Introduction

This document describes the Stanford-PCA Morphware Virtual Machine model, an architecture abstraction mechanism that can be used to “virtualize” all (or at least most) PCA architectures into a common compilation target. This flexible framework will allow our architectures to share a large fraction of the compiler infrastructure necessary to build applications for them, without requiring that we make our underlying architectures exactly the same. In fact, the same, unmodified code should at least run on all of our systems, although it may not run optimally on all systems without a few compiler hints and #pragma statements to customize the applications (especially at first!) and help the compiler process the trickier parts of the code well.

The key feature of the system that allows it to represent a very wide variety of PCA architectures is that, while many low-level details of the underlying hardware are “virtualized” away into generic structures, the overall structure of the architecture is accurately modeled (and fed into the compiler as a form of system metadata) to allow the relatively generic “upper-level” compiler to target the architecture’s key structural features to the extent that the application program allows. Architecture-specific “lower-level” compilers then take the VM API code produced by the “upper-level” compiler and produce actual object code that can execute on a particular PCA architecture.

The first section of this document describes the virtual machine model, which is used to build a simplified model of each PCA architecture for the compiler framework. The second describes some features of the API that is used to communicate between the generic upper-level compiler and the family of architecture-specific low-level compilers. Next, we have included a section that describes how the upper-level compiler uses the VM model information to perform several transformations on application code in an architecture-neutral way to create VM API code for an architecture-specific lower-level compiler. Finally, the last section describes a few enhancements to the basic VM model that may be desirable. An appendix is also included that summarizes the key function calls used within the API.
The Virtual Machine Plan

This section of the document describes the key features of the virtual machine model that acts as a compiler target for the “generic,” upper-level compiler. This model consists of a set of fairly generic nodes representing the processor and memory resources in the architecture connected together into a graph with links between the nodes representing the actual network links in the system. Each node is defined further with several types of metadata that standardize some of the key figures, such as processor performance and memory sizes, that the compiler needs to optimize code to run on the nodes in an effective manner.

This VM model is designed to hide most of the complexity of each individual node in the PCA system, while exposing enough of the structure of the underlying hardware in an architecturally-neutral way to allow the upper-level compiler to perform most of the complex optimization steps that are required to divide up a program so that it can execute well on a tiled architecture. In particular, the upper-level compiler needs to know enough about the target architecture to arrange tasks and data on the architecture’s nodes in a fashion that will permit efficient execution and communication. It also needs to know enough about the local memories attached to each node to correctly stripmine and block data intensive loops and kernels, which must use the local memories as scratchpad and buffer space. These requirements mean that most of the metadata in the VM model describes the sizes of all memories in the system and performance parameters for all processing, memory, and network link elements.
The Hardware Model: Abstract but Detailed

The hardware model consists of an interconnected graph made up of nodes and network segments. Nodes correspond to locations on the PCA architecture that perform useful work such as processors and memories. Network segments are the links between these nodes. The figure on the previous page shows a simple example hardware model consisting of six nodes and the network segments between them. There are three types of nodes used in this example hardware model. The main characteristics of these nodes and how the upper-level compiler targets them are summarized below:

Threaded Processor Nodes: The most important nodes in the model are the threaded processor nodes. These nodes consist of CPUs with RISC or CISC instruction sets much like the microprocessors available today. Each has an instruction cache, data cache, potentially an L2 and/or L3 cache, and a network interface to the outside world. Most generic threaded code may be scheduled to run on these nodes by the underlying compilation framework. For a particular application, each node may be selectively used in “batch” mode, without an underlying thread-scheduling kernel, or in “OS” mode, with one. Each processing element is characterized by several types of metadata, which are summarized in the table below.

<table>
<thead>
<tr>
<th>Metadata</th>
<th>Units</th>
<th>Compiler Interpretation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Processing “Power”</td>
<td>MIPS? MFLOPS? Arbitrary relative factor?</td>
<td>Used as a guideline for allocating threads among unequal-size threaded nodes, when possible. May be ignored when all threaded nodes are the same</td>
</tr>
<tr>
<td>ILP Capability</td>
<td>Instructions per cycle</td>
<td>These factors can be used to help determine the amount of blocking necessary for parallel, data-intensive code running on this node, although memory limitations are more important</td>
</tr>
<tr>
<td>Average operation latency</td>
<td>Processor cycles</td>
<td></td>
</tr>
<tr>
<td>I-Cache Size</td>
<td>Instructions</td>
<td>Used to check code blocks for small, kernel-like threads to ensure that they will be efficiently cached. Can only be done approximately, since final code size will not be available until after low-level compilation is complete.</td>
</tr>
<tr>
<td>I-Cache Linesize</td>
<td>Instructions</td>
<td></td>
</tr>
<tr>
<td>I-Cache Set Associativity</td>
<td>elements per set</td>
<td></td>
</tr>
<tr>
<td>D-Cache Size</td>
<td>KB</td>
<td>Used to block data arrays into the cache when parallelizing loops or inserting prefetch code</td>
</tr>
<tr>
<td>D-Cache Linesize</td>
<td>bytes</td>
<td></td>
</tr>
<tr>
<td>D-Cache Set Associativity</td>
<td>elements per set</td>
<td></td>
</tr>
<tr>
<td>L2/L3 Statistics</td>
<td>Same as L1 caches</td>
<td>Same as L1, if these are present</td>
</tr>
<tr>
<td>Context Switch Time (potentially several)</td>
<td>processor cycles or ms</td>
<td>This is used to intelligently determine how to context switch on this architecture, if more than one choice is possible. We may need more figures than just a delay to do a tradeoff analysis in a reasonable fashion.</td>
</tr>
</tbody>
</table>
Streaming Processor Nodes: The other key nodes in the system are the streaming processor nodes. These nodes execute stream kernels under control of the threaded processing nodes. Each node has an instruction memory, SRF (Stream Register File, a fancy name for the local data memory), and a network interface to the outside world identical to the ones in the threaded nodes. These are normally only used in a more “batch”-like mode, with an input job queue feeding in kernels from nearby threaded processors. Like the threaded processors, each processing element has several metadata figures that summarize its key features for the upper-level compiler.

