated.

1) Memory space reclaimed: In figure 16, demonstrated are the numbers of bytes reclaimed per each scanner invocation for scanning rates *1 kHz* and *0.5 kHz*. When the rate is 1kHz, i.e. 1 scan per timer interrupt, a lot of cycles are wasted by reclaiming 0 bytes. When the rate is decreased to 0.5 kHz, to scan the memory on every other timer interrupt, we recycle more garbage on each iteration but produce less wasted scan cycles. Therefore, if we keep decreasing the rate of scanning, we will increase the number of bytes reclaimed on each iteration and decrease the number of wasted scans. In figure 17, the rate of 0.25kHz produces no wasted scan cycles in the simulation period 800-1000 ms.

2) Memory space utilisation: Figure 19 shows the memory occupation as a simulation progresses, for both garbage collected and non-collected approaches. The memory utilisation in non-garbage collected environment is constantly increasing as simulation progresses and would eventually reach the limit of the allocated memory. On the other hand, the memory usage in garbage collected run jumps to around 4KB at the start and eventually stabilises to approximately 2.5Kb starting with 501ms. Following this, I hypothesise that more neurons could be fit into a single core with enabled garbage collection. If we assume that we have space available for 255 neurons with 12 traces for each, with an additional 3 traces of extra space, the overall space available is:

$$255 \times (12+3) \times TRACE\_SIZE = 255 \times 15 \times 4 = 15.3KB \quad (12)$$

This result indicates that theoretically, we can fit four times more neurons that will stabilise to the garbage collected space occupation of:

$$2.5Kbytes \times 4 = 10Kbytes \quad (13)$$

with 5.3 Kbytes left for expansion. However, the performance of garbage collection, when number of neurons will be increased, will be significantly lower and the real-time requirements can be violated. Additionally, more neurons require other structures to be stored on the core, not just the ones that contain history traces. Future work suggestions in this area are discussed in Section VI.

3) Meeting teal-time requirements: In order to preserve the biological real-time execution property (III-D) of SpiNNaker simulations, any code that is run when the timer interrupt happens must exit before another interrupt occurs, most commonly, after 1ms. The code in each interrupt includes synapse processing and neuron state updates as well as newly introduced garbage collection routines. I have evaluated that without garbage collection, timer interrupt in, the same simulation of 2500 neurons, on average runs for 0.55 ms. When Baker's garbage collection was introduced with a running frequency of 1kHz and the compactor fragmentation factor of 4, the average run-time of a timer interrupt did not change significantly.

Another way to check that the real-time requirements were not violated is to investigate the running program output using Ybug (II-B4) and detect any warnings about SpiNNaker's event queue. If any timer interrupt is running for a longer period than allowed by simulation settings, other events will not be started and thus warnings will be produced. However, upon enabling garbage collector, no warnings have been detected with many trial runs of the simulation.

G. Generational garbage collector

Generational garbage collector (IV-F) is the improvement over copying collector, that tries to lower the size of the scanning space. This is done by maintaining the structure of generations with objects assigned to them. Then, only specific generations are scanned for garbage, this way avoiding the exhaustive search over all objects on the memory heap. In this section the performance statistics of generational garbage collector are compared to statistics of the copying collector.

1) Memory space: Memory space utilisation when using generational garbage collector does not change when compared to the basic copying collector. Figure 18 demonstrates space reclamation as simulation time progress and figure 20 shows the overall space utilisation when simulation is run for 5 seconds. The space utilisation again stabilises to approximately 2,5KB.

2) Performance of the scanner: The scanner (IV-D) is the main operation for which generational collector provides performance improvements. The improvement is a result of not needing to scan the whole memory space to find outdated traces. Instead of the exhaustive scanning, just a certain generation is scanned at any particular time that is certainly going to have garbage in it. Table IV summarises the improved average run times of the scanner and it can be seen the growth is much slower than previously (Table III). The last simulation in the table has a number of traces decreased from 4 to 3 compared to the simulations run for copying collector analysis in Table III. This was done because generations occupy some space on DTCM core and did not allow to fit 4 traces per neuron. However, it does not impact the scanner's performance in any way as buffers will simply be extended to 4 traces when the limit is reached.

type of simulation	time (ns)
40/16/64b	3086 $\pm$48
40/16/128b	3049 $\pm$51
255/4/16b	6449 $\pm$635
255/3/24b	8624 $\pm$640

TABLE IV. Generational collection: running times of the scanning operation (ns). The simulation parameters are, in order: number of neurons, initial number of traces and size of the buffer.

