DREDGE: Dynamic Repartitioning during Dynamic Graph Execution

Andrew McCrabb
University of Michigan
mccrabb@umich.edu

Eric Winsor
University of Michigan
rcwnsr@umich.edu

Valeria Bertacco
University of Michigan
vale@umich.edu

ABSTRACT

Graph-based algorithms have gained significant interest in several application domains. Solutions addressing the computational efficiency of such algorithms have mostly relied on many-core architectures. Cleverly laying out input graphs in storage, by placing adjacent vertices in a same storage unit (memory bank or cache unit), enables fast access during graph traversal. Dynamic graphs, however, must be continuously repartitioned to leverage this benefit. Yet software repartitioning solutions rely on costly, cross-vault communication to query and optimize the graph layout between algorithm iterations.

In this work, we propose DREDGE, a novel hardware solution to provide heuristic repartitioning optimizations in the background without extra communication. Our evaluation indicates that we achieve a 1.9x speedup, on average, over several graph algorithms and datasets, executing on a 24x24-core architecture, when compared against a baseline solution that does not repartition the dynamic graph. We estimated that DREDGE incurs only 1.5% area and 2.1% power overheads over an ARM A5 processor core.

CCS CONCEPTS

• **Hardware** → **Hardware accelerators**; *Application specific processors*; • **Mathematics of computing** → *Graph theory*;

1 INTRODUCTION

Computing on vertex-edge graphs has become increasingly popular as graphs can capture many types of object relations: from social networks, to web connectivity, to road maps, graphs are a common underlying structure for organizing large sets of data. An important, emerging class of applications operates on dynamic graphs: graphs where vertices and edges are added, removed, and modified over time. Dynamic graphs are deployed by e-commerce websites when tracking which products have recently been viewed together – such as sunscreen and jerseys before a local sports event or water and toaster pastries before a major hurricane – and by social media companies, proposing targeted advertisements for users based on their friends' status or events they are planning to attend. Similarly, road navigation algorithms rely on dynamic graphs to deliver optimal vehicle routes in the face of accidents and construction work.

DAC '19, June 2–6, 2019, Las Vegas, NV, USA

ACM ISBN 978-1-4503-6725-7/19/06…$15.00
https://doi.org/10.1145/3316781.3317804

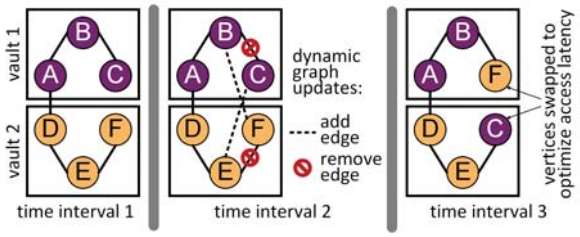

Figure 1: Dynamic graph repartitioning: graph *A-F* is partitioned over two vaults. As its connectivity changes (edges *B-C* and *E-F* are removed, and edges *C-E* and *B-F* are added), vertices *C* and *F* can be swapped to minimize inter-vault message exchanges.

A common trait of the algorithms that compute on graph data structures, whether static or dynamic, is that their performance bottleneck lies in the latency and bandwidth limitations of memory accesses, rather than the computation demands of the algorithms [3]. Indeed, the computation carried out for each vertex of the graph is often light-weight, but triggers updates to other, often several, adjacent vertices. To make matters worse, these adjacent vertices are often not stored nearby in memory. This poor spatial locality leads to low cache hit rates and high latency penalties that cannot be masked by other computation.

Moreover, the graphs entailed by applications like those mentioned above are often very large, and thus stored in multiple discrete storage blocks as, for instance, in distributed systems. Recently, many-core, processing-near-memory systems have been proposed [3] to provide high memory bandwidth and computing power that scales linearly with the size of the system. These systems partition the graph into multiple subgraphs and assign each partition to a distinct storage block in a 3D-stacked memory, also called a "vault". Each vault includes a compute unit and memory controller. It has local access to its own local subset of vertices, and can pay a latency penalty to access non-local vertices (vertices stored in other vaults) via a message-passing system. However, the graph could be partitioned to keep adjacent vertices in the same vault, or at least one close-by, to avoid these penalties.

