University of Delaware
Department of Electrical and Computer Engineering
Computer Architecture and Parallel Systems Laboratory

Massively Multi-Core Systems and Virtual Memory

Guang R. Gao and Jack B. Dennis

CAPSL Technical Memo 128
April 29th, 2014

University of Delaware • 140 Evans Hall • Newark, Delaware 19716 • USA
http://www.capsl.udel.edu • ftp://ftp.capsl.udel.edu • capsladm@capsl.udel.edu

Contents

Abstract

The fundamental limits imposed by semiconductor technology have led to renewed interest in parallel computation, in order to achieve the maximum performance and energy efficiency of massively parallel computer systems. We believe this can be done by extending the virtual memory concept to the domain of parallel processing. First, we explain why virtual memory is essential for massive parallel computing in the age of many-core chip technology. We claim it is desirable and feasible to build an efficient Multicore Operating System that implements a global virtual memory using a codelet based runtime system. Finally, we explain how a codelet based programming model can be supported with modified system architecture to yield high performance and energy efficiency with the benefits of global virtual memory.

1 Introduction

During the last decade, the vendors of processing chips have found that placing multiple processing cores in a single silicon chip is a better way to employ chip area than continuing the quest for greater single thread performance. The prospect is that processing chips with hundreds of processing cores and beyond will become the standard parts of future computer systems.

Users of these new chips face the problem of organizing the computations they wish to perform, as collections of parallel activities that communicate with one another, to implement the essential coordination of these activities. For this, programmers usually use MPI, the Message Passing Interface, which has become the accepted standard software library for writing high performance codes for applications. Research and development of an extension of MPI coupled with a shared-memory parallel programming API - such as OpenMP and its extension - are underway and have been proposed as a potential parallel programming model for new generations of computer systems based on the multi-core chip technology. Our alternative approach to address these issues is the subject of this position paper.

1.1 Virtual Memory

The introduction of virtual memory in the late 1950s has been the most significant innovation in computer architecture for improving programmability. Before virtual memory, programmers had to employ overlays of the main memory address space to run programs using collections of code and data that exceeded the size of the main memory. Programmers also needed to explicitly program data movement within the memory hierarchy to address the issue of storage size limitations as well as the issue of locality.

However, in the new era of parallel computing, programmers have been asked to give up the benefits provided by the virtual memory in which all processes operate. They must be concerned with how the data is distributed, which data to move to less accessible memory and which data to retain for reuse, who has access to which parts, and how processes can avoid interfering with each other.

We believe the best route to a major improvement in programmability of massively parallel systems is to extend the virtual memory concept to the domain of parallel processing. Then, programmers will be freed from managing processor assignment and scheduling, just as virtual memory freed them from involvement in memory management.

Why haven't the benefits of virtual memory been retained in current and prospective massively multi-core computer systems? The thesis considered here is that the weight of legacy software has prohibited consideration of innovative programming models and novel ways of organizing chip components.

1.1.1 Global Virtual memory

The idea of providing hardware supporting addressing for very large data structures has been dormant since the early days of the Multics and AS/400 systems. The following concept was a key element in the design of the Multics computer system at MIT's Project MAC [1]: any object represented in the file system could be dynamically linked into a users address space within limits that have not been approached since then. The result was that the goal to build a computer system that offered the most powerful system support for programming in terms of modularity and security has yet been achieved. Multics, however, did not provide an implementation of global pointers and so its modularity benefits did not extend to parallel programming. The IBM AS/400 series of systems [2] embodied an implementation of global pointers, but the benefits were exploited by system programmers and software package developers, and not promoted for the end user. There is a significant opportunity to extend the benefits of a global virtual address space to systems built of many-core processing chips and to make its benefits available to users through new programming models.

1.1.2 Global Address Space and Virtual memory

Partitioned Global Address Space Languages, or PGAS, such as Co-Array Fortran [3] or UPC (Unified Parallel C) [4] are used to ease the programmer's burden by simulating a fully-shared memory address space. This enables the programmer to transparently access data objects held in remote memories of a distributed memory system. For this purpose, every data object has a global address (hidden from the application programmer) consisting of its local address and an identifier of the processors where the it resides. When a user program accesses an object using its global addresses, the PGAS runtime splits off the processor identifier and uses it to send a message requesting the access at the remote processor. The global address space of a PGAS model is not a true virtual memory because global addresses are not independent of the physical location of the data object. Therefore, a data object cannot be moved to a different processor without invalidating all occurrences of the global address in the program. This fact negates for PGAS models the benefits of virtual memory for general modular program construction.