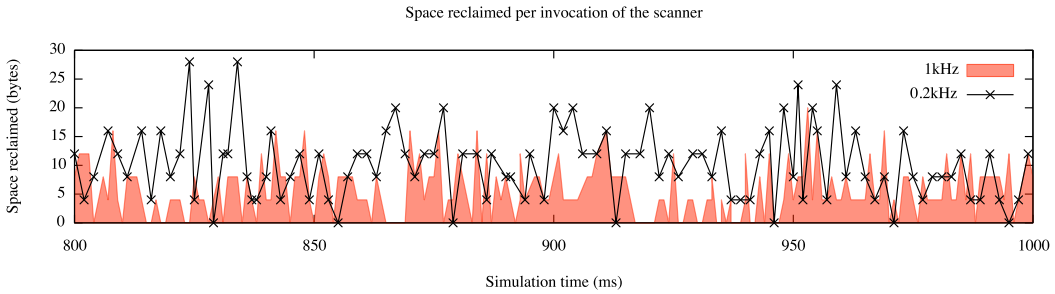

Fig. 16. Space reclaimed per each invocation of the scanner. Rates 1 and 0.5 kHz.

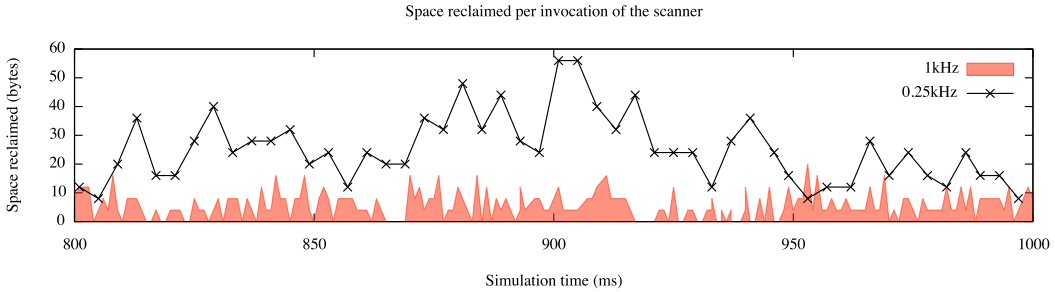

Fig. 17. Space reclaimed per each invocation of the scanner. Rates 1 and 0.25 kHz.

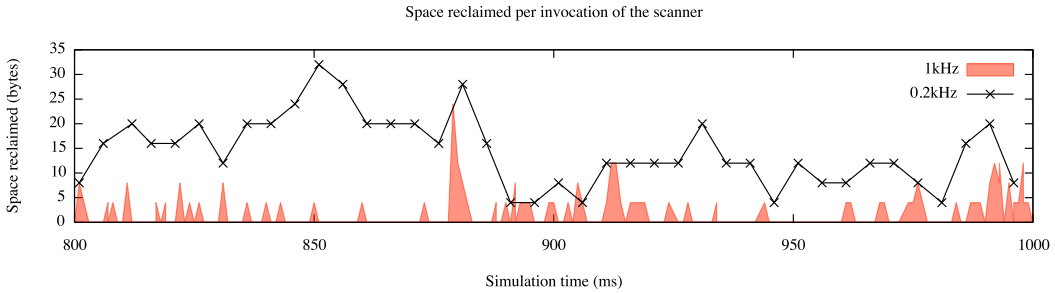

Fig. 18. Space reclaimed per each invocation of the scanner in generational garbage collector.

