Greatly Accelerated Scaling of Streaming Problems with A Migrating Thread Architecture

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 continuous streams of data are passed through large data structures are becoming of increasing importance. However, their execution on conventional architectures, especially when parallelism is desired to boost performance, is highly inefficient. The primary issue is often with the need to stream large numbers of disparate data items through the equivalent of very large hash tables distributed across many nodes. This paper builds on some prior work on the Firehose streaming benchmark where an emerging architecture using threads that can migrate through memory has shown to be much more efficient at such problems. This paper extends that work to use a second generation system to not only show that same improved efficiency (10X) for larger core counts, but even *significantly higher raw performance* (with FPGA-based cores running at 1/10th the clock of conventional systems). Further, this additional data yields insight into what resources represent the bottlenecks to even more performance, and make a reasonable projection that implementation of such an architecture with current technology would lead to 10X performance gain on an apples-to-apples basis with conventional systems.**

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

I. Introduction

Applications where streams of data pass through large data structures such as huge hash tables are of increasing importance. Examples include cyber-security, social networks, interactive messaging, and e-commerce. The ExaBiome bioinformatics project[1] (part of the US ECP exascale effort) is an example. Unfortunately, implementations on conventional architectures become horribly inefficient, especially when attempts are made to scale up performance via parallelism.

Earlier studies [1]–[3] investigated the scalablity of streaming in an unbounded key space using the Lucata[2] migrating thread architecture. In those studies we chose to use the Firehose streaming benchmark [4]–[7] as a framework.

This paper extends that work in two directions. First is a port to a newer and larger migrating thread platform with significantly more, and slightly faster, cores. This allows better and more accurate scaling measurements. Second, a comparison of these new results with those of a prior implementation provides significant insight into exactly how the resources of such an architecture are used. This in turn permits a reasonable projection to be made of what performance might be if the architecture of the migrating thread machine was implemented in the same technology as a conventional system.

[1]https://www.exascaleproject.org/research-project/exabiome/

[2]Lucata formerly EMU Solutions Inc.

The results of this study are rather remarkable, even considering the current implementation base is an FPGA with core clocks at a measly 220MHz:

- On a throughput per compute cycle, the migrating thread platform is 10X more efficient, even where only one instruction can be dispatched per clock cycle.
- One node of such an architecture significantly outperforms in raw throughput a single conventional node.
- A projection to an implementation in technology comparable to modern conventional architectures indicates that something approaching 10X in raw performance is possible.

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 [8] on very sparse data and strong scaling, and one on decision forests [9]. Strong scaling of SpMV (Sparse Matrix-Vector product) on conventional architectures suffers from inefficiencies [10], but results [8] indicates that much better scaling may be possible with migrating threads, versus not only conventional but versus a variety of hybrid architectures [11]–[13]. More general sparse linear algebra operators may also benefit [14].

Other examples of benchmarking results for this machine include radix sort [15], pointer chasing (and an overall evaluation) [16], and approaches to handling sparsity [17].

This paper is organized as follows. Section II reviews the Firehose benchmark and introduces the second generation migrating thread platform Pathfinder-S used in this study. Section III discusses the parallel Deluge algorithm used to implement Firehose. Section IV describes the experimental setup. Section V evaluates the results. Section VI introduces results from other architectures, including a projection of the Pathfinder architecture to an ASIC implementation. Section VII concludes. It should be noted that sections II-A and III are largely equivalent to the prior work [3], but the port to the new machine, its evaluation, and the projection to a possible future implementation are all new.

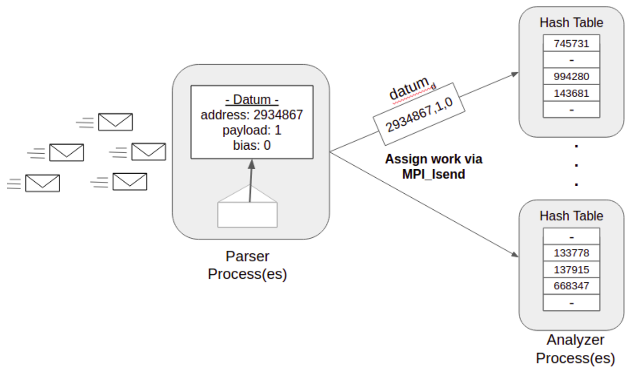

Fig. 1. Firehose PHISH Python/C++ Design: Parser converts datums (ASCII strings) into address, payload, and bias flag triplets. The datum is assigned to an analyzer process via UDP packets. Analyzers look up the address in their local hash table update counters and check for anomalies as necessary.