1.2 Our Position Statements

We believe that for general-purpose high-performance computing systems employing massively multi-core chip technology, the following statements are fundamental and a good solution must be in place:

- Virtual memory is very desirable.
- It is feasible to build an operating system (OS) that implements virtual memory, using a runtime system based on a codelet program execution model (such as FreshBreeze [5]).
- A major advance in performance and programmability can be achieved through the co-design of new architecture features to support virtual memory implementation.

Our position paper is organized as follows: In Section 2, we argue why virtual memory is essential for massive parallel computing. In Section 3, we claim that it is both necessary and feasible to build a multicore OS featuring virtual memory, using a codelet based runtime system. In Section 4, we advocate the need of a novel programming model and architecture to accomplish our goal. Finally, we present our conclusions in Section 5.

2 Why Virtual Memory for Massively Parallel Computing

The field of high performance computing is anxious to move beyond the data parallel programming model and the bulk synchronous programming model implemented by MPI. Users of high performance computing are demanding greater flexibility: Computations that change their needs for memory and intensity of processing from time to time during program execution. This demand arises from several sources: Physics and materials simulations are becoming more sophisticated and scientists wish to model systems made up of components with different physical properties and irregular shapes; with the great advance in understanding biological systems in terms of complex molecular structures, parallel search and classification are now high priority areas for high performance computing; and with the rise of interest in social networking and concern for traffic flows in large networks – highway systems, the Internet, the brain – performing large-scale computation on line graph models of large systems has become enormously important.

This trend in the demand for high performance is counter to the general wisdom about multicore systems that has prevailed in recent years. The system design philosophy has been that programmers must manage resources and plan transfer of data between levels of the memory hierarchy to achieve full utilization of chip processing and memory resources. This has been a very successful approach for an historically important class of high performance applications – those that may be expressed as data parallel computations over a uniform grid of two or three dimensions.

Software tools have evolved to assist the applications programmer in this task: The HPC languages X10, Chapel, and Fortress have been designed to help users specify the distribution of data and work load over the memory and processing resources of a large parallel computing system. In this sense, MPI, the popular library of routines for coding computations that fit the bulk synchronous parallel (BSP) [6] model has become an important standard implementation tool.

Yet these tools do not address the underlying need of new applications for execution time and flexibility in resource use. In response to the new demands for high performance computing, thinking about system design has evolved to a new view: A parallel computer system must efficiently support a user interface through which the dynamic needs for memory and processing may be communicated by the application to the system. Section 3 reviews several current projects with these goals.

2.1 Dynamic Resource Management and Virtual Memory

Central to these proposals are two concepts: a unit of memory and a unit of processing. These concepts form the basis for dynamic management of system resources. The unit of memory, often called a *data block*, is a block of memory words that may be relocated within the address space accessible by a program. A data block may contain references to other data block, for example, in a set of data blocks organized to model a tree data structure. When a data block is relocated in memory, one must ensure that all references to the data block from other data objects remain valid. Unless the references are universally valid identifiers of the data block, they would have to be updated – a messy process if the references are from remote sites in the computer system. By requiring that a data block be relocatable, the system has, in principle, the ability to move data blocks to different locations in physical system memory to optimize data distribution in response to program behavior.

The unit of processing is a *codelet*, a block of machine instructions that define a computation task [7]. A codelet is the unit of work scheduled for execution by a processing core of a multi-core computer system.

- *Flexibility in managing resources requires that any codelet can be executed on any processing core.*

A system that imposes any restriction on where codelets may be executed places a burden on the application programmer, who must understand and live within the constraints.

Providing the ability to execute any codelet anywhere in the system requires that all cores have the same architecture and instruction set. More important, all input data blocks required by the task must be accessible to the codelet regardless of where it is executed. In the envisioned massively parallel systems, data blocks and tasks will be continually created, executed, and discarded. This implies that any data block must be accessible from any processing core. That is, data block access must be effective from any part of the system regardless of where the data

block is located. Thus, location-independent access must be implemented, and the collection of all names of data blocks forms a global virtual address space. We shall use the term *handle* for the unique identifier of a data block or codelet.