<table>
<thead>
<tr>
<th>Metadata</th>
<th>Units</th>
<th>Compiler Interpretation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Processing “Power”</td>
<td>MIPS? MFLOPS? Arbitrary relative factor?</td>
<td>Used as a guideline for allocating stream kernels among unequal-size streaming nodes, when possible. May be ignored when all streaming nodes are the same.</td>
</tr>
<tr>
<td>ILP Capability</td>
<td>Instructions per cycle</td>
<td>These factors can be used to help determine the amount of blocking necessary for parallelized code running on this node, although memory limitations are more important.</td>
</tr>
<tr>
<td>Average operation latency</td>
<td>Processor cycles</td>
<td></td>
</tr>
<tr>
<td>Instruction Memory Size</td>
<td>Instructions or instruction packets</td>
<td>Used as a guideline for scheduling kernel code loading, but upper-level compiler will only be able to make approximations (since actual object code size is architecture-dependent).</td>
</tr>
<tr>
<td>SRF (Data Memory) Size</td>
<td>KB</td>
<td>Used to map out the use of the SRF fully during upper-level compilation.</td>
</tr>
<tr>
<td>Number of DMA Channels</td>
<td>number</td>
<td>This is used to determine how many stream loads and stores a node can handle at once. It can also be broken down into separate #loads and #stores.</td>
</tr>
<tr>
<td>DMA Bandwidth</td>
<td>MB/s</td>
<td>Bandwidth of loads and stores to/from the SRF. May also be broken down into subclasses for strided and scatter-gather DMAs, if speeds are different.</td>
</tr>
<tr>
<td>Kernel Switch Time</td>
<td>processor cycles or ms</td>
<td>This is used to approximately determine the time between kernels, to aid calculation of the overall execution time.</td>
</tr>
</tbody>
</table>

Memory Nodes: The final type of node in the system is the main memory (or just “memory only”) nodes. These are just a bank of memory that may be used by the other nodes, whether it is located off-chip or on-chip. Memory-mapped I/O resources may be modeled as small memory nodes, as well. Like the processor nodes, each memory node has a few numbers to specify it to the upper-level compiler. Half of these only apply to memory nodes that can act as “page caches” for a virtual memory system running on the architecture, and may be ignored in systems without a paging system currently running. Also, these VM parameters are only preliminary, and may need some further expansion on real systems.
<table>
<thead>
<tr>
<th>Metadata</th>
<th>Units</th>
<th>Compiler Interpretation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Size</td>
<td>KB</td>
<td>How much memory is physically in the node.</td>
</tr>
<tr>
<td>Latency</td>
<td>processor cycles</td>
<td>Time required for references to propagate through the node, used to position prefetch code</td>
</tr>
<tr>
<td>Bandwidth</td>
<td>MB/s</td>
<td>How much data can be fetched from or written to the node per second</td>
</tr>
<tr>
<td>Virtual Size</td>
<td>KB</td>
<td>How much virtual memory outside of the system can be accessed through this memory node.</td>
</tr>
<tr>
<td>VM Page Size</td>
<td>KB</td>
<td>Possible page sizes</td>
</tr>
<tr>
<td>VM Resolution</td>
<td>KB</td>
<td>Resolution at which memory can be split between directly-addressed and virtual memory sections. 0 indicates that it’s all one or the other, but not both.</td>
</tr>
</tbody>
</table>

Connecting these three types of nodes together are the network segments. They are just modeled as simple wires that have a latency (in processor cycles) and bandwidth (in MB/s). In the example, there are eight network segments (N1 to N8) connecting together the six main nodes and a pair of junctions (which can be modeled to have a latency, although this may not be necessary). While all of the network segments can be identical, it is more likely that at least the segments crossing the chip boundary will have a much higher latency and lower bandwidth than the others.

To simplify the API for controlling the potentially bewildering array of memory banks on a complex PCA architecture, one final piece of metadata is required to specify the architecture: an overall memory map. All uniquely addressable local memories (i.e. the SRFs on the streaming processors, but not the caches on the thread processors, since they are not directly addressable by any code) and the memories in the memory nodes (in their directly-addressable and/or virtual forms) must be located within different regions of the overall space. The upper-level compiler is responsible for allocating space for code and data within this space, and then must pass this allocation to the lower-level compiler using addresses fixed within the regions of this address space. For the example system depicted previously, here is a potential memory mapping for the final system:
As the figure indicates, these regions may have very different sizes and characteristics. Loads and stores to the different regions may have addresses of differing bit widths. The lower-level compiler may also emit considerably different code when a thread accesses one region or another. For example, accessing even a single byte within a streaming processor’s SRF may require the loading of a very small “stream” of one data element. However, at the level of the VM API, these low-level differences are mostly hidden in order to provide a simplified and more uniform programming interface.
The Virtual Machine: API Definition

This section describes the basics of the “object code” format used to communicate from the upper-level compiler to the lower-level compiler. This format is based on C, and looks much like a C program that has had some extra keywords and library function calls added to it in key locations. The resulting code consists of a number of explicitly marked threads (and stream kernels, if applicable) which have already been parallelized and have had memory access and inter-thread communication marked clearly for the lower-level compiler. This format allows the lower-level compiler to be a fairly conventional optimizing uniprocessor compiler, because it only needs to handle the low-level compiling and optimization of a single thread (or kernel) at a time. Conveniently, most compiler optimizations that are inherently architecture-specific, such as register renaming to an architecture’s register file, actually should be performed at this stage.

In the space allotted for this section, we attempt to cover the basics of all areas in which the API of the VM code enhances upon standard ANSI C. Here is a summary of the key features discussed:

• **Memory Configuration:** The upper-level compiler is responsible for the memory allocation steps of compilation. Hence, the application’s object code must be annotated to indicate the desired memory mapping required from among the complex and heterogeneous memory architecture within each PCA chip.

• **Thread and Kernel Configuration and Control:** Each thread is scheduled to one or more of the PCA architecture processor nodes by the upper-level compiler. This scheduling information, along with some other configuration information for each thread, is added to the program before the VM code is generated for the lower-level compiler. Stream kernels have some configuration information, also, but are simpler.

• **Thread and Kernel Memory Accesses:** All memory accesses in threads and kernels, except to local variables, are annotated so that they explicitly refer to objects within the data memory layout generated by the upper-level compiler. For threads, these transformations are applied to global and heap accesses, while for stream kernels, these are applied to streaming accesses involving the Stream Register File (SRF).

• **Inter-thread Synchronization:** Several “primitive” inter-thread communication routines are included. This portion of the API is designed to be simple, so it is fairly easy to implement, yet comprehensive enough so that it can emulate most of the synchronization functions used in traditional parallel programming languages efficiently with one or more of its function primitives. At the moment, our small library consists of locks, barriers, a direct send-receive message passing mechanism, DMA unit control, and cache memory control.
- **Kernel Control**: A family of routines is included that defines how a threaded processor node can control a streaming one. This includes routines to initiate and synchronize stream load/store operations and kernel executions.

