Abstract—There is growing evidence that current architectures do not well handle cache-unfriendly applications such as sparse math operations, data analytics, and graph algorithms. This is due, in part, to the irregular memory access patterns demonstrated by these applications, and in how remote memory accesses are handled. This paper introduces a new, highly-scalable PGAS memory-centric system architecture where migrating threads travel to the data they access. Scaling both memory capacities and the number of cores can be largely invisible to the programmer.

The first implementation of this architecture, implemented with FPGAs, is discussed in detail. A comparison of key parameters with a variety of today’s systems, of differing architectures, indicates the potential advantages. Early projections of performance against several well-documented kernels translate these advantages into comparative numbers. Future implementations of this architecture may expand the performance advantages by the application of current state of the art silicon technology.

I. INTRODUCTION

Computer architecture has evolved over the last decade from ever faster single-core processors into slower, parallel multi-core designs, mainly as a result of power limitations, and then into many-core designs with a range of different ISAs and internal organizations. Throughout this evolution, “compute-intensive” applications, such as dense linear algebra, have continued to run at 90+% efficiency [1]. This has been achieved by designing architectures that exploit the high temporal and spatial locality of operations and operands inherent in many of these applications. In particular, these designs implement wide channels to memory and deep cache hierarchies to provide sufficient memory bandwidth at low enough latencies to avoid stalls in deeply-pipelined, fast cores.

Unfortunately, this level of success has not held true for all applications. Sparse matrix linear algebra packages [2] achieve only 1% efficiency versus their dense counterparts [3]. Non-numerical benchmarks such as Breadth-First Search [4] scale well, but have had virtually flat performance per node for almost 4 years [5]. “Big Data” analytics applications have emerged that require access to massive amounts of data with no predictable patterns and radically different needs for computational resources [6].

These “data-intensive” applications do not exhibit the same locality traits as compute-intensive applications thus the latency of individual memory accesses overwhelms the advantages of deeply pipelined, fast cores. However, as these applications frequently exhibit “weak-locality,” a form of locality where the irregular accesses occur within a large memory region (i.e., several gigabytes), it is possible to design architectures capable of exploiting this phenomenon.

While multiple solutions have been devised to overcome the memory and performance issues found in data-intensive applications, the majority focus on node-level performance and/or maintain the increasingly strained message-passing model. Multi-threading support, e.g., Pthreads, helps within a single shared memory node, but the inability to easily cross a node boundary without very significant overhead forces a two-paradigm programming model where explicit function calls are needed for non-local actions. PGAS languages such as UPC, Chapel, and X10 blur this boundary, but are still limited by the concept that threads are relatively stationary, and thus operations “here” are different than operations “over there.”

This paper introduces the Emu system architecture, currently being brought up, that is explicitly designed for data-intensive applications exhibiting weak locality. This highly efficient architecture scales to very large systems while providing access to a common global address space via a simple and consistent programming model. Architecturally, the system consists of an interconnected set of nodes, each subdivided into highly multi-threaded nodelets with every nodelet contributing memory to the global address space. Issues associated with irregular and cache-unfriendly memory access behavior are minimized by implementing a highly efficient, fine-grained memory system and migrating threads that move the thread state without explicit programming directives [7] as new memory locations are accessed. This provides a highly efficient “put-only” communication model that clearly reduces thread latency and total network bandwidth load as return trips and cache coherency messages are eliminated.

Our programming model is based on Cilk [8], a language that supports all common threading paradigms and explicitly provides mechanisms for expressing the most irregular cases. To this is added support for atomic operations and libraries for
were beneficial to applications with irregular memory access.

In outline, Section II discusses related work. Section III provides an overview of our programming model and thread structure. Section IV describes the first generation Emu system architecture. Section V provides comparisons to other systems and discusses application performance. Section VI concludes.

II. RELATED WORK

A. Fine-grained Memory Access

The co-processor portion of the Convey systems implemented a memory system with 8 controllers and 16 channels that was cache coherent with the host processor memory. Each memory controller was implemented on an FPGA, and when coupled with a Convey Scatter-Gather DIMM could support single-word memory accesses. Another approach for providing fine-grained memory accesses is the Dynamic Granularity Memory System (DGMS). This hardware-based method provides mechanisms for accessing and caching memory in chunks varying from 8B to 64B. Some modifications to a standard DIMM module are required, but most of the architectural changes are within the CPU to predict the size of a data access from memory.

