Deluge: Achieving Superior Efficiency, Throughput, and Scalability with Actor Based Streaming on Migrating Threads

Brian A. Page
Dept. of Computer Science and Engineering
University of Notre Dame
Notre Dame, IN USA
bpage1@nd.edu

Peter M. Kogge
Dept. of Computer Science and Engineering
University of Notre Dame
Notre Dame, IN USA
kogge@nd.edu

***Abstract*—Applications where streams of data are passed through large data structures are becoming of increasing importance. For instance network intrusion detection and cyber security as a whole rely on real time analysis of network traffic. Unfortunately, when implemented on conventional architectures such applications become horribly inefficient, especially when attempts are made to scale up performance via some sort of parallelism. An earlier paper discussed streaming anomaly detection within a stream having an unbounded range of keys on the Lucata migrating thread architecture. In this paper we introduce *Deluge*, a new implementation that addresses several inadequacies of previous designs and seeks to more directly target the hardware efficiencies inherent to migratory execution within a PGAS address space. Deluge achieves major improvements in hardware efficiency, throughput, and scalability over previous implementations.**

***Index Terms*—Emerging Architectures, Migrating Threads, Streaming, Agent Based Execution, Scalability**

I. Introduction

Applications where streams of data pass through large data structures are of increasing importance. Examples include cyber-security, social networks, interactive messaging, and e-commerce. Unfortunately, implementations on conventional architectures such applications become horribly inefficient, especially when attempts are made to scale up performance via parallelism. This is true even of our own attempts at producing high throughput implementations [1], [2].

An earlier study [2] investigated the scalablity of streaming in an unbounded key space using the Lucata[1] migrating thread architecture. In that study we chose to use the Firehose streaming benchmark [3]–[5], [6] as a framework.

In this paper we introduce Deluge, a new streaming implementation in the nature of Firehose's variant 2, for use on the Lucata migrating thread architecture. Deluge makes many departures from the conventional benchmark implementation code in an effort to optimize for the transient execution pattern inherent to migrating threads. In doing so **Deluge achieves 57X throughput over previously reported implementations on Lucata** [2], in addition to achieving substantially superior scalability and hardware efficiency over a conventional cluster implementation.

The main contributions of this paper are:

- Deluge: a vastly superior implementation of Firehose variant 2 for use on the Lucata migrating threads architecture.
- Agent based execution pattern which obtains over an order of magnitude greater hardware efficiency and throughput over previous designs.
- Performance and scalability projections for a next generation migrating thread system based on existing results.

Finally, it should be noted that there are a growing number of other problems where random or irregular accesses cause major scaling problems for conventional architectures, but early evidence suggests again that a migrating thread architecture has significant benefits. This includes two different machine learning problems: one [7] on very sparse data and strong scaling, and one on decision forests [8]. Strong scaling of SpMV (Sparse Matrix-Vector product) on conventional architectures suffers from inefficiencies [9], but results [7] indicates that much better scaling may be possible with migrating threads, versus not only conventional but versus a variety of hybrid architectures [10]–[12]. More general sparse linear algebra operators may also benefit [13].

II. Background

A. Firehose Streaming Benchmark

Firehose [14] resembles a cyber-security like streaming function where incoming packets are to be monitored. When some number of packets with the same address have been detected, the payload fields are examined for potential anomalies, and if detected, a report issued. The IP address in each incoming packet is used to probe a very large hash table, and when a match is found, data from the packet's payload is merged into the entry, and a match count incremented. When 24 packets have been found, the aggregated payload is analyzed. An "atypical" outcome results in the IP address being flagged. Three variants are proposed: one with a limited key range, a second with an expanded key range, and a more complex third with a nested key extraction.

[1]Lucata formerly EMU Solutions Inc.