To keep this preliminary specification relatively brief, syntax for most code headers and other “metadata”-like items that do not directly impact the executable portion of the code has largely been omitted. As examples tend to be rather sizable, only one code example is included in this section, covering the coding of a stream reduction. Finally, specification of a few areas, particularly dynamic thread startup and shutdown protocols, is still incomplete at this point in time. Full specification for this syntax will need to be finalized before we can actually build a compilation framework to target the VM, but we felt that it is necessary to settle the more critical parts of the VM definition before we produce the lengthy document required to specify all of these details.

### Memory Setup Information

Prior to the definition of threads or kernels themselves for the lower-level compiler, the upper-level compiler must specify how the global memory environment presented by the architecture to all of the threaded processors will be used by this particular application. The resulting map is a key form of application-specific metadata presented to the lower-level compiler along with the application code itself. This process consists of dividing the physical memory provided by the architecture into a series of ranges described by their behavior.

**Cache Control Ranges**: For the caches in the threaded processors, the upper level compiler must explicitly encode how each portion of application’s memory may be cached. This is necessary to determine which portions of memory are considered “uncached” to the node, which parts may be cached, and which parts are shared by multiple nodes (and which nodes, in this case). The following table summarizes the possible modes that may be used by each range of memory. It may also be desirable to specify whether ranges are read-only, write-only (primarily for message-passing “output” buffers), or read-write.

<table>
<thead>
<tr>
<th>Mode Name</th>
<th>Characteristics</th>
</tr>
</thead>
<tbody>
<tr>
<td>Incoherent cache</td>
<td>A range of memory that does not necessarily need to be monitored by hardware to guarantee coherence. Multithreaded software must be specially designed to work without coherence guarantees. This memory should be used for code, data for single-threaded tasks, or local memory structures such as stacks.</td>
</tr>
<tr>
<td>Coherently cached</td>
<td>Same as above, but HW and/or low-level software provides a coherent memory model for all threads that are using this range of memory. This should be chosen from the least restrictive model required by the application from a list of models like sequential, release, TSO, etc.</td>
</tr>
<tr>
<td>Remote Only</td>
<td>Entire block is never cached. Primarily for I/O. The SRFs of streaming processors should be mapped into “remote only” space as well to allow threaded processors to access these memories.</td>
</tr>
</tbody>
</table>
Virtual Memory Ranges: Memory ranges that are selected from “virtually managed” sections of the architecture memory space (if these sections are present at all) may be paged by the architecture’s underlying OS kernel(s). We may want to allow additional specifications for ranges in these memory sections, such as allowable page sizes. On a more practical level, some architectures may allow the same physical memory to be used both directly OR for the page cache of a virtual memory system. Obviously, the upper-level compiler must avoid allocating the same memory for both purposes.

The figure below shows how the memory example from the first section of the document might be mapped among three different threads:

<table>
<thead>
<tr>
<th>Architectural Memory Map</th>
<th>This Application's Memory Map</th>
<th>Final Software View of Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stream Processor #1 I/SRF</td>
<td>Remote, T1</td>
<td>SRF #1</td>
</tr>
<tr>
<td>Stream Processor #2 I/SRF</td>
<td>Remote, T1</td>
<td>SRF #2</td>
</tr>
<tr>
<td>I/O Space</td>
<td>Remote, T1</td>
<td>I/O</td>
</tr>
<tr>
<td>I/O Space</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Off-Chip Main Memory, Direct Addressing</td>
<td>Remote, T1/2/3</td>
<td>Inter-thread synchronization</td>
</tr>
<tr>
<td></td>
<td>Incoherent, T1</td>
<td>Key Values</td>
</tr>
<tr>
<td></td>
<td>Left alone, will act as page cache for virtual memory</td>
<td></td>
</tr>
<tr>
<td>Off-Chip Main Memory, Virtual Addressing</td>
<td>Incoherent, T1</td>
<td>Code &amp; Data</td>
</tr>
<tr>
<td></td>
<td>Incoherent, T2</td>
<td>Code &amp; Locals</td>
</tr>
<tr>
<td></td>
<td>Incoherent, T3</td>
<td>Code &amp; Locals</td>
</tr>
<tr>
<td></td>
<td>Coherent, T 2/3</td>
<td>Shared Data</td>
</tr>
<tr>
<td></td>
<td>Unmapped for this application</td>
<td></td>
</tr>
</tbody>
</table>
In this example, thread #1 is sort of a “master” thread, controlling both of the streaming processors and handling all I/O traffic. Neither of the other two threads can access the necessary portions of memory at all. Meanwhile, the other two threads are a pair of tightly coupled threads that share a large section of main memory cache-coherently. Whenever one of these threads wants to communicate with thread #1, it passes data through the shared “remote-only” section of main memory, where cache coherence is a moot point. In addition, all three threads have some local-only memory for data structures like stacks that are never shared. Thread #1 actually keeps most of its data there.

**Thread Setup Information**

The previous example also brings up another potential problem. When these threads are allocated to the example hardware, which only has two threaded processors, either two threads will have to share a single processing node (with a kernel running only on that node) while the third thread gets a node to itself, or a kernel running on both nodes will timeshare all three threads across both nodes. The compiler may be able to make a decision about the best way to allocate the threads based on its metadata and some analysis of the complexity of the various threads, but this is probably a case where the user will have to give explicit hints to the compiler. Based on the memory situation, having thread #1 on one node and threads #2 and #3 on another is probably the optimal solution, because they will be able to share a single cache — eliminating the need for a complex cache coherence scheme in one fell swoop. However, the relative computation requirements of the various threads or other factors may make this simple model unoptimal.

No matter which of these solutions is chosen, we must encode the selected method of thread-to-node assignment explicitly in the VM API code (kernel-to-node assignment will be covered in the next section). Hence, the first part of each API code section is a summary of the mappings for the thread(s) encoded in that section. A few key facts about the machine must be established with this code:

**Processor Assignment:** Each thread must be assigned to a single processing node, or to an underlying kernel running on a group of processing nodes with equivalent configurations (i.e. same memory setup). This allows the local compiler to generate code that addresses memory resources associated with that processing node properly. While it is possible to assign a thread to two processors with radically different configurations, the low-level compiler will probably have to compile the thread multiple times into separate binaries.

**Startup/Control Technique:** There are several ways that threads or kernels may start up or shut down. Here are a few of the most common possibilities that could be indicated in the header information:
• Some threads or sets of threads will just start up immediately and run on one or more processor node(s) from the time of initial configuration. This results in a static machine layout.

• Other threads may start when an earlier thread completes — in other words, when the node running the thread(s) self-“morphs” to a new state, with one or more new threads replacing the old set.

• A few may be started “on demand” by a remote procedure call mechanism. See the streaming API section, which always uses this model for kernel initiation, for an example of how this API might also be coded for threads. This methodology effectively allows one node to “morph” another one dynamically during execution.

