An Overview of the NUMAchine Multiprocessor Project*

T. Abdelrahman S. Brown T. Mowry K. Sevcik M. Stumm
Z. Vranesic S. Zhou A. Elkateeb M. Gusat P. Pereira B. Gamsa
R. Grindley O. Krieger G. Lemieux K. Loveless N. Manjikian
G. Ravindran S. Srbljic Z. Zilic

Departments of Electrical & Computer Engineering and Computer Science
The University of Toronto
Toronto, Ontario M5S 1A4

May 9, 1994

Abstract

The NUMAchine multiprocessor project is a large research effort at the University of Toronto aimed to investigate and develop novel software techniques to support efficient parallel computing. An integral part of the project is to design and build the NUMAchine multiprocessor– a large-scale, cache-coherent, non uniform memory access (NUMA), shared memory multiprocessor. The NUMAchine has a number of hardware innovations designed to facilitate our software techniques. This integrated hardware–software approach is the major theme of the project.

In this talk we will present an overview of the NUMAchine project, and will describe the NUMAchine multiprocessor architecture. In particular, we will describe the unique features of the architecture: network caches, cache-coherence protocol, support for block transfers, monitoring capabilities, and the FPGA-based flexible hardware control. Our software techniques will be address fundamental issues in software support for multiprocessors in the areas of operating systems, compilers, and run-time support. We will describe our research and progress in each area.

1 Introduction

The NUMAchine project at the University of Toronto is a research effort aimed at developing a shared-memory multiprocessor architecture and software support for easy and efficient use of this architecture. A key objective is to define an architecture that is: modular, cost-effective, and scalable to a size of 1024 processors. A prototype 64-processor machine will be built and system software developed to allow programmers to transport their applications onto this multiprocessor with ease, and exploit the full potential of the hardware. The project builds on our recent experience in developing the Hector multiprocessor system [1]. It is funded by a Strategic Grant from the Natural Sciences and Engineering Research Council of Canada.

*This research has been supported by NSERC Strategic Research Grants STR0149404 and STR0149815.

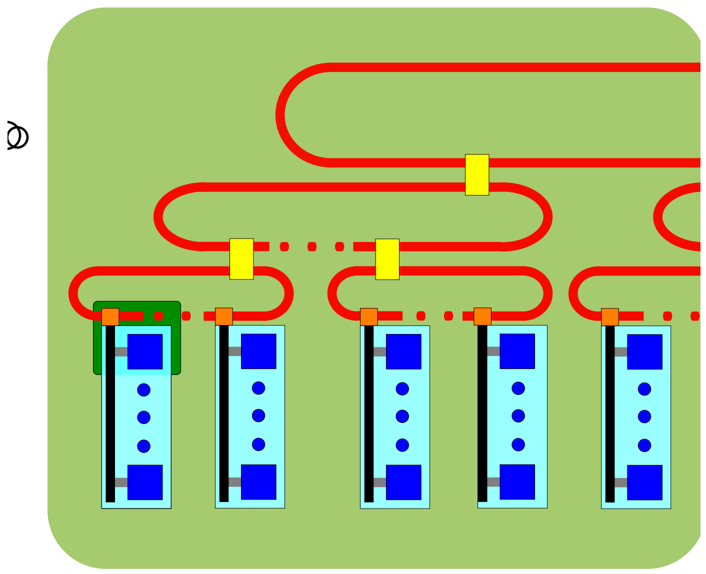
Figure 1: The NUMAchine Hierarchy.

2 Architecture

The NUMAchine architecture uses a hierarchical ring structure, which was demonstrated to be attractive in the Hector project. The architecture features include: cache coherence scheme based on limited broadcasts, network caches, block transfer support, monitoring capability, and flexible control that allows more than one protocol to be used.

2.1 Hierarchical Structure

At the lowest level of hierarchy, processor modules are connected into small clusters, called *stations*. A simple bus is used for interconnection within a station, which is the most effective choice when only a small number of functional units is involved. The stations are interconnected through a hierarchy of bit-parallel rings, as illustrated in Figure 1. The ring structure provides some important advantages. Point-to-point connections are used between successive nodes, resulting in simple interfaces because the ring connects to a given node by means of only one input and one output port. Such connections avoid loading and signal reflections from multiple connectors which plague bus-based schemes and limit the number of processors that can be supported. Signals can be transmitted reliably at very high clock rates on rings.

