Structuring the Unstructured Middle with Chunk Computing

Justin Mazzola Paluska Hubert Pham Steve Ward
MIT Computer Science and Artificial Intelligence Laboratory
Cambridge, MA, U.S.A.

Abstract

Modern computing substrates like cloud computing clusters, massively multi-core processors, and general-purpose GPUs offer a wealth of computing power with the caveat that programmers must carefully structure their programs to fit within various hardware and communication limits in order to get high performance. Unfortunately, modern abstractions expressly hide machine structure from the program and program structure from the machine, hindering automatic optimization. We introduce an alternative computing abstraction—a linked graph of finite-sized memory *chunks*—that explicitly exposes the size and structure of programs to the operating system. The chunk graph enables the operating system to use size and structure to optimize how programs are mapped onto complex machine structure.

1 Introduction

In the past few years, the computer industry has seen the growth of three new computing substrates: elastic cloud computing, massively multi-core computing, and general-purpose GPU computing.

At a high-level, these three substrates are wildly different. Elastic cloud computing systems like Amazon's EC2 and Microsoft's Azure provide dynamically expandable clusters of hosts, each connected over a virtualized network. Massively multi-core processors like the Tilera TILE [1] series of processors provide a plethora of processor cores connected by on-chip buses and local caches. General-purpose GPUs like NVIDIA's Tesla [2] provide an abundance of ALUs connected by a hierarchy of memories.

While the granularity and specifics of these new substrates differ, what is common to all of them is (1) an abundance of processing elements and (2) programmer-visible communication and storage hierarchies whose proper utilization determines how efficiently the processing elements may be used. For example, in the cloud, intra-core operations are faster than intra-host operations, which are faster than inter-host operations. Likewise, in the GPU, communication and data sharing within a single processing element is fast compared to communication and sharing processing between elements, which itself is fast compared to CPU to GPU communication.

In contrast to "normal" CPU-based platforms that automatically manage the memory and communication hierarchy for high performance, to wring the most performance out of these new substrates, a programmer must carefully manage the communications and storage hierarchy so that the processing elements are continually fed with instructions and data. Typically this is done by making sure that related computations are on processing elements close to each other in the computing substrate and that computations "fit" into the variously sized buckets that each level of the substrate provides. Consequently, the hardware has natural, if non-linear, set of "size" and "distance" metrics that correlate with how well a given program may perform on the machine.

2 Space: the Next Frontier

The previous paragraph points to using the spatial concepts of "size" and "distance", as well as "structure" that determines how space is used, to optimize the runtime performance of programs. Unfortunately, modern machine interfaces—strings of assembly code operating in a flat memory—expressly hide machine structure behind the machine language abstraction. At the same time, assembly code obfuscates software structure from the runtime managing the hardware. As Figure 1 illustrates, machine langauge is the "thin middle" of the computing hourglass. In order to provide opportunities for optimization on new computing substrates, we believe that we may need to expand the thin middle.

We would like to exploit the structure of software in order to chop apart software elements (like data struc-

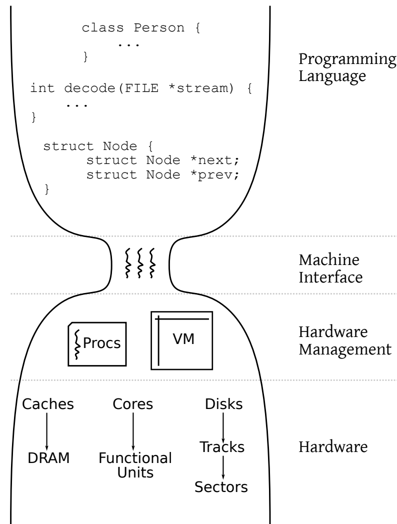

Figure 1: The structural hourglass of computing.

tures or functions) to fit inside the processing, storage, and communication elements of the computing substrate, ideally at runtime by an external hardware manager (e.g. a cloud manager, a multi-core aware OS, or a GPU runtime). Unfortunately, the assembly code compilation process flattens explicit structural elements into "implicit" structure. Links between data structures and program parts are obfuscated by pointers indistinguishable from other scalar data: their status as edges in a computation graph becomes apparent only at the time when the pointers are used as pointers by the processor. The flattened assembly also loses all size indicators: the sizes of both basic blocks and data structures are hidden behind the flat memory abstraction, which itself hides the complexity of the memory system.