3) Extra memory requirements: Generational garbage collector introduces more data structures on the already limited DTCM memory block. For each generation, a new data structure of length equal to the number of neurons, is allocated that can store set of integers. The integers are indices of the neurons that are currently residing in that particular generation. Therefore, for n neurons and k generations the space occupied in bytes will be:

$$n \times k \times sizeof(integer) \tag{14}$$

As an example, in a simulation with n=255 and k=4, generational data structure will occupy 4080 bytes.

One of the solutions to reduce the memory occupied by generation structures would be re-using the generation space of those generations that are already too old to contain any more objects. Another approach could be speculating[5] the maximum number of buffers that might be added into a single generation and reducing it to occupy less memory. An investigation of a simulation with 255 neurons per core has shown that the average highest number of neuron history trace buffers in a generation was 80. Generally, this number depends on the spiking rate of the neurons in the simulation and therefore can be speculated to save space on DTCM.

VI. Future work

This section documents the ideas for further work related to SpiNNaker garbage collection.

[5]Speculative execution is an optimisation technique of predicting the resources required by the system, before it is executed. It is commonly used in optimising microprocessor branch prediction. For example, speculation would be allocating smaller number of resources than theoretical number and detecting whether the system still works in the same way.

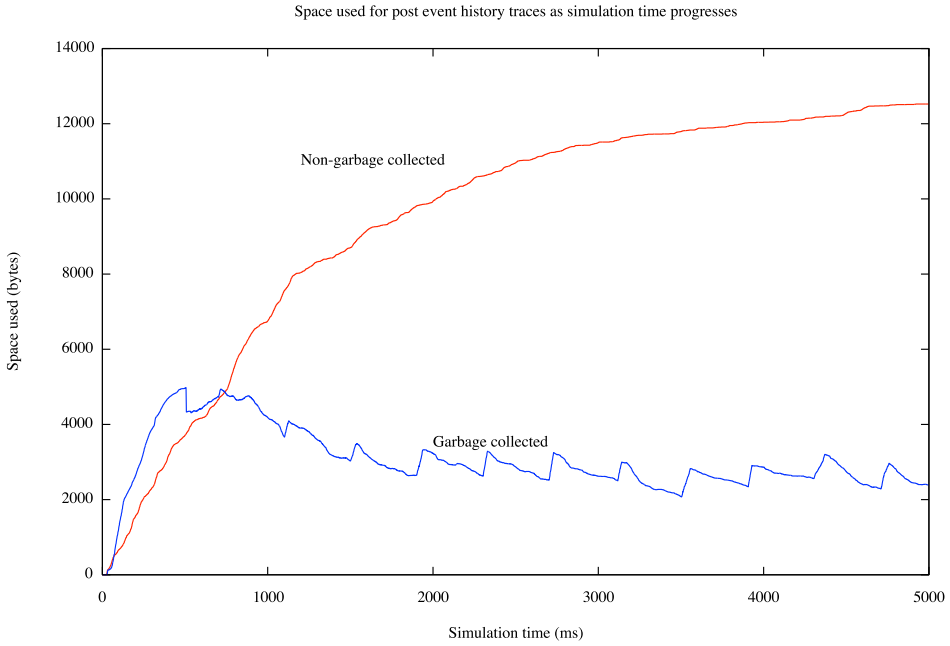

Fig. 19. A simulation with 255 neurons per core. Initial number of traces in non-garbage-collected run was 16, and for garbage collected run it was 12, with an extra space of 3 traces allocated for each neuron. Garbage collection was set to collect all traces older than 500ms.