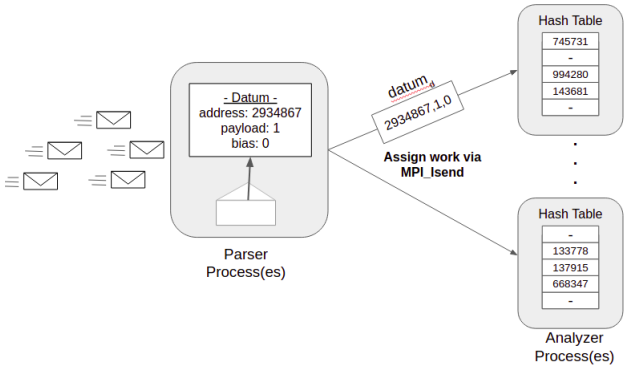

Fig. 1. Firehose PHISH Python/C++ Design: Parser converts datums (ASCII strings) into address, payload, and bias flag triplets. The datum is assigned to an analyzer process via UDP packets. Analyzers look up the address in their local hash table update counters and check for anomalies as necessary.

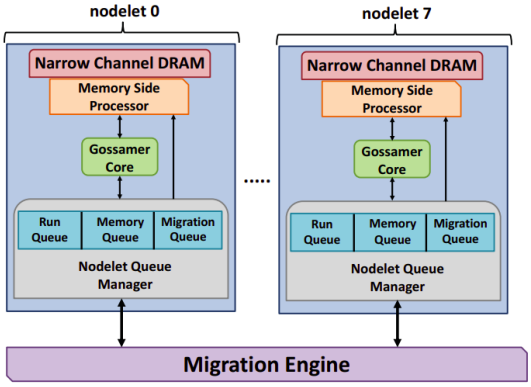

Fig. 2. A single node in the Emu Chick system. There are 8 nodelets within a single node. Nodes are connected over a Serial RapidIO interconnect (not shown here).

Using an active set key generator [14], variant 2 possesses a potentially infinite key range (2^{64} possible keys) from an existing computational capability standpoint. Due to resource constraints, implementations center around limiting memory footprint by "aging" keys out of a hash table when they are likely to no longer be present in the current active set.

The Firehose website contains several reference implementations. However in this study we focus on their PHISH Python/C++ version which utilizes UDP for multi-process communication in a distributed or hybrid environment. The benchmark is run for some predetermined amount of time or total datum volume and statistical data is output for review. Runs may be done with multiple parsing processes, multiple analysis processes, or a combination of the two. This creates the possibility for the following producer-consumer relationships: one-to-one, one-to-many, and many-to-many. Fig. 1 shows the execution flow for the PHISH/C++ implementations in which a single datum parser process (producer) assigns work to multiple analyzer processes (consumers).

Analysis of an arbitrary datum occurs only within the analyzer process to which it was assigned. The PHISH C++ code uses std::unordered_map for the hash table functionality of storing and looking up keys, while a Least Recently Used (LRU) eviction mechanism using doubly linked lists tracks keys based on occurrence for removal or reuse. The total hash table coverage amongst all analyzer processes is subdivided into segments equivalent to $global\_size/analyzer\_count$, where $global\_size$ is some multiple of the generator's active set size, and $analyzer\_count$ is the number of analyzers.

It is worth noting that performance can be dependent on workload distribution which is directly determined by the active set generator, system size, and key hashing function used for datum assignment.

B. Migrating Thread Architecture

A migrating thread architecture [15] is one where the underlying hardware, not software, moves the state of a thread as required during execution. Fig. 2 diagrams such an architecture as implemented by Lucata Solutions [16]. The basic unit, a **nodelet**, is a memory module, its controller and some number of multi-threaded cores. All nodelet memory resides in a common address space. A network connects all nodelets. A thread runs in a multi-threaded "**GC core**" until it makes a memory reference that is not contained in that nodelet's memory. The hardware then puts the thread to sleep, packages it, and moves it over the network to the correct nodelet, where it is unpacked and restarted. A thread can spawn independent child threads. Also, the memory controller contains hardware to implement atomic operations as close to memory as possible. Finally, very lightweight threads can be spawned to perform remote memory operations without moving the parent.