Flat assembly code has served us well for many years, but in this "new age" of computing with multiple and varied processing elements inside complex hierarchies, the flat abstraction might impose too high of a performance cost and lead to too many lost optimization opportunities.

2.1 Domain-specific Solutions

There are already new domain-specific abstractions that "thicken" the thin middle and enable the programmer to match their problem to that of the available hardware. In the cloud, large-scale data processing systems like Google's MapReduce [3] and Microsoft's Dryad [4, 5] expose the structure of data parallel tasks to cluster management software. The cluster management software, in turn, manages code and data placement on hosts in the cluster so as to achieve high hardware utilization and task throughput. At the opposite end of the hardware spectrum, NVIDIA's CUDA [6] programming model exposes three levels of memory (private, shared, and device) to programmers, allowing the programs to pre-fetch data they know they will need, rather than relying on guesswork by the processor.

2.2 An Application-Generic Approach

However, rather than requiring a specialized new programming model, we would like to exploit the structure already inside both programs and the computer systems that run them. Our key insight is that a new machine interface may allow us to better map program structures onto the structure of the computing substrate.

This paper explores a model for computation that explicitly exposes three properties of both code and data:

Structure How code and data relate to each other.

Distance How far apart related elements are.

Volume How large programs and data are relative to available hardware resources.

Concretely, we focus on a particular kind of structured execution environment centered around small, simple, finite-sized blocks of memory called *chunks* that are dynamically mapped by a hardware management layer to different levels of the communications, storage, and compute hierarchy. Chunk-based systems explicitly expose program structure to the hardware management runtime while at the same time providing natural distance and volume measurements of code and data. Exposing distance, volume, and structure information allows runtime systems to optimize program placement, data flow, and communication.

3 Chunks

A chunk is an ordered, fixed-size array of fixed-sized *slots*. Each slot is typed and may be empty, contain scalar data, or hold a *link* that points to another chunk. Chunk links are explicit and can be used to create large data structures out of networks of chunks. Each chunk has an unique identifier; chunk links are simply slots typed as references and filled with the identifier of the referent chunk. Chunk links are not addresses and have no semantics beyond identity, giving us the freedom to dynamically mapped chunks to any location in the memory while avoiding "implicit locality" implications of nearby addresses. Each chunk is also annotated with a type, allowing a runtime to distinguish and interpret chunks differently.

3.1 Data Structures

Since chunks have a fixed number of fixed-sized slots, there is a fixed limit to the amount of information a chunk may contain as well as limit to the number of outgoing links a chunk may have.

While fixed sizes may be seen as a restriction, we view them as an advantage since large data structures must be decomposed into a graph of inter-linked chunks. By using the number of pointer traversals between two chunks as a distance metric, we now gain a natural notion of volume as the maximum size a group of chunks may take up. More concretely, the volume of the neighborhood a radius R edges around some chunk is at most $s \cdot d^R$, where s is the size of a single chunk and d is the maximum out degree of a chunk. While this is a conservative upper bound on the volume a computation may take up, more detailed analysis that takes into account the actual link structure in a chunk graph may reveal tighter bounds.

A "small" data structure is one whose working set's volume is smaller than a hardware limit, e.g., a cloud node's available memory, a core's L1 cache, or bus message buffer. Large data structures are those that overflow hardware resources. The hardware manager must actively manage data flow of large data structures in the chunk graph, e.g., by pre-fetching data into processing elements, to maintain high performance. This is the primary advantage of our model since it reflects how large data sets must be time-shared into finite hardware resources, while at the same time exposing large structures in small granules that may be partitioned and distributed across clusters of hardware resources.