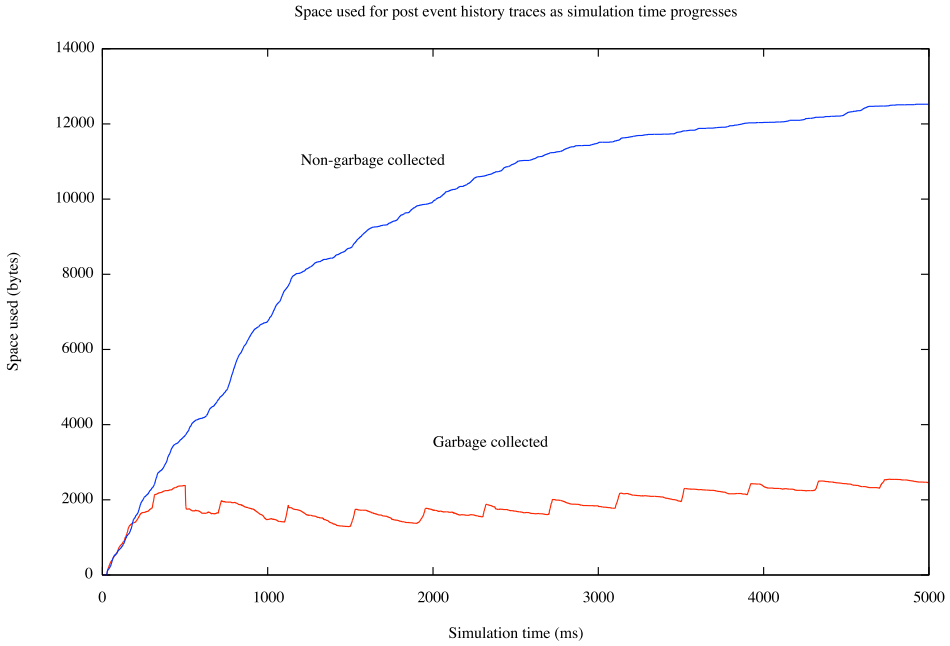

Fig. 20. A simulation with 255 neurons per core. Initial number of traces in non-garbage-collected run was 16, and for garbage collected run it was 4, with an extra space of 3 traces allocated for each neuron. Garbage collection was set to collect all traces older than 500ms and they were managed into 8 generations.

A. More efficient usage of DMA controller

DMA controller is an independent device, and therefore can copy data without requiring CPU time. Further speed optimisations of garbage collection could include the investigation into parallelizing compactor operations with any other neural simulation work that must take place. Garbage collector can use DMA controller completely to compact data, and at the same time allow ARM core to do spike processing instead of locking it. In order to achieve such optimisation, a protocol for managing memory read and write operations must be established that will make sure that ARM core is not reading or writing to memory which, at the same time, is being processed by the compactor.

B. General garbage collector

This study have investigated a garbage collector specifically for implementation of synaptic plasticity on SpiNNaker. Further work can include a general garbage collector that will manage heap independently of the application. The main challenge in the general garbage collector is scavenging that is widely documented by Lieberman and Hewitt [3]. Scavenging refers to the operation of finding all the references that point to the memory region that was relocated as part of copying garbage collection. Finding such references is the slowest process of Lieberman's garbage collector, and therefore would require many hours of man power to implement, test and analyse. However, the type of applications, for which general garbage collector on SpiNNaker would be required, is not yet clear.

C. Fitting more neurons into a core

With 255 neurons per ARM968 core, the limitations of the DTCM are reached with 16 history traces for each neuron stored. Future work will include the investigations into whether garbage collection techniques introduced in this study can provide a way to store more neurons per core. The difficulties in this include identifying all the parts of the SpiNNaker toolchain, that control how much neurons are stored in each core, increasing it and investigating the effects while enabling garbage collection. The implications of the numbers of neurons stored in a single core are extensively discussed by Furber et al [22], [23].

D. Other rules for finding dead objects

In this project, a rule of 500ms moving window has been used to find dead objects: If the object does not fit into a window from *current_time - 500* to *current_time*, then it is dead and can be recycled. Other techniques might be available and are worth investigating. One of them could be counting how much times history traces have been read by the synaptic processes similarly as shown in Appendix of [24]. Then, when a number of 'reads' is equal to the number of synapses that the spike associated to history trace is destined to, we are able to throw away the trace. This technique might provide a better rule for dead traces. However, the memory for extra data structures required would also be a significant constraint in this study.