• Finally, some threads may be “forked” off from an existing thread in a manner analogous to the UNIX fork() system call. We may choose to include special versions of this call in the VM API, or this may be left as a part of OS kernel APIs.

Threads initiated in either of the latter two methods may be scheduled onto a processing node using either a “slave” processor node that just waits to receive new threads from an input FIFO (the technique usually used for streaming kernels) or to a processor that is dynamically scheduled using a runtime kernel (most common for threads, especially ones using fork() calls).

Thread Identity: The header sets a predefined global variable VM_THREAD_ID that may be used by either the C preprocessor or the thread code itself to determine which thread is which. We may choose to make the selection of thread ID information within a thread more complicated than a simple integer, if users find it helpful to have more information available for access within thread code.

Code Location: The upper-level compiler is responsible for reserving a section of memory where the code for each thread is kept. As the size of the actual object code for a thread can only be estimated before compilation completes, the space allocated should normally be rather generous so that the lower-level compiler may complete successfully. However, if necessary this space may be resized intelligently by re-running the upper-level compiler after a test compilation of the code has been performed by the lower-level one.

Stack Location: The upper-level compiler is responsible for reserving a section of memory for each thread to use as a stack. Normally, this section of memory should be only accessible by that particular thread. The lower-level compiler then manages the stack itself within the supplied memory area.

Message Passing Buffers: If this thread uses the VM_RECEIVE function at all, then the thread will need to reserve some message-passing buffers in its setup. The upper-level compiler should attempt to determine a reasonable size for these buffers depending upon how the message passing functions are used (frequency, data sizes passed, etc.) and the latencies in the architecture’s network between various nodes.
The lower-level compiler can then attempt to allocate these buffers in a way that is most advantageous for a particular architecture: special queueing hardware, simple local memories, or perhaps just global memory controlled using synchronization operations.

Because of the numerous options and requirements that may need to be encoded here, a specification of this API is beyond the scope of this early-stage document. It will be specified fully in a longer revision.

**Thread Code API**

The basic thread code API is just standard, single-threaded C code, with a few minor modifications. First, each section of code must be associated with a single named thread from the thread setup information in order to establish the memory environment that may be used by the thread’s code. Associating code with a family of equivalent threads (ones having a similar memory environment) is also possible. For the sake of brevity, this document will not go into details of syntax for this header code. Second, and more critically, accesses to globals and heap variables must be converted to a form that explicitly notes where the memory reference will fall in the named thread’s allowed address space. The upper-level compiler performs all of the compile-time memory management, so the VM must reflect this. There are several types of possible memory transformations, listed below:

**Local Variables:** These are left unchanged. The lower-level compiler may allocate space for these or for temporary values on the stack associated with this thread, according to the stack handling conventions for the underlying machine architecture.

**Static Memory:** Variables that may be assigned to an address at compile time (primarily globals) must be statically allocated by the upper-level compiler. Access to these should be passed to the lower-level compiler using a fixed address in the global memory space. Here is a possible syntax for each reference:

```c
data = _LOAD(region, address, #bytes)
_STORE(region, address, #bytes, data)
```

The lower-level compiler can then translate this into an actual load, store, or more complex operation (such as a node-to-node communication operation) for the architecture, as is appropriate.

**Dynamic Memory:** Variables that are allocated at runtime by routines such as `malloc()` can only have their memory block fixed by the upper-level compiler, but not the address itself. The block information is used by the lower-level compiler to generate code for dereferencing pointers to the proper type of memory. Pointer dereferences would use an API identical to the static memory reference API, except that the address field is variable instead of constant.
The other major type of transformations that will need to be made to the source code within threads involves communication between parallel threads. The upper-level compiler is responsible for generating single-threaded code from each application. Hence, it may need to insert inter-thread communication to connect any independent threads that either it or the programmer chose to make. The family of communication mechanisms supplied must be a fundamental part of the VM API:

**Explicit Synchronization:** Uses of explicitly VM-supported synchronization primitives are expressed using special notation. We should select a set of primitives that allows all (or at least most) types of hardware synchronization acceleration to be encapsulated within the API primitives, while keeping the collection of primitives small enough to keep low-level implementations simple. The precise scope of this collection is a topic that will need to be discussed, but we suggest locks (and/or semaphores) and barriers as a good starting selection. Here is a suggestion for the type of API we might want:

```c
VM_LOCKINIT(region, lock_ptr)  // Set up lock
VM_LOCKFREE(region, lock_ptr)
VM_LOCKACQUIRE_B(region, lock_ptr)  // Use lock
got_it = VM_LOCKACQUIRE_NB(region, lock_ptr)
state = VM_LOCKTEST(region, lock_ptr)
VM_LOCKRELEASE(region, lock_ptr)

VM_BARRIERINIT(region, bar_ptr)  // Set up barrier
VM_BARRIERFREE(region, bar_ptr)
VM_BARRIER(region, bar_ptr, num_threads)  // Use barrier
num_threads = VM_BARRIERTEST(region, bar_ptr)
```

Each primitive is allocated a block of memory in one of the memory spaces that must be initialized and released with the INIT and FREE functions. Metadata supplied by the architecture specification to the upper-level compiler defines the size of the blocks of memory required by each primitive, along with any restrictions on where the block may be placed in memory. In particular, some architectures may have specialized memory regions reserved purely for the use of high-speed synchronization primitives. The lower-level compiler may also take these lists and allocate registers, accelerator hardware, or other resources as are required to perform the actual synchronization in the most optimal way for the architecture. Like normal memory variables, the memory blocks associated with each primitive may be statically or dynamically allocated, with appropriate syntax for each case.

The function of most of the routines should be fairly obvious to any programmer familiar with locks and barriers. Locks may be acquired in a blocking manner with the LOCKACQUIRE_B function, or in a nonblocking manner using LOCKACQUIRE_NB repeatedly until it returns TRUE. At the end of a critical region, the lock should be released using LOCKRELEASE. Using a barrier is simply a matter of calling BARRIER with an appropriate number of threads. Both locks and barriers
also provide functions for testing the current state of the lock or barrier without actually entering into competition for it. It should be noted that implicit memory operation barriers are associated with all actual locking and barrier functions.

