On the Feasibility of a Codelet Based Multi-core Operating System

Jack B. Dennis
MIT Computer Science and Artificial Intelligence Laboratory
Guang R. Gao
University of Delaware

***Abstract*—We believe it is feasible to build a multi-core operating system that implements virtual memory, and honors the principles of modular software construction, using runtime software that implements a codelet program execution model. Performance and energy efficiency can be enhanced through co-design of new architecture features that replace resource management functions of runtime software with efficient hardware mechanisms. The resulting systems will offer benefits in programmability, application portability and reuse absent in current systems.**

I. Introduction

During the past decade, 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. For this, programmers often use MPI, the Message Passing Interface [1], which has become an accepted standard software library for writing high performance application codes.

In this position paper we argue that virtual memory is required to achieve the flexibility of resource management demanded for future applications of massively parallel computer systems. Moreover, a virtual memory implementation is a prerequisite for building systems that can support the general composability of parallel programs [2]. 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. We believe it is feasible to implement a multi-core operating system that will support composability of parallel programs, an achievement that will yield benefits in progrmmability, ease of reuse and portability, as well as performance and energy efficiency.

A. Our Position Statements

- It is feasible to build a multi-core operating system (OS) that implements virtual memory, and honors the principles of modular software construction, using runtime software that implements a codelet program execution model.
- Performance and energy efficiency can be enhanced through co-design of new architecture features that replace resource management functions of runtime software with efficient hardware mechanisms.
- The resulting systems will offer benefits in programmability, application portability and reuse absent in current systems.

B. Synopsis

In Section II, we argue that virtual memory is essential for massively parallel computing in the age of many-core chip technology. Section III explains why a virtual memory implementation is a precondition for efficiently supporting composability of parallel program modules. In Section IV, we explain our claim that it is feasible to build a Multicore Operating System, using codelet based runtime software, that implements a global virtual memory and supports modular construction of general parallel programs. In Section V, we advocate development of an enhanced hardware architecture that will gain performance and energy efficiency by replacing software resource management functions with efficient hardware mechanisms. Finally, we mention some related work and present conclusions.

II. Meeting the Demand for Flexibility

Users of high performance computing wish to run applications that have changing needs for memory and processing from time to time during program execution. This demand arises from several sources: Physics and materials simulations are becoming more sophisticated. Scientists wish to model systems made up of components with different physical properties and irregular shapes; they wish to model what happens at the boundaries between solid, liquid and gaseous phases of substances. Parallel search and classification are now high priority areas for high performance computing. Performing large-scale computation on line graph models of large systems has become important.

This demand for flexible resource management is counter to the conventional wisdom about multi-core systems that has prevailed in recent years. The dominant philosophy is 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 applications

that may be expressed as data parallel computations over a uniform grid of two or three dimensions. Software tools have evolved to assist the application programmer in this task: The HPC languages X10 [3], Chapel [4], and Fortress [5] have been designed and implemented to help users specify the distribution of data and work load over the memory and processing resources of a large parallel computing system. MPI, the popular library of routines for coding computations that fit the bulk synchronous parallel (BSP) model, has become an important standard implementation tool.

Yet these tools do not address the underlying need of new applications for execution time flexibility in resource use. In response to the new demands for high performance computing, thinking about system design has evolved to a new view:

- The 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.

A. Virtual Memory

Introduction of virtual memory in the late 1950s was 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 a 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 this need not be the case, and that massively parallel systems with virtual memory and major programmability benefits can be built.

B. Flexible Resource Management

Two concepts, a unit of memory and a unit of processing, are essential as 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 to a program. A data block may contain references to other data blocks, as, 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 could be 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 block of machine instructions that defines a computation task. We will use the term *codelet* for this concept, as it is becoming widely accepted with this meaning. A codelet is the unit of work scheduled for execution by a processing core of a multi-core computer system. The greatest flexibility in managing resources will be available if any codelet can be executed on any processing core. This property is provided if all cores have the same architecture and instruction set since the codelets are simply memory objects that (with flexible memory allocation and relocatable code) can be moved to any memory location in the system and remain useable. Moreover, the ability to execute any task on any core requires that any input data blocks required by the task be accessible to the codelet.

