Brewing the first ever automatic memory management utility for SpiNNaker: Real-Time Garbage Collection for STDP simulations

DOI:
10.1109/IJCNN.2017.7966229

Document Version
Accepted author manuscript

Brewing the first ever automatic memory management utility for SpiNNaker: Real-Time Garbage Collection for STDP simulations

Mantas Mikaitis, David R Lester

The Human Brain Project
APT Research Group
University of Manchester, Manchester, UK
{mikaitis, dlester}@cs.man.ac.uk

***Abstract*—First generation SpiNNaker chip uses ARM968, with highly limited internal memory space, as its core element. In simulations of learning algorithms, many biologically plausible learning rules require history traces of each neuron's activity to be stored. As a result, the history traces of neurons rapidly fill the internal memory space eventually reaching the limits of ARM968. To lower the possibility of memory overflow, we propose to introduce a memory management routine working in the background, which must 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**

I. Introduction

One of automatic computer memory management techniques is garbage collection. 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.

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 [1]. A large part of these events leave traces in the memory that are used in further simulation activities. Current approach uses fixed size buffers to store traces. If a buffer becomes full, the last trace will be dropped out of memory unconditionally, even if it is still useful for future simulation activities. In comparison, other neuromorphic systems apply a manual memory management [2] or build mathematical models to detect memory critical areas of the system [3]. 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 for simulation activities mentioned above. The demonstrated work should be useful to computational neuroscience community and in particular to builders and users of massively parallel neuromorphic platforms. This study will provide useful insights into how memory can be managed on SpiNNaker and similar neuromorphic systems, and where specifically automatic management could be useful for such architectures.

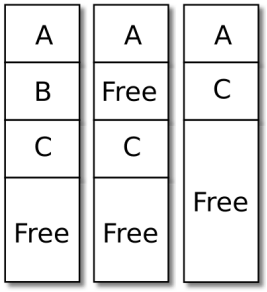

Fig. 1. Basic garbage collector operation: Reclaiming memory space occupied by a dead object B and compacting live objects A and C

The contributions are provided in the following order: We start by introducing background information about SpiNNaker computer (Sec.II). We then present two foundational garbage collection algorithms: copying collector (Subsection III-A) and generational collector (Subsection III-B). Then, the manuscript is continued with the details of implementing two aforementioned garbage collection algorithms on SpiNNaker chips (Sec.IV). After that, we evaluate garbage collection on SpiNNaker, given the memory limitations described in section II-A, and biological real-time constraints of learning mechanisms introduced in sections III-C and III-D. The analysis is given in section V.

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 [4]. 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. SpiNNaker Chip

SpiNNaker chip is made out of 18 ARM968 processors as well as a block of shared SDRAM of 128Mbytes. The board in Fig.2, on which the project was also developed, comprises 4 SpiNNaker chips. Typically each ARM968 in the SpiNNaker chip will be allocated up to 255 neurons.

ARM968 contains 64Kbytes of data storage memory, DTCM[1], and also 32Kbytes of instruction memory, named ITCM. 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.

B. 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 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 [6].

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 [7]. 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[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 [5]

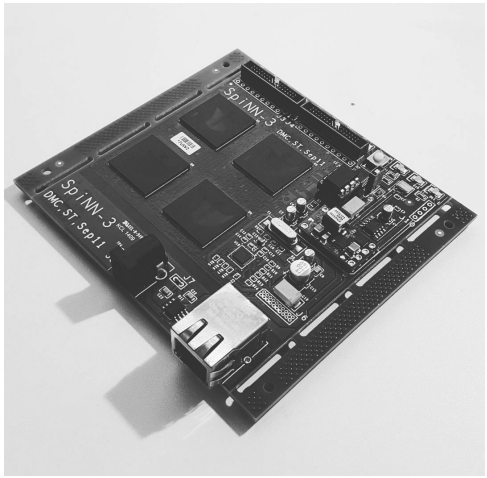

Fig. 2. 4-Node SpiNNaker development board

III. Theory

A. Copying garbage collector

One of the first ideas of real-time garbage collection was studied and presented by H. Baker [9] with useful notes on it written by Lieberman et al [7]. Baker proposes the available memory space to be divided into two parts, called *fromspace* and *tospace*. 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 pointers to point to the new locations. Noteworthy, the type of objects that we will collect in SpiNNaker never have pointers to other memory areas, therefore we will not need to undertake scavenging.