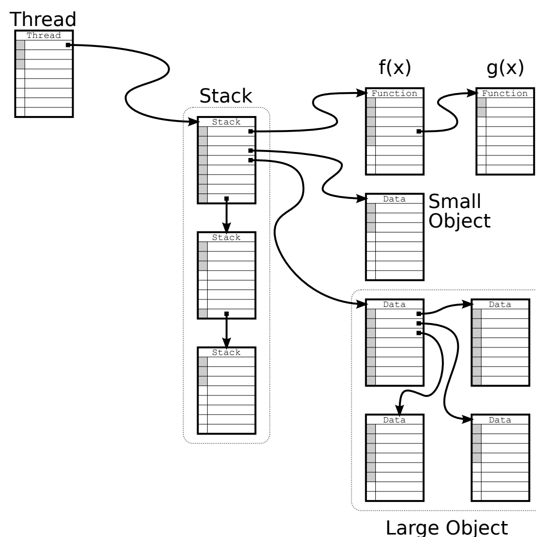

Figure 2: Snapshot of a program executing with limited-size chunks. Each slot may contain scalar data or a link to another chunk. Slots that are in use are marked in gray. Because each chunk is limited in size, the large stack and large object must be split into multiple chunks.

3.2 Computation

We integrate computation into the chunk model by embedding code and active thread state inside chunks, either by direct or just-in-time compilation, and executing programs within a management runtime that places chunks on processing elements and translates them to native machine code and pointers as necessary.

Figure 2 shows one representation of an executing program. In this particular representation, each computation is rooted in a `Thread` chunk. Each `Thread` chunk links to a `Stack` chunk that represents the current state of the computation that the thread is executing. The `Stack` chunk links to a `Function` chunk that holds the code of the currently executing chunk, which in turn links to any global data structure chunks to which the function may access. The `Function` chunk also contains links to other `Function` chunks that it may potentially call. Local variables, including any data available to the function by way of closures, are linked from the `Stack` chunk.

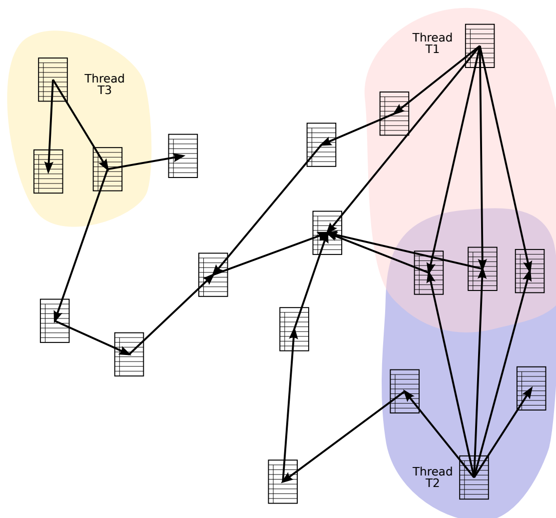

Figure 3: Three threads operating in a chunk space with their volumes of influence for $R_i = 1$ highlighted. Threads t_1 and t_2 are close to each other and must to be synchronized, while thread t_3 may run in parallel with the other threads.

Since there is a limit to number of instructions in each `Function` chunk and a limit to how many other chunks any given instruction can access, it follows that there is some volume of *influence* defined by a radius R_i from a chunk that encompasses everything that the chunk can read or update during its execution. If `Thread` chunks of two threads have roots that are $2R_i$ apart from each other in the chunk graph, then they cannot influence each other during their execution. If two threads are closer than $2R_i$, their volumes of influence may overlap and may need to be considered together by the hardware management layer.

For example, consider Figure 3, in which three threads execute in a chunk space. Threads t_1 and t_3 are far away from each other in the space and may be able to run concurrently without competing for resources like memory bandwidth. In contrast, threads t_1 and t_2 have overlapping volumes of influence and must time-share memory bandwidth and synchronize access to their shared chunks. The operating system may be able to use this information to, e.g., place t_1 and t_2 on the same processor core so they can share synchronization objects, while placing t_3 on a separate core that runs in near-isolation with no synchronization overhead.

4 Discussion

The chunk abstraction may enable new kinds of optimizations. We outline three below.