The current prototype used in this study, a Lucata Chick, is housed at Georgia Tech's CRNCH center[2]. It has up to 64 nodelets, each with 8GB of memory and one 175MHz multi-threaded core. These nodelets are packaged 8 to a **node board** which supports a RapidIO-based network that connects 8 such boards in a single chassis. A dual core POWER microprocessor on each node board runs Linux, manages a local SSD, and launches migrating threads into the system. The nodelet logic on each board is implemented in an FPGA. Table I compares its characteristics to that of the conventional system used as a baseline.

In comparison, the core clock rate of the baseline is 15.2X that of the CRNCH Chick. A more complete comparison is probably even higher than this in favor of the baseline as the nodelet cores are single issue and the baseline cores are multi-issue. Also, on a per core basis, the baseline has about 4.4X a pro-rated memory bandwidth of a core in the Chick, but, because of the memory channel design used in the nodelets, the ability of a Chick core to handle different independent memory accesses is actually 1.8X higher. Finally, the average network injection bandwidth per core is higher for the Chick than the baseline[3].

[2]https://crnch.gatech.edu/rogues-Lucata

[3]It should be noted that the baseline system has much higher bandwidth between its on-node 48 cores, and thus this ratio has a lot of caveats

System	Baseline	CRNCH	CRNCH	Ratio:	Ratio:
Type	HPE DL385 Gen10	Chick	Pathfinder-S	Baseline/Chick	Pathfinder/Chick
Socket	AMD 7451	Arria FPGA	Stratix FPGA		
Cores/Socket	24	8	24	3	3
Core Clock (GHz)	2.3	0.175	0.220	15.2	1.3
Compute Cycles per Socket (G/s)	63.8	1.4	5.3	45.6	3.8
Mem. B/W per Socket (GB/s)	170.62	12.8	34.1	13.3	2.7

TABLE I
COMPARISON TO BASELINE IMPLEMENTATION.

The programming tool chain is based on Cilk: C with a prefix to function calls to spawn new threads, a sync to wait for a set of children to complete, and a parallel *forall* to have a set of independent threads cooperate on a loop. Supported intrinsics include a rich set of remote atomic operations.

A second generation system, Pathfinder-S, is currently being installed in the CRNCH Center, and should be available in the near future. This system has 3X the cores per node board and 2.7X the memory bandwidth. Also, unlike the Chick, a thread running on any core can access any of the 8 memory channels without a migration. This improves load balancing. Only accessing memory on some other board causes a migration.

III. STREAMING ON LUCATA MIGRATING THREADS

A. Previous Implementation

In a previous study [2] we implemented unbounded anomaly detection on the Lucata Chick by creating 2 different thread pools, producer and consumer, on every nodelet. Fig. 3 illustrates the general concept of the previous design. Producer threads on each nodelet parse datums, stored as ASCII strings, into usable address, payload, and bias flag values. The address field is then hashed to determine which nodelet and consumer thread the datum is to be assigned to for analysis. The producer thread then migrates to the destination nodelet and adds the datum's fields to the appropriate consumer thread's work queue. The producer thread returns to the nodelet on which it was spawned and parses the next datum..

Consumer threads never migrate and continually check for new work assigned to them and if present analyze their assigned datums by performing a look of the address field in their local hash table. Consumers also check for anomalies and update statistics counters where appropriate. As was shown in [2] decent throughput scalability was achieved using this method, however exact performance was highly dependant on total datum volume, hash table size, and the number of nodelets/threads in use.

