A Streaming Supercomputer

Bill Dally, Pat Hanrahan, and Ron Fedkiw
September 18, 2001

1 Introduction

1.1 We are starving in an era of plenty

We are in an era where computational building blocks are plentiful and inexpensive. A single chip today can hold over 100 1GHz floating-point units for a total performance of 100 GFLOPS/chip. Many graphics chips achieve 80GFLOPS and over 1TOP rendering performance, and cost less than $100. Embedded processors are less powerful, but incredibly cheap. It is fair to say that a raw GFLOPS costs less than $1. Memory is currently selling for less than 20 cents a MByte. Bandwidth has become less expensive as well. Chips with a Tb/s of aggregate bandwidth have recently been demonstrated.

In this era of plenty, however, we have not developed technology to cost effectively scale computing. Supercomputers cost significantly more per GFLOPS and GByte than their low-end counterparts. For example, it is estimated that total cost of future large-scale ASCI machines with 10's of thousands of nodes is greater than $1,000 per GFLOPS. This factor of a 1000:1 in cost effectiveness is paradoxical: it should be possible to reap economies of scale with computing, just as in other major acquisitions. Although scalability has long been a focus of computer science research, it has not been transferred into practical commercial systems. Now more than ever we need to build the technological infrastructure to cost-effectively scale computation.

In addition to being cost inefficient, contemporary high-end computers, constructed from clusters of workstations or servers, do not deliver their promised performance. They achieve a small fraction of peak performance on many key applications that are dominated by global communication. Critical calculations, such as verifying nuclear weapons, performing signal intelligence, calculating the dynamics of protein folding, and fluid flow through complex turbomachinary, do not map well to these machines.

The performance of the microprocessors from which these clusters are composed is no longer scaling at the historic rate of 50% per year. Microprocessors have reached a point of diminishing returns in terms of gates per clock and clocks per instruction. As we enter an era of billion transistor chips, there is not enough explicit parallelism in conventional programs to efficiently use these resources. For example, a modern graphics processor has at least 64 floating point ALUS and 1000's of integer ALUs, almost a hundred times the arithmetic density of a microprocessor. In contrast, most of the chip area in a microprocessor is devoted to cache memory or the support infrastructure (e.g. supporting out-of-order execution) to keep a few ALUS running at their peak clock rate. It is expected that without new innovations in parallel processor designs, microprocessor performance will only increase with the increase in gate speed, at a rate of about 20% per year. Such as change would have a major effect on the computer business, and the entire economy.

Cluster supercomputers, like the microprocessors they are constructed from, are inefficient because they are poorly matched to the technology from which they are constructed and the applications which they run. They are unable to efficiently exploit the large numbers of floating-point units that can be fabricated on a chip. They also have low global bandwidth and have register and cache architectures that do not capture large amounts of application locality and hence make excessive demands on this bandwidth. Because these systems are not well-designed, they are difficult to program. Programmers spend all their time working around the limitations of the machine, rather than on developing efficient algorithms for their application.

1.2 Streaming processors leverage emerging technology

Recent developments enable streaming architectures that efficiently convert the capabilities of emerging technology into realized performance on scientific applications. We envision a streaming supercomputer that delivers orders of magnitude more performance per dollar than clusters of servers ($50 per GFLOPS

and $2 per million GUPS[1]) and is scalable to a machine with one PFLOPS of peak performance and 10^{13} GUPS. We expect that applications will achieve a large fraction of the peak FLOPS (at least 25%) on arithmetic-limited code sections and a large fraction of peak GUPS on memory-limited code sections.

Streaming supercomputers with this level of performance and efficiency are made possible by the confluence of three recent innovations: stream architecture, high-speed signaling, and efficient interconnection network architecture. Stream architectures expose and exploit parallelism and locality in applications. This in turn enables architectures with a high degree of *arithmetic intensity*, that is applications with a high ratio of arithmetic to memory bandwidth. Streams also offer an easy way to hide the inherent latency of global memory references. Stream architectures have been proven on signal- and image-processing applications. Our initial investigations show that they are equally applicable to a broad class of scientific applications. High-speed signaling and efficient network architecture together enable economical memory systems with very high global bandwidth.

The streaming architecture we envision leverages commodity technology to economically achieve high performance. It does not, however, use commodity processors. Processors, in fact, are not really a commodity - they are not interchangeably available from multiple vendors at prices close to cost. Commodity technology is leveraged in three ways. First, the main memory of the system is built entirely out of commodity high-bandwidth memory chips. Such memory chips truly are a commodity - they are available in volume from a number of different suppliers at competitive prices. Second, the streaming processor chips are fabricated using a standard CMOS process. As with memories, CMOS wafers are a commodity - being available in volume from multiple vendors. Finally, the system interconnect is constructed using off-the-shelf connectors and backplane technology.

Realizing the performance of a streaming supercomputer, of course, requires recoding applications in a streaming style - as *streams* of records passing through networks of arithmetic *kernels*. Coding applications in this style makes communication explicit, making it easy for the software tools to efficiently map the application to a streaming architecture.

1.3 Domain-specific languages simplify mapping problems to streaming supercomputers

We envision a three-level programmign system that will simplify the mapping of applications to a stream architecture and at the same time make the resulting code more portable. At the top level, several domain specific languages will target specific classes of applications, e.g., Monte-Carlo integration, ODEs, PDEs, etc.... Each of these languages will enable a scientist to describe their problem's equations, its geometry, constraints on its solution, and solution methods. A compiler then uses this description to map the problem to a streaming programming model. Domain specific languages have proven successful in many applications. In particular, graphics shading languages have been effective in describing complex shading calculations in terms of high-level primitives and mapping these calculations to a variety of hardware, including stream processors.

The target of the domain-specific language compiler is a stream language that describes the application in terms of streams of records passing through kernels of computation. We envision generalizing streams so they can describe not just linear sequences of records but also unordered collections of records, higher-dimensional arrays of records, and arbitrary graph structures (e.g., to describe a finite-element mesh). These collections will be operated on by kernels that *map* a function over the collection, *filter* the collection, selecting certain elements, *expand* a collection, producing several results for each input, or *reduce* a collection, combining several inputs into a smaller set - or even a single - result. By describing the application at an abstract stream level, this stream language will be completely hardware independent but yet will expose available parallelism and locality. A stream compiler will accept such a stream program and a machine description and generate output in our low-level programming language.

The output of the stream compiler is a program in a low-level stream language. In addition to streams, this language includes constructs to describe DSP and SIMD operations, threads and synchronization, and memory management. We anticipate writing several back-ends for the low-level stream compiler

[1]A GUPS (global updates per second) is the number of single word memory references to random locations across its entire memory space that a machine can support per second.

that will enable us to map programs not only to a streaming supercomputer, but also to conventional hardware.