The hierarchical structure permits a number of localized transfers to take place concurrently on several rings, thus providing high total bandwidth. The highest transfer rates are achieved if most transfers are between stations on the same ring (i.e. with high locality). The longest transfers are those that traverse all levels in the hierarchy, but even these transfer times are in general shorter than they would be if all nodes were connected to a single long ring. A particularly useful feature of the ring-hierarchy is that there exists a unique path between any two nodes in the system. This feature can be exploited effectively in providing hardware support for cache coherence.

The memory is distributed across the stations in the system. However, the user sees a single coherent view of the memory, because all processors can access all memory locations in the same way. The time needed to access a given memory location depends upon the physical place of this memory in the hierarchy with respect to the accessing processor. Since the access times vary, the entire architecture falls into the class of NUMA (Non-Uniform Memory Access) multiprocessors.

The NUMAchine prototype will be a 64-processor system, organized into a 3-level hierarchy. One central ring will be used to connect four local rings, each of which will have four stations of four processors each.

2.2 Station

Each station may contain a small number of processors. For practical reasons, this number should be limited to eight. For our NUMAchine prototype, a station will have four 64-bit processors. Figure 2 shows the organization of a station. Each processor has its own cache (perhaps both primary and secondary caches). A single memory unit is provided on the station.

A *network cache* is included to reduce the traffic in the communications network. It is used to cache data from non-local memory modules (but not the data from the local memory). Since this cache only needs to appear to be fast compared to non-local memory modules (which involve transfers through the communications network), it can be built using the same DRAM technology as used for the local memory.

The *station interface module* contains the control circuitry and buffers needed to connect to the local ring. In order to eliminate problems with flow control, the communications protocol gives priority to higher levels in the hierarchy. Therefore, a packet on the local ring is always accepted by a destination station. A station can place a packet on the local ring only when it sees an empty slot (in a slotted-ring protocol).

An *I/O module* provides interface to input/output devices. At least one disk storage unit will be included in each station.

2.3 Cache Consistency

NUMAchine has hardware-supported cache consistency, based on the natural broadcast mechanism available in the hierarchical structure and the fact that there is a unique path between any two nodes in the machine. Since any full broadcast scheme does not scale well, our protocol limits the scope of broadcasts by means of a packet-filtering mechanism. Simple bit-masks are used to keep track of positions in the hierarchy of various copies of cache lines. Each bit-mask is kept at the memory module that is the home location of the corresponding cache line. The bit-masks provide addressing information for reaching the copies in various caches (e.g. during invalidation process). At each node in the network, a filter circuit examines a single bit in the bit-mask field in the packet and decides if a copy of the broadcast packet should be sent to the lower level in the hierarchy.

The cache consistency scheme can be implemented at low cost. It needs very simple routing control, and it performs well. We believe that it will prove to be superior to the well-known directory schemes.