Thus an essential element of system implemented resource management is that data blocks have *unique identifiers* that may be used to access the data block form anywhere in the system, regardless of where the data block is located. A unique identifer (UID) is assigned to each data block when it is created – memory is allocated for it. There must be a sufficient supply of UIDs to make assignments to as many objects as may exist at any time during system operation. The set of all UIDs forms a space of virtual addresses. A computer system supporting dynamic resource management must provide (in runtime software or in hardware) a mapping of virtual addresses to physical memory locations that may be readily changed as new objects are created and others fall out of use.

For some time, multi-core virtual memory implementations have been available in multi-core servers from Oracle[6] and SGI[7]; however these implementtions use extentions of the memory mapping, paging and TLB hardware that have become standard features of microprocessor chips for many generations. For use in massively parallel high performance systems, these implementations have unacceptable complexity and energy consumption. Fortunately, this need not be the case: we have demonstrated a memory model impelmentation, using trees of fixed size memory chunks, that provides an efficient global virtual memory for all executing tasks in a multi-core system[8].

- Through reconsideration of virtual memory support in the context of data blocks and codelets, an energy efficient implementation of virtual memory with competitive performance can be achieved.

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. Some mechanism is needed to return memory occupied by data blocks to the free memory pool when they are no longer needed by the executing program. One choice is to require that the programmer explicitly release data blocks when they are no longer needed. There are two problems with this plan: The first problem arises because 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 principles of modular program construction[2]. 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 automatic garbage collection implemented by software or hardware.

The fact that the task label is a complete specification of a task for execution makes possible extremely efficient scheduling of tasks. In conventional systems, switching a processor from one task to another is very expensive because the addressing environment of the new task can be different from that of the old task. In a system where data blocks and codelets have globally unique identifiers, all tasks may run in the same addressing environment and starting task execution on a processor is accomplished just by loading the UID of the codelet and the UID of its initial environment of data objects.

III. Composability of Parallel Programs

By *Composability* we mean the ability to use any program module, without any changes, as a component of other program modules. To support composability of program modules, a computer system must honor a set of six principles set forth in [2]. The principles concern the interface between a program module and the program that is making use of it. The interface supported must meet these requirements:

- **Information Hiding Principle:** The user of a module must not need to know anything about the internal mechanism of the module to make effective use of it.
- **Invariant Behavior Principle:** The functional behavior of a module must be independent of the site or context from which it is invoked.
- **Data Generality Principle:** The interface to a module must be capable of passing any data object an application may require.
- **Secure Arguments Principle:** The interface to a module must not allow side-effects on arguments supplied to the interface.
- **Recursive Construction Principle:** A program constructed from modules must be useable as a component in building larger programs or modules.
- **System Resource Management Principle:** Resource management for program modules must be performed by the computer system and not by individual program modules.

Many of these principles follow from the observation of Parnas[9] that the user of a program module must not need to know anything about the internal working of the module to make effective use of it. In particular, the System Resource Management Principle is needed because a program module cannot be aware of resource assignments made for other program modules. The principles are the same whether or not the computer system performs parallel execution of modules.

Without a system implementation of memory management, it would be impractical for a computer system to honor the six principles. Lacking the ability of a program module to ask the sytem to allocate memory, no module would be able to create new data objects. In the absense of automatic garbage collection, abandoned memory space could not be reused because no module can know when no references to an object remain.

- System management of memory in a global virtual memory is essential in a computer system that supports composability of parallel program modules.

IV. Design for a Multi-Core OS

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 three issues must be addressed: (1) how blocks of memory are allocated from a pool of free memory; (2) how blocks no longer needed may be returned to the free memory pool; and (3) an efficient representation for the free memory pool. 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 exist 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 [8] 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.

V. Enhanced Architecture

Implementation of virtual memory through a multi-core Operating System runing 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 to the goal.