1.4 Paper outline

The remainder of this paper describes our vision of a streaming supercomputer in more detail. We start by sketching the architecture of a streaming supercomputer in Section 2. Section 3 describes our vision of a three-level programming system in more detail. Applications and the domain-specific languages to describe them are discussed in Section 4. Finally, we outline a plan to accomplish this research in Section 5.

2 Architecture of a Streaming Supercomputer

In this section we sketch a possible architecture of a streaming superocomputer to demonstrate the feasibility of this approach. There are many details that remain to be worked out and many parameters and ratios may change by as much as a factor of two. However, the sketch here demonstrates the feasibility of a machine of the class we propose.

2.1 Overview

Figure 1 shows a block diagram of a streaming supercomputer. Each node contains a streaming processor with 64 1-GHz floating-point units (FPUs) and a local memory with 16 1Gb DRDRAM chips with a bandwidth of 2.4GB/s each[2]. The local memory capacity is 2GBytes and the local memory bandwidth is 38GB/s. Each node has a 20GB/s channel to the global interconnection network. Nodes can sustain simultaneous accesses to the memory of adjacent nodes at this rate - half the local memory bandwidth.

The global network enables any processor to access any memory location in the system. The network has a bisection bandwidth of $4N$GB/s, that is 4GB/s *per node*. Thus, each node can simultaneously sustain accesses to global memory at greater than 10% of the local memory bandwidth of the node. We expect that a global memory access in a N =16,384 node machine, including a round trip over the global network and remote memory access time will have a total latency of less than 500ns - 500 processor cycles.

To sustain full global bandwidth - 4GB/s or one word every two 1ns cycles - while tolerating this latency, the processors make streaming memory references- stream loads and stream stores. A *stream load* operation loads a stream of records. The individual records may be addressed with unit-stride, arbitrary-stride, or indexed addressing modes. An indexed stream load gathers individual records (possibly as small as a single word) from arbitrary global locations. A single stream load can request thousands of multi-word records, more than enough to fill the 250-word deep memory pipeline. By fetching contiguous multi-word records, rather than individual words (like a vector load), stream loads result in more efficient access to modern memory chips.

We plan to package 16 nodes of the streaming supercomputer on a circuit card measuring 300mm by 400mm. This card will contain 16 processor chips, 256 DRAM chips, and will have a peak performance of 1TFLOPS[3]. Each cabinet will hold 64 of these cards, in 4 rows of 16, along with associated power supplies and cooling for a total of 1K nodes with 64TFLOPS and 2TBytes of memory per cabinet. Machines larger than 1K nodes are assembled by cabling cabinets together with optical fibers. We anticipate being able to scale the machine to 16K nodes (1PFLOPS) while maintaining our global latency and global to local bandwidth ratio.

The major properties of the streaming supercomputer we envision are summarized in Table 1.