While graph partitioning has been well-studied for static graphs [10] [9], these algorithms do not provide a good solution for graphs whose connectivity changes over time: even when the initial graph is perfectly partitioned, subsequent edge additions and deletions cause spatial locality to degrade over time, with deteriorating performance. As an example, Figure 1 illustrates a repartition on a simple dynamic graph. As edges are added and removed to vertices *C* and *F*, costly inter-vault messages must be issued during execution. Vertices *C* and *F* can move to the partition holding their neighbors after the dynamic changes, avoiding inter-vault messages and boosting performance.

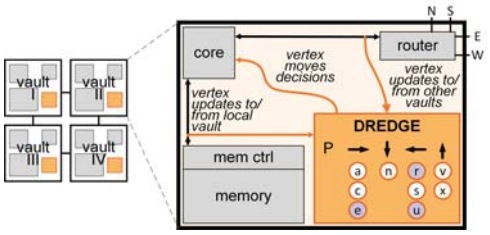

Figure 2: Baseline multi-node architecture, augmented by DREDGE modules. DREDGE can be deployed in a chip multiprocessor or a high-bandwidth HMC, among other systems, by adding our novel hardware unit to each node. DREDGE only requires access to messages in transit within the node and operates in a completely distributed fashion, thus it is seamlessly scalable to many-node systems.

To address this challenge, this work proposes DREDGE, that is, Dynamic REpartitioning during Dynamic Graph Execution, consisting of a small hardware module deployed next to each computation node. It operates in the background to repartition a graph during an application's execution to maintain a balanced, min-cut partitioning of the graph without explicit access to main memory. In contrast with prior solutions in this domain [7], DREDGE does not suspend the application to repartition the graph, nor entails costly queries of the graph edges to determine the best repartitioning moves. In addition, it leverages a small hardware module to accomplish its goal and entails no performance overhead on the graph application.

Contributions. DREDGE makes the following contributions:

• We present a novel heuristic for repartitioning dynamic graphs without querying the graph structure. It relies on monitoring past vertex-update messages to estimate future traffic patterns.

• We present a hardware solution to make repartitioning decisions based on this heuristic. The hardware module lies next to each computing core. Its silicon footprint at 45nm technology is only $0.017mm^2$, equivalent to 1.5% of the area of an ARM A5 core. When deployed in a HMC architecture as in [3], DREDGE incurs no area overhead because of the low silicon logic utilization. We believe this is the first hardware solution for repartitioning dynamic graphs.

• We evaluate DREDGE on several datasets and algorithms and demonstrate a performance improvement of $1.86x$ over a state-of-the-art dynamic graph repartitioning solution [7]. Moreover, we show that DREDGE's performance benefit increases as we increase the size of the underlying architecture or the amount of computation entailed by the application.

2 DREDGE ARCHITECTURE

Dynamic graphs experience variations in the graph topology over time: edges and vertices may be added or removed at a variable pace, depending on the nature of the dataset they represent. For instance, graphs representing products or movies recommendation systems change slowly (order of hours or days between changes), while graphs modeling messages exchanged between users of a social network may change more often, particularly for large social networks (order of thousands of changes per millisecond). Fast changing graphs present more challenges in preserving a balanced partition and to the algorithms executing on them. Such algorithms are bound to produce approximate results, as the graph is subject to many changes throughout the application's execution.

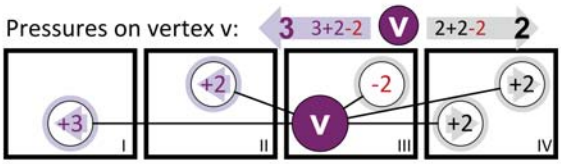

Figure 3: Example of vertex pressure computation based on accesses to v from five adjacent vertices.

The goal of this work is to accelerate graph repartitioning decisions in graphs by monitoring the messages exchanged between processor core, memory block and router within a processing node. To this end, DREDGE tracks vertex accesses, and then decides when to move a vertex to another vault and where. As illustrated in Figure 2, our solution is completely distributed, with one hardware unit per node and no external connection nor communication outside the vault, thus it can scale to a system with any number of vaults (or computing nodes). Note that messages between router and core indicate communication from a remote vertex in another vault, while those between core and local memory reveal access from a locally-stored vertex.

In contrast, other repartitioning schemes rely on examining the mapping of a vertex's neighbors to make repartition decisions. For example, one state-of-the-art solution in repartitioning of dynamic graphs [7] reaches out to the neighbors of vertices affected by recent graph changes to query their location and determines whether to recommend a vertex move. These query messages introduce significant performance overhead, which increases linearly with the rate of change of the graph. On the other hand, all of DREDGE's move decisions are based exclusively on locally-saved "vertex pressure" measurements, a synthetic measure of the net pull of a vertex to other partitions. DREDGE tracks the vertex pressure measure locally for a portion of the vertices in each partition; if and when a vertex's pressure rises above a threshold, the vertex is moved to a neighboring partition in the direction that would lower the pressure metric.