VII. Conclusion

Two implementations of the automatic memory management on SpiNNaker were investigated throughout this study. First implementation was the copying collector (IV-E) that proved to be slow due to linear search of the whole memory heap. The second implementation allowed to improve the first collector by managing memory into generations and searching for garbage only in the certain areas of the memory (IV-F). The implementation that was introduced, provides an easy way to tune garbage collector by changing frequency of scanning and compacting, tuning the generation step as well as controlling the size of extra space allocated for buffer extension (IV-C). By running and evaluating garbage collection on a development board containing physical SpiNNaker chips, the results have shown that it does not violate the real time of the system (V-F3). Additionally, the statistics of memory utilisation when garbage collection is turned on (V-F2), provided many ideas for future investigations. Last but not least, the source code was contributed to the official SpiNNaker repository which will allow SpiNNaker users to consider the use of garbage collection for simulations of synaptic plasticity, and will be a foundation for further research in the automatic memory management.

SpiNNaker is a very young architecture with many non-standard technologies, thus it presented a steep learning curve at the start of the project. Because of this, some of the project activities, that were laid out in the initial project plan, eventually did not fit into the schedule. More investigations could have been done for finding the use cases for garbage collection on SpiNNaker. This would require more time for collaborating with the SpiNNaker team as well as users. Additionally, garbage collection introduced changes to some parts of the current SpiNNaker toolchain (IV-A2). Not enough time was left to fully integrate garbage collection into the main code base. It must be done by either replacing the fixed buffers by the proposed dynamic buffers, or providing conditional compilation directives that will allow to switch between the two approaches.

The first ever implementation of garbage collection on SpiNNaker, was analysed in this document. All the work that did not fit into the schedule of this project, was also presented for future investigations. To sum everything up, the project has been a success, both in terms of research contribution to the SpiNNaker project, as well as the scope of techniques learned throughout.

VIII. Acknowledgements

I thank my project supervisor Dr. David Lester for support, constant inspiration to learn and the introduction to learning mechanisms. I am also grateful to the members of the SpiNNaker team: James Knight, Andrew Rowley and Alan Stokes for helping me to understand the programming model of SpiNNaker and the implementation of Synaptic Plasticity.

Appendix A
Profiler

Profiler on SpiNNaker [23] provides a capability of measuring the time performance of particular pieces of code.

The main principle is to add special entry and exit calls in the application, which will then time stamp these particular execution points and record time stamp data into SDRAM. At the end of simulation, data is gathered and summarised into average run-time statistics. The usage of profiler is as follows:

```
profiler_entry(ENTER | <UNIQUE_TAG>);
<Function source code>
...
profiler_entry(EXIT | <UNIQUE_TAG>);
```

Then the difference between exit and enter times for this unique tag are taken to evaluate the running time of the enclosed program. If the program is called many times, many time stamps are recorded and average is taken. Here is an example output of the profiling of the compactor:

```
Tag: Compactor
  Mean time: 0.267614ms
  With standard deviation 0.156149ms
  Standard error: 0.049379ms
  Mean samples per timestep: 0.011086
  Last sample time: 902.000000ms
  Mean time per timestep: 0.002967ms
```

REFERENCES

[1] Steve B. Furber, Francesco Galluppi, Steve Temple, Luis A. Plana. *"The SpiNNaker Project"*. Proceedings of the IEEE, Vol. 102, No. 5, May 2014.

[2] John McCarthy *"Recursive Functions of Symbolic Expressions and Their Computation by Machine, Part I"*. Massachusetts Institute of Technology, Cambridge, Mass, April 1960.

[3] Henry Lieberman, Carl Hewitt. *"A Real-Time Garbage Collector Based on the Lifetimes of Objects"*. MIT Artificial Intelligence Laboratory, 1983 ACM 001- 0782/83/.

[4] David Detkefs. *"A Hard Look at Hard Real-Time Garbage Collection"*. Sun Microsystems, Proceedings of the IEEE (ISORC'04).

[5] Henry Baker. *"List processing in real time on a serial computer"*. Commun. ACM 21, 4 (April 1978) 280-294.

[6] *"SpiNNaker project, http://apt.cs.manchester.ac.uk/projects/SpiNNaker/"*. APT Advanced Processor Technologies Research Group, The University of Manchester.

[7] *"SpiNNaker datasheet"*. APT Advanced Processor Technologies Research Group, The University of Manchester, Version 2.02, 6 January 2011.