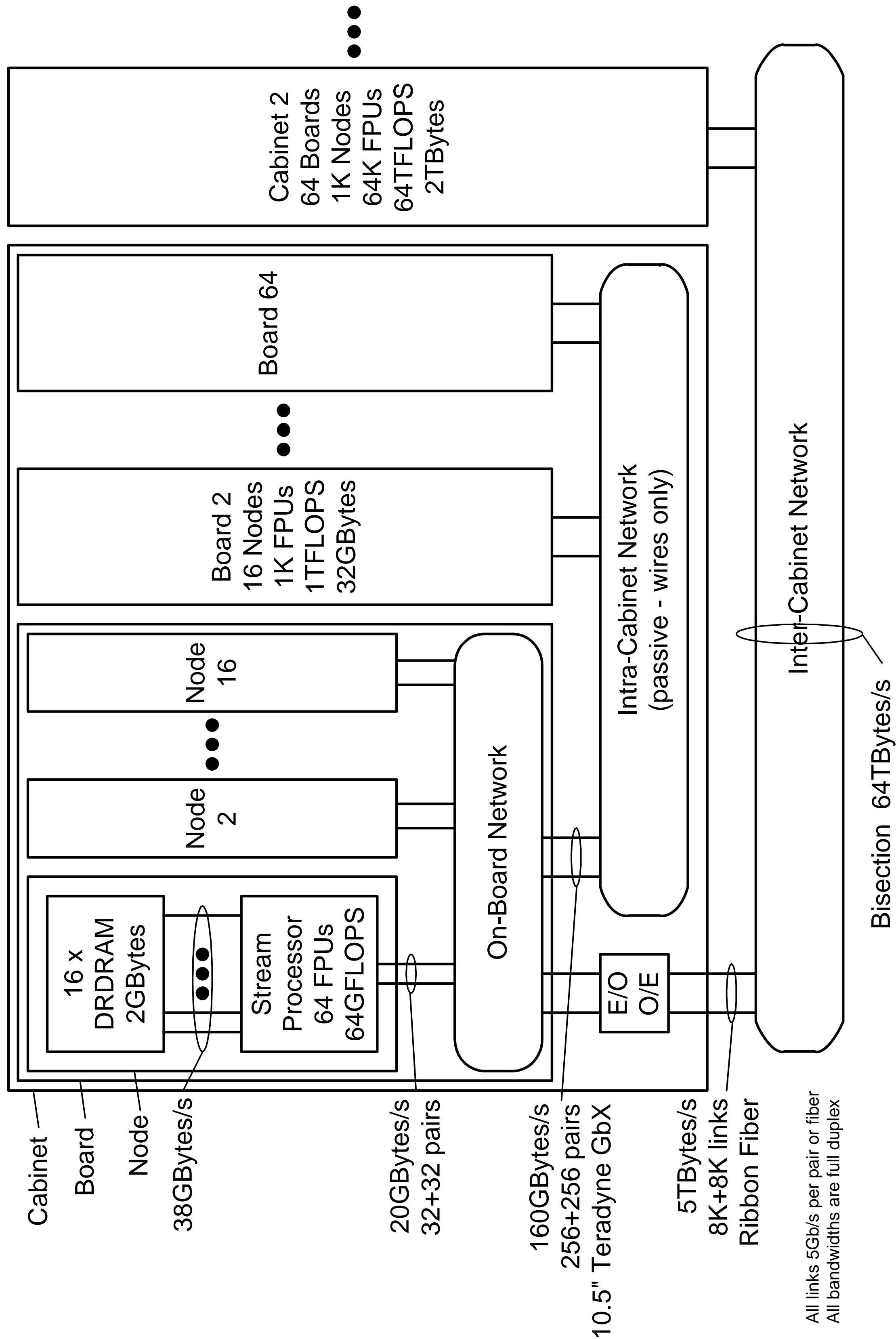

Figure 1: Block diagram of a streaming supercomputer

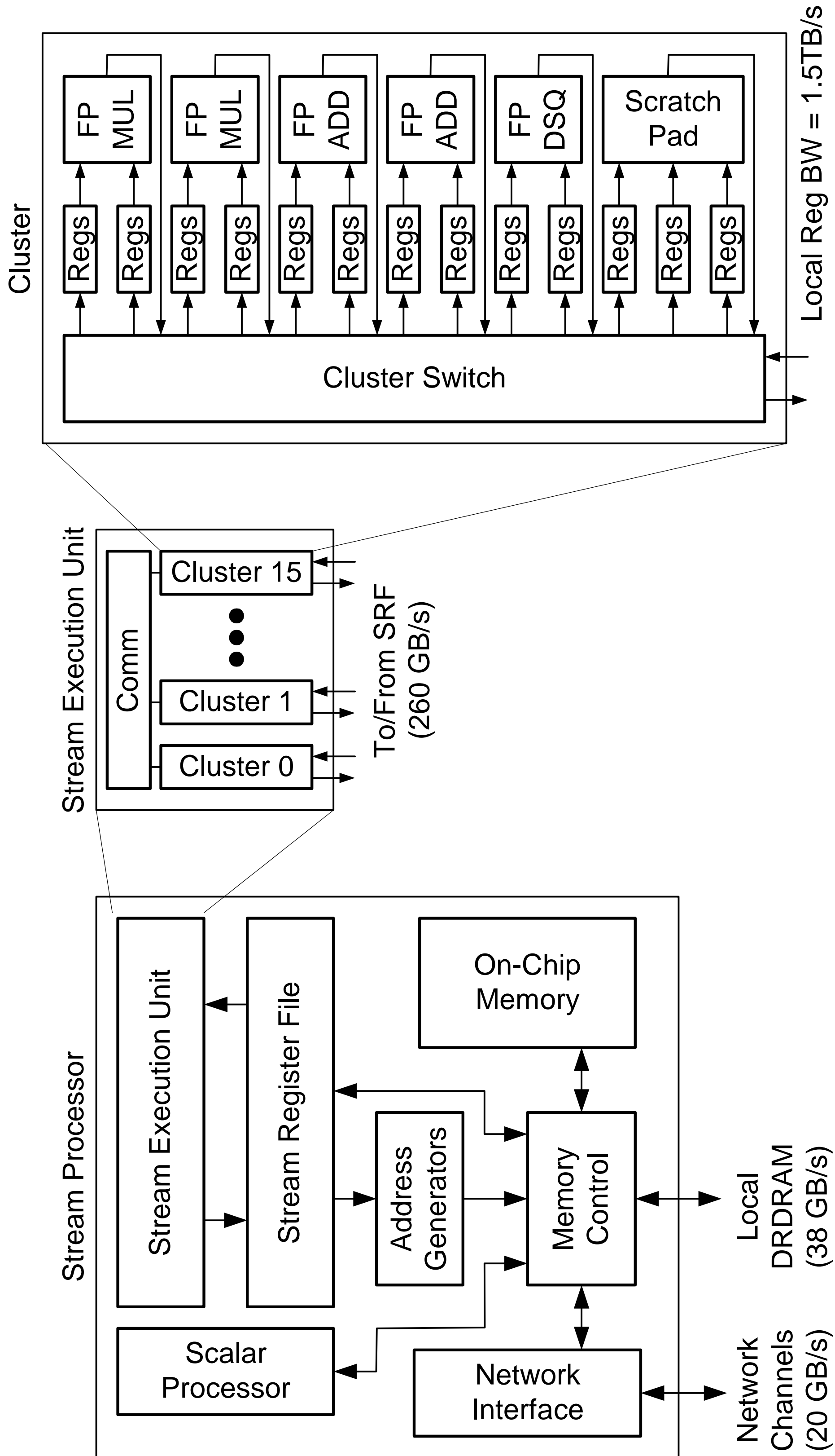

Figure 2: Block diagram of a streaming processor

Parameter	f(N)	N=4,096	N=16,384	Units
Memory Capacity	$2 \times 10^9 N$	2.8×10^{12}	3.3×10^{13}	Bytes
Local Memory BW	$3.8 \times 10^{10} N$	1.6×10^{14}	6.3×10^{14}	Bytes/sec
Global Memory BW	$3.8 \times 10^9 N$	1.6×10^{13}	6.3×10^{13}	Bytes/sec
Global Memory Accesses	$4.8 \times 10^8 N$	2.0×10^{12}	7.9×10^{12}	GUPS
Peak Arithmetic	$6.4 \times 10^{10} N$	2.6×10^{14}	1.0×10^{15}	FLOPS
Processor Chips	N	4,096	16,384	
Memory Chips	$16N$	6.6×10^4	2.6×10^5	
Boards	$N/16$	256	1,024	
Cabinets	$N/1,024$	4	16	
Power (est)	$50N$	2.0×10^5	8.2×10^5	Watts
Parts Cost (est)	$1 \times 10^3 N$	4×10^6	1.6×10^7	2001 Dollars

Table 1: Properties of proposed streaming supercomputer as a function of the number of nodes N, and for N =4,096 and N =16,384

2.2 Streaming Processor

Figure 2 shows a high-level view of the processor chip that provides the arithmetic capability of the streaming supercomputer. The core of the processor is a stream execution unit containing 64 64-bit floating-point units[4]. The stream execution unit also contains 4,096 64-bit *local* registers, 32 on each input of each arithmetic unit, and 8,192 64-bit *scratch-pad* registers for holding intermediate results of arithmetic kernels. The stream execution units read and write streams from a 32K word stream register file that stages stream data to and from memory. The stream execution unit is controlled by stream-operate instructions each of which causes a small subroutine to be executed on each element of the input streams to generate each element of the output streams.

A pair of address generators execute stream load and store instructions to transfer streams between the stream register file and the memory system. The local portion of the memory system is contained on the processor chip. This consists of the DRAM controllers, a network interface, and a cache memory (size to be determined). We plan to make the cache partitionable so some of this on-chip memory space can be used as an explicitly addressed staging memory.

The stream processor exploits a bandwidth hierarchy to efficiently keep such a large number of arithmetic units supplied with data and productively occupied. The levels of the hierarchy are listed in Table 2. For each level of the hierarchy, the table shows both the bandwidth in words/sec and the number of arithmetic operations per word of bandwidth at that level. The 64 arithmetic units in the stream execution unit each consume three 64-bit words of bandwidth each 1ns cycle for an aggregate per node bandwidth of 1.9×10^{11} 64-bit words/sec. The locality exposed by casting applications into kernels keeps most of this bandwidth local, so it can be provided inexpensively out of the local registers with the write bandwidth traversing small per-cluster switches. At the next level, the stream register file provides sufficient global register bandwidth so that one word can be read from each of these levels for every two arithmetic operations. Producer/consumer locality exposed within the stream model is exploited to capture most of inter-kernel bandwidth at this level. There are then three levels to the memory system, with the on-chip memory (staging memory and cache), local DRAM, and global DRAM providing progressively lower amounts of bandwidth.