II. Background

A. Firehose Streaming Benchmark

Firehose [18] resembles a cyber-security like streaming function where incoming IP 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.

Using an active set key generator [18], from an existing computational capability standpoint variant 2 possesses a potentially infinite key range (2^{64} possible keys). 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 either 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. Lucata Pathfinder-S Migrating Thread Architecture

The Lucata architecture [14] follows the migratory memory side processing principle. It is a parallel architecture based on a partitioned global address space, where threads can be spawned remotely across the system in addition to being able to migrate automatically when attempting to access memory addresses which are not local to the node on which they are currently executing. A thread runs in a multi-threaded "**GC core**" until it makes a memory reference that is not contained in that node's memory. The hardware then puts the thread to sleep, packages it, and moves it over the network to the correct node, 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. This is possible thanks to the PGAS address space which is accessible by any thread form any location within the system at all times.

The first version of the Lucata architecture was the Lucata Chick [19]. Fundamental to the Chick's design was the concept of "nodelets" as the basic unit. A **nodelet** is a memory module, its controller, and a single multi-threaded core. Each of the 8 nodes in the Chick contained 8 nodelets, therefore an 8 node Chick would have a total of 64 nodelets (cores). A node's logic is implemented in an FPGA, and as a result core clock speed is rather low 175MHz. A dual core POWER microprocessor on each node board runs Linux, manages a local SSD, and launches migrating threads into the system. Each node had 6 links to other nodes in a hypercube topology.

While Pathfinder-S builds upon the concepts of the Chick system, as well as the intermediate Pathfinder-A which provided further insight [20], the Pathfinder-S does away with the nodelet concept in favor of increased core count per node. The current prototype used in this study is housed at Georgia Tech's CRNCH center[3]. Fig. 2 illustrates to the Pathfinder-S architecture for a single chasis consisting of 8 node boards. The design methodology is nearly identical to that of the Chick. FGPAs are still used for the hardware implementation, but instead of Intel Arria FPGAs, Pathfinder-S uses Intel Stratix chips. The increased size of the Stratix enabled the use of 24 cores per node at a faster 220MHz. However each core still maintains a single instruction issue pipeline.

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

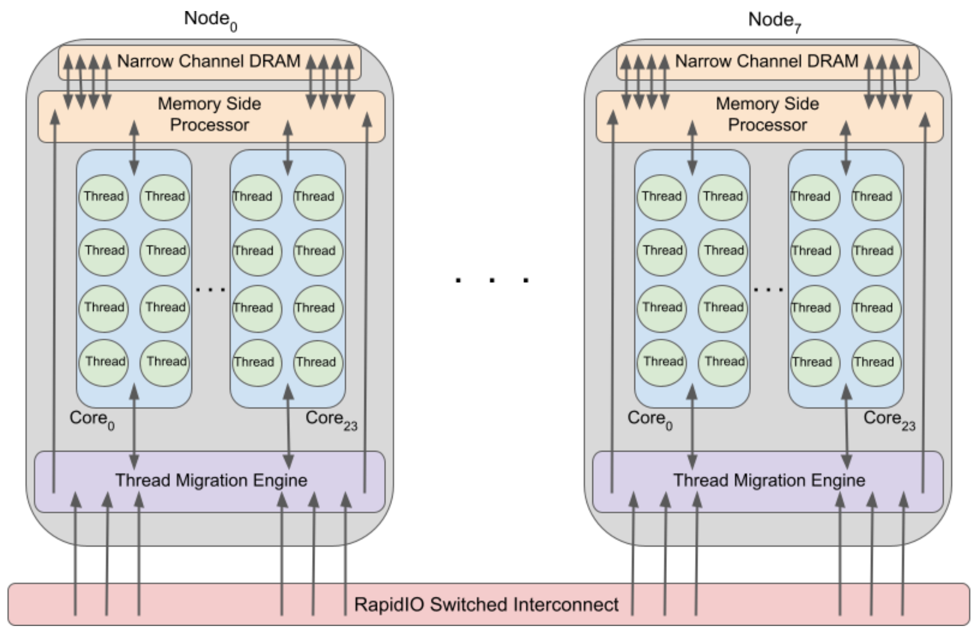

Fig. 2. A single chassis in the Lucata Pathfinder-S system. There are 8 nodes within a single chassis