4.1 Synchronizing the Volume of Influence

The volume of influence metric allows us to find "independent subsets" of programs—those parts of the program that operate on distinct subsets of data—at runtime. For example, the chunk graph rooted at t_3 in Figure 3 is one independent subset since it is so far away from the other threads, while the combined graph of t_1 and t_2 might form another functional subset that can run in parallel with the t_3 graph. We also may be able to create independent subsets by adding additional "synchronizing" chunks to artificially increase the distance between `Thread` chunks beyond the $2R_i$ threshold. If two threads access a "synchronizing" chunk, it becomes a hint to the operating system to automatically synchronize them.

4.2 Volume versus Communication

Chunk links are references, and as such, do not refer to particular addresses of particular chunks, but rather are names for chunks that may reside on the same processing element or a processing located across the network. The chunk graph of an application exposes potential communication paths between parts of the program and where a hardware runtime may need to manage communication costs.

If two threads of computation on two separate processing elements start operating on the same data sets—that is, the volumes of their working sets overlap—the hardware manager might choose to migrate the threads to processing elements that are closer to each other, or even to timeshare the threads on the same processing element in order to reduce communication costs.

The hardware management software could, in the degenerate case, reduce communication costs to zero at the expense of parallelism by placing all threads on the same processing element. Of course, doing so will cause a large volume of chunks to operate on a single processing element. In order to balance the volume of chunks on the machine with communication costs, the volume of the chunks used in the computation can be used as inputs to a utility function that accounts for parallel speedup versus communications costs.

In an elastic environments where it is possible to increase and decrease available hardware, e.g., dynamic allocation of cloud hosts or power-constrained management of processing elements in a GPU, we can extend the chunk placement algorithm to also take into account the dynamism of the set of processing elements. For example, if a particular cloud application suddenly starts creating many new chunks in response to user requests and the number of chunks residing on one processing element climbs over some threshold, the hardware manager may automatically redirect some subset of the requests to another already-running processing element or, if all of the processing elements are already running at capacity, to a newly powered-up processing element, ideally splitting requests so the each machine hosts a disjoint set of chunks and can operate independently without synchronization. When the spike in chunk creation subsides and the total volume of active chunks falls, the hardware manager may start to coalesce chunks on fewer processing elements and shut down the redundant ones.

4.3 Chunk Sizing

We require that chunks be a fixed size in order to induce nice distance and volume metrics. It is an open question what the best fixed size is. From a graph and resource management standpoint, smaller chunks allow fine-grained placement of program structures, while larger chunks amortize overheads over more bytes of data. On the other hand, since we use chunks to model hardware resources, the "native" packet size of network hardware or page boundaries of memory hardware might provide better guidelines for chunk sizes. A network-heavy application might benefit from chunks based on a standard Ethernet frame of 1500 bytes, while more tradi-

tional memory-oriented applications might favor a chunk size based on a 4 KB page.

Rather than fixing physical chunk size, we virtualize the physical chunk size. Virtualization gives the hardware management layer freedom to compose a virtual graph of fine-grained chunks out of a physical graph with coarse-grained *superchunks* that better match the actual hardware resources. Superchunks reduce link resolution overheads since virtual chunk links that point within the superchunk could be implemented as constant offsets to slots elsewhere in the superchunk. They also present an additional optimization path for the operating system: collocating closely linked chunks on the same superchunk. For example, when executing in a network environment that supports 9000-byte "jumbo" Ethernet frames, the hardware management layer might coalesce smaller chunks into 9000-byte superchunks. Since the application uses only standard-sized chunks and not superchunks, it may also run on a system with only standard-sized chunks without modification.

Superchunks are not exposed at the application layer, but at the hardware management layer, allowing the hardware management layer to provide compatibility over a broad range of different-scale chunk environments—from very small hardware like GPU processing elements to large, Internet-scale data clusters—all sharing the same set of small virtual chunks.