Both approaches demonstrated that fine-grained accesses were beneficial to applications with irregular memory access patterns, however, both are more complex than the memory system implemented here.

B. Processing-in-Memory

Processing-in-memory (PIM) combines memory and logic on the same die to minimize the effects of the memory wall. The EXECUBE chip was an early PIM implementation targeting large parallel systems, in particular, each chip contained 8 processing elements each having a bank of memory. These processing elements were connected to each other and provided off-chip links to other EXECUBEs to provide direct memory access transfers between processing elements. DIVA was a later PIM based system that combined the PIM-to-PIM communication abilities of the EXECUBE with support for a host processor to memory interface. This provided support for standard general purpose computing models and allowed for the gradual migration of applications from non-PIM to PIM based execution. Intelligent RAM (IRAM) was another PIM implementation focused mainly on per-node performance and eventually targeted vector-based instruction processing.

A consistent theme throughout these projects was the increase in performance and decrease in power required for executing applications. However, given the different fabrication methods for DRAM and logic, PIM fell out of favor until recently when various groups began investigating how to utilize the logic layer in stacked memory products like Hybrid Memory Cubes. These works have investigated a variety of different customization options for the logic die, and, similar to the original PIM results, estimate significant improvements in performance and reductions in power. However, as they generally target single node performance, issues such as cache coherency and the ability to efficiently access off-node data remain.

C. Migrating Threads

A key aspect of the Emu system is that threads automatically migrate to the memory they reference. Active Messages was an early approach that performed similar operations, and, as implemented in packages such as GASnet, permits a thread running on one core to spawn a function executing as a thread on another core. Nomadic Threads was a software based approach that allowed activations (groups of threads) to migrate through the system when accessing remote memory and demonstrated a significant reduction in communication when doing large distributed memory problems.

Thread migration within cores in chip multiprocessors (CMPs) has become important for factors such as load balancing, power-moderation, wear-leveling, speculative execution, and utilizing specialized processing cores. Challenges in migrating threads highlighted by are the amount of thread state that must move as well as the cold cache effects when the thread starts execution on a new core. The Execution Migration Machine is a recent example of a CMP implementing migrating threads, and like the Emu system, seeks to minimize the number of registers sent with each migrating thread.

D. Other Systems

The J-machine was a fine-grained parallel computer that attempted to balance computation and memory at each node while providing mechanisms for communication and synchronization that enabled a variety of parallel execution models. One of the key takeaways from this effort, particularly as it relates to the Emu architecture, is that with the appropriate communication mechanisms in place, fine-grained computing with small threads could be made efficient and cost-effective.

The Tera/Cray Multi-Threaded Architecture (MTA) and its follow-ons are highly multithreaded architectures that required nearly all memory references to go through the system network. In these systems, each processor maintained 128 active, but stationary, threads and issued an instruction from a different thread each cycle. Since nearly all memory references required a round trip traversal through the network, the programmer and compiler did not attempt to create data locality. However, this limited performance and scalability as it increased network bandwidth requirements and prevented intelligent programmers from taking advantage of any locality that may exist in their application. The Emu system shares many similarities with the MTA in the threading and execution model, but migrating threads are used to reduce network band-
width requirements, and hardware and software mechanisms are provided to take advantage of locality that may exist.

III. PROGRAMMING MODEL

A. Address Space

The Emu system supports a partitioned global address space (PGAS) system with memory contributed by each nodelet. A portion of this address space is marked as “Replicated,” so that the same address accessed by a thread resident on any particular nodelet will access a location on that nodelet. This Replicated range can be used by an application programmer to ensure data locality and is particularly valuable for providing local copies of global variables, important constants, and pointers into application data structures that are dispersed across the system. Utilizing this range can help minimize the number of hotspots that may exist in an application. Instruction code is always maintained in replicated memory so that a thread can always find its code locally without knowledge of where the thread is executing.

B. Threads and ISA Features

One advantage of the Emu architecture is that it implements thread migration, without programmer intervention, as memory locations on different nodelets are accessed. This automatic thread migration is handled completely in hardware, which first determines if the memory access is local to the current nodelet. If not, the thread context is removed from the current nodelet, transmitted across the system interconnect to the correct nodelet, and restarted there.

Thread contexts in the Emu system are compact, typically 10-20 64-bit words, and consist of the thread status word (TSW), address register, and assorted data registers. To minimize bandwidth requirements as threads migrate, the TSW tracks which registers are live, and instructions are provided to modify this state to remove dead registers. The compiler inserts these state modifications immediately before anticipated migrations based on its knowledge of the set of registers containing meaningful data.