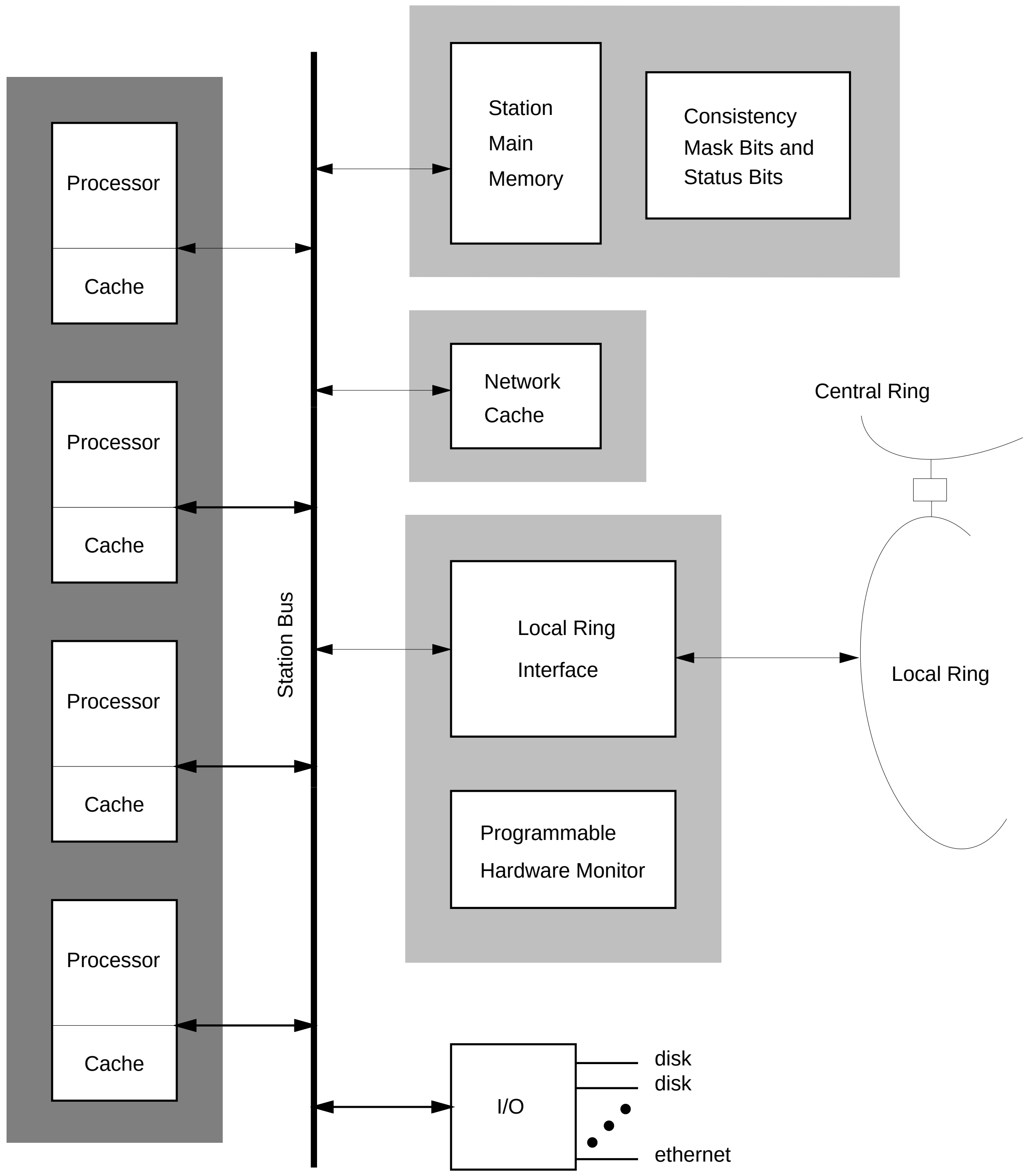

Figure 2: Block Diagram of a Station.

2.4 Performance Monitoring

Since our NUMAchine prototype hardware will be used as a research vehicle for both software and architecture investigations, it is important that there exist facilities for measuring its performance. Consequently, NUMAchine will include hardware that provides for non-intrusive monitoring of different aspects of the system's behavior. The monitoring is reconfigurable, based on high-capacity programmable logic device technology.

2.5 Practical Considerations

One of the key advantages of NUMAchine architecture is its modularity. A small system can comprise a single-ring (or even a single-station) configuration. When a larger machine is needed, it can be easily realized by adding a higher-level ring, etc. Thus, small configurations are not penalized by an up-front cost, and the cost essentially grows linearly as additional modules are included.

Scalability is an important factor. In a hierarchical system there will arise problems if a constantly large amount of traffic must pass through the highest level in the hierarchy. Also, it is clear that any protocol that uses full broadcasts will not scale well. We have solved the broadcast problem by means of the filtering mechanism. Our simulations indicate that the proposed architecture will scale well at least up to 1024 processors. This is likely to be the range of most commercially sold multiprocessors in the next decade.

All parts of NUMAchine hardware are based on standard workstation technology. This, combined with the modular structure, allows a very cost-effective implementation.

3 Tornado Operating System

Tornado is a multiuser, NUMA-aware operating system being developed for NUMAchine. It is a performance-oriented, microkernel-based system, where most services are provided by servers and application-level run-time libraries. Tornado has a highly modular structure and is implemented in C++, making it well suited as a test bed for experimental operating systems research. The programming model provided to the application programmer includes multiple light-weight processes per address space and a file system based on mapped files.

The Tornado development effort will build on our recent experiences in implementing the Hurricane operating system for the Hector multiprocessor [2]. Being able to temporarily use some of Hurricane's servers under Tornado will allow us to quickly exercise Tornado code as it becomes available, and will provide users of NUMAchine an operating environment complete with internet access and X-Window library support shortly after the hardware becomes operational.