	CRC	Skybridge	Chick	Pathfinder-S	ASIC	Path/Chick	ASIC/Path
Cores/node	24	16	8	24	24	3X	1X
Mem. Channels/node	8	4	8	8	32	1X	4X
Links/node	1	2	6	6	6	1X	1X
Core Clock (GHz)	2.3	2.6	0.175	0.22	2.0	1.26X	9X
C: Cycles/node (G/s)	55.2	41.6	1.4	5.3	48	3.78X	1.5X
M: Mem. B/W/node (GB/s)	170.6	51.2	12.8	34.1	1638	2.67X	48X
A: Mem. Accesses/node (GB/s)	21.3	12.8	1.6	4.3	25.6	1.33X	6X
L: Link B/W/node (GB/s)	3.4	3.4	2.5	2.5	16	1X	6.4X
p: packet size/datum (B)		1236	1605	615	*615*		**26**
c: cycles/datum		15127	899	1298	*899*		**53**
m: mem. bytes/datum (B)		18618	8216	8389	*8216*		**199**
a: Mem. Accesses/datum		4655	1027	1049	*1027*		**25**
D: Datums/s /node (M/s)		2.75	1.56	4.07	*25*		*25*

TABLE I
SYSTEM COMPARISONS WITH ONE NODE FOR GENERATOR AND ONE NODE FOR HASH TABLE. THE LAST TWO COLUMNS ARE RATIOS OF RESOURCES. THE RED NUMBERS ARE THE SMALLER OF THE CHICK AND PATHFINDER VALUES. THE BLUE NUMBERS ARE THE EXTRAPOLATED DATUMS/S USING JUST THAT RESOURCE. THE GREEN IS THE SMALLEST OF THE BLUE NUMBERS.

Table I compares Pathfinder-S's characteristics to that of the Chick as well as two conventional systems and projections to a future ASIC implementation, all discussed later. For performance comparison we list the total compute cycles (core count times core clock) for each system, along with the improvement ratio between the Pathfinder-S and the Chick. Given the 3X increase in cores, the Pathfinder-S is almost 3.8X more capable than the Chick. Additionally, each Pathfinder memory channel returns 16 bytes for each access compared to the Chicks 8 byte access. This leads to an increase in memory bandwidth per node of 2.67X over that of the Chick hardware.

Programming uses a version of the Cilk language [21]:

1) **Parallelism:** basic parallelism is obtained using *cilk spawn* for local thread spawns, as well as *cilk spawn at(ptr)* and *cilk migrate hint(ptr)* for remote spawns. Loop level parallelism is introduced through the use of cilk for. Explicit synchronizations are performed using *cilk sync*.
2) **Atomic operations:** remote atomic operations are available for 64-bit integers, and do not require a migration to the node containing the variable. They are used when the result is not immediately needed. Versions returning the result (hence incurring a migration) are also available. The hardware also supports an atomic compare-and-swap on 64-bit integers.
3) **Data distribution:** data can be distributed using different patterns. The most commonly used are the roundrobin allocation of one element or one structure at a time. More dynamic allocations allocate a new chunk of memory collocated to a pointer.
4) **Helper libraries:** helper libraries provide a blocked distribution of the data, as well as efficient functions to spawn threads that will work on local chunks.