The ISA executed by the migrating threads provides a standard set of instructions, including basic floating point operations, enhanced with several key features. One key feature is support for single instruction spawns by which a parent thread can create a child thread. The child then pursues an independent existence including the ability to spawn further generations of threads. The thread ISA also contains a rich set of atomic instructions which perform read-modify-writes on memory without any possibility of intervening accesses.

Stores and atomic instructions that do not require returning a value to the thread can be handled as remote operations. To execute a remote operation, the thread does not migrate but instead generates a short packet with the data, operation to be executed, and the address to be updated. This remote packet traverses the system to arrive at the correct nodelet where the atomic update is completed. If desired, the remote nodelet sends an acknowledgment packet back to the sending thread. This is done when the sender wishes to know that an operation or set of operations is complete and globally visible before proceeding to a new phase of the algorithm. Remote operations are especially advantageous for threads that need to update information in a “fire-and-forget” manner, as they do not return any data to the sending thread.

C. Cilk

To fully utilize the Emu system, mechanisms for expressing parallelism and creating large numbers of threads are required. The general model for doing this is based on Cilk [8], an extension of C. We have implemented a LLVM-based toolchain that supports three Cilk keyword extensions to C: cilk_spawn, cilk_sync, and cilk_for. The cilk_spawn keyword is a prefix to a conventional function call that converts it into a non-blocking call that executes independently of the code following the call (called the “continuation”). This is equivalent to starting a new thread to do the function call and corresponds almost directly to the spawn instruction in the ISA.

The cilk_sync keyword causes the thread executing it to wait for all child threads (created by cilk_spawn) previously created by the enclosing function to complete. Given a block that contains one or more cilk_spawns, there is an implicit cilk_sync at the end of the block.

The cilk_for keyword looks syntactically like a conventional for. The difference is that there is no assumed or implicit ordering to the loop evaluations when using cilk_for as each loop iteration can be executed by a separate thread. Notionally, the loop is converted into a tree of cilk_spawns where each such spawn uses the body of the loop as an anonymous function.

As threads are spawned in the system, they are given a priority value that ensures that the child thread has a lower priority than the parent thread. This is done to conform to the recommended operating behavior of Cilk programs. This is logical as threads with higher priorities can dispatch many descendants, and therefore will typically exercise greater influence over the progress of the application than their descendants. Limits are in place to prevent too many generations of threads from being created.

D. Intrinsics and Libraries

In addition to the Cilk-based keywords, the toolchain includes intrinsic functions that allow the programmer to directly access parts of the architecture and ISA that are not utilized in the vanilla Cilk model. Key elements of the ISA accessible via intrinsics include the atomic memory operations such as compare-and-swap and in-place arithmetic and logical functions (e.g., fetch-and-add) along with several thread control functions (save state, reschedule, etc.). By using these intrinsics, the programmer can exercise finer control over the scheduling of threads and memory operations.

The musl C library1 is currently being ported to the Emu system with success. Additionally, we have created other

1www.musl-libc.org
libraries for application developers to assist with managing distributed memory and the initialization and access of replicated variables. We anticipate developing additional libraries in the future to aid with other common application tasks (e.g., spawn distributed memory and the initialization and access of replicated variables).

The Gossamer cores execute the threads and are general-purpose pipelined processors capable of supporting 64 concurrent threads; when a thread slot is available, the GC requests a thread from the NQM. Internally, the GCs implement fine-grained multithreading and issue a single instruction each cycle. Each thread is limited to one active instruction at any given time, but since the system is expected to maintain large numbers of active threads, limiting the execution rate of a single thread should have a minimal performance impact.

Limiting each thread to a single active instruction reduces the amount of logic needed by each GC, as features like branch prediction and data hazard mitigation are not required. Logic requirements within the GCs are further reduced as they do not include data caches; memory transactions are either atomics or done as loads and stores between the thread registers and memory. Removing data caches eliminates coherency issues and reduces communication requirements in the system.

When a thread performs a memory transaction on a GC, the transaction is submitted to the Memory Front End (MFE). The NQM generates the (optional) acknowledgment packet and the remote operation, or acknowledgment as defined in Sec. III-B.

A thread will continue to execute in a GC until it migrates, quits, or is evicted by a timeslice interrupt.