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 [7] 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

[2]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 [8] where two different processors were used for executing both programs.

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.

C. Synaptic plasticity on SpiNNaker

Spike-Timing-Dependant Plasticity(STDP) [10] 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 [11]. In order to calculate relationships between spikes, spiking history traces must be stored at the end of post-synaptic neuron.

How do we handle memory objects generated by STDP? 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 [12]). 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.

D. SpiNNaker Real-Time Constraints

A major goal of the SpiNNaker architecture is to be able to run brain simulations in real time [4]. SpiNNaker has a timer

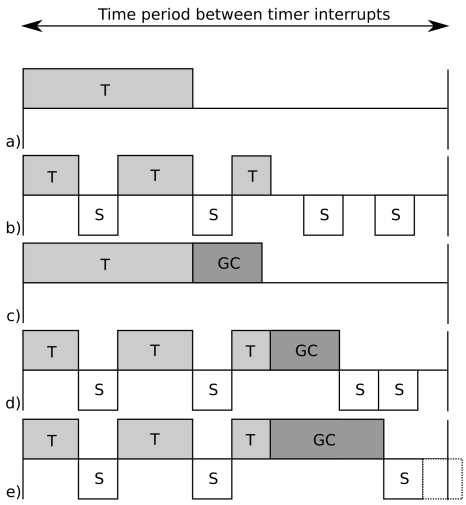

Fig. 3. Simulation activities that are taking place on each timer interrupt. T - Timer, GC - Garbage Collection, S - Spike processing. a) Activities that are marked T run for some fraction of time per each timer interrupt. b) Spike processing events have higher priority, therefore timer interrupt activities 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.

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. The period of timer interrupt can be chosen by the user, but most commonly it is set to 1ms. Any code 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 3 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 collects all the garbage or none. Due to these reasons, it can defer some spikes.

We predict 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 simulation capabilities of the SpiNNaker system will be preserved. 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 main algorithms that were developed. The development consisted of 3 main components that when combined, form a variant of Henry Baker's copying garbage collector: scanner, buffer extender and memory compactor.

A. Synaptic event history trace buffers

Current SpiNNaker implementation of history trace buffers allocates a fixed amount of memory for each buffer and stores them in a single consecutive block. These data structures are then considered to be standard C arrays.

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 base address of the array is constant and therefore cannot be updated [13]. 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 4. 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.

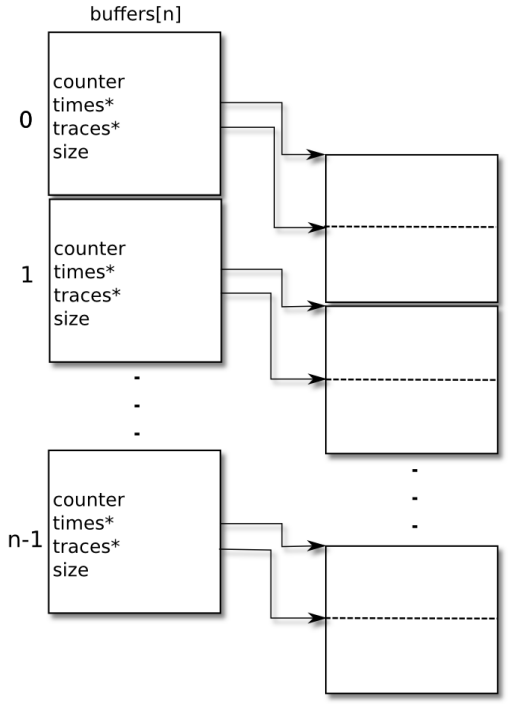

Fig. 4. Experimental synaptic history trace buffer structure

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: 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 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 (Fig.5).

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 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.