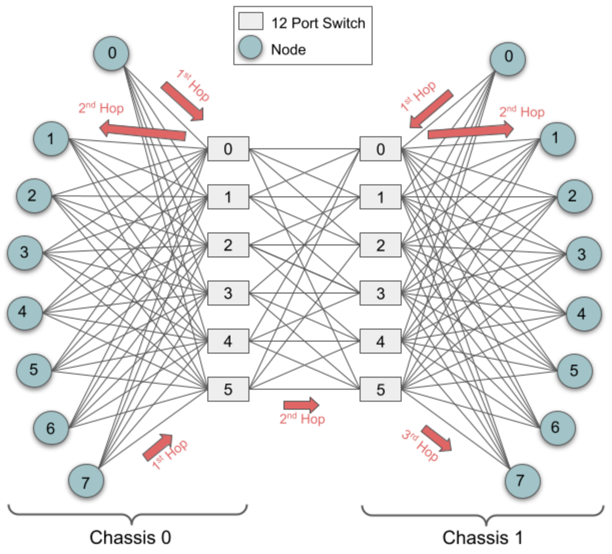

Fig. 3. Pathfinder Topology

C. Pathfinder-S Interconnect Topology

Migrating thread architectures utilize a PGAS address space in which memory is logically shared but physically distributed. When a thread accesses any arbitrary memory address it does so as though it was a read from local memory, even if the thread is currently executing on a node that does not control the address being accessed. As mentioned in the previous section the thread context is packaged up and sent to the node governing the desired memory address such that when execution continues a true local memory access takes place. Thread migrations of any kind require communication much like messages via MPI on conventional distributed systems with the key difference the migrating thread hardware performs this functionality very efficiently at the hardware level.

In order to facilitate efficient transfer of thread contexts for intra-chassis as well as inter-chassis communication the network interconnect must be robust enough to handle high traffic volume at low latency. In Fig. 3 the network topology for a 2 chassis Pathfinder-S is shown. Each node board has 6 ports each of which connects to one of 6 12-port switches. In doing so, within a chassis, there are 6 disjoint star topologies each providing a route to every other node within the chassis, for a total of 6 2-hop routes between any arbitrary node pairing. This provides very high available bandwidth for inter-node communication within a single chassis.

As mentioned 12-port switches are currently used for each chassis, with 8 ports on each switch delegated to local node connections. The remaining 4 ports are then used for connections to additional chassis in a multi-chassis system such as the we evaluated in this study. As can be seen in Fig. 3 the reduced number of connections between chassis switches means that a full mesh topology at the chassis level is not present. This limits the number of routes between among an arbitrary cross-chassis node pairing to 4 3-hop routes. The reduction of routes effectively reduces the available aggregate bandwidth between chassis to $2/3$ that present within a chassis. Additionally the link between chassis switches adds an extra layer of latency that is not present for inter-node communications.

Despite this the network topology currently implemented for the Pathfinder-S is an improvement over that of the Lucata Chick, in which an incomplete hypercube, in which the center diagonals were missing, since it created the necessity for 2-hop routes for some inter-chassis node pairings.

III. DELUGE

Deluge [3] 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 is split into producer (generator) and consumer (analyzer) nodes. Actor threads are spawned on producer nodes and begin generating datums using the active set generator as done in the official Firehose benchmark spec. An actor will hash the address in the datum it generates to determine the consumer node which governs the hash table portion the address should be checked against.

At this point the Actor's thread context is packaged up and subsequently migrated to the appropriate consumer node. Once on the destination node the actor is rebuilt and scheduled on one of the node's cores where it then proceeds with analysis of the datum. Being shared memory, all Actors currently executing on a consumer node 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 the benchmark spec in which a least recently used (LRU) list maintains the order of key occurrence 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 is packaged up for its return trip to the producer on which it was spawned, is migrated, rebuilt and scheduled on an arbitrary core in preparation for generator a new datum. Unlike today's conventional MPI-based model, the recognition of the need to migrate, the packing, and the unpacking is all done by hardware.

This process continues until the desired datum volume is reached. The number of actors spawned on a producer node 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 have migrated or are in the process of ”flooding” consumer(s) at any one time. This constant *”flood”* of threads also insures greater overlap between in-flight threads and computation on consumer nodes.