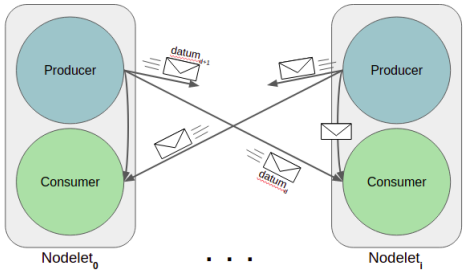

Fig. 3. Previous Lucata Chick Implementation [2]: Each Nodelet shared hardware resources between a producer and consumer thread pool. Producers performed ASCII string conversion and assignment. Consumers performed hash table lookup, update, and anomaly detection.

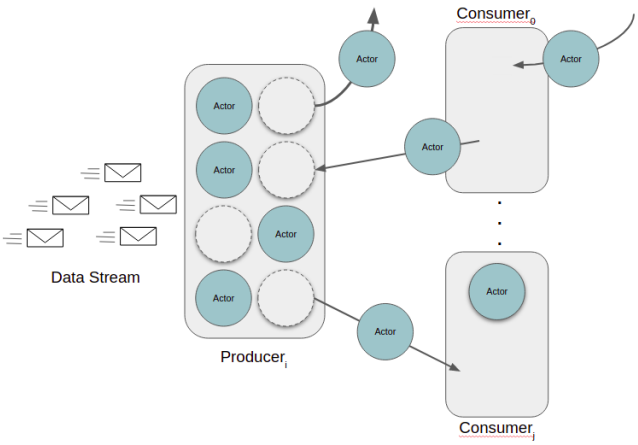

Fig. 4. Deluge on Migrating Threads: Producer nodelets spawn a local collection of "actor" threads. Actors generate datums and then migrate to a consumer nodelet to perform analysis based on the datum's contents. When analysis of the current datum is complete actors migrate back to nodelet they were spawned on. Consumers have no resident execution.

B. Deluge: Actor Based Execution

Deluge utilizes the actor execution model via the use of migrating threads in order to perform analysis of the generated datums. Unlike our previous attempts, here there is no distinction between "producer" and "consumer" threads throughout the system, but rather each thread is an "actor" which both generates a new datum to evaluate and performs the analysis of that datum itself.

In Fig. 4 we see that the system itself is split into producer and consumer nodelets. Actor threads are spawned on a producer nodelet and begin generating datums using the active set generator as done in the official Firehose benchmark specification. An actor will hash the address in the datum it generates to determine the nodelet which governs the hash table portion the address should be checked against.

At this point the Actor migrates to the appropriate consumer nodelet and proceeds with analysis of the datum. Being a shared memory environment, all Actors currently executing on a consumer nodelet share the local hash table, and therefore must acquire a lock on the given hash table slot they wish to perform insertions, updates, or deletions on. At this point, analysis occurs in the same manner as in the prior imple-

mentation, and a least recently used (LRU) list maintains the order of key occurrence in order to select a good candidate for eviction in the event the hash table has become full.

Once the actor completes evaluation of its current datum, the actor thread migrates back to the producer nodelet it was spawned on and generates a new datum. This process continues until the desired datum volume is reached. The number of actors spawned on a producer nodelet can be in the thousands. With such a large actor thread pool to draw from producers are able to maintain high utilization even though many of their resident actors are have migrated or are in the process of ”flooding” consumer(s) at any one time.

By using the actor based execution model in this way, only producer nodelets function as the permanent home for actor threads. **Consumer nodelets have no resident execution.** Because of this a consumer nodelet's hardware resources are dedicated to the analysis of datums only, and do not have to be shared with the parsing of ASCII strings as was necessary in previous versions.

IV. Experimental Setup

A. Conventional Baseline Implementation

To form a baseline for our comparisons we developed an implementation of Firehose variant 2 for use on a conventional cluster (multiple nodes of the type in the second column of Table I). Our baseline code functions in the same manner as described in Sec. II-A in which datum parsing and datum analysis is carried out by separate specialized processes running on different distributed cores and or nodes.Each process resides on its own physical node so that any arbitrary process has complete control of the resources present on its host node.