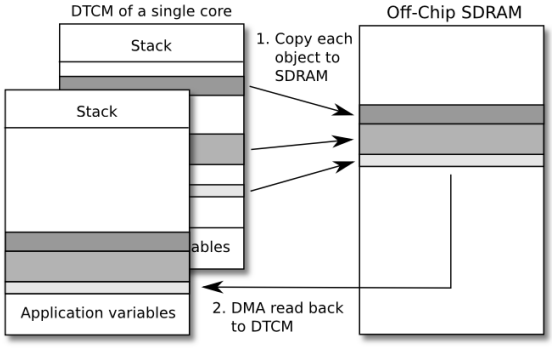

Fig. 5. A visualisation of the compactor

C. Buffer extender

In order to use the space of the buffers, that are not filled completely, 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-A.

D. Scanner

Scanning is the main operation that finds garbage on the heap and recovers the memory space that the outdated objects occupy. In general, scanner finds garbage in each of the history trace buffers. Because history traces are in fact time stamps, they occur in sequential order and the oldest ones, that will be collected first are always at the top. Then, in order to collect them we can simply move the base pointer down and decrease the size variable in the buffer, to redefine the boundaries.

E. Generational garbage collector for SpiNNaker

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 be categorised into specific groups, further called generations (Fig.7). 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. In order to accommodate this new functionality, scanner needs a small modification in order to scan only buffers that are in the current generation.

V. Analysis

For testing purposes, a simulation of 2500 neurons with a timer interrupt period of 1ms was run. 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.

A. Efficient data copying using ARM block-copy

An implementation of ARM block copy [14] 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:

$$4(\lfloor \frac{n}{16} \rfloor + \lceil \frac{n \pmod{16}}{4} \rceil) + 9 \quad (1)$$

The following figure compares ARM block copy with other available copying routines on SpiNNaker:

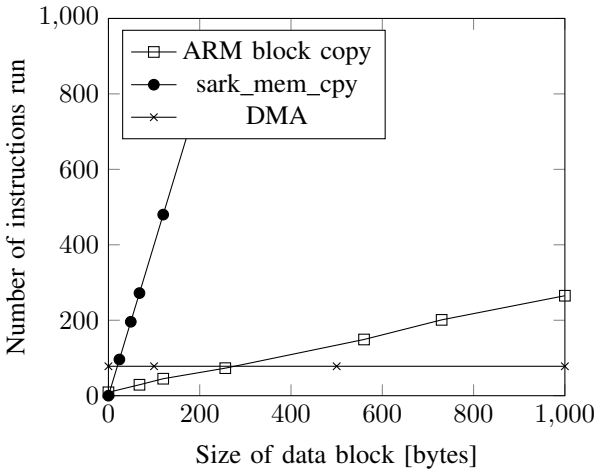

Fig. 6. Growth of the number of instructions required to copy data using different copying routines. sark_mem_cpy copies data byte per loop iteration.

B. Memory compactor

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

The timing statistics also show 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 (Fig.8) and on each call of the compactor only one of the regions will be compacted, but different region each time.

C. Buffer extender

The buffer extender (IV-C) operates by copying a single buffer and shifting elements down by a small number of bytes. Table IV summarises the running times of extension