<table>
<thead>
<tr>
<th>SP</th>
<th>NLS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stationary Core (SC)</td>
<td>Nodelet</td>
</tr>
<tr>
<td>SATA Interface</td>
<td>PCie Interface</td>
</tr>
<tr>
<td>DRAM for SP</td>
<td>Narrow Channel DRAM</td>
</tr>
<tr>
<td>Peripheral (e.g., Infiniband)</td>
<td></td>
</tr>
<tr>
<td>Solid State Drive</td>
<td></td>
</tr>
<tr>
<td>PCIe Interface</td>
<td></td>
</tr>
</tbody>
</table>

Fig. 1. Emu Node Architecture

The NCDRAM system does return more bits than necessary for the ECC, but future implementations may provide more elaborate ECC encodings to support multi-bit errors or special memory functionality such as full/empty bits. This overhead is a small price to pay to avoid reading unused data from a standard DIMM based memory system.

B. Node Architecture

Eight nodelets are replicated and combined with a Migration Engine to form the majority of the NLS (NodeLet Set) portion of the node as shown in Fig. 1. The NLS also maintains two PCIEexpress interfaces where one provides an interface to the Stationary Processor (SP) and the other can be optionally used for connecting other PCIEexpress cards (e.g., Infiniband). The Stationary Processor portion maintains the Stationary Cores (SCs), a solid state drive, a boot NVRAM, and a DRAM for code and data that is private to the SP.

The NLS is the workhorse of the Emu system, containing all of the logic except the SP and its associated functions.

2A small instruction cache is included in each GC.
The Migration Engine (ME) is a crossbar with a direct connection to the Nodelet Queue Manager within each nodelet, a Stationary Core Proxy, and six RapidIO interfaces.

The SP contains conventional 64-bit processing cores running Linux that act as service processors on each node to handle I/O requests and provide exception handling for threads running on the nodelets. These cores are labeled “stationary” because they run conventional threads that stay on the same processing core from birth to death.

**NLS-SP Communications:** The Stationary Core Proxy is an input block within the Migration Engine that allows the SP to inject migrating threads into the system. Once injected, these threads can migrate to any nodelet in the system and access memory or perform work on behalf of the sending SP. Additionally, the NLS supports an on-chip network that allows the SP to read/write data directly from local nodelet memory via PCIExpress transactions.

Communication from the nodelets to the SP is accomplished by having the GCs transfer thread contexts into a Service Queue implemented in each nodelet’s memory. When moving a context into this queue, the GC tags it to indicate the specific service needed. The SP polls the Service Queues and removes thread contexts from them. The SP then performs the service request and will return the thread to the nodelets via the Stationary Core Proxy, alert the user, or end the program (e.g., the application called `exit()`) depending upon the request.

**Node Packaging:** All components of the node shown in Fig. 1, except for the optional peripherals, are packaged onto a single custom card approximately 100mm x 275mm in size.

**C. System Interconnect and Organization**

The system interconnect between nodes in the Emu architecture utilizes the Serial RapidIO (SRI0) network standard [36]. RapidIO was chosen as it scales from chip-to-chip through chassis-to-chassis connections in a variety of applications. The Emu system has been architected to guarantee in-order delivery of remote operations to ensure that remote operations sent by a single thread to the same address occur in the correct order. Additionally, we have mapped the packet types used in the Emu system to basic RapidIO message types thus enabling the use of standard off-the-shelf RapidIO switches and IP.

Our initial system implementation, currently being brought up, is designed to reside in a standard office environment and consists of a tower chassis with 8 node cards placed onto a custom motherboard. This implementation is organized into a cube topology, as shown in Fig. 2, to provide direct links between a majority of the nodes. Since each node lacks the seventh link needed to form a fully connected system, the cross diagonal links are not made; packets needing to traverse these links are instead routed through a neighboring node before reaching their destination. This configuration provides a platform for early adopters as well as building and testing the system without any RapidIO switches.

A second configuration, under development, utilizes the same node cards and a second custom motherboard to hold 32 node cards and 24 RapidIO switches. The motherboard and attached components are encased in a rack-based 3U tall chassis and is a tray. A custom network topology has been designed for this system configuration to provide efficient any-to-any communications. For systems with more than two trays, RapidIO switch boxes, such as those from Prodrive Technologies [37], will be used to connect trays to one another.