The key difference between our implementation and the benchmark spec is that our producer processes generate datums using the active set generator, then send them via UDP packet to the appropriate analysis process. This was done to match the Lucata Deluge code since the FPGA based design does not currently allow for efficient network access in the same way as an ASIC based system and therefore provides a more accurate comparison between systems.

Additionally, though possible, we did not design this as a hybrid implementation: meaning that each process, be it a parser or analyzer has only one execution thread. This makes comparison to Lucata chick hardware much simpler as the relationship between cores and nodelets is notionally 1-to-1.

B. Testing Migrating Threads

For our weak scaling tests, we increase the number of consumer nodelets in powers of 2. Additionally we increase the number of producer nodelets, and the number of actor threads spawned within each producer, by powers of 2 up to a total of 1024 accross all producers. The sum of the consumer and producer nodelet count must be no more than 64.

Generation of datums is done by each actor thread, with each thread generating from its own key distribution and active set. Run time measurements are started before the recursive spawn which generates worker threads in each team on each nodelet. A *cilk_sync* prevents further program execution until all nodelets have completed, upon which the stop time is measured and total runtime determined. The time required by the asynchronous updates to statistic counters is included, and is consistent with the benchmark specification and our conventional cluster's baseline implementation.

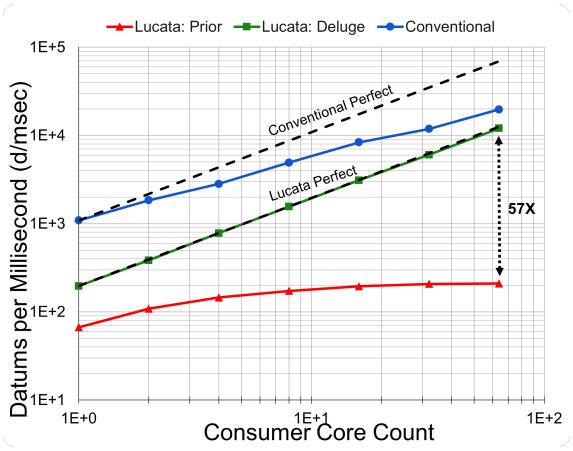

Fig. 5. Observed throughput scalability. Each curve represents the maximum observed throughput for each consumer core count.

V. Evaluation

Comparing two systems is often done by looking at how performance varies as a function of some resource. For systems that are close in architecture (such as conventional clusters) this is typically using the number of cores or nodes in each system. For a migrating thread (especially a Chick) versus a conventional system, however, things get a bit more difficult. For hardware references, we could use “cores,” but also a case could be made for comparing on the basis of sockets or nodes. Alternatively, we could use metrics such as number of compute cycles available (clock times core count), or perhaps as aggregate memory bandwidth or access rate. As each gives somewhat different insight we will utilize several here.

A. Throughput Scalability

The Firehose benchmark defines throughput as the maximum number of datums per millisecond (d/msec) that can be handled by a system before problems occur. Using consumer core count is an obvious metric. Since we did not run multiple threads within any process, core count is equivalent to process count in the conventional implementation. Conversely with respect to Deluge on the Lucata Chick, a nodelet contains a single processing core (that may be handling 100s of threads at any point in time). Thus consumer nodelet count is equivalent to core count. Fig. 5 shows the observed maximum throughput achieved at each consumer core count up to 64 (the maximum for this system). Additionally we show the theoretical perfect scaling for each system, computed as $max(single\_core\_throughput) * consumer\_count$. Less than perfect scalability of our previous streaming attempt with migrating threads is evidenced by a rapidly flattening curve