- *The ability to execute any codelet anywhere requires that data blocks have globally valid, location-independent identifiers. The set of such identifiers, called* handles *constitutes a global virtual address space.*

A memory allocation mechanism is required to create new data blocks as required by program execution. This implies that the system must maintain a pool of available memory from which new data blocks may be allocated. Furthermore, a mechanism is needed for returning memory occupied by data blocks to the free memory pool when they are no longer needed by the processes. Such mechanism could be explicitly implemented by the programmer. However, there are two problems with such approach: The first one arises as references to a data block may have been passed to other tasks, or incorporated into data blocks that are part of a data object that will be used in later processing. The upshot is that it is, in general, impossible for a program unit to know when it is safe to free memory for a data object it has created. Requiring programmers to have this knowledge is a violation of the principle of modular program construction[8]. The second problem concerns *space leaks*, failure to release memory no longer in use, which can lead to memory exhaustion and system failure. Avoiding this problem requires that programmers meticulously ensure that unused data blocks are returned to the pool of free memory. The alternative is system implemented automatic garbage collection [9]. Therefore, dynamic creation of data blocks and tasks requires system implemented resource management, including automatic garbage collection of data and memory blocks that are no longer needed.

2.2 Benefits of Virtual Memory

A very significant benefit of a large scale parallel computer system supporting a global virtual address space will be the ability to use any parallel program (constructed for the system) as a component of other parallel programs. Moreover, this is possible without any modification of component programs. Note that this property depends for its universality on the ability to access a data object from any execution site in the computer system.

Another benefit is the ability of multiple jobs to run concurrently, each being able to utilize resources that other jobs are not able to use. Privacy is ensured by the property that the data objects an executing task may access is limited to those objects for which it has been given handles.

- *The benefits of virtual memory include the ability to build parallel programs from modular components and the ability of multiple programs to share use of data and code objects during execution.*

3 The Role of An Execution Model and RUNTIME

A number of DOE XStack projects [10], such as Dynax [11] and SWARM [12, 13], aim to develop dynamic adaptive even-driven execution models that will provide a better API for application programmers and applications with varying demands for memory and processing resources.

In designing a runtime system to implement memory management, there are three issues to be addressed: (1) how blocks of memory are allocated from a pool of free memory; (2) how blocks that are no longer needed may be returned to the free memory pool; and (3) what an efficient representation for the free memory pool is.

In a distributed system with many processing cores, these issues become more complex. For example, the free memory pool should be distributed across the processors, so requests for new memory blocks at any processor may be quickly met. But how should the system ensure that roughly equal amounts of free memory exit at each processor?

Implementing memory allocation from a free memory pool is fairly straightforward; for instance the buddy system is a popular solution [14]. However, implementing distributed garbage collection is another matter. Much effort has been devoted to develop methods for adapting the classic mark/sweep algorithm [15] for distributed systems, with disappointing results. The critical problem is that the references that must be traced may cross system boundaries at any level of the memory hierarchy.

A way out of the garbage collection dilemma is to adopt a program execution model in which data blocks, once defined by execution of a codelet, are never modified. This *write-once* policy guarantees that cycles of references can never be created, and therefore the reference count method of garbage collection [16] may be used. Moreover, reference count garbage collection is readily implemented in hardware as a distributed activity, concurrent with user task execution. Another major benefit of the *write-once* policy is that cache memories may be used with no concern for multiprocessor coherence issues, as the content of shared data blocks never changes.

Thus our conclusion is:

- *An efficient runtime implementation of global virtual memory is possible if the write-once policy for management of data blocks is adopted.*

4 A Novel Programming Model and Architecture

Implementation of virtual memory through a multi-core operating system running on hardware with inadequate support for virtual memory yields a computer system that cannot reach the performance level achievable with hardware co-designed with the programming model for the best match of hardware technology for achieving the goal.

One candidate for this objective is the Fresh Breeze programming model and system architecture [5]. The foundation for this envisioned system is the idea of representing all data objects

as trees of fixed sized *chunks* of memory. We present the Fresh Breeze memory model and Fresh Breeze support for task management. Then, we discuss its overall system organization, experimental evaluation, and how it meets the goal of efficient virtual memory for massively parallel computation.