By using the actor based execution model in this way, only producer nodes function as the permanent home for actor threads. **Consumer nodes have no resident execution.** Because of this a consumer node's hardware resources are dedicated to the analysis of datums only, and do not have to be

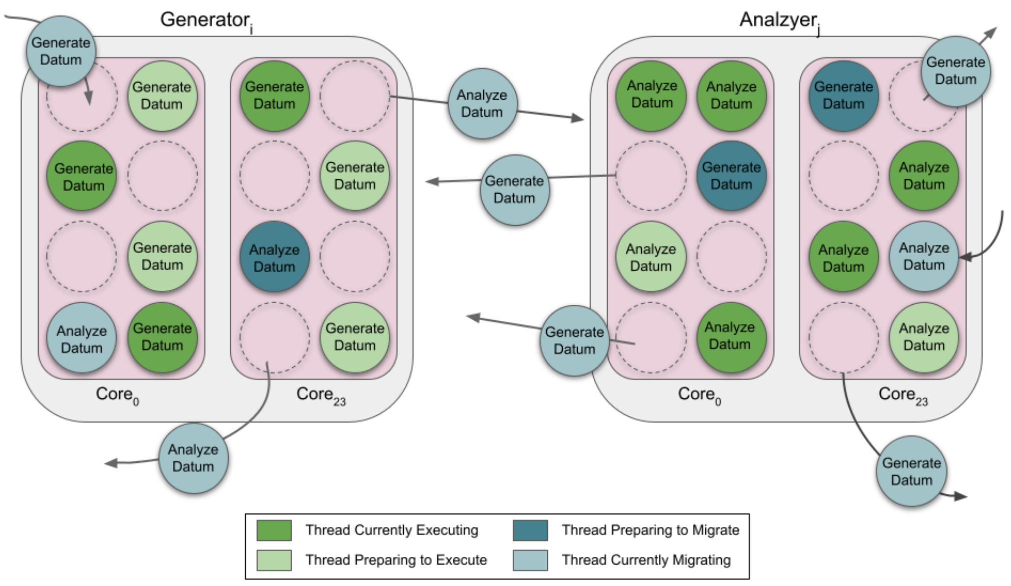
Generator$_i$
Analzyer$_j$
Generate Datum
Analyze Datum
Core$_0$
Core$_{23}$
Thread Currently Executing
Thread Preparing to Execute
Thread Preparing to Migrate
Thread Currently Migrating

Fig. 4. Deluge execution pattern on Pathfinder-S.

shared with generation or parsing of datums as was necessary in previous versions.

IV. Experimental Setup

In this study we utilized the Pathfinder-S available at Georgia Tech's CRNCH facility which consists of 2 chassis of 8 nodes each for a total of 16 nodes (384 cores). For our weak scaling tests, we increase the number of consumer nodes in powers of 2. Additionally we increase the number of producer nodes, and the number of actor threads spawned within each producer, by powers of 2 up to a total of 8192 across all producers. One key note is that for evaluating 16 consumer nodes, due to system size constraints, we were forced to perform datum generation on the same nodes as datum analysis. As such each producer is also a consumer since at 16 consumers there would be no additional nodes in the current system to act as dedicated producer nodes.

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 node. A *cilk_sync* prevents further program execution until all nodes 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.

V. Scaling Results

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 versus a conventional

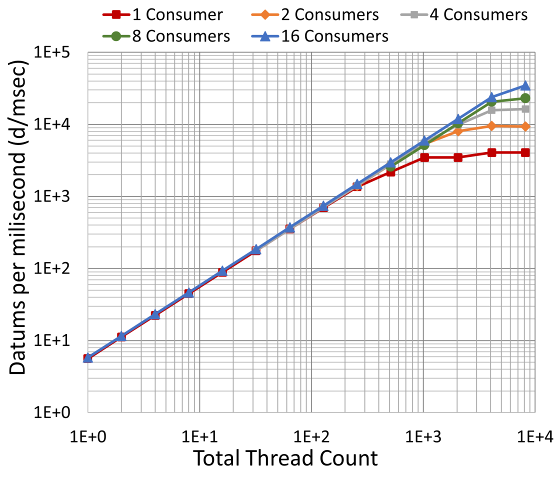

Fig. 5. Observed throughput scalability. Each curve represents the maximum observed throughput for each consumer core count.

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.