**Scalar Message Passing:** While it is possible to simply pass data from one thread node to another using loads, stores, and synchronization operations, it may be desirable to pass local data directly from one node to another without going through off-chip memories. The following routines normally support communication of this kind of data directly from one processor to another, although they will be implemented as moves through memory if this is impossible on an architecture. For communication of ranges of memory, the DMA communication described next should be used, instead.

```c
// Blind send
VM_SEND_BLIND(int target_node, data, int #bytes)

// Thread-blocking send with error checking
error_code = VM_SEND_B(int target_node, data, int #bytes)

// Non-blocking send with receipt
DONE finished = VM_SEND_NB(int target_node, data, int #bytes)
bool state = VM_SEND_TESTDONE(DONE finished)
int error_code = VM_SEND_SYNCDONE(DONE finished)

// Guaranteed send
VM_SEND_ASSURED(int target_node, data, int #bytes)
```

The first routine just sends out data to the receiving node and doesn’t wait around to see if the data was actually received. The second blocks until it knows whether or not the receiver was actually able to buffer the transmission. The third is similar, except that it is nonblocking, allowing the thread to do other work while it waits to see if its send was successful. It uses a special DONE pointer, described more fully in the streaming API section, to allow the sending thread to monitor the progress of the block move using the TESTDONE and SYNCDONE functions. The former function is non-blocking, and reports on the progress of the send. The second is a blocking function that waits until the move is complete and then returns an error code to indicate when the receiver is unable to accept the transmission. This function also releases any resources that may be allocated in the DMA engine, so it should be called after every nonblocking send. Finally, the last form is a “guaranteed” send. On most architectures, this will simply be implemented using a small loop that retries a VM_SEND_B function until it succeeds.

At the other end, a small selection of primitives allow the receipt of data sent from another threaded node:
// Receive data
data = VM_RECEIVE (int source_node, int #bytes)

// Input buffer checks
bool present = VM_DATAPRESENT(int source_node)
int #bytes = VM_BUFFERSIZE(int source_node)
bool full = VM_BUFFERFULL(int source_node)

The first routine performs a blocking receive. If no data is present in the input buffer from a particular target node, the thread blocks and waits. If this behavior is undesirable, then the various input buffer checking functions should be used to verify the state of the input buffer before calling RECEIVE.

DMA Communication: Many PCA architectures will have one or more DMA engines that can be used to accelerate large block transfers of data from one part of the system to another. The VM supplies a few functions that can utilize this hardware when it is present — but will convert into a loop of loads and stores that will still work if it is not. High-level multithreading APIs using message passing, in particular, will find frequent occasions to use these routines in order to accelerate communication. Here is a potential selection of low-level routines for accessing these engines:

// Thread-blocking block move
error_code = VM_BLOCKMOVE(int start_region, void *start_ptr, int end_region, void *end_ptr, int #bytes)

// Non-blocking block move
DONE finished = VM_BLOCKMOVE_NB(int start_region, void *start_ptr, int end_region, void *end_ptr, int #bytes)
bool state = VM_BLOCK_TESTDONE(DONE finished)
int error_code = VM_BLOCK_SYNCDONE(DONE finished)

The first routine is just a simple function call that blocks the calling thread while the move is performed. If multiple threads may be scheduled on the node, they may be scheduled to execute while a DMA engine actually performs the work. The second is a non-blocking variant. It uses a special DONE pointer, similar to the one used previously for the SEND functions and described more fully in the streaming API section. Their function is essentially identical to those for SEND functions, except that potential errors will generally be load/store kinds of errors (address errors, page faults, etc.) instead of buffering errors.

Cache Control: We will need to include a few primitives to allow the upper-level compiler to explicitly control the threaded processor’s LD/ST unit and cache memory enough to allow additional synchronization primitives to be built in software, if necessary. These are especially required if we choose to use incoherently cached...
memory for synchronization. Some primitives that we might want are memory sync, prefetch operations, and cache flush. Here are a few potential API suggestions:

```c
// Memory barrier operations
VM_MEMSYNC() // Blocking
state = VM_MEMSYNCTest() // Nonblocking

// Memory barrier for a particular address
VM_MEMFENCE(region, address)
state = VM_MEMFENCETEST(region, address)

// Cache control operations
VM_CACHE_FLUSH(region, address, #bytes) // MESI = I
VM_PREFETCH_READ(region, address, #bytes) // MESI = S
VM_PREFETCH_WRITE(region, address, #bytes) // MESI = E
VM_TOUCH_WRITE(region, address, #bytes) // MESI = M
```

The barrier operations either force all LD/ST operations for the entire thread (or just for a particular address accessed by that thread) to go to completion, or check on the current status of LD/ST operations for that thread (in the non-blocking form). Cache operations prefetch or flush out all of the cache lines in a range of memory, and leave the MESI protocol for shared memory (if present) in the indicated state. For incoherent memory, the two prefetch operations are functionally identical. The “touch” operation is an alternative to prefetching before writes that simply clears out a range of cache memory instead of loading the current state into the cache. On some hardware, this can be performed without actually wasting bandwidth fetching the current state of memory. If the extended local memory model from the next chapter is used, some additional API elements of this nature will be required to explicitly control these specialized local memories.

This set of functions will handle most code transformations within threads that we may need, but it should by no means be considered exhaustive. In particular, we are interested in getting feedback on additional types of primitives that may allow the full selection of hardware acceleration that may be possible on PCA architectures to be fully utilized.

**Stream Code API: The Control Thread**

The most important part of the streaming code API is in the control code that must be inserted into a “control thread” that actually performs the task of sequencing stream loads, stores, and kernel execution within the PCA system. As such, it is technically a part of the threaded VM API. The following overview diagram shows a typical streaming node and its environment in a PCA architecture. In it, the threaded processor at the lower left is actually responsible for controlling the entire streaming system.
The main control functions associated with the threaded control processor are sequencing of DMA operations (managing the local memories in the stream processor) and starting the execution of kernels. The remainder of this section discusses these routines, a few other useful control routines, and some of the basic techniques used with them.

Before we continue with a discussion of the routines themselves, it is important to note that all of the main routines are nonblocking. The threaded processor can use the finished record returned by each routine to block as necessary using a variety of synchronization functions described near the end of this section. The finished data record, of a “DONE” data type, should never be used for anything except input to the synchronization functions, as its internal structure is defined on an architecture-by-architecture basis. With these routines, it is possible to implement and control a full streaming pipeline (potentially even an out-of-order one), with a single thread handling the scheduling of the stream operation “instructions” for one or even several stream execution engines. If multiple stream engines are present on the PCA chip, then the one being addressed by any particular operation is selected using its memory mapping region number. Alternatively, blocking versions of the stream operations may be built quite easily by always following the operations with an immediate, blocking DONESYNC function.