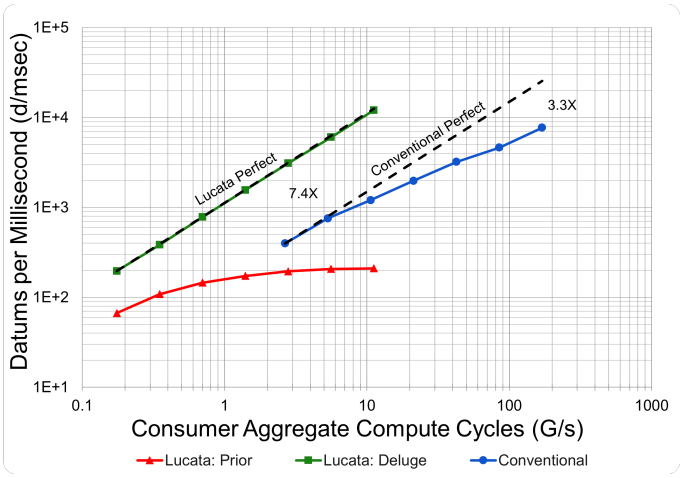

Fig. 6. Observed throughput scalability as a function of compute cycles.

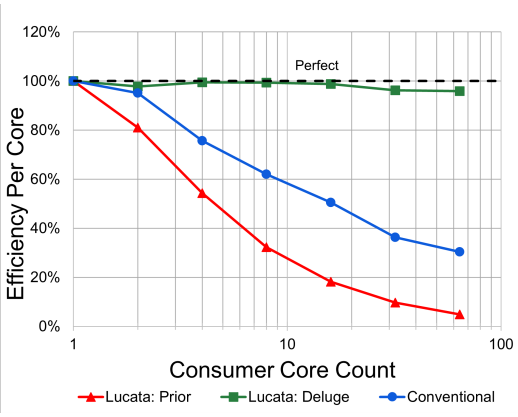

Fig. 7. Per core hardware efficiency as a percentage of single core efficiency.

as core counts increase. The conventional cluster implementation doesn't flatten, but deviates significantly from perfect as the core count increases. While the conventional system throughput appears unusually low in relation to previously reported data, to improve comparison both systems perform datum generation in the exact same way and therefore incur added computational requirements thus reducing throughput.

In contrast Deluge achieves near perfect throughput scaling as we increase the number of consumer cores. In fact Deluge's peak throughput was approximately **57X that of our previous implementation** (listed as "Lucata: Prior") [2]. This suggests that Deluge obtains vastly improved hardware utilization thanks to a code which better targets the capabilities of the migrating thread system. Also, Deluge and the conventional system cross over at 16 cores, with Deluge's lead growing at larger systems (and this is with far slower cores).

Perhaps a better resource to use as a basis of comparison is a measure of computational capability available. We chose aggregate compute cycles $clock\_rate * consumer\_count$, Fig. 6. Again the excellent scaling of Deluge is apparent, but what is also apparent is that each cycle of execution in a Chick core produces *at least 7 times as much performance* as a single cycle of the conventional system. This is not counting the fact that the Lucata is a single-issue per cycle design, and that, besides being much faster, the conventional cores can issue multiple instructions per cycle. There is something significant in the Lucata architecture that makes it far more efficient than the conventional architecture.

Other measures for reference are possible, including use of memory bandwidth, where again we would think that the conventional system holds a significant advantage in aggregate numbers. A graph like Fig. 6 but using memory bandwidth as the horizontal axis reaches a similar conclusion, but the fact that not all cores in the conventional sockets are used confuses the accounting for how much of a conventional socket's bandwidth should be counted.

B. Hardware Efficiency

A common metric for benchmarking HPC systems is hardware efficiency. Fig. 6 suggests there is a significant difference. Here we are considering hardware efficiency as what percentage of the observed datums per millisecond is attributable on average to an individual consumer as core counts increase, compared to that of the single consumer core case.

Fig. 7 calculates the efficiency for Deluge, the baseline conventional cluster, and our previous attempt on migrating threads. As can be seen our previous attempt obtained the poorest hardware efficiency, dipping as low as just 5% when 64 consumers were used.

Our baseline implementation using the conventional cluster fared only marginally better. While the conventional system's efficiency fell as low as 28% its general trend was the roughly the same as our original implementation, decreasing rapidly as system size increases.