Vertex pressure is a measure we devised to keep track of data access for a given vertex from local and remote nodes. If a vertex is accessed by a neighbor node, that vertex experiences pressure to move towards that node and its corresponding partition. Inversely, if a vertex has neighboring vertices within its own partition, it experiences pressure to stay in place, an aspect we incorporate by reducing the all pressures to move elsewhere. Note that, in general, a vertex will experience pressure from multiple directions simultaneously: in our metrics we measure all these pressure vectors by tracking the vertex ID and direction for each vertex-update message incoming to the local vault. Specifically, the vertex pressure for a vertex v toward a given direction, noted as $pres(v)$, is updated whenever v is accessed by another vertex through a message coming from that direction. Vertex pressure is computed with the following function:

$$pres(v1) \begin{cases} += d(v1, v2) + 1 & part(v1) \neq part(v2) \\ -= 2 & part(v1) = part(v2) \end{cases}$$

For update messages coming from remote nodes, we increase the pressure by 1 above the distance d from the node, to take into account the baseline cost of transferring data out of the local node. In addition, the pressure is decreased by 2 for local updates, so to

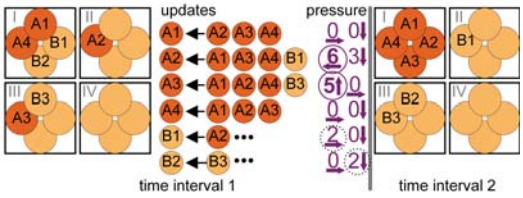

Figure 4: Graph repartitioning example. Vertices A2 and A3 are tagged for moving because their pressure towards vault I is above threshold. To keep partitions balanced, vertices B1 and B2 must move in the opposite direction.

avoid instability situations where a vertex keeps moving back and forth between adjacent vaults. Computing these simple functions is trivial and is entirely masked by other computation in the vault.

Figure 3 provides an example for our metric. Five vertices access vertex **v**: two from the east, two west and one local. The two east neighbors are one vault away, causing a pressure of 4 to the east (2 each). The local neighbor reduces overall pressure by 2, resulting in a net pressure of moving east by 2. From the west, vertices are one and two vaults away, causing a total pressure of 5 (2 and 3, respectively). The local neighbor reduces the pressure from the west by 2 as well, resulting in a final pressure to move west of 3.

Graph repartitioning. Vertex pressure metrics are maintained in the background, in storage within the DREDGE hardware unit, while graph-based applications compute in the foreground. Because DREDGE storage is finite, vertex pressure is tracked only for a portion of the vertices, the first n vertices referenced in the messages DREDGE observes when it begins to monitor activity. At the end of each computation interval, some vertices are moved to other vaults, making space for others to be tracked by DREDGE. Those new vertices to be monitored are also identified by grabbing the destination of the next messages observed. If a vertex pressure falls to ≤ 0, then the vertex is deemed to best stay in the current vault, and thus removed from further analysis by DREDGE. That same vertex could be considered again later if external vertex-update messages reference it. When the pressure value for a vertex is above a preset threshold at the end of a time interval, the vertex is migrated to the adjacent partition in the direction of that pressure. Whenever a vertex moves to another vault, the other vault must move a vertex in the opposite direction to keep partitions balanced. This vertex is selected as the one in the neighbor vault with the highest pressure to move in the relevant direction.

In our experiments we set the threshold value heuristically: we noted that low threshold values cause vertices to move back and forth between adjacent transitions, while high values dampen the repartitioning of the graph. An example of the repartitioning process is illustrated in Figure 4, which assumes an interconnect adopting X-Y routing on a mesh topology. The example considers a system with 4 vaults executing an algorithm on a graph with 16 vertices. The example illustrates the analysis carried out for all the *Ax* vertices: in the middle of the diagram we show all the vertex update message sources observed during the first interval of execution. For instance, *A*1 receives update messages from *A*2, *A*3 and *A*4. The pressures for *A*1 are computed as follows: $p \rightarrow = 2 - 2 = 0$ (*A*2 pulls east, while *A*4 compensates back). and $p \downarrow = 2 - 2 = 0$. Pressures for the other *Ax* are reported in the example. At the end of the

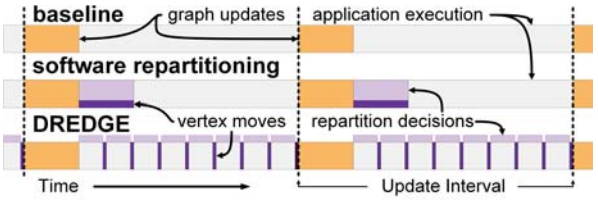

Figure 5: Execution flow for DREDGE, a state-of-the-art software-based solution for dynamic graphs, and the baseline, a static graph framework with no repartitioning.