Across the entire machine, this bandwidth hierarchy spans over two orders of magnitude. Experience with stream architectures on signal- and image-processing applications gives us confidence that the intra-kernel locality and inter-kernel producer-consumer locality will provide sufficient localization of

[2]By convention, lower case 'b' denotes 'bits' while upper case 'B' denotes Bytes.

[3]This board may also hold a number of network chips depending on whether the network is folded into the processor chips or not

[4]These are tentatively arranged as 16 clusters of two adders, two multipliers, and one divide square-root unit each - actually 80 units - but this is subject to change.

Level of hierarchy	BW (Words/s)	(ops/Word)
Local registers	1.9×10^{11}	0.33
Stream registers	3.2×10^{10}	2
On-chip memory	8.0×10^{9}	8
Local DRAM	4.8×10^{9}	13
Global DRAM	4.8×10^{8}	133

Table 2: Bandwidth hierarchy of a streaming supercomputer. Per-processor bandwidth at each level of the hierarchy.

data movement to match many important problems to this hierarchy.

A scalar execution unit executes scalar instructions and dispatches stream instructions to the stream processor and address generators. We plan to leverage an off-the-shelf core for this processor.

2.3 Memory system and network

The streaming supercomputer provides high-bandwidth access at single word granularity across a flat global address space that covers the entire memory of the machine. Memory is accessed via scalar load and store instructions and via stream load and store instructions. Stream load and store instructions hide latency by issuing a stream of memory operations with a single instruction. Each node's memory is implemented as 16 future DRAM chips with a bandwidth of 2.4GB/s each[5]. Memory on remote nodes is accessed via a high-bandwidth interconnection network.

To isolate processes running on the machine without causing performance issues historically associated with TLBs, all memory accesses are translated via a set of eight segment registers. Each segment register specifies the segment length, the subset of nodes over which the segment is mapped (to support space sharing), whether the segment is writeable, the interleave factor for the segment, and the caching options for that segment. Segments are restricted to be aligned in a manner that facilitates fast address formation.

The network employs a hierarchical topology, uses high-speed (5Gb/s per signal) signaling to give high global bandwidth and uses flit-reservation flow control to minimize memory latency. The network organization, sketched in Figure 1, matches the physical packaging hierarchy of the machine. The network is composed of *channels* connected by *routers*. Each channel consists of eight 5Gb/s differential signals giving it a raw bandwidth of 40Gb/s[6]. Messages are switched between channels by routers. There are four routers on each circuit card. Corresponding routers are connected together across the circuit cards to form four completely independent routing planes.

Each router connects to 28 bidirectional channels (eight signal pairs in each direction). Sixteen of the channels are *local* channels, eight of the channels are *backplane* channels, and the remaining four channels are *global* channels. One local channel is connected to each of the sixteen streaming processors on the circuit card. Processors on a circuit card can communicate directly with one another by traversing one router and two local channels and, using all four planes, have a raw bandwidth of 20GB/s over each of these connections. This permits processors to access the memory of other processors on the same circuit card with half the bandwidth that they can access their own memory. The local channels of the 64 circuit cards in a backplane are connected together in a backplane interconnection network (details remain to be worked out). This permits all nodes in a cabinet to sustain a usable bandwidth of 10GB/s each to random locations in the cabinet. Finally, the global channels are converted on the backplane to ribbon fibers and connected in a global interconnection network that permits all nodes in a system to sustain 4GB/s of global memory bandwidth each. Table 3 summarizes how this network tapers bandwidth as more distant memory is referenced.

[5] Rambus' roadmap indicates that DRDRAM chips will have this bandwidth in the appropriate timeframe.

[6] The usable bandwidth will be substantially less than this due to address and control overhead.

Level	Size (Bytes)	Bandwidth (Bytes/s)
Node	2.0×10^9	3.8×10^{10}
Circuit Card	3.2×10^{10}	2.0×10^{10}
Backplane	2.0×10^{12}	1.0×10^{10}
System (16 backplanes)	3.3×10^{13}	4.0×10^9

Table 3: Memory bandwidth vs. accessible memory size

To simplify the task of coordinating operation between the nodes, the network and memory system incorporate a set of synchronization mechanisms. Presence tags can be allocated for each record in memory to synchronize producers and consumers of data. The producing store (scalar or stream) sets the tag to a present state, a consuming load (scalar or stream) blocks until the tag is in this state. Atomic remote operations including fetch and (integer) add or compare and swap are also implemented by the memory controllers to permit common synchronization constructs to be implemented without traversing the network multiple times. More complex remote operations can be implemented using a memory-mapped message send that is received by a message handling thread on the remote scalar processor.

2.4 Input/Output and Mass Storage

I/O and mass storage are attached to the machine via four 12-wide (30Gb/s) infiniband I/O channels on each processor card. Off-the shelf disk arrays, network interfaces, and user input and output are expected to be available to interface with the infiniband network.

3 Programming Models

As mentioned previously, the streaming supercomputer achieves high performance because of two key ideas: data parallelism and arithmetic intensity. A streaming computation involves passing the records of streams through a network of *kernels*. For the problems we envision there are 10^8 to 10^{10} records (and potentially even more) providing large amounts of data parallelism. All calculation within a kernel are local to a processor as are streams that are passed between a pipeline of related kernels. This locality maps well to the bandwidth hierarchy of a streaming computer. Finally, data dependencies are explicitly managed through stream buffers. This prevents read-write hazards and allows data to be efficiently moved throughout the system.

A key challenge is to develop a programming environment for the streaming supercomputer. This programming environment should naturally reflect the capabilities of the machine, so programmers are encouraged to program in an efficient way. Thus, the programming environment must expose parallelism and make data dependencies explicit. It also must encourage local calculations with high compute to memory ratios.

Since a significant investment will need to be made in recoding algorithms for such computers, the programming environment must be carefully designed to be portable and to run on future hardware of this type. Low-level machine parameters that are expected to change over time should be hidden from the programmers and managed by the compiler.

Compiler technology is critical to the success of the programming environment. However, the compiler technology that is needed is quite different than the existing focus of parallel compiler research. The goal of our compiler is not to *discover* parallelism hidden in sequential codes. This has proved to be a difficult task in the past and limits the ultimate scalability of the system. We will assume the programming environment makes parallelism explicit. The goal of our compiler is to *map* the calculation onto the machine in the most efficient way. This is more in the spirit of code generation, a problem that has proved tractable.

To support the streaming supercomputer we envision a three-level system.

1. A low-level language close to the virtual machine that manages platform specific resource constraints. This low-level system would be analogous to UPC or StreaMIT.
2. A mid-level progamming language that supports data parallel calculations. This level would be analogous to the Connection Machine programming environment C*.
3. Finally, we envision a high-level, domain-specific set of languages that make it easy to support the development of specific applications. This would be analogous to the Stanford real-time shading language for programming photorealistic rendering effects.