The first group of routines allow the threaded processor to control the contents of the local memories within the stream processor, generally by using the DMA controller associated with the streaming processor.
**Stream Load/Store:** The most common tasks performed by the control processor are telling the DMA controller on a stream processor to load strided streams of data into its SRF from main memory or to store strided result streams from its SRF to main memory. These operations are performed using a pair of routines:

```c
DONE finished = VM_STREAMLOAD(int source_region, STREAM *address_from_main, int dest_srf_number, STREAM *address_to_srf, int #records, int recordSize, int stride);
DONE finished = VM_STREAMSTORE_op(int source_srf_number, STREAM *address_from_srf, int dest_region, STREAM *address_to_main, int #records, int recordSize, int stride);
```

These both have identical parameters, except that the location of the source and destination are in opposite sections of memory. Both move \( #records \) data records of size \( recordSize \), which are packed into a dense stream in the SRF of size \( #records\times recordSize \), to or from another memory in the architecture, where each record is separated from the next by \( stride - recordSize \) bytes. Obviously, having \( stride \) equal to \( recordSize \) will move a dense stream directly into or from the SRF.

The optional _op addition to the STREAMSTORE routine, if present, specifies that the stream store is not just a simple store to memory. Instead, the following sequence of events occurs for each stream element:

1) The pre-existing contents of the destination memory location are read.

2) The result of the calculation: (existing memory contents) OP (the stream element being stored) is calculated, where OP is any one of several commutative operations such as AND, OR, XOR, ADD, or MULTIPLY.

3) The result of this operation is then stored into the memory location.

This mechanism is designed to provide support for intelligent future memory system interfaces that can accelerate updates to the existing contents of their memories. On systems lacking this support, its effect can be simulated by loading the existing contents of the memory into the SRF and then running a small kernel that just performs the OP prior to performing a conventional store of the results.

**Stream Gather/Scatter:** A similar but slightly more complex operation is to tell the DMA controller on the stream processor to load indexed streams of data into the SRF from main memory or to store the resulting indexed result streams from the SRF to main memory. These operations are also performed using a pair of routines:

```c
DONE finished = VM_STREAMGATHER(int dest_srf_number, STREAM *address_to_srf, PTR_STREAM *record_ptrs_srf, int source_region, int #records, int recordSize);
```
DONE finished = VM_STREAMSCATTER_op(int
source_srf_number, STREAM *address_from_srf,
PTR_STREAM *record_ptrs_srf, int dest_region, int
#records, int recordSize);

The interface to both routines is very similar to that for the strided LD/ST routines. Both move #records data records of size recordSize, which are packed into a dense stream in the SRF of size #records•recordSize, to and from another memory in the architecture. Unlike the strided forms, however, the record_ptrs_srf stream within the SRF is used as the source for 64-bit pointers to the locations in the global address space where the records should be stored in memory. For STREAMSCATTER, the optional _op field works the same as it does with STREAMSTORE.

Stream Move: This routine is optional, but it may be desirable to have a routine that can move a stream directly from one on-chip memory to another (i.e. between SRFs):

DONE finished = VM_STREAMMOVE(int source_srf_number,
STREAM *address_from, int dest_srf_number, STREAM
*address_to, int #records, int recordSize);

This just performs a block DMA of #records•recordSize bytes from one memory in the architecture to another. It could actually be used to move dense, non-strided streams around anywhere using DMA controllers (and could therefore be compiled into different forms of object code by the lower-level compiler).

Kernel Load: Loading of kernels into the fixed kernel memory of each stream processor must also be managed by the compiler. This is handled by a late stage of the upper-level compiler, after it has already invoked the lower-level compiler to compile stream kernels and measured the size of the resulting kernels (in other words, stream kernels must be compiled before threads):

DONE finished = VM_KERNELLOAD(int code_region, byte
*address_from_main, int dest_stream_region, byte
*address_to_kmem, int kernelSize);

This just performs a simple DMA of kernelSize bytes from the main memory to the kernel memory in a node.

It is also possible to use simple loads and stores to the addresses within the SRF or kernel memory to handle data movement. However, this is not recommended for anything larger than a scalar or two that needs to be communicated into or out of a kernel, as it may be highly inefficient on some architectures and will sometimes block the threaded processor.

The next routine is used for actually starting up the thread processor once you’ve got the SRF arranged the way you want it to be for execution purposes.
Kernel Execute: This executes a single stream kernel starting at \textit{kmem\_start\_address} on the stream processor containing the global \textit{kmem\_start\_address} memory:

```c
DONE\ finished = VM\_KERNEL\_START(int target\_stream\_region, byte *kmem\_start\_address, 
STREAM\_DESCRIPTOR *ios);

// Where “ios” is an array of ...
typedef struct {
    STREAM *address\_from;
    int stream\_length;
} STREAM\_DESCRIPTOR;
```

When the kernel is started on the appropriate streaming node, it is fed the inputs described in an array of “stream descriptor” records pointed to by \textit{ios}. The size of each record and the direction of the data flow (input, output, or I/O) for each parameter may be obtained directly from the prototype of the stream kernel, or we may choose to add these fields to the STREAM\_DESCRIPTOR records. For outputs, the length of the stream is the \textbf{maximum} size of the stream that will be allowed, and not necessarily the size of the \textbf{actual} output stream produced. Any attempt to exceed this limit will result in the kernel stopping and returning an error. Scalar I/Os, such as scaling parameters or reduction results, should be passed through the SRF using streams of length 1 for the purposes of this API, although the lower-level compiler may translate these references into specialized scalar passing code if the underlying hardware cannot handle such short streams well.

The previously listed routines are all that is fundamentally necessary to control a streaming processor. However, the fact that they are all nonblocking can make sequencing of kernels difficult without some way of tracking when the various operations actually complete. Hence, there is a small family of routines for monitoring the progress of active stream operations. These routines all make use of the DONE pointer associated with each of the main stream routines to select the stream operation(s) that the thread is actively checking.

The upper-level compiler normally inserts these routines just before the next stream operation that is dependent upon the results of a previous one. In general, this forces the system to execute in a manner analogous to instruction issue on a conventional scoreboarded out-of-order issue processor. It should be noted, however, that streaming architectures should not be assumed to have store buffers. Hence, any later load or execute operation that attempts to fill a previously stored portion of the SRF with new data is considered to be dependent upon the store operation.

Blocking Completion Test: These functions test one or more DONE record(s) to see whether or not the associated stream operation(s) have completed. They block if the operation has not completed, and normally release the DONE record(s) for re-use by the system (which must be done with every DONE pointer eventually, to avoid resource leakage on some architectures).
int error = VM_DONESYNC(DONE tested);
VM_DONESYNC_N(int #dones, DONE rel1, DONE rel2, DONE rel3, ...);
VM_DONESYNC_NOCLEAR_N(int #dones, DONE rel1, DONE rel2, DONE rel3, ...);

When these routines return, the operations indicated by the given DONE records will have definitely completed, and the DONE records are recycled for reuse by other stream operations. Because of this, the DONESYNC_N form of the function normally does not allow error codes to be checked, but the special NOCLEAR version can be used if the compiler wants to wait for several operations simultaneously, but then keep their error codes around for checking with subsequent single DONESYNC operations.