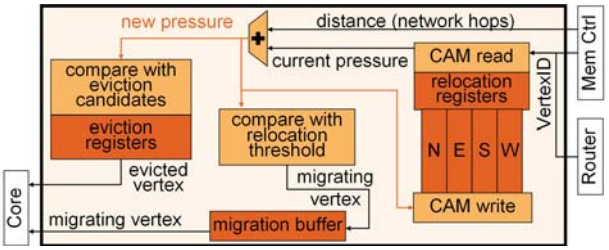

Figure 6: Microarchitecture of the DREDGE hardware unit, which includes relocation registers to track the pressure of some vertices within the vault, a migration buffer to set up vertex transfers to other vaults, and eviction registers to track candidate vertices to send when a new vertex arrives from another vault. Two comparators support updates to this storage.

interval, because the pressure for *A*2 to move west and for *A*3 to move north are above the threshold (set to 4 in our example), those two vertices are marked for moving to vault I. In order to keep the partition balanced, two vertices must leave vault I to vault II (east) and III (south): among the vertices available, *B*1 and *B*2 are selected because they have the highest pressure towards those directions. The right side of the Figure shows the partitions after these moves.

Application execution and graph updates. As mentioned above, DREDGE computes pressure measures and makes repartition decisions concurrently with the graph application, since it leverages its own dedicated hardware unit. The execution model we consider is a classic approach to dynamic graphs computations: the application runs for a fixed interval, then all graph modifications are processed, and finally the application resumes, now operating on the modified graph. As illustrated in the top part of Figure 5: a baseline graph computation framework does not consider any repartitioning and simply alternates between execution and graph updates. The general model is preserved in our solution, with the addition of brief vertex-move activities interleaved with the application execution, illustrated at the bottom of Figure 5. A state-of-the-art software solution for repartitioning [7] dedicates a significant portion of time to calculating repartitions after each graph update and leaves less time for computation, as illustrated in the middle of Figure 5,

3 HARDWARE IMPLEMENTATION

DREDGE uses a dedicated hardware unit to track vertex pressures and make relocation decisions concurrently with the application's execution. Figure 6 provides a schematic of the unit's design. Below we discuss each major component. When a vertex-update message is snooped from the router or memory bus, the relocation registers

Table 1: Graph datasets for our evaluation

Name	Description	Type	Avg Vertices	$\frac{Changes}{1M\ clk\ cycles}$
amazon	product co-purchasing	bipartite	98,000	6,500
lj	livejournal network	power law	20,500	1,200
friendster	gaming social network	power law	127,000	5,000
roadnet	CA road network	uniform	89,000	6,900

are consulted to retrieve the destination vertex's pressure. If the vertex is present, its pressure in the relevant direction is updated and then compared against a preset threshold, and also against the corresponding pressure of the migration buffer's entries, then migration and eviction registers are updated as needed.

Relocation registers (RRs) are used to store vertex pressures in each direction. As our experimental evaluation assumes a mesh architecture, we require 4 sets of registers to be tracked, one for each router's I/O port. Note however, that it would be straightforward to adapt this design to other topologies, where routers have a different number of I/O ports. The RRs are stored in CAMs addressed by vertexID, unique to each direction. The first time a vertex is observed in a core-router message, an RR for the appropriate direction is allocated if available. Whenever a subsequent vertex-update message from core or router is observed to that same vertex, the RRs are updated according to our vertex pressure function. Whenever a vertex is removed from consideration because it has moved to a different node or its pressure has returned to zero, its corresponding RR entry can be reallocated to another vertex. Note that, while only a portion of the vertices are being tracked at any given time, any single vertex is unlikely to use an RR entry for long, as they soon graduate to a determination for their location.

A **migration buffer** is used to store vertexID and direction for vertices whose relocation direction has been determined. They wait for the next window when all pending vertex movements are carried out. At that time, the core will read all the vertex movements and complete them sequentially.