All these languages would be based on C. There would be a single low-level and mid-level language. A metacompiler (or an extendable source-to-source compiler) would convert the mid-level to the low-level. We envision multiple high-level domain-specific languages. The same metacompiler infrastructure would be used to translate these domain-specific languages to the mid-level language.

In the following subsections we describe the properties of each of these languages in more detail.

3.1 Low-level language presents an abstract view of the hardware

The goal of the low-level language is to expose the features of future streaming computers to the programmer. We believe this is an entirely new class of machines that will have a long life-span. Just like C originally exposed the features of PDP-11 class minicomputers to programmers, our language exposes the features of streaming computers. However, streaming computers are different than current microprocessors, and need additional language support.

The design of the low-level language is incorporates ideas from several parallel programming environments, most notably the Imagine StreamC and KernelC languages, the CILK run-time environment, the StreaMIT language, and the Stanford DSP-C and Smart Memories Virtual Machine. Finally, this language would leverage the current efforts underway to design UPC.

This language is based on the communicating sequential processes (CSP) model of programming, but enhanced to support streams, thread management and scheduling, and memory placement and data management. We include with the language the run-time environment, partly just for convenience, but also because we see a tighter and tighter coupling between the parallel run-time environment and the hardware architecture.

The language and run-time environment must provide the following capabilities:

- Explicit support for streams.

  As in StreaMIT and Stream/Kernel C, streams will be explicitly declared and kernels explicitly identified. This makes all of the communication in the program explicit and exposes it to the metacompiler so it can be optimized.

- Support for modern DSP and SIMD instruction sets.

  This includes support for fixed point calculation and segmented instruction sets. The language should be able to express and generate code for current microprocessors such as the Intel Pentium family with SSE, and the AMD Athlon family with 3DNow; and for current programmable streaming graphics chips such as the NVIDIA GeForce3 with vertex programs.

- Support for threads and synchronization primitives.

  The language should also provide control over the scheduling of threads. It should be possible to express data-affinity; that is, schedule threads after the data has been prefetched or on the processor which has a local copy of the data. The thread scheduler should also be smart. For example, the CILK run-time system is designed to perform load-balancing by assigning tasks to threads and then running those threads.

- Support for memory management and communication primitives.

  Our inspiration for these features are the memory management primitives in UPC. Memory management includes partitioning between global shared memory and local memory. The language

should also support different memory consistency models. It should also be possible to explicitly manage the cache, both by prefetching data and by segmenting the cache into subcaches. The language should also support the management of stream register files and stream buffers. It should be possible to express strided access, and to coordinate prefetched gathers with the execution of different kernels. This requires tight coupling between thread execution and data movement.

3.2 Mid-level collection oriented program expresses data parallelism

The goal of the mid-level programming language is to provide support for data parallelism. We believe almost all the major applications of high-performance computing are data-parallel; in particular, signal processing, image or media processing, scientific computing and database engines are data parallel.

Although data parallel computations may be (and are) written using a CSP or thread + communication programming model, we believe it is better to use a data parallel programming model. The data parallel programming model explicitly exposes parallelism and communication at a high-level. A metacompiler can than map this program onto a particular machine, relieving the programmer from dealing with particular machine parameters or limitations. This makes these programs significantly more portable, and hence long-living, encouraging programmers to rewrite their algorithms in this language.

The design of our mid-level data parallel programming environment is based on languages such as C* (and its Lisp counterpart Lisp*), the Connection Machine programming language. Other features are derived from signal, array and vector programming languages and libraries such as ZPL and VSIPL, as well as matlab and APL. Finally, we are motivated to include recent research on high-order functional languages such as Haskell and Scheme, and collection-oriented languages such as NESL.

The language will have the following features:

- Support for collections of records of various types.

  A record may be a primitive type such as a float, or a struct. Programmers will be encouraged to use records as the basic type. An example of a record might the state variables associated with a finite element mesh.

  Collections represent many records. We propose that there be at least three types of collections: sets, lists (or streams), and arrays. Sets are meant to capture the idea of an unordered collection. Certain parallel operations may take advantage of this unordered semantics. Lists or streams capture the idea of an ordered set. Lists and streams are meant to be processed in order. Finally, arrays capture the idea of random accessing or indexable calculations. Collections may have fixed or variable length.

  We may also want to support multidimensional arrays and a *graph* collection that can represent the connectivity of an arbitrary graph, e.g., a finite-element mesh. Later versions might also allow collections of collections (as is done in NESL).

- Kernel functions that represent operations on records.

  In some sense these kernel functions are the atomic operations in the language. Kernels take one or more records as input and produce one or more records (or as we will see a set of records) as output. Kernels represent entirely local calculations. However, unlike the Imagine KernelC programming model, kernels may contain loops and conditionals. Kernels may also have read-only or write-only access to global arrays. These arrays are declared as parameters to to kernel.

- High-level operators that apply kernels to collections.

  These operators would include:

  MAP: map applies a kernel to each element of a collection. For example, we could apply a transformation by a matrix to each vertex in a polygon mesh. Map is polymorphic across collection types, but respects the properties of the collection. For example, mapping a kernel onto an ordered collection will result in an ordered collection in the same order. However, mapping a kernel onto an unordered collection allows the order of the result to be different than the order of the input.

REDUCE: reduce applies a kernel to each element of a collection in the process producing a single result. Examples of reductions include max, sum, any, all, etc. Reduction operations may or may not be associative or commutative. Programmers are encouraged to use the least strict semantics. Reduce is often called SCAN or FOLD and corresponds to the Lisp* β operator.

EXPAND: expand creates a collection from a single record, or from a collection. For example, expand could be used by a rasterization kernel to produce a set of fragments from a triangle.

FILTER: filter produces a subset of a collection; the elements of the subset is determined by the value of a predicate kernel.

SCATTER, GATHER and PERMUTE: these operators rearrange collections.

In time, and as required, additional data parallel operations will be added. However, it has already been demonstrated that many important scientific applications may be written using these operators. In the final section on applications, we will discuss our research plan for different application areas.

Taking inspiration from modern functional programming languages such as Haskell, we would like to formally specify the semantics of these data parallel operators. For example, the expression MAP(f, MAP(g, c)) would be formally equivalent to MAP(f o g, c). That is, two consecutive maps involving f and g is equivalent to a single map of the composition of f and g. The reverse would also be possible: a complex function could be broken into two separate functions. This formal analysis would be very powerful. For example, by composing two functions we increase the arithmetic intensity, since both functions are executed although only a single read is performed. Formally breaking apart functions is also useful. Some implementations may want to break functions with global references into separate kernels with an intervening gather (as, for example, required by the Imagine architecture). Splitting functions might be useful for load balancing. Splitting functions creates more tasks, or potentially more uniform-sized tasks, which could be useful on future fine-grained, massively parallel machines. Reasoning about arithmetic intensity as machines evolve over time is one of the major research challenges for streaming computers.