Meanwhile Deluge on the Lucata chick maintained a relatively consistent efficiency percentage with a maximum, minimum, and average of 99.5%, 96%, and 98% respectively. This means that as we increase consumer core count each new core added is providing at least 96% of the throughput of the single core case. Deluge's substantially improved, and near perfect hardware efficiency, is what allows it to obtain superior scalability over previous attempts and even that of the conventional cluster.

VI. Other Implementations

Firehose variant 1 is a much simpler problem. We previously implemented a migrating thread for that variant [17] and saw throughputs of 222K d/msec, with good (but not perfect) scalability.

There are several other reported implementations of Firehose variant 2. The website discusses two other implementations. One is a shared memory implementation on a single node where performance on one core was 1900 d/ms, and only 3400 for 7 cores. These numbers are higher than either of our implementations, but it must be remembered that the multi-process overheads of are not present in a shared memory

implementation. However, the efficiency at 7 cores is only about 25%, so clearly this is not a scalable implementation.

Another very relevant implementation discussed on the website and in [5] is a cluster implementation on a Cray CS-300 with data from dual-socket 16 core 2.6GHz nodes in configurations from 40 up to 300 nodes. Scalability was decent, but performance on a per core basis was about 62 d/msec, considerably less than any of our implementations. Efficiencies appear to be even less than the ones for our conventional implementation here.

A final implementation used NVIDIA Tesla GPUs [6]. The limited published data indicated a performance of 61K d/ms for 2 Tesla M40s, and 122K d/msec for 4 M40s. This is higher than the implementations here, but it is unclear how such a system would scale to very large numbers of such nodes.

A. Extrapolation to Pathfinder-S

As mentioned earlier, a new migrating thread system is being installed in the CRNCH center, with characteristics in Table I. There are more, faster, cores and more memory bandwidth. The 30% increase in clock rate implies that we should expect a 30% jump for the Lucata implementations in both Fig. 5 and 6. The 3X jump in cores, a concurrent almost 3X jump in memory bandwidth, and a doubling in the number of chassis suggests that we could now see in a revised Fig. 5 a Deluge line that beats the conventional implementation at 6 cores, and continues to scale up to around 384 cores. If scaling remains near perfect, this puts the numbers near what was reported on the Firehose website for a 1000 cores of a Cray CS-300 cluster, again where each core has a 10X or better clock. This represents a potentially tremendous improvement in the state of the art for this problem.

VII. Conclusion

In this study we introduced Deluge, a new implementation of streaming anomaly detection on migrating threads, which achieved over an order of magnitude improvement to both hardware efficiency and throughput over prior designs. It is clear that our previous attempt at unbounded key space anomaly detection while fruitful from a knowledge gained perspective, were lack luster in their performance and scalability. By tailoring Deluge to more closely reflect the encapsulated yet freely mobile nature of execution throughout the migrating thread architecture **we were able to achieve near perfect throughout scaling as well as hardware efficiency at scale**.

What makes these results so startling is that the Deluge implementation is on a hardware base that per core is considerably slower, and has an equally lower equivalent bandwidth. Regardless of possessing comparatively lower compute resources, as system sizes increased the dramatically superior hardware efficiency achieved by Deluge and migrating threads lead to a significantly superior system performance after just a few cores. The ability to support huge numbers of threads relative to the number of cores makes for excellent and essentially self-managing load balancing.

As previously stated the Lucata chick exhibited many novel architectural features and provided a plethora of benchmark and proof of concept data for such a system. The new Pathfinder-S may prove to be even more useful for the execution of irregular applications thanks to its numerous hardware equipment upgrades/advancements. In the very near term we will be focusing on characterizing the new system and its uses. We also look forward to adapting the paradigm used here to other streaming problems.