A large source of energy ineficiency in current systems is in execution of instructions of the runtime and OS software which do not contribute to progress in performing the computation specified by application code. A large portion of runtime software is devoted to implementation of resource management policies. Replacing these functions of the runtime software with efficient harware mechanisms has the potential to yield major improvements in performance and energy efficiency. The benefits of hardware implementation of resource mangagment functions has been demonstrated in simulations of the Fresh Breeze system architecture [8].

In the choice between implementation of a PXM with runtime software versus implementation with a new hardware architecture, there are advantages and drawbacks to each alternative. The software approach offers flexibility and ease of making changes and improvements, but has limited potential in performance and energy efficiency. Using novel hardware can yield better performance and energy efficiency, but design errors are difficult to repair, and interesting avenues for futher advance may be stiffled.

A best path toward systems meeting the demands of future high performance applictions will likely be to build a multi-core operating system supporting virtual memory and modular software principles, then incorporate the tested design elements into a system architecture with superior programmability, performance and energy efficiency.

VI. Related Work

Partitioned Global Address Space (PGAS) languages such as Co-Array Fortran [16] or UPC (Unified Parallel C) [17] are used to ease the programmers burden by implementing access to remode data objects using global addresses. This enables the programmer to transparently access data objects held in remote memories of a distributed memory system. Every data object has a global address consisting of its local address and the identifier of the processor where the object 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.

VII. Conclusion

The advent of the multi-core era has opened opportunities for exploration of new directions in computer system architecture. We see two paths of development that can lead to systems with programmability advantages never seen before. The first is a codelet based multi-core operating system; the second employs improved processor architecture and memory organization to replace software implementation of resource management functions with hardware mechanisms, thereby realizing additional gains in performance and energy efficiency. Either path will lead to environments for parallel computing in which any parallel program can be used (subject to total resource constraints) without modification, in the construction of new parallel programs.

VIII. Acknowledgement

The authors appreciate the help from members of the Computer Architecture an Parallel Systems Laboratory (CAPSL) at the University of Delaware, especially, Jose M. Diaz and Jaime Arteaga, and collegues at ET International, Inc. who have made publication of this paper possible.

The ideas presented here have grown out of research funded by the DOE Grant *DE-SC0008717*, NSF Grant CCF-1217498 and an AFOSR Grant.

References

[1] M. P. Forum, "Mpi: A message-passing interface standard," Knoxville, TN, USA, Tech. Rep., 1994.

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

[3] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun, and V. Sarkar, "X10: An object-oriented approach to non-uniform cluster computing," *SIGPLAN Not.*, vol. 40, no. 10, pp. 519–538, Oct. 2005. [Online]. Available: http://doi.acm.org/10.1145/1103845.1094852

[4] B. Chamberlain, D. Callahan, and H. Zima, "Parallel programmability and the chapel language," *Int. J. High Perform. Comput. Appl.*, vol. 21, no. 3, pp. 291–312, Aug. 2007. [Online]. Available: http://dx.doi.org/10.1177/1094342007078442

[5] E. Allen, D. Chase, J. Hallett, V. Luchangco, J.-W. Maessen, S. Ryu, G. L. Steele, and S. Tobin-Hochstadt, "The Fortress Language Specification," Sun Microsystems, Inc., Tech. Rep., March 2008, version 1.0.

[6] A. Leon, B. Langley, and J. Shin, "The ultrasparc t1 processor: Cmt reliability," in *Custom Integrated Circuits Conference, 2006. CICC '06. IEEE*, Sept 2006, pp. 555–562.

[7] G. Thorson and M. Woodacre, "Sgi ®UV2: A fused computation and data analysis machine," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, ser. SC '12. Los Alamitos, CA, USA: IEEE Computer Society Press, 2012, pp. 105:1–105:9. [Online]. Available: http://dl.acm.org/citation.cfm?id=2388996.2389138

[8] 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.

[9] D. L. Parnas, "On the criteria to be used in decomposing systems into modules," *Commun. ACM*, vol. 15, no. 12, pp. 1053–1058, Dec. 1972. [Online]. Available: http://doi.acm.org/10.1145/361598.361623

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

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

[15] 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

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

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