Fig. 5 shows the observed throughput in datums per millisecond analyzed (d/msec) as a function total thread count within the system. We conducted tests on a range of consumer node counts and thread counts. As can be seen in Fig. 5 we observed near perfect throughput scaling for all consumer node counts up through 256 total threads at which the single consumer throughput begins to deviate and eventually saturate by 1024 total threads. Throughput for 2 consumer nodes

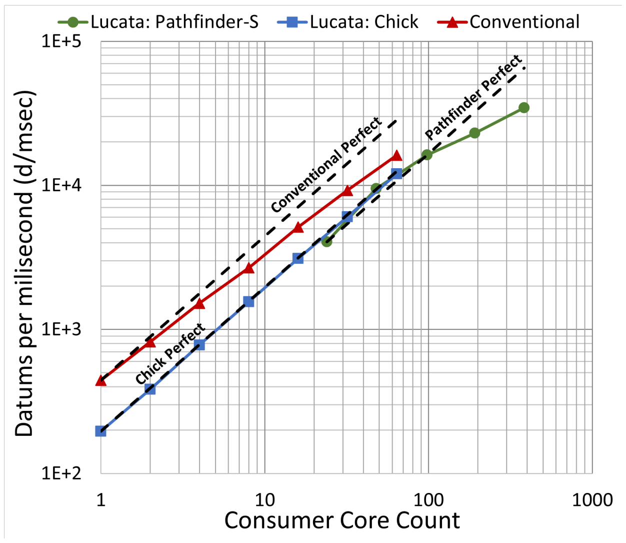

Fig. 6. Observed throughput scalability. Each curve represents the maximum observed throughput for each consumer core count.

deviates from perfect scaling at 1024 threads, followed by 8 and 16 nodes at 8192 threads respectively. Based on the results we observed in our study of Deluge on the Lucata Chick [3] we had expected to see good throughput scalability at higher system sizes just like those visible in Fig. 5.

As stated, for our throughput tests all nodes that which are not currently used as consumer nodes are used as producer nodes on which threads are spawned and generate datums. Therefore we were able to produce enough threads in the system to saturate the consumer nodes in our tests. For 16 consumers this caused produced an issue as the maximum node count on a 2 chassis system is 16, leaving no dedicated producer nodes. In order to evaluate throughput on 16 consumer nodes, we were required to spawn threads on all 16 nodes making each node both a producer and consumer node.

In an earlier study we evaluated an implementation of Firehose variant 2 for the Chick [2] using 2 thread types, again producer and consumer. However those threads were stationary and remained on their original nodelets. Throughput scaling in that study was lackluster due to limited compute capability of individual nodelets having a single core which was then split between both datum generation and datum analysis tasks. Due to the increased core count, clock rate, and memory bandwidth available per node on the Pathfinder-S we were able to achieve superior performance in a shared producer/consumer role environment as observed by the scaling obtained by 16 consumer nodes, with the eventual saturation at higher thread counts. It is important to note that we expect increased throughput is possible if enough system chassis were available to facilitate the use of dedicated producer nodes.

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. In [3] we performed a comparison between Deluge on the Lucata Chick as well as a version designed for a conventional cluster using the Message Passing Interface (MPI). For the conventional implementation we did not run multiple threads within any process, therefore core count is equivalent to process count. Similarly with respect to the 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. Conversely each Pathfinder-S node has 24 cores, on which threads can be arbitrarily scheduled for execution without any method for limiting intra-node core use. This means that the base core count for a Pathfinder single consumer node must be 24 compared to 1 in the case of the Chick and conventional tests.

Fig. 6 shows the observed maximum throughput achieved at each consumer core count up to 384 (the maximum for a 2 chassis Pathfinder-S system). Results from the prior study have been included for comparison. Additionally we show the theoretical perfect scaling for each system, computed as $max(single\_core\_throughput) * consumer\_count$.

As detailed in our earlier study the conventional system experienced good initial throughput scaling but quickly deviates as node count increased. This is most likely due to increased MPI communication overhead, as well as poor cache behavior as a result of the highly irregular memory access pattern inherent to streaming applications. Surprisingly, for the Pathfinder we begin to see similar behavior emerge around consumer 8 nodes, and continue to worsen at 16 nodes. We believe this to be a result of the added communication overhead associated with migrating thread contexts across the chassis boundary. In Section II-C we discussed how inter-chassis communication using the current network topology has 4 routes per node pairing as opposed to the 6 available for intra-chassis traffic. This reduction in available routes effectively limits inter-chassis bandwidth to $2/3$ of that within a single chassis. The Deluge implementation we used here allocates consumer nodes sequentially meaning that for an 8 consumer node test all 8 consumers are on 1 chassis with all 8 nodes of the 2nd chassis serving as producer nodes. As a result virtually **all** thread migrations, and therefore every datum analysis, must be sent between chassis and incur the reduced connection quantity as well as the increased latency of additional switch-to-switch connection.