[8] *"SpiNNaker Software Specification and Design"*. APT Advanced Processor Technologies Research Group, The University of Manchester, Version 0.0, October 2012.

[9] Steve Temple *"SARK - SpiNNaker Application Runtime Kernel"*. SpiNNaker Group, School of Computer Science, University of Manchester, April 2013, Version 1.30.

[10] Steve Temple *"AppNote 1 - SpiNN-3 Development Board"*. SpiNNaker Group, School of Computer Science, University of Manchester, November 2011, Version 1.00.

[11] Steve Temple *"AppNote 2 - Programming SpiNNaker with ARM and GNU tools"*. SpiNNaker Group, School of Computer Science, University of Manchester, November 2011, Version 1.00.

[12] Steve Temple *"AppNote 3 - The APLX File Format"* SpiNNaker Group, School of Computer Science, University of Manchester, November 2011, Version 1.00.

[13] Steve Temple *"ybug - System Control Tool for SpiNNaker"*. SpiNNaker Group, School of Computer Science, University of Manchester, April 2014, Version 1.30.

[14] Steve Temple. *"SpiNNaker System Software presentation slides"*. SpiNNaker Workshop - Manchester - Sep. 2015.

[15] Michael Robert Bernstein. *"Real-Time Garbage Collection Is Real"*. Blog post on real time garbage collection: http://michaelrbernste.in/2013/06/03/real-time-garbage-collection-is-real.html.

[16] Eustace Painkras, Luis A. Plana, Jim Garside, Steve Temple, Francesco Galluppi, Student Member, Cameron Patterson, David R. Lester, Andrew D. Brown, Senior Member, and Steve B. Furber. *"SpiNNaker: A 1-W 18-Core System-on-Chip for Massively-Parallel Neural Network Simulation"*. IEEE Journal of Solid State Circuit, Vol. 48, No. 8, August 2013.

[17] Nick Parlante and Julie Zelenski. *"The Ins and Outs of C Arrays, Computer Science class Handout"*. Standford University, CS107, Spring 2008.

[18] Richard Jones, Antony Hosking, Eliot Moss *"The Garbage Collection Handbook"*. Chapman & Hall/CRC, Applied Algorithms and Data Structures Series, 2012.

[19] Edsgar W. Dijkstra, Leslie Lamport, A.J.Martin, C.S.Scholten, and E.F.M.Steffens. *"On-the-fly garbage collection: An exercise in cooperation"*. Communications of the ACM, 21(11):965-975, November 1978.

[20] https://github.com/SpiNNakerManchester, *"SpiNNaker project on GitHub"*.

[21] ARM Information Center: http://infocenter.arm.com/. *"Tightly-coupled memory"*. Last accessed: 12/03/2016.

[22] T.Sharp and S.Furber. *"Correctness and performance of the SpiNNaker architecture"*. Neural Networks (IJCNN), The 2013 International Joint Conference on, 2013, pp. 1-8.

[23] Mundy, A.; Knight, J.; Stewart, T.C.; Furber, S. *"An efficient SpiNNaker implementation of the Neural Engineering Framework"*. Neural Networks (IJCNN), 2015 International Joint Conference on. USA: IEEE: 2015: 1-8.

[24] A. Morrison, A. Aertsen, M. Diesmann. *Spike-Timing-Dependent Plasticity in Balanced Random Networks.* Neural Computation 19, 14371467(2007).

[25] ARM Information center: http://infocenter.arm.com/. *"Block copy with LDM and STM"*. Last accessed: 15/03/2016.

[26] Sadegh Nabavi, Rocky Fox, Christophe D. Proulx, John Y. Lin, Roger Y. Tsien, Roberto Malinow. *Engineering a memory with LTD and LTP.* 348, Nature, VOL 511, 17 July 2014.

[27] Henry Markram, Joachim Lubke, Michael Frotscher, Bert Sakmann. *Regulation of Synaptic Efficacy by Coincidence of Postsynaptic APs and EPSPs.* Science, VOL 275, 10 January 1997.