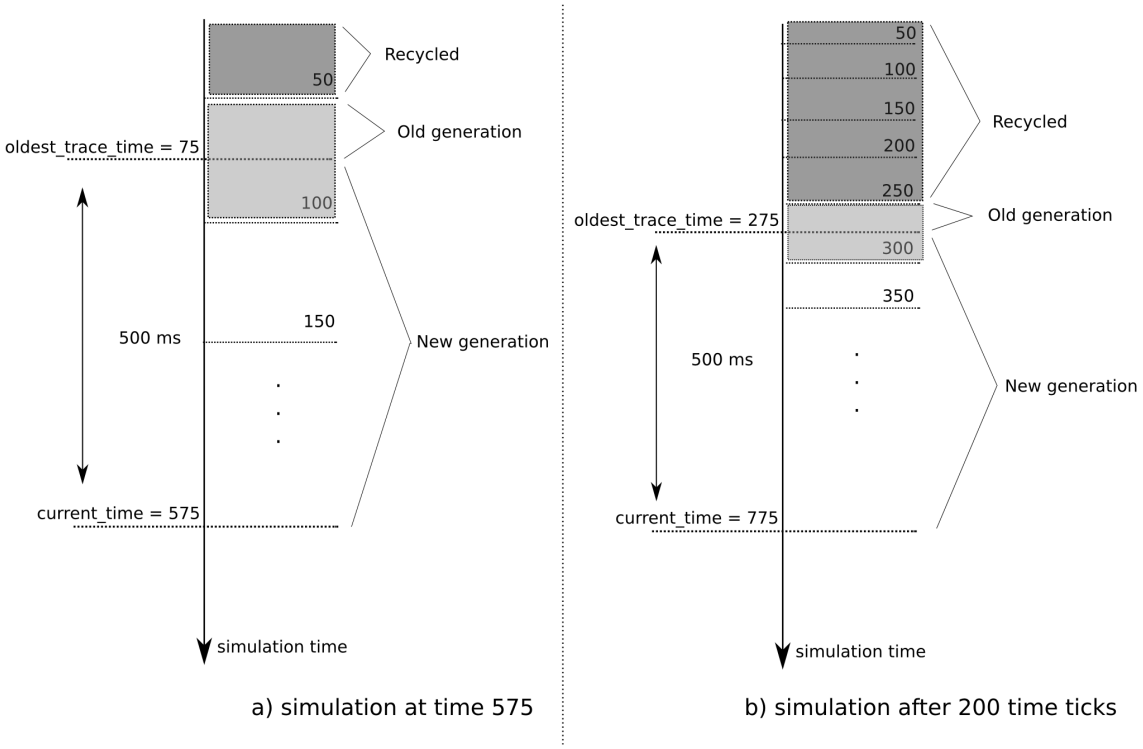

Fig. 7. Generational garbage collection at two specific time instants. The oldest trace time was set to $current\_time - 500ms$. The generation step is 50ms. All traces are additionally categorised into two higher level generations, *old generation* and *new generation* similarly as demonstrated in Garbage Collection Handbook, Chapter 9 [15]. 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.

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. Each column represents the copying method used to copy buffers to SDRAM.

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 II. 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.

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

TABLE III. 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.

operations with various, differently sized buffers. On average, the extender is called *3.5 times* per SpiNNaker time step thus it approximately occupies only 1% of the time available in a single timer interrupt.

D. 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. The run times of the scanner are summarised in Table II. 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.

E. Combined system performance analysis

In figure 10 (a), 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 10 (b), the rate of 0.25kHz produces no wasted scan cycles in the simulation period 800-1000 ms.

Figure 9 shows the memory occupation as simulation progresses, for both garbage collected and non-collected approaches. 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.

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. It was observed that without garbage collection, timer interrupt callback code, 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.

F. Generational garbage collector

Generational garbage collector (IV-E) 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 we discuss the improvements and downsides that generational collector provides.

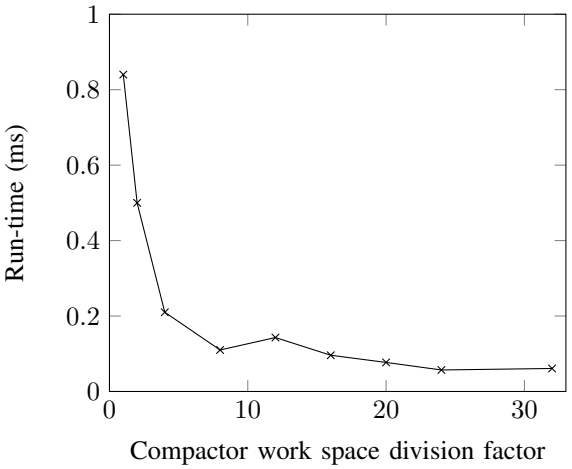

Fig. 8. Run times of the single invocation of the compactor as values of the compactor space division factor are increased.

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

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. Table III summarises the improved average run times of the scanner and it can be seen that the growth is much slower than previously (Table II). Additionally, figure 10 (c) demonstrates that generational memory management does not reduce the numbers of bytes collected per invocation of the scanner.