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. 7. Again the excellent scaling of Deluge is apparent. Throughput on the chick followed the perfect scaling for nearly all system configurations. As predicted in [3] the results for Pathfinder-S appear above and to the right of those observed for the Chick system due largely to the dramatic increase in compute capability per Pathfinder node. Throughput closely follows the perfect scaling line deviating noticeably at the highest core count due to having to share hardware resources between datum generation and analysis.

Additionally at approximately 85 gigacycles, or 8 nodes (192 cores), **throughput on the Pathfinder-S is about an**

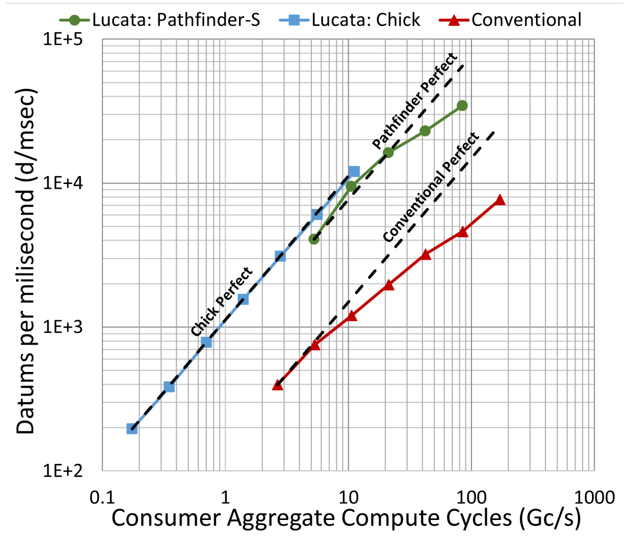

Fig. 7. Observed throughput scalability as a function of compute cycles.

order of magnitude greater than that observed on the conventional cluster. This is a direct result of dramatically improved efficiency per core on Pathfinder over the conventional AMD EPYC 7541 used in the previous study, along with other hardware improvements such as increased memory channel width and memory access rates.

VI. Other Implementations

Other than our previous study in which we evaluated Deluge on the Lucata Chick [3] 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 on a single core basis, but it must be remembered that the multi-process overheads of messaging 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 [6] 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 62K 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 [7]. 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 ASIC

Given the fact that both systems we have run on so far are FPGA-based, an obvious question is what would happen if the same architecture were to be implemented in a modern ASIC technology. Table I has some additional cells to begin an exploration of that idea. The top entries in the “ASIC” column assume an ASIC implementation that looks like modern designs such as the A64FX[4] (the chip used in the top system in the TOP500 list for June 2021 - the Fugaku machine). The A64FX has 48 ARM cores at 1.8GHz and 4 HBM2 memory stacks, with each core having a vector extension, and significant on-chip caches. For our study we assume just 24 cores at 2GHz (simpler in architecture, without either vector extensions or L2 caches), and 6 network links of 4 lanes each at PCIe-5 rates. Our study assumes 4 HBM2e memory stacks.

Table I has 4 rows near the bottom labelled with lower case letters. For the Skybridge, Chick, and Pathfinder-S columns the numbers in these rows represent how different system resources are used on a “per datum” basis. The “p” row is the ratio of the bandwidth on one link (in one direction) divided by the datum rate achieved by the system. Likewise, “c” is the aggregate compute cycles divided by the datum rate, with “m” the aggregate memory bandwidth and “a” the aggregate access rate both divided by the datum rate. Each number represents how much of that resource would be needed if it were used up completely handling the datum traffic.

Clearly not all resources are fully utilized at the same time, so each of these numbers represents an upper bound on how much was actually used. The red numbers in italics under the ASIC column of Table I thus represent the smaller of the two implementation numbers for each resource, and was used to estimate ASIC performance. Each smaller number may still be larger than the actual needed resource amount, but are guaranteed not to exceed that number.

The comparative numbers for these 4 resources between the Chick and Pathfinder-S system give insight into what resources are needed for Firehose on this class of architecture. The one-way link bandwidth of the two systems is identical, but the “l” parameter decreases (even while performance on the Pathfinder-S increased). This decrease implies that network bandwidth is not yet a problem. Whether or not 615 bytes/datum is in fact the amount needed cannot be determined, but the odds are the real number is considerably less (even with a header, the maximum state size of a migrating thread is considerably less).