**D. System Specifications**

The NodeLet Set is currently implemented on an Altera Arria10 FPGA with an expectation of 4 Gossamer Cores per nodelet; a clock rate of 300 MHz is targeted. The NCDRAM system is implemented with memory devices using DDR4-2133 and provides a capacity of 8GB per nodelet (64 GB/node, 512 GB in the 8 node system).

The Stationary Processors are NXP (née Freescale) T1024 processors with PowerPC e5500 cores. Given the generic purpose of these processors, this part could be replaced with another processor.

The system interconnect uses 4-lane RapidIO Gen2 which provides a full-duplex bandwidth of 5 GB/s per link (2.5 GB/s each direction).

Future generations will replace the FPGA with an ASIC, use RapidIO Gen 3 with additional ports per NLS, and use new memory technologies (e.g., Hybrid Memory Cubes).

**V. COMPARATIVE SYSTEMS AND RESULTS**

The Emu architecture is designed around improving the efficiency with which we access memory when such accesses are to “random” addresses throughout a large PGAS memory space. Performance evaluation of such a new architecture must have two components: first a comparison of aggregate characteristics to leading examples of current scalable parallel architectures, and second a comparison of kernels that exercise these characteristics in realistic programs.

A. Comparisons to Other Systems

The comparison reported here considers multiple modern systems taken from the Top500 and Graph500 lists that can be grouped into one of three different architectural classes.

**Loosely Coupled Distributed Memory (LCDM)** systems utilize commodity server blades and commodity networking protocols such as Ethernet or Infiniband. The basic hardware is unaware that the system has been scaled up to multiple nodes. The software infrastructure is typically based on MPI. A thread on a core can access memory directly only within the node that contains it, and explicit software intervention is needed to send or receive packets (data) between nodes. Frequently, each node
maintains a buffer for each destination node and aggregates messages in the buffers to prevent inefficiencies due to short messages. Accessing and maintaining these buffers skews the memory accesses patterns of real programs and incurs a fair amount of overhead.

**Tightly Coupled Distributed Memory (TCDM) systems** are highly scalable and make up the bulk of the highest performing entries on the TOP500 and GRAPH500 lists. The networking protocol is typically purpose-built, often with separate routing and network interface chips embedded with the compute processors, and expressly designed to scale up to 100s of racks. In these cases, a thread on a core can typically access memory directly only within the node that contains it, but there is a specially-designed dedicated high speed network coupling nodes that typically provides some low level RDMA operations, usually in a non-coherent fashion. While MPI is still a common software programming paradigm, the underlying software uses the network’s ability to perform operations against remote memory to avoid much of the overhead present in LCDM systems. There still is, however, a need for an application to understand what is local and what is non-local, with explicit calls to routines to handle the latter.

**Shared Memory (SM)** systems are purpose-built to support large multi-node shared memory architectures. Programming paradigms include multi-threading (e.g. Pthreads and OpenMP), where each thread is notionally associated with a specified core on a specific node. Networks are largely invisible to applications as memory references can access any location without knowledge of where the target memory location lies. Depending on the system, remote memory references may take on different degrees of coherency. Some systems in this category use hardware-level multi-threading to hide the latency of remote accesses, such as in the MTA machine and its descendants. Such systems have proved particularly efficient at non-numeric codes that involve many random, and remote, memory references, as in the GRAPH500 benchmark.

The SM class is closest to the Emu architecture, but lacks the ability to scale much beyond a few 10s of racks. Also, when commodity processors are used, cores quickly go dormant when remote memory operations are in progress, and a complete round-trip latency must be endured.

Figure 3 diagrams the two most important memory characteristics for the sparse applications addressed here: for a single microprocessor chip (“socket”) the number of memory channels and the peak access rate (the number of independent memory accesses made per second) from those channels. The closest competitor is an LCDM system using the most recent high end microprocessor chip; however, its effective access rate will be lower due to the message passing overheads as described earlier. Looking forward, conventional systems may be hard-pressed to advance much further as (1) single chip pin limitations are reached and (2) an increase in the memory access rate leads to an increase in coherency traffic.

Given the scale of systems being targeted (many racks), comparisons must be made on an equal basis. While “per node” or “per rack” are appealing, the systems considered have wildly varying physical footprints. Consequently, the remaining comparisons are normalized to “per square foot of rack space,” and “per watt” bases. The Emu system results include both the 8-node tower and a 256-node rack constructed from 8 trays along with the RapidIO switch boxes used for interconnect. The 8-node system is treated as physically equivalent to one-eighth of a rack for “per square foot” comparisons.