It may also be desirable to allow the error-checking and/or DONE clearing operations to be executed separately from the synchronization functionality. Since these can only be performed after the operation has finished, however, they are not entirely necessary. If desired, here are the appropriate sub-operations:

int error = VM_DONEERROR(DONE tested);
VM_DONERELEASE(DONE rel);
VM_DONERELEASE_N(int #dones, DONE rel1, DONE rel2, DONE rel3, ...);

Nonblocking Completion Test and Release: This function tests a single DONE pointer to see whether or not the associated stream operation has completed, but without blocking:

bool complete = VM_DONEQUERY(DONE tested);

If the thread wants to check and see whether a stream operation is complete, but has other work to perform if it still isn’t done, then this routine can be used to perform a nonblocking test on the completion. Note that a releasing DONESYNC function should still be used afterwards to clear the DONE pointer.

It is possible to get error codes from any of the routines using the DONESYNC function. Most stream operations should not return errors, but there are a few obvious ones that might be returned. The various load and store operations could return any of the same errors as a standard load or store — address errors, virtual memory page faults, etc. Similarly, the kernels could return a user-defined error code specified by each kernel as it completes.
Stream Code API: The Stream Kernels

As with the threaded code, a single-threaded, C-function-like notation will normally be the “language” of choice as the baseline for the stream kernels. The alternative decompositions mentioned previously for basic threaded code could also be applied here.

While we have discussed the elimination of some C constructs such as IF-THEN statements, for now we believe that we will simply let the user use a full selection of C programming constructs in kernels. If the constructs prove too complex for the thread kernel hardware on a particular architecture to handle, then the fallback position is for the compiler to transform the stream kernels into highly parallel threads (while warning the user, of course, so that he or she will know about the potential performance degradation). If this system proves to be a major problem, then we may rein back the legal allowed source code structure in future revisions of the API.

Also as with threads, the majority of the code transformations required within stream kernels have to do with memory operation transformation. Specifically, all references to the SRF, which is managed by the upper-level compiler, must be converted into an internal API form similar to the memory transformations performed for threads. On the other hand, accesses to locally allocated memory (in register files or small memories like Imagine’s scratchpad) are left alone and handled entirely by the lower-level compiler.

Direct SRF Accesses: Stream accesses into the SRF could be transformed into accesses similar to the array memory accesses performed by threads. These may be necessary in some cases, but their use is discouraged since these general accesses may be very inefficient on some architectures. In this case, the upper level compiler accessing a stream could produce something like:

```c
// Arbitrary LD/ST in kernel
data = _SRF_R(address, #bytes)
_SRF_W(address, #bytes, data)

// Indexed stream access in kernel
data_record = _SRF_RA(starting_address, record_size, record_number)
_SRF_WA(starting_address, record_size, record_number, data_record)

// Indexed kernel parameter stream access
data_record = _SRF_IOR(parameter_number, record_size, record_number)
_SRF_IOW(parameter_number, record_size, record_number, data_record)
```

The first forms just read or write data out of the SRF in a completely arbitrary manner, much like a normal threaded LD/ST. The middle group is for direct accesses
to arbitrary streams located within the SRF. The final routines are for access to specifically indexed elements within kernel’s I/O parameter streams. “Parameter number,” in this case, is the 0-based array reference into the list of stream descriptors given to the KERNELSTART call.

**I/O Streaming Accesses:** A simpler version of the previous notation should be used for simple sequential streaming of I/O operands, the most common “memory” accesses made by streaming kernels. Several different varieties of these are possible:

```c
// Basic stream I/O
data_record = _IN(parameter_number)
_OUT(parameter_number, data_record)
_OUT_INORDER(parameter_number, data_record)
```

The **IN** and **OUT** functions are the most common way to access I/O streams from a kernel. They pop the next element out of an input stream or push a new result into an output stream. It should be noted, however, that the **OUT** function does not necessarily guarantee any particular ordering of elements in the output stream. As a result, the default is for the architectures to produce output streams as fast as possible, irregardless of the original input ordering. If the output must actually remain in the same order as the original input, then the **INORDER** version of the function may be used to force it to comply with this requirement.

```c
// Peek at nearby stream elements, without advancing
data_record = _PEEK(parameter_number, const offset)
```

The **PEEK** function allows the kernel to look at the element in the stream about to be popped off (with an offset of zero), further ahead into the stream (with a positive offset), or back at recently popped-off elements (with a negative offset). This offset must be a constant, and may need to be fairly small on some architectures. If the architecture is incapable of handling a large offset, then this may be converted into an equivalent **SRF_IOR** function, instead.

```c
// Stream I/O that may conditionally occur (optional)
data_record = _IN_CON(execute, parameter_number)
_OUT_CON(execute, parameter_number, data_record)
_OUT_CON_INORDER(execute, parameter_number, data_record)
```

Finally, it may be desirable to have a special operator for the conditional input or output of a stream element. These operators are exactly like the non-conditional equivalent, except for the fact that they only execute if the **execute** boolean parameter evaluates to TRUE. They are not truly necessary, however, since they can be easily created simply by putting a normal version of the stream operators within a normal C IF-THEN statement.
Unlike with threads, there are very few specialized operators functions that can be used within streaming kernels. While this collection may grow in the future, at the moment we have only three specialized operators:

**Size of Stream**: This function-like operator returns the number of records in a variable-length input stream. The `parameter_number` is the same number used to reference the parameters to the kernel as the previous streaming accesses.

```c
int numRecords = VM_SOS(int parameter_number);
```

**End-of-Stream**: This function-like operator returns TRUE when a variable-length input stream has been emptied. The `parameter_number` is the same number used to reference the parameters to the kernel as the previous streaming accesses.

```c
bool ended = VM_EOS(int parameter_number);
```

**Output Full**: This function-like operator returns TRUE when an output buffer for a variable-length output stream has been filled. Most likely, the kernel will want to return an error to the controlling thread when this occurs. The `parameter_number` is the same number used to reference the parameters to the kernel as the previous streaming accesses.

```c
bool full_buffer = VM_OF(int parameter_number);
```

**Stream Code API: Reduction Operations**

While we have considered including additional functions to help mark reduction operations for the lower-level compiler, our current plan is to simply build up custom reduction operations using the more fundamental components already presented in this document. This actually offers a good example to show how many of the operations presented can work together effectively. This example shows how the reduction for a simple “sigma” (sum across stream elements) function might be assembled from these elements.