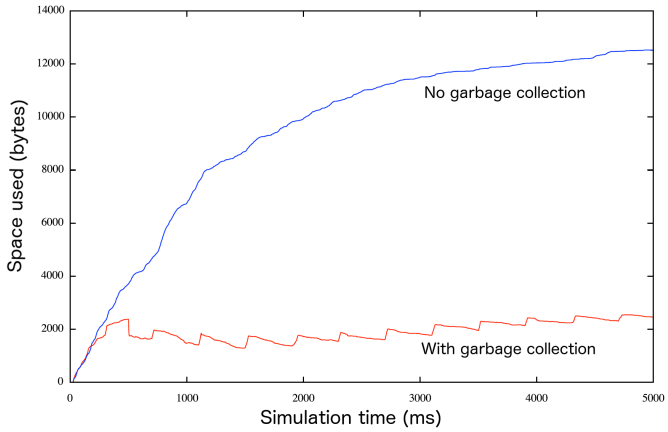

Fig. 9. Space used for history traces in a simulation with 255 neurons per core.

VI. Conclusion

Using SpiNNaker chip, we have demonstrated that by instantiating a structure of dynamic history trace buffers (IV-A) we can add and remove 'live' objects by simply shifting the data and moving the boundary pointers (IV-D). We then employed a periodical event that compacts memory on the heap by referring to these 'live' pointers, thus discarding any 'dead' objects to free memory (IV-B). Due to usage of relatively slow SDRAM in memory compaction operation, we have developed a basic approach to fragment the compaction operation across multiple timer interrupts. The removal of data is done by the scanner (IV-D) using exhaustive search. To reduce the scanning area, we have developed a variant of generational garbage collector on SpiNNaker, that accurately finds

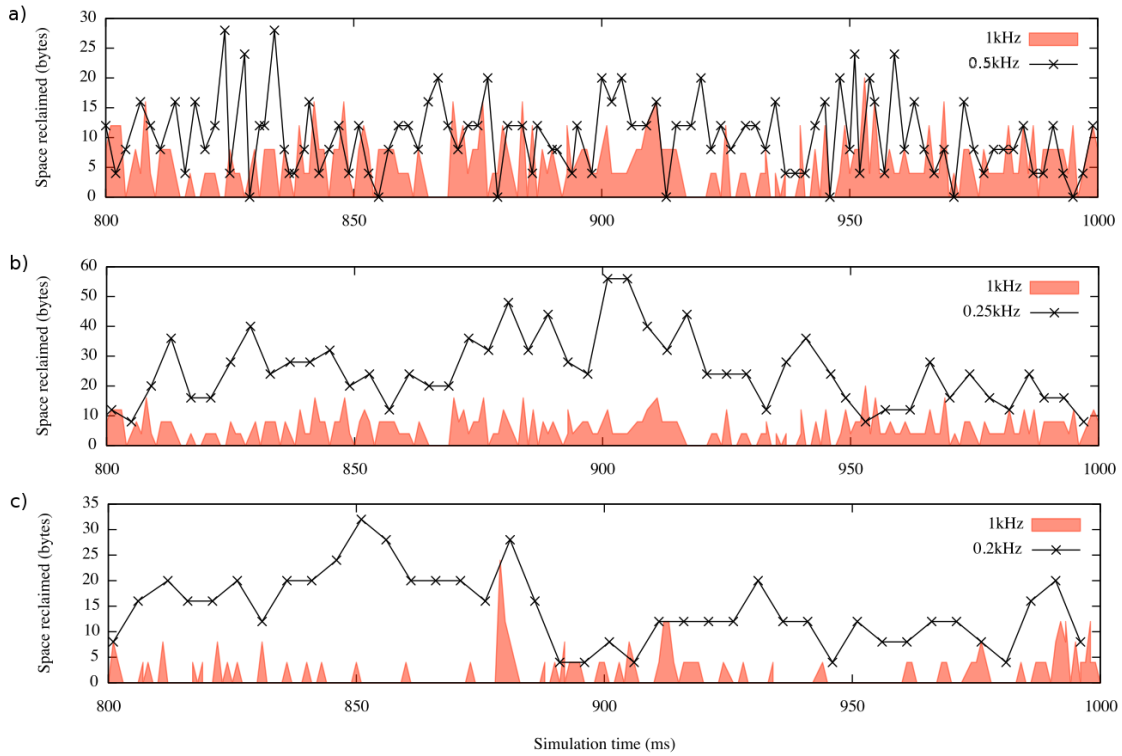

Fig. 10. Space reclaimed per each invocation of the scanner: a) scanning rates 1 and 0.5 kHz, b) rates 1 and 0.25 kHz and c) generational collector with rates 1 and 0.2 kHz.

all 'dead' objects without the exhaustive search (IV-E). Finally, we have demonstrated how garbage collection, incorporated into each timer interrupt together with neuron and synapse updating routines, runs and completes in SpiNNaker's time boundary of 1ms per clock tick (III-D, V-E). With these contributions we have demonstrated how classical garbage collection algorithms can be transformed to apply to such problems as history trace management in simulations of synaptic plasticity as well as provided a basic foundation of automatic memory management for SpiNNaker, which previously it did not have.