One natural question is whether the mid-level and the low-level languages could be merged. Although that is possible, we prefer our proposed design because it allows us to leverage commodity compiler infrastructure. In particular, the low-level language is entirely responsible for code generation and of the additional capabilities that we need could be added through run-time libraries, as is traditionally done with dynamic memory management and threads libraries. In fact, it should be easy to develop a low-level language for existing machines, and hence most of our efforts would be on the mid-level language.

3.3 Domain-specific high-level languages map applications to collections

Finally, we envision building several high-level domain-specific languages for important application areas. These high-level languages will expose the capabilities of the streaming supercomputer to application programmers in an easy-to-use way, thus encouraging the adoption of the technology. We describe candidate areas for domain-specific languages in the section on applications.

Domain-specific languages have a long history in computer science. Our goal of providing this layer is motivated by the shading language that we have recently developed for programmable graphics hardware such as the NVIDIA GeForce3 or the ATI Radeon 8500. Shading languages have been developed by graphics researchers to describe the appearance of different objects, materials and environments. Shading languages have built-in functions for common operations, for example, to compute the light reflection from a matte surface using Lambert's Law. They also provide data types unique to that application, for example, vertices, fragments and textures.

Our shading language compiler maps this language onto the data parallel programming model. In terms of the primitives described in the last section, the compiler produces three kernels. A kernel to applied to each primitive, a kernel to be applied to each vertex, and a kernel to be applied to each fragment. The resulting data-parallel calculation is then the sequential execution of three MAPS, one for each collection. (Note in this case programmable graphics hardware handles automatically the conversion

of one type of record to the other. However, in our new system, these conversions could be handled by the EXPAND operator.)

Besides providing a high-level programming environment for applications, we believe domain-specific languages will lead to very efficient implementations. For example, many applications require solving linear systems of equations. However, the types of matrices generated by the application varies. In some cases, asynchronous iteration may be used to solve for the unknowns. These leads to an efficient parallel algorithm since updates may occur out of order. Another advantage of application specific languages is that certain sections of code may be difficult to parallelize. By encapsulating them in highly optimized, built-in functions or libraries, these difficulties may be avoided. An example of this is in the shading language; execution paths that involve complex data dependencies such as clipping and rasterization are handled by built-in functions that use specially designed algorithms (and, in the case of graphics chips, hardware).

3.4 Metacompilation

A critical enabling component of the programming environment is the metacompiler. The metacompiler is an extensible compiler infrastructure that performs source to source translation. The same metacompiler will be used to map the high-level language to the mid-level as well as map the mid-level to the low-level language.

The metacompiler performs source-to-source translations. The first stage of the metacompiler is to read in the source language and build a suitable intermediate form. The last step is to translate the intermediate form back into the source language. The core of the metacompiler is extendible methods for analyzing and manipulating the intermediate form. In order to do this, we will build a *program transformation language*. This language will make it easy to match source patterns in the input, and to rewrite that part of the code. This part of the system will be based on the existing metacompiler framework developed by Dawson Engler as part of his research on for checking programs.

The most interesting metacompiler will be the one that transforms from the mid-level language to the low-level langauge. This compiler will also read in a machine description file. This machine description file will include key parameters of the machines, in particular, the computational and communication resources available in the machine (similar to the table presented in previous sections). The size of different parts of the memory hierarchy will also be available. The metacompiler will perform similar analysis to that performed by the Imagine StreamC compiler. It will allocate memory for collections, it will break up collections into smaller chunks to take advantage of producer-consumer locality, it will schedule data transfers from global memory to local stream register files, and it will schedule kernels when the data is available.

However, unlike the Imagine StreamC compiler this compiler will be retargetable. By changing the machine configuration file, future variants of our architecture can be used. It will also be possible to use our programming environment on current clusters and shared memory multiprocessors. Since even conventional shared memory and message-passing multiprocessors benefit from regular access patterns and locality, our system should run efficiently on these machines. That is, the resources of these machines would still be used efficiently, even though such machines would be much less cost-effective than our streaming supercomputer architecture. The ability to use existing machines as platforms will allow us to begin development of the programming environment before the architecture is complete.

4 Applications

In previous work, we have demonstrated that streaming architectures perform extremely well on media applications, include signal processing, image processing and graphics. The Imagine architecture yields an order of magnitude performance increase over conventional processors [**?**]. Even complex algorithms such as MPEG encoding, depth extraction through correlation, and the conventional graphics pipeline can be mapped onto streaming architectures. Graphics is a particular success story. Last year several vendors have introduced programmable graphics processors that resemble special-purpose stream processors. These graphics processors have created a major new generation of graphics chips

with new capabilities. However, since these special-purpose streaming processors are not as general as our streaming computer, there is great interest in industry in more general streaming processor designs.

A major goal of this project is to extend our application domain from media processing to scientific computing. We will explore three major classes of scientific computing problems in roughly increasing degree of difficulty (that is, difficulty for efficient adaption into the streaming pipeline): Monte Carlo (MC) algorithms, ordinary differential equations (ODE), and partial differential equations (PDE). Numerical techniques for these types of problems underlie many important application areas. For example, Monte Carlo algorithms are regularly used in radiation transport and solid state physics, ODEs are used in molecular dynamics, astrophysics and rigid body dynamics, and PDEs are necessary in mechanical and structural design as well as fluid flow. Since simulating complex multi-physics processes is a major goal of the ASCI University Program, we are particularly interested in multi-physics applications, such as turbomachinary simulation which couples combustion with fluid flow and compressor turbine motion.

Some of these applications are embarassingly parallel, and we expect it will be easy to map them onto streaming computers. In fact, embarassingly parallel applications run well on current clusters, since they involve very little communication. However, even in these cases it is worthwhile to map them onto the stream programming model, since this will allow these applications to be run on much more cost-effective and scalable machines.

Other applications are more challenging to map onto parallel computers. These typically involve extensive global communication. These applications do not run well on existing clusters because of their limited global memory bandwidth. Our hypothesis is that they will run much better on the streaming computer because of its high-bandwidth interconnection network and memory system. However, these algorithms will need to be carefully mapped onto the stream programming model, and this will involve close interaction with colleagues in scientific computing. Successfully mapping any of these applications onto the streaming supercomputer could potentially open up whole new areas of computational science.

4.1 Monte-Carlo Radiation Transport

The simplest scientific computing problem that we will tackle is Monte Carlo integration, in particular, Monte Carlo simulation of transport equations. The key application of this technique is radiation transport, which is important in heat transfer and the design of nuclear devices. Monte Carlo of course has many other applications; for example, Markov Chain Monte Carlo or the Metropolis Algorithm is widely user in Bayesian inference, which is a major method used in statistical computing and artificial intelligence.