Figure 4(a) updates the per socket results of Fig. 3 to address the issue of how densely can such sockets be packaged in complete systems by switching to a “per square foot” metric. Here the nearest competitor to the Emu 8-node system is a very high end TCDM system with very advanced packaging. The full-rack versions of the Emu system still provide significantly more capability.

A second key performance metric for comparing the Emu system to other systems is the network injection bandwidth. Figure 4(b) plots the injection rate versus the memory access rate on a “per square foot” basis. As energy efficiency is also an important metric for large systems, Fig. 4(c) does a “per watt” rather than “per square foot” comparison. Only the highest-end TCDM systems have good injection bandwidth on a per square foot basis, and no system comes even close when energy is concerned. For the former, however, the actual topology is also a factor, as many TCDM systems employ simple multi-dimensional torii interconnect. In such cases, the through node re-routing can consume a large fraction of the node’s injection bandwidth when messages are to “random” targets rather than just nearest neighbors. The topology of the Emu system is designed to favor such random paths.

**B. Application Results**

Several applications, including Random Access (aka “GUPS”), Breadth-First Search (BFS), sparse matrix-vector multiply, and particle swarm have been written in Cilk, compiled with our toolchain, and executed on a SystemC based near cycle accurate simulator that models the Emu system hardware.

We focus on the GUPS benchmark because of the availability of comparative reports on different systems. This
benchmark was developed as part of the HPC Challenge Benchmark suite [38]. A large table $T$ of 64-bit integers is set up to consume about 1/2 of all system memory, ideally ensuring part of the table is found on each node. A sequence of pseudo-random integers $x_1, \ldots, x_N$ is computed and used to update entries in the table as $T[x_i] = x_0 \leftarrow x_0 (T[x_i], x_i)$. While algorithmically simple, considerable complexity is often needed for good performance on current parallel systems. For example, Ref. [39] describes complex algorithmic optimizations for IBM Cell and GPU implementations. Both involve very extensive packing, coalescing, locking, and streaming operations, all of which greatly increase code complexity, and with performance at best less than our results.

By comparison, the Cilk code is short and simple. $T$ is spread across memory and local data structures for random number generation are maintained in Replicated memory. A set of threads is started on each nodelet with each thread given the number of updates it is to complete. Each thread then executes a loop which computes a random number, uses this to construct an index into the table, and then performs a remote operation to perform the update.

The HPC Challenge Awards Competition\(^3\) included results from this benchmark and the top two performers were IBM POWER7\(^4\) systems at 2,021 and 1572 GUP/s\(^5\). The next two systems were the 864-rack K computer at 472 GUP/s and a 48-rack IBM Blue Gene/Q at 418 GUP/s.

Analysis of these results over time has shown that the GUP/s per socket rate has been essentially flat at about 0.01, with the exception of the POWER 775 results at about 0.25\(^6\). This is due to the inclusion of remote atomics in that system’s hardware.

The current simulation-based projections for the 8-node Emu architecture is approximately 2 GUP/s, and over 60 GUP/s for a 256-node single rack system. Since each node uses a single FPGA “socket,” this relates to about 0.25 GUP/s per socket - nearly the same as the currently-best POWER systems. The one rack system would place 9th in these rankings, versus the approximately 21 racks needed for the POWER and 864 racks for the K Computer.

Figure 5 compares the Emu systems to the best of breed of the above on a “per square foot” and “per KW” basis. The “x86” point represents the highest-ranked system that uses an x86-style chip for which complete data was available. The advantage of the Emu architecture is clear.

VI. CONCLUSION

The Emu system introduced here can improve the performance of data-intensive applications that run poorly on other architectures. This is done through the unique combination of having threads migrate to where the data they are using is stored and a highly efficient narrow channel memory system. The Emu architecture is highly scalable and presents a common global address space to application programmers which

---

\(^{3}\)From 2014: http://www.hpcchallenge.org/

\(^{4}\)http://www-03.ibm.com/systems/power/hardware/775/

\(^{5}\)http://icl.cs.utk.edu/hpcc/hpcc_record.cgi?id=495

\(^{6}\)The POWER 775 reports list under “processors” a number that correlates to a 4-chip MCM, each chip of which is 8 cores. The GUP/s per socket here were prorated to each of the processor chips.
is accessible via a simple, consistent programming model that allows massive numbers of active threads. Our results show significant advantages to the Emu architecture, particularly when measured on the “per square foot” and “per KW” bases.

REFERENCES