5 Related Work

This work is heavily influenced by a plethora of alternative computer architectures and operating system designs. The MuNet [7] machine and corresponding operating environment allowed a computation to span multiple processors and gradually migrate between them. MuNet inspired our cloud computing migration ideas. The $\mathcal{L}$ project [8] outlines a chunk memory system nearly identical to ours. We expand on the $\mathcal{L}$ model in two ways. First, $\mathcal{L}$ is limited to a fixed topology of nodes, while we allow fluid expansion and contraction of computation nodes. Second, our chunks are network-accessible, while $\mathcal{L}$'s chunks work only in the context of a single machine.

Chunks serve the purpose that fixed-sized virtual memory pages do for typical operating system kernels, namely that they abstract resource allocation and placement. The virtual memory system in the Mach microkernel [9] allows memory objects, like chunks, to be mapped dynamically by applications. Unlike chunks, however, Mach's memory objects are unstructured and of arbitrary size, limiting the analysis that may be performed on them. Alternatively, the Barrelfish multikernel [10] uses information from explicit message passing to optimize communication, much as we plan to optimize memory allocation with chunks.

A chunk-based system may be able to use optimizations developed for Non-Uniform Memory Architectures (NUMA). For example, Bolosky et al.'s NUMA kernel [11] uses VM pages as the unit of sharing and replication. Unfortunately, VM pages are prone to "false sharing" of different objects on the same VM page. Chunks may have the same problem if a compiler puts two objects in the same chunk. Fortunately, Granston et al. [12] argue that the compiler may be able to reduce false sharing by separating objects into different pages or chunks. Chilimbli et al. [13, 14] show that the order of fields in a data structure may help cache performance; a chunk compiler may re-order slots in a chunk to increase locality of reference.

6 Conclusions and Future Work

Chunks expose program structure and machine structure to the operating system and hardware management layers of the computing stack, allowing optimization based on structure. We have successfully used chunks as the data structures of a collaborative video editing application called ChunkStream [15]. We are implementing a distributed chunk naming system and interpreter.

References

[1] D. Wentzlaff et al. On-Chip interconnection architecture of the tile processor. *IEEE Micro*, 27(5):15–31, 2007.

[2] E. Lindholm et al. NVIDIA tesla: A unified graphics and computing architecture. *IEEE Micro*, 28(2):39–55, 2008.

[3] Jeffrey Dean and Sanjay Ghemawat. MapReduce: simplified data processing on large clusters. In *OSDI*, 2004.

[4] Michael Isard et al. Dryad: distributed data-parallel programs from sequential building blocks. In *EuroSys*, 2007.

[5] Yuan Yu et al. DryadLINQ: a system for General-Purpose distributed Data-Parallel computing using a High-Level language. In *OSDI*, 2008.

[6] John Nickolls et al. Scalable parallel programming with CUDA. *Queue*, 6(2):40, 2008.

[7] Robert H. Halstead Jr. and Stephen A. Ward. The MuNet: a scalable decentralized architecture for parallel computation. In *ISCA*, 1980.

[8] Joseph Derek Morrison. *A scalable multiprocessor architecture using Cartesian Network-Relative Addressing*. M.S. thesis, Massachusetts Institute of Technology, 1989.

[9] M. Young et al. The duality of memory and communication in the implementation of a multiprocessor operating system. In *SOSP*, 1987.

[10] Andrew Baumann et al. The multikernel: A new OS architecture for scalable multicore systems. In *SOSP*, 2009.

[11] W. Bolosky, R. Fitzgerald, and M. Scott. Simple but effective techniques for NUMA memory management. In *SOSP*, 1989.

[12] Elana D Granston and Harry A. G Wijshoff. Managing pages in shared virtual memory systems: getting the compiler into the game. In *ICS*, 1993.

[13] Trishul M. Chilimbi, Mark D. Hill, and James R. Larus. Cache-conscious structure layout. In *PLDI*, 1999.

[14] Trishul M. Chilimbi, Bob Davidson, and James R. Larus. Cache-conscious structure definition. In *PLDI*, 1999.

[15] Justin Mazzola Paluska and Hubert Pham. Interactive streaming of structured data. In *PerCom*, 2010.