Finally, the **eviction buffer** stores a short list of the best known candidate vertices for transfer in each router's direction. Each time a vertex is transferred to a neighbor vault, another vertex must be exchanged in the opposite direction to preserve a balanced partitioning. This buffer's purpose is to maintain a list of transfer candidates: we store the vertex with the highest pressure, including its ID and pressure values. During each exchange, a vertex is removed from here and transferred to the vault of the incoming vertex. If the eviction buffer is empty for a direction, a signal is sent to the core to evict a random vertex. This rare situation only occurs when neighboring vaults are significantly more active in repartitioning.

4 EXPERIMENTAL EVALUATION

To evaluate our solution, we augmented the BookSim network simulator [8] to include structures for modeling the vaults, simulating accesses to local memory, and creating inter-vault messages based on updates to non-local vertices. We assume that an in-stride access requires 20 clock cycles and a random access requires 200 cycles. We did not model the cores because graph applications are far from being compute-bound. We implemented both DREDGE and the software-based repartitioning solution [7] outlined in Section 1 in this simulator. We then generated four dynamic datasets from four real-world graphs summarized in Table 1. A random subset of edges are activated in the static graph every million cycles. An edge lives in the graph for a preset lifetime (25 million cycles for LiveJournal and Friendster and 50 million cycles for Amazon and Roadnet) before it is removed. An existing edge resets its lifetime if reactivated before removal. A vertex is added once there is a living edge connecting it to another vertex and it is removed when its last incoming edge dies off.

LiveJournal [4] and Friendster [20] are social networks, where a dynamic graph can represent interactions between users. CA RoadNet [12] is the California road network, where a dynamic graph can represent traffic between intersections. Amazon [11] is a co-purchasing network of a subset of products on Amazon, where an edge represents two products that are often bought together.

We compare DREDGE against two other solutions: repartitioning using an unoptimized baseline with no repartitioning (Baseline) and an optimized baseline, [7] (Leopard) which uses software-based repartitioning. A new set of changes to the graph is available every one million clock cycles, beginning as soon as the current algorithm iteration finishes. When simulating without repartitioning, computation resumes immediately after the graph is updated. When simulating the software repartitioning solution [7], a repartitioning phase is executed immediately after the graph changes before computation. When simulating DREDGE, repartitioning occurs at the beginning of every algorithm iteration, based on decisions computed in the background. In addition, we used hash-based streaming partitioning in our experiments when vertices are added to the graph to reduce the time taken to update the graph. While a more complex streaming partitioning scheme would yield us a lower immediate hopcount, a hash-based scheme allows for a more direct comparison between DREDGE, [7], and the unoptimized baseline.

To compare partitioning quality, we measure hopcount: the sum of the distances (in network hops) of all edges in the graph. For example, the graph shown in Figure 3 has a hopcount of five. Hopcount is a synthetic metric to quantify spatial locality in the graph partitioning. Figure 7 reports our findings on total graph hopcount for the datasets we considered while executing PageRank. The plots include four traces for each solution: each trace corresponds to a different system size, ranging from 8x8 nodes, up to 32x32 nodes. The software repartitioning solution and the baseline performs within 1% of each other, and are represented by the same traces.

DREDGE produces a lower total hopcount than both the other solutions in all cases, often an order of magnitude difference. The Leopard solution [7] waits until a vertex is affected by a graph change (adjacent edges being added or removed) multiple times before calculating the optimal partition, so to avoid repartitioning a large subset of the total graph each time. Since these datasets are highly dynamic and vertices are removed quickly, many vertices are either never repartitioned before being removed or are removed soon after being optimized, providing little benefit. As a result, software repartitioning provides approximately the same hopcount than the unoptimized repartitioning. In contrast, DREDGE is capable of moving vertices as soon as they appear in the system, especially if their neighbors are far away. Also, opportunities for movement occur much more frequently (after an application iteration, instead of only after a set of graph changes). As a result, vertices are moved faster and the layout of the graph converges more quickly to a stable state, achieving lower hopcount.