Tornado will pursue a number of objectives that set it apart from other operating system efforts. Three of these include scalability, predictable performance behavior, and support for data-intensive applications:

- **Scalability**: An operating system must be as scalable as the underlying hardware if the performance potential of the hardware is to be exploited in a cost-effective way. For shared-memory multiprocessors, scalability entails both concurrency and locality. However, existing operating systems for large-scale multiprocessors have been scaled to accommodate many processors only in an *ad hoc* manner, by repeatedly identifying and then removing the most contended bottlenecks, thus addressing only concurrency. This approach results in systems that *i*) have a large number of locks that have to be held for common operations, easily doubling the cost of these operations, and *ii*) do not take locality into account.

- **Predictable performance behavior**: Tornado is a multiprogrammed system rather than a uniprogrammed or partitioned system, because a multiprogrammed system makes more effective use of the hardware resources, and because it can provide improved response times to users. However, application performance is less predictable in a multiprogrammed system, because concurrently executing applications compete for common resources. The fact that the physical resources allocated to an application may change at any time makes it difficult for the programmer or compiler to tune a parallel program to run well, or for the run-time system to appropriately parameterize the algorithm.

  As a simple example, a program assuming the availability of P processors will perform poorly if less than P processors are available during execution. While the availability of P processors can easily be achieved for example with co-scheduling, we believe that guaranteed resource availability during program execution is also necessary for the other physical resources such as physical memory and I/O bandwidth if high performance is to be achieved. As another example, if an application prefetches from disk a large data set, then increased paging activity with an attendant degradation in performance can result if sufficient physical memory is not available at that time. Optimizations, such as prefetching in this example, are effective only if the resources they require are actually available; otherwise they are detrimental.

  Support for data intensive applications: Tornado will be tuned for the class of applications with large data sets and hence high I/O demands. Few systems support this class of applications well; we have found it easy to be able to cripple most existing parallel systems with even simple I/O-bound programs. Providing a high capacity I/O system alone is not sufficient. A more integrated approach is required where the file system, the memory management system, the scheduler, and the application program cooperate in order to be able to efficiently exploit the potential bandwidth of a parallel I/O subsystem.

A number of techniques can help achieve these objectives. We briefly describe three.

- **Hierarchical Clustering**: Hierarchical clustering is a structuring technique that addresses scalability. In a clustered system, a set of resources of one type (virtual or physical) that correspond to a cluster of processors are managed together in a tightly-coupled and semi-autonomous fashion. The data structures used for this purpose are constrained to remain local to the cluster and to be accessed only by the processors of the local cluster. The basic idea behind clustering is that individual easy to implement modules provide a small degree of concurrency with simple locking structures and much locality, while multiple modules are used to cover the entire system, providing the concurrency required. The clusters interact in a loosely-coupled fashion to provide the applications with the view of a single, integrated and consistent large system. The size of the cluster is determined largely by the expected workload, the expected degree of sharing, and the topology of the underlying hardware.

Clustering incorporates structuring principles from both tightly-coupled and distributed systems, exploiting the advantages of both. By using the structuring principles of distributed systems, the services and data are replicated and migrated to: a) distribute the demand, b) increase concurrency, c) decrease contention, and d) increase locality, thus making the system scalable. On the other hand, there is tight coupling within a cluster, using the structuring principles of small-scale multiprocessor operating systems, so the system can perform well for the common case where interactions occur primarily between objects located in the same cluster.

The Hurricane operating system was originally implemented as a clustered system, and to the best of our knowledge, is the first and only such system. Tornado will also use this structure, although modified to accommodate the lessons we have learned from our experience. For example, our existing implementation supports only two levels of hierarchy: C/P clusters of size C span the system. Moreover, the cluster size is the same for each resource type. Tornado will support additional levels of hierarchy and will allow each resource class to have an independent cluster size. In Hurricane, we originally underestimated the importance of restraining data accesses to a single processor whenever possible. In Tornado, the clusters at the lowest level of the hierarchy will mostly be of size one in order to increase processor locality and hence maximize the cache hit rate, minimize cache invalidations, and minimize the need for locking.