Acknowledgements

This work was supported in part by NSF grant CCF-1822939, and in part by the University of Notre Dame. We would also like to acknowledge the CRNCH Center at Georgia Tech for allowing us to use the Chick system there.

References

[1] B. A. Page and P. M. Kogge, "Scalability of streaming on migrating threads," *High Performance Extreme Computing (HPEC)*, September 2020.

[2] ——, "Scalability of streaming anomaly detection in an unbounded key space using migrating threads," in *High Performance Computing*, B. L. Chamberlain, A.-L. Varbanescu, H. Ltaief, and P. Luszczek, Eds. Cham: Springer International Publishing, 2021, pp. 157–175.

[3] J. Eaton, "firehose, pagerank, and nvgraph: Gpu accelerated analytics."

[4] K. Anderson, "FIREHOSE: Benchmarking Streaming Architectures," in *Chesapeake Large Scale Data Analytics Conf.*, 2016.

[5] J. Berry and A. Porter, "Stateful streaming in distributed memory supercomputers," in *Chesapeake Large Scale Data Analytics Conf.*, 2016.

[6] M. Bisson, M. Bernaschi, and M. Fatica, "GPU Processing of Streaming Data: a CUDA Implementation of the FireHose Benchmark," *High Performance Extreme Computing (HPEC)*, October 2016.

[7] B. A. Page, "Scalability of irregular problems," Ph.D. dissertation, 2020.

[8] P. L. Springer, T. Schibler, G. Krawezik, J. Lightholder, and P. M. Kogge, "Machine learning algorithm performance on the lucata computer," *IEEE High Performance Extreme Computing Conf. (HPEC)*, Sept. 2020.

[9] B. Bylina, J. Bylina, P. Stpiczyński, and D. Szałkowski, "Performance Analysis of Multicore and Multinodal Implementation of SPMV Operation," vol. 2, pp. 569–576, 2014. [Online]. Available: https://fedcsis.org/Proc./2014/drp/313.html

[10] B. A. Page and P. M. Kogge, "Scalability of hybrid sparse matrix dense vector (spmv) multiplication," *Int. Conf. on High Performance Computing & Simulation*, Jul 2018. [Online]. Available: http://par.nsf.gov/biblio/10064735

[11] ——, "Scalability of hybrid spmv on intel xeon phi knights landing," *Int. Conf. on High Performance Computing & Simulation*, Jul 2019. [Online]. Available: https://par.nsf.gov/biblio/10109480

[12] ——, "Scalability of hybrid spmv with hypergraph partitioning and vertex delegation for communication avoidance," *Int. Conf. on High Performance Computing & Simulation*, Mar 2020.

[13] G. P. Krawezik, S. K. Kuntz, and P. M. Kogge, "Implementing sparse linear algebra kernels on the Lucata Pathfinder-A computer," in *IEEE High Performance Extreme Computing Conf. (HPEC)*, Sept. 2020.

[14] S. N. Labs, "Firehose benchmarks," http://firehose.sandia.gov/.

[15] P. Kogge, "Of piglets and threadlets: Architectures for self-contained, mobile, memory programming," *Innovative Architecture for Future Generation High-Performance Processors and Systems*, pp. 130–138, Jan. 2004.

[16] T. Dysart, P. Kogge, M. Deneroff, E. Bovell, P. Briggs, J. B. Brockman, K. Jacobsen, Y. Juan, S. Kuntz, R. Lethin, J. McMahon, C. Pawar, M. Perrigo, S. Rucker, J. Ruttenberg, M. Ruttenberg, and S. Stein, "Highly scalable near memory processing with migrating threads on the emu system architecture," Piscataway, NJ, USA, pp. 2–9, Nov. 2016. [Online]. Available: https://doi.org/10.1109/IA3.2016.7

[17] B. A. Page and P. M. Kogge, "Migrating threads to accelerate streaming - extended abstract," *IWIA 2020: Innovative Architecture for Future Generation High-Performance Processors and Systems*, Jan. 2020.