4.1 Fresh Breeze Memory Model

The Fresh Breeze memory model uses trees of 128-byte *chunks* of memory to represent all data objects. Each chunk has a unique identifier, its *handle*, that serves as a globally valid means to locate the chunk within the storage system. Chunks are created and filled with data, but are frozen before being shared with concurrent tasks. This is the *write-once* policy introduced in Section 3 that eliminates data consistency issues and simplifies memory management by precluding creation of cycles in the heap of chunks. Another benefit is that low-cost, reference-count based garbage collection may be used to reuse memory chunks for which no references exist in the system. Such a memory model provides a global addressing environment, a *virtual memory*, shared by all user jobs and all cores of a many-core multi-user computing system, with benefits for modular parallel programming. It can cover the entire online storage, replacing the separate means for accessing files and databases in conventional systems, extending the benefits of modular software to programs accessing data anywhere in online memory.

The power of the tree-of-chunks memory model has been demonstrated, using a simulated model, for linear algebra kernels [17, 18] and the Graph500 breadth first search challenge[19].

A compiler under development [5] will convert programs from a functional-style high level source programming language to fine-grain codelets for execution by a Fresh Breeze system.

4.2 Fresh Breeze Task Management

The unit of program execution in a Fresh Breeze system is a *codelet*, a block of instructions that is scheduled for execution by a processor core and remains assigned to the core until its execution has completed. Codelets may include instructions to spawn worker tasks that execute other codelets and combine their results for processing by a *continuation* codelet. The mechanism to support codelets is similar to the spawn/join model for parallel programming of Cilk [20]: A master task executing a codelet may spawn one or more worker tasks to execute one or more codelets. Worker tasks may receive data objects as arguments provided by the master task, and each worker task contributes the results of its activity to the *continuation task* using a join mechanism [21]. The scheme matches the data parallel features of a programming language such as Sisal [22] or NESL [23]. Through recursive use of this scheme, a program can generate an arbitrary hierarchy of concurrent tasks corresponding to available parallelism in the computation being performed.

4.3 Fresh Breeze System Architecture

The architecture of a Fresh Breeze computing system looks conventional from a high level perspective. Multi-core processing chips are coupled to an off-chip shared memory hierarchy. The principal differences are that the unit of data held at each memory level is a 128-byte chunk and that chunks are located by fully associative but economical search mechanisms at each memory level. In place of a conventional L1 cache, a set of one-chunk buffers is used and managed in correspondence with chunk handles held in processor registers, avoiding costly tag memory and lookup circuitry. A hardware scheduler maintains a queue of pending tasks at each processor core and performs load balancing by moving task records from the task queues of highly loaded cores to the task queues of cores with lesser load.

The Fresh Breeze concepts of memory and tasking realize a virtual memory for massively parallel computation because the collection of handles for data chunks forms a global virtual address space, handles being location-independent identifiers of memory chunks.

It is expected that a Fresh Breeze system will achieve superior performance and energy efficiency in comparison with the multi-core operating system described in Section 3 for several reasons:

- The use of a single size of memory block for allocation and garbage collection yields major simplifications of memory management, yielding benefits in performance and energy efficiency.
- No runtime or operating system software is involved in resource management.
- Hardware task schedulers permit efficient, fine grain task scheduling and load distribution.
- Fine grain task management extends strong scaling for application codes by distributing many small tasks over system resources.

5 Conclusions

The advent of the multi-core era has opened opportunities for exploration of new directions in computer system architecture. It now seems possible that the logjam freezing out the introduction of new approaches and ways of supporting parallel computation has been broken, after 40 years of being saddled with an obsolete programming model (the sequential process and a limited linear address space). We have described two paths we believe will lead to systems with the programmability advantages of global virtual memory: The first is a codelet based multi-core operating system; the second employs a novel system and processor architecture. Both paths lead to systems in which any parallel program (subject to resource constrains) can be used, without modification, in the construction of new parallel programs. We are currently studying the connections of the two paths and hope a co-design of many-core OS, runtime, and architecture will eventually lead to an efficient solution of the virtual memory system support.

References

[1] A. Bensoussan, C. T. Clingen, and R. C. Daley, "The Multics virtual memory," in *SOSP '69: Proceedings of the Second Symposium on Operating Systems Principles*, 1969, pp. 30–42.

[2] Q. Schmierer and A. Wottreng, "Ibm as/400 processor architecture and design methodology," in *Computer Design: VLSI in Computers and Processors, 1991. ICCD '91. Proceedings, 1991 IEEE International Conference on*, Oct 1991, pp. 440–443.

[3] R. W. Numrich and J. Reid, "Co-Array Fortran for Parallel Programming," *SIGPLAN Fortran Forum*, vol. 17, no. 2, pp. 1–31, Aug. 1998. [Online]. Available: http://doi.acm.org/10.1145/289918.289920

[4] W. W. Carlson, J. M. Draper, and D. E. Culler, "S-246, 187 introduction to upc and language specification."