We have designed and implemented a Protected Procedure Call (PPC) facility that nicely demonstrates the advantages of using hierarchical clustering. The PPC facility requires no access to shared data and no locking, so end-to-end performance is comparable to the fastest uniprocessor IPC times, yet can sustain this performance independent of the number of concurrent PPC operations, even if all such operations are directed to the same server [3]. Moreover, the PPC facility enables servers to be structured using the same approach by ensuring that local resources are made available to the target server so that it can equally exploit locality as much as possible.

- **Physical Resource-based Virtual Machine**: Tornado will guarantee physical resource availability to each application, and will allow each application to determine how these resources are to be managed. It will support a model in which each parallel application negotiates for a set of physical resources. Once granted, the application has exclusive use of those resources throughout the course of the computation or until renegotiated. At this time, we envision negotiating for a fixed set of processors, physical memory pages and a portion of the I/O bandwidth to specific disks. (It is still an open question how best to treat backplane traffic.) Processes of an application will be scheduled only if its physical resources are available, and their state will be restored to what they were when the application last ran. Moreover, Tornado will allow the resources of each physical resource-based virtual machine to be managed in an application-specific way.

These two features, guaranteed resource availability and application-specific resource management, will make the performance behavior of applications predictable, regardless of other concurrently running applications, and hence simplify the performance tuning process significantly. The application appears to have a physical machine to itself, running in virtual time. Hence, the system will also respond well to programs designed for uniprogrammed systems. Moreover, the system will allow valid run-time measurements for performance evaluation purposes (for example to obtain speedup curves) without having to reserve the hardware for single-user access. The challenge is to support this model while at the same time ensuring that throughput remains at an acceptable level. We expect the potential performance benefits of being able to properly tune an application to be significantly larger than the overhead caused by supporting this system model.

- **User-level I/O support**: We have designed and implemented a parallel file system under Hurricane capable of supporting concurrent access to numerous disks distributed across the system [4]. In principle, this file system is capable of providing very large aggregate I/O bandwidth. In practice, however, it is difficult to effectively exploit this bandwidth, unless the application, the virtual memory system, the scheduler, and the file system collaborate closely. Tornado intends to provide I/O primarily through mapped files, because of its simpler interface, its natural support for parallel operation, and the reduced amount of copying and address space crossings required. But to provide high-performance I/O, it is necessary to supplement the system-level servers with application-level support.

  In previous work [5], we have been able to improve I/O performance of uniprocessor applications by moving much of the I/O functionality into the application space to reduce the number of address space crossings and the amount of data copying. We feel that application-level support is even more important in the case of data-intensive parallel applications. Overall, we believe that a proactive approach to I/O as opposed to a reactive one will lead to much improved performance. For example, application-level prefetching and post-storing is necessary to keep the disks busy and streaming. Moreover, the application must ensure that sequential access patterns are seen by the disk as much as possible, possibly by assigning exclusive access to the disk to one process at a time.

4 The NUMAchine Compiler

We are currently developing a compiler system for NUMAchine. The compiler is intended to support high-performance data-parallel programming for large-scale scientific applications. It specifically targets shared memory multiprocessors with non-uniform access memory systems and coherent caches. It will optimize programs to maximize cache utilization, enhance locality of reference in a physically-distributed shared memory, reduce data movement and cache coherence overhead, and reduce network contention. These goals distinguish our work from other compiler research efforts such as Parafrase II [6], Polaris [7], Fortran-D [8], SUIF [9], and Superb [10], that either target uniform memory access multiprocessors, and are mainly concerned with the detection of parallelism and the formation of parallel tasks, or that target distributed memory multiprocessors, and are mainly concerned with communication optimizations.

Our compiler will include a number of techniques that: (1) enhance locality of reference; (2) reduce memory access latency; and (3) hide remaining latency. We will utilize the prefetching and block transfer capabilities of the hardware to support our techniques.

4.1 User-Level Locality Management

For a large class of applications, user-level specification of data mapping is essential to maintain locality and reduce false sharing. We will provide support for the specification of data mapping through High Performance Fortran (HPF) language extensions for data partitioning and distribution. These extensions enable our compiler to partition arrays and to distribute the partitions in the physically distributed shared memory. A critical performance-related issue is the implementation of array indexing and storage management for the resulting distributed arrays in the presence of sharing. A possible approach is that of Fortran-D in which all program references to arrays are translated into local references to array partitions for each processor, and in which buffers are used to store shared data. However, this approach adds complexity to the compiler and does not result in good performance. Instead, our compiler will take advantage of NUMAchine's shared memory architecture to efficiently implement array indexing and storage management [11]. We will implement and evaluate the performance of a number of possible implementations in our compiler. We also intend to support irregular and dynamic data distributions– the shared address space of NUMAchine will facilitate more efficient implementation of these distributions in comparison to distributed memory multiprocessors.