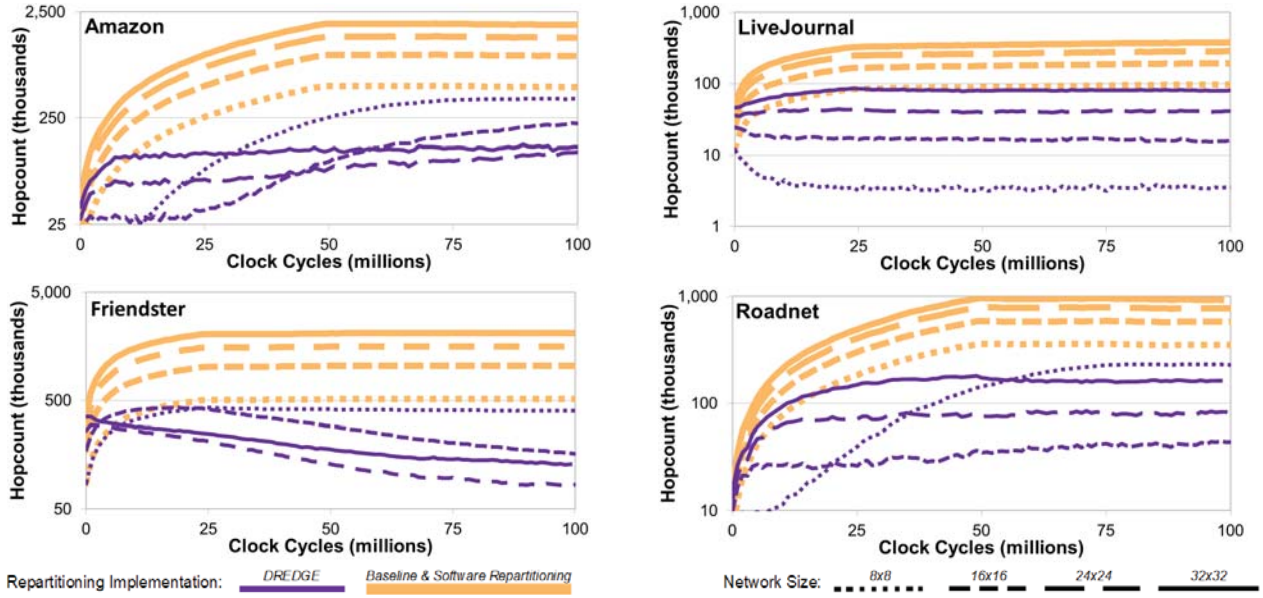

Figure 7: Aggregated hopcount for all edges over time

System Size	Amazon	Friendster	LiveJournal	Roadnet	Average
8x8	1.28x	1.12x	0.81x	0.96x	**1.04x**
16x16	1.68x	1.55x	1.44x	1.21x	**1.47x**
24x24	1.90x	2.20x	2.05x	1.38x	**1.88x**
32x32	1.80x	2.42x	1.86x	1.33x	**1.85x**

Figure 8: Performance improvement of DREDGE over the baseline solution of iterations completed within one million clock cycles for varying systems sizes (64 to 1,024 nodes).

Algorithm	Edges Accessed	Speedup
PageRank	100%	2.46x
Centrality	90%	1.98x
Compression	50%	2.50x
Random Walk (15%)	15%	1.41x
Random Walk (5%)	5%	0.48x

Figure 9: Performance at 100M clock cycles after running the Friendster dataset in a 32x32 system.

Though counter-intuitive, DREDGE may obtain a lower hopcount for larger hardware systems with more nodes than for smaller ones. A larger network allows for pressure to rise faster and for more iterations to finish in the same amount of time, increasing the opportunities for repartitioning. This shows an important feature: DREDGE performs better with larger compute systems.

To measure performance, we compared the total number of algorithm iterations completed between graph updates (one million clock cycles in our experiments). Because the update interval is fixed, this metric takes into account both the partitioning quality and the time taken to repartition. The results of this analysis are reported in Figure 8, where we report our findings over a range of system sizes, from 64 to 1,024 nodes. The values reported are the performance ratios between DREDGE and the baseline solution. Note again, the performance improvement provided by DREDGE trends up as the system scales to larger sizes.

To evaluate DREDGE's performance over a range of applications, we executed five different popular graph algorithms on the Friendster dataset, simulating a 32x32 node system. This study evaluates DREDGE in presence of a wide range of access patterns. Each application traverses a different number of edges per iteration. DREDGE's performance deteriorates significantly for Random Walk 5%, as this application entails a minimal level of activity with very few messages exchanged, giving DREDGE very few opportunities to improve the graph partitioning.