In terms of available compute cycles, the Pathfinder-S has more faster cores, so has more compute power. However, the “c” ratio actually goes up! This can only be if compute is not the limiting factor, and the number of idle cycles has increased.

The two memory parameters “m” and “a” however are nearly flat from system to system, with a ratio that mirrors almost perfectly the ratio of of datums/s between the two systems. The obvious conclusion is that memory is the resource that bottlenecks first, although from these numbers it is hard to say if that is due to bandwidth or access rate.

[4]https://www.fujitsu.com/global/products/computing/servers/supercomputer/a64fx/

Given these bounds we can thus go backwards from them through the estimates for the ASIC system to come up with a series of estimates for ASIC throughput assuming each resource was the limiting factor. They are computed as the ratio of an uppercase letter with the matching lowercase parameter. The blue numbers in the table are these individual rate estimates; for example $A/a = 25.6/1027 = 25e-3$. The green number at the bottom left is then the minimum of these blue numbers, and represents a minimum datum processing rate that we would expect, given all else was equal.

The estimate of 25M datums/s for the ASIC system is 6.1X that of the Pathfinder-S, and almost 10X that of the distributed system documented in the Firehose website. If our analysis is correct, the bounding resource is memory access rate, followed closely by network bandwidth (which as previously discussed is liable to be better in real life than that assumed here). There is more than twice the available compute cycles and 8X the memory bandwidth, which would be available for enhanced analytics without dragging down expected performance. The extra bandwidth comes primarily from the much longer memory lines read out for each access, where much of the extra data is never used.

VII. Conclusion

In this study we evaluated the new Lucata Pathfinder-S migrating thread architecture via streaming anomaly detection in an unbounded keyspace. **Pathfinder was able to achieve up to approximately 10X throughput over a conventional MPI based cluster and greater than 2X over the Lucata Chick system**. Increases to Pathfinder's core count, clock rate, and memory access rate per node all aid in providing superior performance. 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 systems exhibit many novel architectural features and provided good proof of concept data. The results indicate that further refinement may obtain improved performance at much greater systems sizes. This is evidenced by the emergence of communication overhead impacting overall performance, which had not been observed previously for the migrating thread architecture. More robust interconnects may rectify this issue.

Finally, the results of the projection to an ASIC implementation are significant enough that additional studies are in order to project such architectures to even larger scales such as found in exascale computing. 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)*, Sept. 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] ——, "Deluge: Achieving superior efficiency, throughput, and scalability with actor based streaming on migrating threads," in *High Performance Extreme Computing Conf. (HPEC)*, Sept. 2021.

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

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

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

[7] 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.

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

[9] 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.

[10] 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

[11] 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

[12] ——, "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

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

[14] 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.

[15] M. Minutoli, S. Kuntz, A. Tumeo, and P. M. Kogge, "Implementing radix sort on Emu 1," in *3rd Workshop on Near-Data Processing in conjunction with 48th IEEE/ACM Int. Symp. on Microarchitecture (MICRO-48)*, Dec. 2015.

[16] J. Young, E. Hein, S. Eswar, P. Lavin, J. Li, J. Riedy, R. Vuduc, and T. M. Conte, "A microbenchmark characterization of the Emu Chick," *Parallel Computing*, Sep. 2019.

[17] T. B. Rolinger and C. D. Krieger, "Impact of traditional sparse optimizations on a migratory thread architecture," in *2018 IEEE/ACM 8th Workshop on Irregular Applications: Architectures and Algorithms (IA3)*, 2018, pp. 45–52.

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

[19] 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

[20] G. P. Krawezik, S. K. Kuntz, and P. M. Kogge, "Implementing sparse linear algebra kernels on the lucata pathfinder-a computer," in *2020 IEEE High Performance Extreme Computing Conf. (HPEC)*, 2020, pp. 1–6.

[21] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou, "Cilk: An efficient multithreaded runtime system," in *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPOPP '95. New York, NY, USA: Association for Computing Machinery, 1995, p. 207–216. [Online]. Available: https://doi.org/10.1145/209936.209958