The basic Monte Carlo algorithm is particle tracing. Particles are created in certain states according to a source distribution functions. These particles make transitions to other states using a scattering distribution function. Finally, particles are terminated according to absorption distribution function. In classic Monte Carlo, each particle or sample is independent of the others and thus the algorithm is easily parallelized. Further, the inner loops involve generating random variables according to probability distribution functions. Although in toy problems these distribution functions are simple (e.g. uniform), in physical simulation they can be quite complex. Thus, not only are the calculations local, but they have high arithmetic intensity.

Monte Carlo is complicated if the model cannot be replicated. For example, when doing radiative transport in complex geometries, the geometric database may be very large and not easily replicated. Also, the interactions of particles with the database is not localized, since particles may be in different parts of the environment. In related work, we have shown how ray tracing may be mapped onto a streaming architecture. The key idea is to formulate the problem of finding a ray-surface intersection as a streaming calculation. Thus, we believe even in this case, we will be able to map radiation transport onto a streaming computer.

4.2 Ordinary Differential Equations

A more complicated problem is the solution of ordinary differential equations, in particular, coupled differential equations. Classic applications of these techniques include molecular dynamics and astrophysics,

or N-body problems, which involve pairwise forces between particles. Other applications include chemical kinetics which involves solving chemical rate equations for the concentrations of different species. Another example is rigid body dynamics of articulated figures, or robotics.

As a first example consider chemical reactions as might take place in combustion. Sometimes an overall time step is determined in order to gurantee stability and accuracy, but many times Godonuv (or Strang) time splitting is used to separate the chemical kinetics update from the fluid dynamics. While the fluid dynamics is "frozen", one has to solve a possibly stiff system of coupled ode's in each element of the computational mesh. One can easily have tens of species and hundreds of governing reactions, but these are usually simplified as much as possible employing a reduced chemical mechanism in order to defray the computational cost. These reduced mechanisms can be based on asymptotic theory or experiments and are currently an area of active research (notable methods include using reduced manifolds [Maas and Pope]), especially since they are not generally robust and many times need to be adjusted on a case by case basis. These type of computations are ideal for a streaming computer where one thrives with the full reaction mechanism with a high arithmetic cost per node. Hopefully, our new design will alleviate some of the need for reduced mechanisms allowing full mechanisms to be employed more often.

Another important example is solving equations of motion of large particle systems. This can be done in linear time using the fast multipole method developed by Greengard and Rocklin. The idea of this method is to approximate the field in a cell with a k-term multipole expansion. The field is then propagated up a hierarchy by translating and scaling the coefficients of the expansion. Computationally, this is a linear transformation on the coefficients and may be performed by a matrix-vector multiply. This field is pulled up the hierarchy in this way, and then pushed back down to the leaves. Again, this calculation may be easily mapped onto a streaming computer, and in fact has been efficiently implemented on the connection machine and other data parallel machines.

As a final example, in molecular dynamics a simpler method is often used. Since molecular forces fall off faster than $1/r^2$, it is typical to only compute forces amongst some small set of neighbors. Streams would need to access their neighbors with a GATHER, and then compute forces. But again, since these force terms are reasonably complicated, this would have high arithmetic intensity. Of course, there are still many details to work out, such as how to update the spatial data structure as particles move.

Problems of this type would benefit from high-level languages. In particular, Sussman and Wisdom have recently developed a high-level language based on Scheme for classical mechanics. They are able to specify the Lagrangian of a system from that declaration automatically derive the equations of motion. They then map the equations of motion onto a set of solvers. A similar approach could be used for coupled ODEs and highly parallel systems of ODEs. In addition to choosing methods for solving the ODEs, this new system could automatically parallelize the application.

4.3 Partial Differential Equations

The more complex problem domain we intend to study is methods for solving partial differential equations including finite volume, finite element and finite difference methods. Example applications include elastic and inelastic deformations of solids, and fluid flow, and fluid-solid coupling problems similar to those at the Caltech and Illinois ASCI centers. We further subdivide these techniques into Lagrangian (where the mesh is attached to a local reference frame moving with the material) and Eulerian (where the reference frame is global and the material moves between mesh elements). Of course, ALE schemes (where the mesh has and arbitrary velocity in between the Eulerian and Lagrangian mesh) are also of interest, especially at interfaces, but we intend to first take the approach of the Caltech ASCI center (and researchers such as Noh) and couple Eulerian and Lagrangian schemes directly at a fluid solid interface.

There are two major classes initial-value partial differential equations: Hyperbolic and parabolic equations (here we classify elliptic equations as boundary-value problems). However, for our purposes we will subdivide the problems into those that are *stiff*, and those that are not. Stiffness implies that some part of the problem requires much smaller steps than others in order to guarantee stability, and that this smaller time step is not needed to obtain the desired accuracy. Practically, this means that in stiff systems *implicit* methods are used for efficiency reasons. Implicit methods involve solving a matrix equation - for example a pressure solver in low Mach number or incompressible flow - and are similar (for

our purposes) to elliptic partial differential equations. If implicit methods are not needed, then *explicit* methods, which only require local neighborhood computation, can be used.

Explicit methods may be mapped efficiently onto parallel computers using domain decomposition. A typical time step in an explicit method requires a small neighborhood around a position in space. These neighborhoods are required in order to estimate spatial derivatives. So each step involves fetching information from your neighbors, and then updating your value. In many PDEs, the calculation involved is minimal and only involves a few arithmetic operations. Thus, the communication- to-compute requirements are large, although this ratio reduces as one applies more complex numerical methods, e.g. nonlinear limiters to treat discontinuous phenonmena or special algorithms for interface treatment. But, fortunately, there is an easy solution to this case: domain decomposition. By choosing as the unit of computation a region of space, then neighbors need only be communicated at the boundary of the domain. Since the boundary grows as n^2 and the interior grows as n^3 (for 3D problems), the ratio of communication to bandwidth decreases as larger domains as chosen.

Domain decomposition works well even on clusters, although the domains need to be large. The downside is that if the domains become too large, than there are fewer independent tasks. As the number of processors becomes large there may not be enough tasks for each processor, of if there is variability in the amount of work between tasks, load balancing problems may arise. Choosing the right domain size is a well-studied problem, and we again note that the situation improves when using more complex algorthims, e.g. those that are higher order accurate or those needed for the treatment of interfaces and discontinuities. We should be able to use these algorithms for the streaming computer. Again, this could be done by the compiler of a higher-level language.

Methods that involve implicit techniques are more complex. Fundamentally they involve solving linear systems of equations at each time step. This linear system of equations is usually sparse, which means that it can have a very irregular memory access pattern. A typical method involves first applying a preconditioner to the matrix and then using an iterative algorithm such as the conjugate gradient algorithm. In some cases, e.g., incompressible fluid flow, this matrix solve is the most expensive part of the calculation, consuming as much as 90% of the CPU time. Another approach to solving the matrix equation arising from this approach is to use a multigrid algorithm.