In general, as graph-based applications are usually memory-bound, memory access patterns have a key impact on performance. Because DREDGE uses the messages sent during algorithm execution to inform the vertex movement, there is a greater speedup for algorithms that traverse many edges, like PageRank. DREDGE performance is worse than the baseline solution for algorithms that exhibit very low parallelism and experience very few edge traversals. We argue that, if parallelization and execution time are not critical for the application, repartitioning is also not critical.

We also analyzed the impact of varying the number of graph changes per update interval. Because DREDGE's repartitioning decisions are based on interconnect packets during meaningful computation, not the number of changes, performance is not directly affected by the rate of change in the graph. Instead, the rate of change impacts how quickly the partition quality (hopcount) degrades, affecting all repartitioning schemes. As DREDGE maintains the same speedup for all rates of change, we observe that DREDGE's rate of spatial locality improvement is proportional to the amount of repartitioning work necessary: DREDGE works faster for poorly-partitioned graphs, slower for well-partitioned graphs, and more effectively than state-of-the-art baselines for all rates of change.

Area and Power overheads. We synthesized DREDGE using Synopsys' Design Compiler and the IBM 45nm technology library based on the microarchitecture presented in Figure 6. We found that the DREDGE unit requires 4.84 mW of power at peak usage. Based on prior work [16], we scaled the power usage of DREDGE to a

Changes per Update Interval	725	750	800	1,200	4,800
Speedup	2.01x	2.04x	1.99x	2.12x	2.24x

Figure 10: Performance improvements of DREDGE over a varying intensity of graph changes for PageRank running on the LiveJournal dataset in a 24x24 node system after 100M clock cycles.

32nm technology node, and found its power demands to be 3.14 mW, corresponding to a 2.1% overhead over an ARM Cortex A5-like core (150.4 mW) in a 3D-stacked memory system [14]. DREDGE also requires an additional $0.0077mm^2$ of silicon area when scaled to the 32nm node. corresponding to a 1.5% area overhead over the cores used in Tesseract [14].

5 RELATED WORK

Dynamic Graph Partitioning There are three standard ways to repartition graphs: scratch-map, streaming, and incremental. Scratch-map partitioning periodically applies a static graph partitioning heuristic to the whole graph to achieve close-to-optimal repartitioning. While state-of-the-art solutions like ParMETIS [10] and PT-Scotch [5] leverage highly-parallel systems to create well-partitioned results, the computation overhead is too large to continuously repartition large graphs.

Several streaming graphs like Fennel [17] and Linear Deterministic Greedy [15] are used to partition vertices as they arrive. For an incoming vertex, they prioritize locations that house immediate neighbors and penalize locations that already have many vertices. These were originally designed to partition a large, static graph, working while the graph is loaded into a system, but can be used for dynamic graphs as well.

Continuous partitioning applies a repartitioning stage at the end of a graph update, relocating vertices before meaningful computation resumes. [18] determines the location of all neighbors for each vertex and randomly selects a subset of vertices to move to better partitions. [1] proposed only examining and moving vertices based on the most recent changes to the graph, so that only a subset of vertices are examined. Leopard [7] also proposed tracking and examining only those vertices affected by recent changes to the graph, using a more efficient heuristic.

Hardware for Graph Analytics To the best of our knowledge, we are the first to use specialized hardware for graph partitioning. Other graph accelerators have been presented using specialized additional storage in the core to improve the locality visible to the core [19] [21] [13] [2]. Graphicionado [6] uses a specialized execution pipeline with optimized communication to the memory system. Tesseract [3] demonstrated benefits using high-bandwidth memory for graph processing. They show performance improvements for well-partitioned graphs, but do not address dynamic graphs.

6 CONCLUSION

In this paper, we present DREDGE, a novel heuristic to repartition dynamic graphs in a many-core system with private memory vaults, such as Hybrid Memory Cubes. This heuristic leverages information gathered during an application's execution to make repartitioning decisions, avoiding costly inter-vault communication required by prior work. We also present a hardware implementation of this heuristic which shows $1.9x$ speedup to application execution without modifying the core or the algorithms and only 1.5% area and 2.1% power overheads.

Acknowledgments This work was supported by the Applications Driving Architectures (ADA) Research Center, a JUMP Center co-sponsored by SRC and DARPA.