In addition to providing a mechanism for locality specification, our support for HPF language extensions will facilitate porting to the NUMAchine applications that use these extensions.

4.2 Automatic Data Layout

Data layout encompasses the placement, organization, and possible movement of data in a physically distributed shared address space. We believe that data layout must take into account low-level issues such as storage management and the role of caches in order to maximize parallel execution performance. Locality is determined by the manner in which application data is partitioned and placed through the storage management facilities of the operating system. Effective cache utilization requires proper data organization with respect to the data access patterns. Finally, reduction of communication and coherence overhead arising from data sharing during parallel execution requires minimization of data movement. Automated derivation of data layout requires adequate support for: analysis and representation of data access patterns, evaluation and selection of data layout alternatives with consideration of low-level issues such as cache effectiveness, and transformations of computation and data required to enforce the chosen data layout, including interactions with the storage management facilities of the operating system. We will investigate these issues and implement automatic data layout support in our compiler.

4.3 Latency Hiding by Block Transfers

The block transfer facility in the NUMAchine hardware provides compilers with the opportunity to hide memory access latency by overlapping communication time to access remote data with local computations. In order to realize this overlap, a program segment must be restructured to initiate block transfers of remote data, proceed to perform computations that require only local data, and then upon receipt of the remote data perform computations that require the remote data. There are a number of open research issues that must be addressed before effective use of block transfers can be possible. These include: (1) analysis and representation of data access patterns in parallel applications to determine local and remote data accesses; (2) identification of program segments that can benefit from block transfers (the use of block transfers can degrade parallel program performance due to the overhead associated with block transfers, and hence, benefits of block transfers must be ascertained before they are used); and (3) restructuring of programs to initiate block transfers and receive block transferred data. We will research these issues and develop appropriate support for block transfers in our compiler.

4.4 Loop Scheduling

Loop scheduling assigns iterations of a parallel loop to processors for parallel execution. It must strike a balance between two conflicting goals: to minimize run-time overhead of scheduling decisions, and to balance the workload across the processors. On NUMAchine, the non-uniform nature of memory access dictates yet another goal– to minimize data movement and cache coherence overhead. The scheduling of an iteration must take into consideration the location of the data referenced by the iteration. We have developed in the run-time system of Hurricane a dynamic loop scheduling algorithm, called LDS [12], which relies on the user to provide locality information. We will continue to pursue locality-based scheduling by investigating loop scheduling techniques that assign iterations to processors at compile-time in such a way to minimize data movement and cache coherence overhead. We will use the techniques we develop to describe data access patterns in parallel applications to estimate the number and cost of memory accesses in a parallel loop, and use this estimate to determine a loop scheduling strategy that strikes a good balance among the three goals.

4.5 Organization of the Compiler System

A block diagram of the compiler system we will develop is depicted in Figure 3. It consists of a front end which parses Fortran and C programs to produce an intermediate representation. Control flow analysis and data flow analysis are then applied to the intermediate representation to perform common optimizations such as constant propagation, copy propagation and common subexpression elimination. We have integrated the Omega test [13] into the compiler to perform exact and symbolic data dependence analysis. We have also implemented a number of standard transformations, such as unimodular transformations, strip-mining, tiling, loop distribution and loop fusion. The transformations are implemented in a modular fashion to facilitate the integration of new transformations. Data layout is performed after a set of primary transformations have been applied, and may require additional transformations to be applied. Once data layout is determined, latency hiding techniques are applied. Loop scheduling is then applied and a parallel source program is generated by the back end. The resulting program is then compiled using the native compilers of the target machine to generate an executable parallel program.

References

[1] Z. Vranesic, M. Stumm, D. Lewis, and R. White, "Hector: A hierarchically structured shared memory multiprocessor," *IEEE Computer,* vol. 24, no. 1, pp. 72–79, 1991.