Mapping this matrix solve onto a streaming computer is a major research challenge. We should do much better than traditional clusters and shared memory multicomputers because we have greater global bandwidth and can tolerate the latency of irregular accesses. However, we believe we can do better. We will work with the numerical analysts at Stanford to develop new algorithms that map well to streaming computers. This will probably lead to a family of algorithms, since the algorithm of choice will depend on the application domain.

5 Research Plan

There are many challenging problems that must be solved to realize our vision of a streaming supercomputer. Many architecture issues must be resolved, there are many unknowns in the design of a layered stream programming system, and methods for most efficiently mapping our target applications to the stream model remain to be discovered.

To address these problems, we propose a six year research effort that proceeds through phases of *exploration*, *refinement*, and, if the early phases are successful *prototyping*. The overall program is illustrated in Figure 3. Our effort has three main threads: *architecture*, *programming systems*, and *applications*. The threads are tightly coupled with the results of one thread being used by and influencing the other threads.

During the first two years of the program we focus on solving the fundamental issues involved with architecture (e.g., network topologies, the interaction of streaming and cache coherence, and control mechanisms for streaming processors), streaming programming systems (e.g., collection-oriented languages, optimization methods, and program transformation technology), and applications (e.g., numerical methods that are best suited for streaming, and mapping techniques). At the end of this period we expect to be able to run simple Monte-Carlo radiation transport and ODE applications, compiled using our three-layer software system, on our architectural simulator.

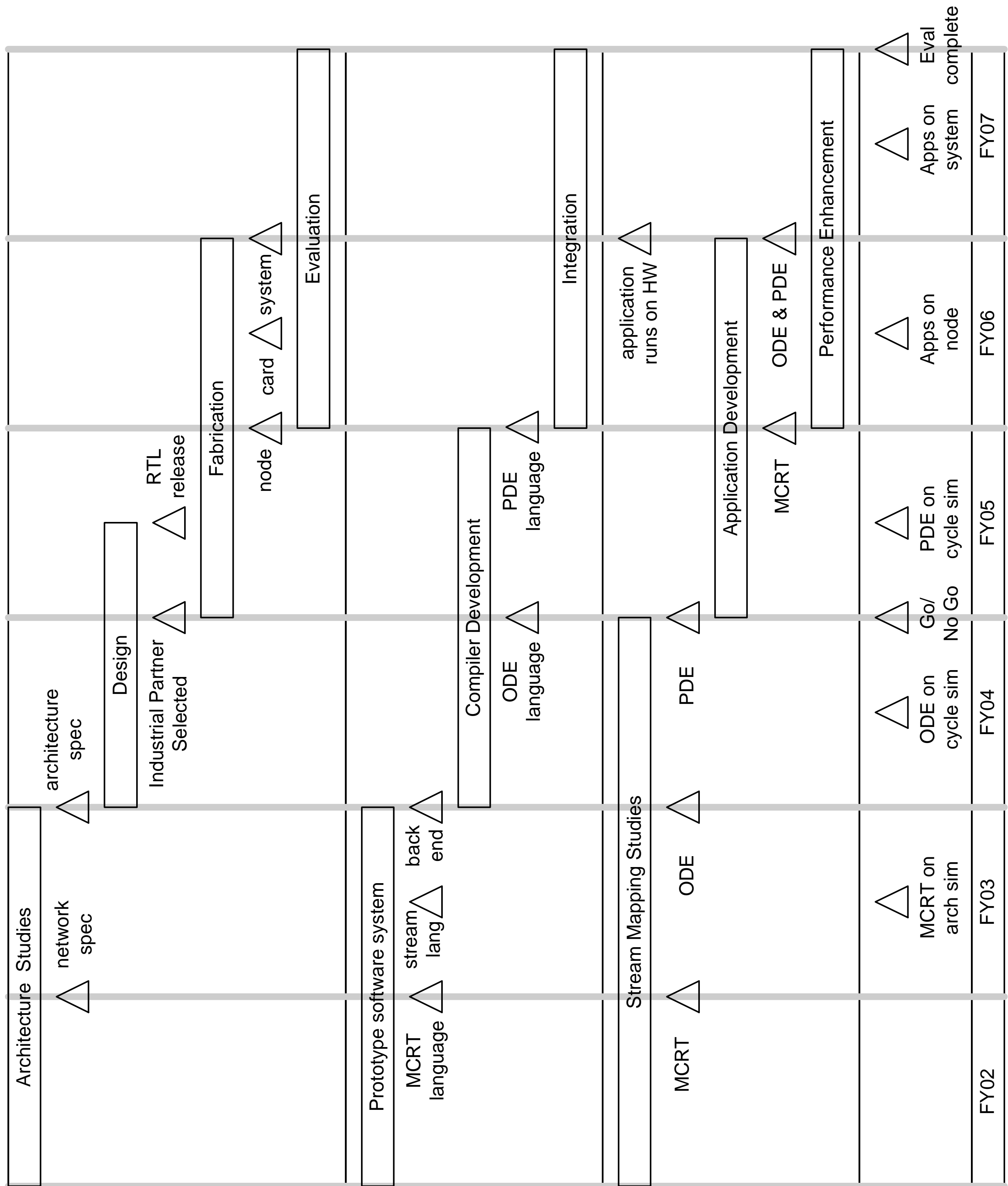

Figure 3: Plan for streaming supercomputer research and development

The next 18 months are a refinement period during which feedback from early simulation experiments will be used to guide the development of the architecture and programming system. During this period, the architecture will be reduced to a more detailed design and the programming system will be expanded. Toward the end of this refinement period we plan to make a go/no-go decision as to whether to continue the streaming supercomputer program into the prototyping phase or not. This decision will be based on whether our experiments suggest we can meet our cost-performance goals for the system and whether our architecture and programming system can yield high sustained performance on meaningful applications.

If the decision is made to proceed, an industrial partner will be selected to take on most of the detailed design, physical design, and fabrication of the streaming supercomputer. At Stanford we will work with our industrial partner on these hardware tasks. At the same time we will be increasing the range, size, and sophistication of applications that our programming system can handle. Our goal is to evolve our system to the point where we can handle the combined turbomachinery/combustion code.

The machine will be brought up in stages: first a single streaming processor node with local memory - no network, then a 16-processor card, and then systems of increasing size. As we bring up the hardware, we will integrate our programming system and evaluate the hardware on our application suite. The final period of the program is devoted to evaluation of the machine to learn what works and what doesn't and to discover how streaming supercomputers should be built in the future.

Key personnel

- Bill Dally
- Pat Hanrahan
- Ron Fedkiw
- Dawson Engler

[Do we want to add Mendel? Mark?]

References