[5] J. B. Dennis, "Compiling fresh breeze codelets," in *International Workshop on Programming Models and Applications for Multicores and Manycores*. IEEE, 2014.

[6] L. Valiant, "A Bridging Model for Parallel Computation," *Communications of the ACM*, vol. 8, no. 33, pp. 103–111, 1990.

[7] S. Zuckerman, J. Suetterlein, R. Knauerhase, and G. R. Gao, "Position Paper: Using a "Codelet" Program Execution Model for Exascale Machines," in *In Proceedings of ACM SIGPLAN 1st International Workshop on Adaptive Self-Tuning Computing Systems for the Exaflop Era (EXADAPT 2011); Programming Language Design and Implementation (PLDI 2011)*, Jun. 2011.

[8] J. B. Dennis, "A parallel program execution model supporting modular software construction," in *Massively Parallel Programming Models*. IEEE, 1997, pp. 50–60.

[9] R. Jones, A. Hosking, and E. Moss, *The Garbage Collection Handbook: The Art of Automatic Memory Management*. Chapman and Hall, 2011.

[10] US Department of Energy. (2014, April) Advanced scientific computing research - x-stack portfolio. [Online]. Available: http://science.energy.gov/ascr/research/computer-science/ascr-x-stack-portfolio/

[11] Dynax Project. (2014, April) Dynax project website. [Online]. Available: https://www.xstackwiki.com/index.php/DynAX

[12] ETI International, "Swarm (swift adaptive runtime machine). scalable performance optimization for multi-core/multi-node."

[13] C. Lauderdale and R. Khan, "Towards a codelet-based runtime for exascale computing: Position paper," in *Proceedings of the 2Nd International Workshop on Adaptive Self-Tuning*

Computing Systems for the Exaflop Era, ser. EXADAPT '12. New York, NY, USA: ACM, 2012, pp. 21–26. [Online]. Available: http://doi.acm.org/10.1145/2185475.2185478

[14] D. Knuth, "Fundamental algorithms. the art of computer programming." Addison-Wesley, 1997, vol. 1, pp. 435 – 455.

[15] J. McCarthy, "Recursive functions of symbolic expressions and their computation by machine, part i," *ACM Communications*, April 1960.

[16] D. F. Bacon, C. R. Attanasio, R. V.T., and S. E. Smith, "A Pure Reference Counting Garbage Collector," Aug. 2001. [Online]. Available: http://researcher.watson.ibm.com/researcher/files/us-bacon/Bacon03Pure.pdf

[17] J. B. Dennis, G. R. Gao, and X. X. Meng, "Experiments with the Fresh Breeze tree-based memory model," in *International Symposium on Supercomputing, Hamburg*, June 2011.

[18] J. B. Dennis, G. R. Gao, X. X. Meng, B. Lucas, and J. Slucom, "The Fresh Breeze program execution model," in *Parallel Computing, Ghent, Belgium*, August 2011.

[19] T. S. John, J. B. Dennis, and G. R. Gao, "Massively parallel breadth-first search using a tree-structured memory model," in *Proceedings of International Workshop on Programming Models and Applications for Multicores and Manycores*. ACM SIGPLAN, February 2012.

[20] M. Frigo, C. E. Leiserson, and K. H. Randall, "The Implementation of the Cilk-5 Multithreaded Language." in *PLDI*, J. W. Davidson, K. D. Cooper, and A. M. Berman, Eds. ACM, 1998, pp. 212–223. [Online]. Available: http://dblp.uni-trier.de/db/conf/pldi/pldi98.html

[21] J. B. Dennis, "The Fresh Breeze model of thread execution," in *Workshop on Programming Models for Ubiquitous Parallelism*. IEEE, 2006, published with PACT-2006.

[22] J. Mcgraw, S. Skedzielewski, S. Allan, O. Oldehoeft, J. Glauert, C. Kirkham, B. Noyce, and R. Thomas, *SISAL: Streams and iteration in a single assignment language, language reference manual version 1.2*. Lawrence-Livermore-National-Laboratory, Mar. 1985.

[23] G. E. Blelloch and J. Greiner, "A provable time and space efficient implementation of nesl," in *In ACM SIGPLAN International Conference on Functional Programming*, 1996, pp. 213–225.