[2] M. Stumm, R. Unrau, and O. Krieger, "Hierarchical clustering: A structure for scalable multiprocessor operating system design", CSRI Technical Report CSRI-268, 1993. (Extended version of "Clustering Micro-Kernels for Scalability," Proc. of the Usenix Workshop on Micro-Kernels and Other Kernel Architectures, April, 1992.

[3] B. Gamsa, O. Krieger, and M. Stumm, "Optimizing IPC performance for shared-memory multiprocessors," CSRI Technical Report CSRI-294, 1994, (submitted for publication).

[4] O. Krieger, *The HFS parallel file system*, PhD thesis, in progress, expected 1994.

[5] O. Krieger, M. Stumm, and R. Unrau, "The Alloc Stream Facility: A redesign of application-level Stream I/O," *IEEE Computer*, 27(3) March, 1994, pp. 75–82.

[6] C.D. Polychronopoulos, M.B. Girkar, M.R. Haghighat, C.L. Lee, B.P. Leung and D.A. Schouten, "The structure of Parafrase-2: and advanced parallelizing compiler for C and Fortran," in *Languages and Compilers for Parallel Computing,* D. Gelenter, A. Nicolau, and D. Padua, eds., pp. 423–453, The MIT Press, 1990.

[7] D.A. Pudua, R. Eigenmann, J. Hoeflinger, P. Petersen, P. Tu, S. Weatherford and K. Faigin, "Polaris: A new generation parallelizing compiler for MPPs," CSRD Report No. 1306, Center for Supercomputing Research and Development, University of Illinois, 1993.

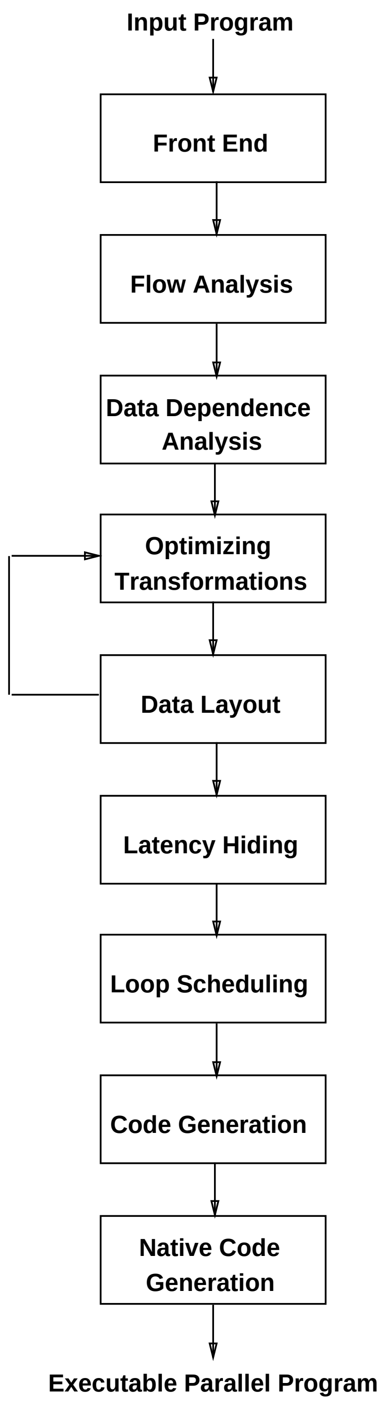

Figure 3: A Block Diagram of the Proposed Compiler System.

[8] S. Hiranandani, K. Kennedy and C. Tseng, "Compiling Fortran D," *Comm. of the ACM,* vol. 35, no. 8, pp. 66–80, 1992.

[9] S. Tjiang, M. Wolf, M. Lam, and K.Pieper, "Integrating scalar optimization and parallelization," in Languages and Compilers for Parallel Computing, pp. 137–151, Springer-Verlag, Berlin, 1992.

[10] H. Zima and B. Chapman, "Compiling for distributed memory multiprocessors," *Proceedings of the IEEE,* 1993.

[11] T. Abdelrahman and T. Wong, "Distributed Array Data Management on NUMA Multiprocessors," *Proc. Scalable High-Performance Computing Conference,* 1994.

[12] H. Li, S. Tandri, M. Stumm and K. Sevcik, "Locality-Based Dynamic Scheduling," *Int'l Conf. on Parallel Processing,* 1993.

[13] W. Pugh, "A practical algorithm for exact array dependence analysis," *Comm. of the ACM,* vol. 35, no. 8, pp. 102–114, 1992.