REFERENCES

[1] Amirreza Abdolrashidi and Lakshmish Ramaswamy. 2016. Continual and Cost-Effective Partitioning of Dynamic Graphs for Optimizing Big Graph Processing Systems. In *Proc. Big Data Congress.*
[2] Abraham Addisie, Hiwot Kassa, Opeoluwa Matthews, and Valeria Bertacco. 2018. Heterogeneous Memory Subsystem for Natural Graph Analytics. In *Proc. IISWC.*
[3] Junwhan Ahn, Sungpack Hong, Sungjoo Yoo, Onur Mutlu, and Kiyoung Choi. 2016. A scalable processing-in-memory accelerator for parallel graph processing. *SIGARCH Computer Architecture News* 43, 3.
[4] Lars Backstrom, Dan Huttenlocher, Jon Kleinberg, and Xiangyang Lan. 2006. Group formation in large social networks: membership, growth, and evolution. In *Proc. SIGKDD.*
[5] Cédric Chevalier and François Pellegrini. 2008. PT-Scotch: A tool for efficient parallel graph ordering. *Parallel Computing* 34, 6-8.
[6] Tae Jun Ham, Lisa Wu, Narayanan Sundaram, Nadathur Satish, and Margaret Martonosi. 2016. Graphicionado: A high-performance and energy-efficient accelerator for graph analytics. *Proc. MICRO.*
[7] Jiewen Huang and Daniel J Abadi. 2016. Leopard: Lightweight edge-oriented partitioning and replication for dynamic graphs. *Proc. VLDB.*
[8] Nan Jiang, Daniel U Becker, George Michelogiannakis, James Balfour, Brian Towles, David E Shaw, John Kim, and William J Dally. 2013. A detailed and flexible cycle-accurate network-on-chip simulator. In *Proc. ISPASS.*
[9] George Karypis, Rajat Aggarwal, Vipin Kumar, and Shashi Shekhar. 1999. Multilevel hypergraph partitioning: applications in VLSI domain. *IEEE Trans. VLSI Systems* 7, 1.
[10] George Karypis and Vipin Kumar. 1998. A fast and high quality multilevel scheme for partitioning irregular graphs. *SISC* 20, 1.
[11] Jure Leskovec, Lada A Adamic, and Bernardo A Huberman. 2007. The dynamics of viral marketing. *ACM Trans. on the Web* 1, 1.
[12] Jure Leskovec, Kevin J Lang, Anirban Dasgupta, and Michael W Mahoney. 2009. Community structure in large networks: Natural cluster sizes and the absence of large well-defined clusters. *Internet Mathematics* 6, 1.
[13] Muhammet Ozdal, Serif Yesil, Taemin Kim, Andrey Ayupov, John Greth, Steven Burns, and Ozcan Ozturk. 2016. Energy efficient architecture for graph analytics accelerators. *Proc. ISCA.*
[14] Seth Pugsley, Jeffrey Jestes, Huihui Zhang, Rajeev Balasubramonian, Vijayalakshmi Srinivasan, Alper Buyuktosunoglu, Al Davis, and Feifei Li. 2014. NDC: Analyzing the impact of 3D-stacked memory+ logic devices on MapReduce workloads. In *Proc. ISPASS.*
[15] Isabelle Stanton and Gabriel Kliot. 2012. Streaming graph partitioning for large distributed graphs. *Proc. SIGKDD.*
[16] Aaron Stillmaker and Bevan Baas. 2017. Scaling equations for the accurate prediction of CMOS device performance from 180nm to 7nm. *Integration, the VLSI Journal* 58.
[17] Charalampos Tsourakakis, Christos Gkantsidis, Bozidar Radunovic, and Milan Vojnovic. 2014. Fennel: Streaming graph partitioning for massive scale graphs. In *Proc. WSDM.*
[18] Luis Vaquero, Félix Cuadrado, Dionysios Logothetis, and Claudio Martella. 2013. xDGP: A dynamic graph processing system with adaptive partitioning. *arXiv:1309.1049.*
[19] Chongchong Xu, Chao Wang, Lei Gong, Lihui Jin, Xi Li, and Xuehai Zhou. 2018. Domino: Graph Processing Services on Energy-Efficient Hardware Accelerator. In *Proc. ICWS.*
[20] Jaewon Yang and Jure Leskovec. 2015. Defining and evaluating network communities based on ground-truth. *Knowledge and Information Systems* 42, 1.
[21] Jinhong Zhou, Shaoli Liu, Qi Guo, Xuda Zhou, Tian Zhi, Daofu Liu, Chao Wang, Xuehai Zhou, Yunji Chen, and Tianshi Chen. 2017. Tunao: A high-performance and energy-efficient reconfigurable accelerator for graph processing. In *Proc. CCGrid.*