VII. Acknowledgements

This work was partially supported by ERC Advanced Grant 320689, FET grants FP7-604102 and DLV-720270, and an EPSRC PhD studentship. We would also like to thank SpiNNaker team members: Jamie Knight, Andrew Rowley and Alan Stokes for their invaluable help in this project and reviewers for useful feedback on the manuscript.

References

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

[2] S. Friedmann, J. Schemmel, A. Grübl, A. Hartel, M. Hock, and K. Meier, "Demonstrating hybrid learning in a flexible neuromorphic hardware system," *IEEE Transactions on Biomedical Circuits and Systems*, vol. 11, no. 1, pp. 128–142, Feb 2017.

[3] S. Kunkel, T. Potjans, J. Eppler, H. E. Plesser, A. Morrison, and M. Diesmann, "Meeting the memory challenges of brain-scale network simulation," *Frontiers in Neuroinformatics*, vol. 5, p. 35, 2012. [Online]. Available: http://journal.frontiersin.org/article/10.3389/fninf.2011.00035

[4] S. B. Furber, F. Galluppi, S. Temple, and L. A. Plana, "The spinnaker project," *Proceedings of the IEEE*, vol. 102, no. 5, pp. 652–665, May 2014.

[5] "Arm information center, tightly-coupled memory." [Online]. Available: http://infocenter.arm.com/

[6] J. McCarthy, "Recursive functions of symbolic expressions and their computation by machine, part i," *Commun. ACM*, vol. 3, no. 4, pp. 184–195, Apr. 1960. [Online]. Available: http://doi.acm.org/10.1145/367177.367199

[7] H. Lieberman and C. Hewitt, "A real-time garbage collector based on the lifetimes of objects," *Commun. ACM*, vol. 26, no. 6, pp. 419–429, Jun. 1983. [Online]. Available: http://doi.acm.org/10.1145/358141.358147

[8] E. W. Dijkstra, L. Lamport, A. J. Martin, C. S. Scholten, and E. F. M. Steffens, "On-the-fly garbage collection: An exercise in cooperation," *Commun. ACM*, vol. 21, no. 11, pp. 966–975, Nov. 1978. [Online]. Available: http://doi.acm.org/10.1145/359642.359655

[9] H. G. Baker, Jr., "List processing in real time on a serial computer," *Commun. ACM*, vol. 21, no. 4, pp. 280–294, Apr. 1978. [Online]. Available: http://doi.acm.org/10.1145/359460.359470

[10] H. Markram, J. Lubke, M. Frotscher, and B. Sakmann, "Regulation of synaptic efficacy by coincidence of postsynaptic APs and EPSPs," *Science*, vol. 275, no. 5297, pp. 213–215, Jan 1997.

[11] S. Nabavi, R. Fox, C. D. Proulx, J. Y. Lin, R. Y. Tsien, and R. Malinow, "Engineering a memory with LTD and LTP," *Nature*, vol. 511, no. 7509, pp. 348–352, Jul 2014.

[12] A. Morrison, A. Aertsen, and M. Diesmann, "Spike-timing-dependent plasticity in balanced random networks," *Neural Comput*, vol. 19, no. 6, pp. 1437–1467, Jun 2007.

[13] N. Parlante and J. Zelenski, "The ins and outs of c arrays, computer science class handout, cs107," Standford University, 2008.

[14] "Arm information center: Block copy with ldm and stm." [Online]. Available: http://infocenter.arm.com/

[15] R. Jones, A. Hosking, and E. Moss, *The Garbage Collection Handbook*, ser. Applied Algorithms and Data Structures Series, 2012.