**Within a single stream kernel**: Within a stream kernel, loop-carried dependencies trigger the need for optimized reduction operations. Hence, the code for this operation is quite simple:

```c
// Kernel headers here . . .

sum = 0; // Initialize
while(!VM_EOS(0)) // Stream loop
    sum += _IN(0);
_OUT(1, sum); // Output result
```
The loop carried dependence on `sum` inherent in the `+=` operation automatically notifies the compiler to the presence of some form of reduction. As this reduction is fairly simple (just an add), it may even be possible to use some form of hardware acceleration to speed it along. The simple stream-access functions are used to read the input stream (parameter #0) and output a single scalar to the output “stream” (parameter #1). When the kernel is complete, the controlling thread may recover the result from the SRF.

On a related note, it is fairly easy for a compiler to determine that parameter #1 is a scalar because it is only written once, and outside of the kernel’s loop. On some architectures, the lower-level compiler may be able to use special scalar-only mechanisms to accelerate this process.

**Within a thread controlling multiple streaming engines:** When a single thread controls a pair of streaming engines, and passes chunks of a sigma kernel out to all of them in parallel, then that thread must run something like the following code. The control code for a single streaming engine control thread looks almost exactly the same, but without the reference to two different stream processors. However, we chose to present the example with two streaming engines just to show how it could be done using the API. This code assumes that setup such as loading the kernel into the streaming node and building up the contents of the 2000-entry input array has already been performed. Also, no error checking is performed in order to keep the code short. For the purposes of this example, data and code are loaded into fairly arbitrary addresses within each SRF (assumed to have already been determined by the upper-level compiler during memory allocation), and the different memory region numbers have been preassigned to the various `_RGN` constants.

```c
// Headers and initial code here . . .

// Load 1/2 of the input array in each SRF in parallel
flag1 = VM_STREAMLOAD(MEM_RGN, &values[0], SRF1_RGN,
   0x1000, 1000, sizeof(double), sizeof(double));
flag2 = VM_STREAMLOAD(MEM_RGN, &values[1000],
   SRF2_RGN, 0x1000, 1000, sizeof(double),
   sizeof(double));

// Define the input and output streams
myStreams[0].address_from = 0x1000;
myStreams[0].stream_length = 1000;
myStreams[1].address_from = 0x0FF0;
myStreams[1].stream_length = 1;

// Run kernels in both streaming nodes in parallel
VM_DONESYNC(flag1);
flag1 = VM_KERNELSTART(SRF1_RGN, 0x0E10, &myStreams);
VM_DONESYNC(flag2);
flag2 = VM_KERNELSTART(SRF2_RGN, 0x0E10, &myStreams);
```
// Get the scalar results and add them locally
VM_DONESYNC_N(2, flag1, flag2);
result = _LOAD(SRF1_RGN, 0x0FF0, sizeof(double)) +
        _LOAD(SRF2_RGN, 0x0FF0, sizeof(double));

// . . . other code afterwards

The execution of most of this should be fairly straightforward to anyone who has
some experience with stream programming. The code just loads the two halves of a
stream into two different nodes, executes kernels in each node in parallel to compute
the reduction result, and then finishes up by fetching the two partial sums and adding
them together to get the final result. If one looks closely it becomes clear that all
DONESYNC operations are performed just before the results of their operations are
used. This is the way they should almost always be used in order to get the best
system performance — and extract the most parallelism — with the nonblocking
stream execution commands.

There is another subtle detail in the parameter passing. Streams are loaded into the
SRF using the STREAMLOAD function, which should seem fairly straightforward.
However, the final results are fetched directly from the SRF to the thread node at the
end using normal-looking load operations. The lower-level compiler may turn these
“simple” loads into actual LD instructions, some form of explicit inter-node message
passing, or even rather roundabout STREAMSTOREs of a single element to memory
followed by a load from that memory address. Since the LOAD operation specifically
notes that it is loading from the SRF regions, the lower-level compiler will have no
trouble determining that it must generate code for reading a scalar from a streaming
node’s hardware (either the SRF itself or special, accelerated scalar logic) instead of a
more “normal” bank of memory. The API is maintained at a high enough level so
that the lower-level compiler is free to perform the data movement in the most
efficient manner for its situation.

With multiple control threads: Once multiple control threads are being used, inter-
thread communication will be needed to communicate the results from each “slave”
control thread up to a “master” thread that can then finish the sigma function. There
are two ways to do it, using either shared memory synchronization or message
passing primitives. Metadata about the underlying architecture will guide the upper-
level compiler in choosing which of these techniques is the most efficient to use when
generating code. First, we present the shared memory version.

    // Headers and initial code here . . .
    // . . . code from previous section to generate
    “result”

    // Slave sends data
    if (VM_THREAD_ID == SLAVE_NODE) _STORE(MEM_RGN,
        &result_buffer, sizeof(double), result);
// Wait at barrier to sync up the two processors
VM_BARRIER(MEM_RGN, &result_barrier, 2);

// Master (node #0) receives data
if (VM_THREAD_ID == MASTER_NODE) result +=
   _LOAD(MEM_RGN, &result_buffer, sizeof(double));

// . . . other code afterwards

This uses a combination of synchronization with normal load and store operations,
and should seem fairly straightforward. The alternate version uses direct send and
receive operations in an effort to avoid going through memory.

// Headers and initial code here . . .
// . . . code from previous section to generate
   “result”

// Slave sends data
if (VM_THREAD_ID == SLAVE_NODE)
   VM_SEND_ASSURED(MASTER_NODE, result,
                   sizeof(double));

// Master receives data
if (VM_THREAD_ID == MASTER_NODE) result += VM_RECEIVE
   (SLAVE_NODE, sizeof(double));

// . . . other code afterwards

This performs the same operation without the explicit synchronization and routing of
data through a memory buffer. The send and receive operations perform their own
queueing-based synchronization and internal buffering. On architectures with explicit
hardware support for message passing operations, this second set of code may run
much more quickly because indirect communication through shared memory are
completely replaced by direct communication between processor nodes.
The Resulting Compiler Architecture

We now get to the major point of this VM and API exercise. The information in the VM model and the VM API presented previously allows us to split the compilation framework in half, along the lines noted in this figure:

The advantage to this configuration is that the “upper-level” compiler, above the VM, can perform most of the “hard” tasks associated with compilation of an application to a sophisticated, multi-node architecture. This way, we can design this compilation framework once and use it across the full space of PCA architectures. In contrast, the simpler, architecture-specific tasks are left for a lower-level compiler that must be customized for each architecture. This is a big win because the lower-level compilers are much simpler than the shared, upper-level one. The remainder of this section describes the major tasks required for compilation to a PCA architecture, and how these tasks are intelligently divided between the two halves of the compiler.

The Upper-Level Compiler

The “upper” portion of the compiler attempts to take a complex PCA application and break it down into its fundamental components (individual threads, kernels, etc.). This portion of the compilation is dependent upon the language, but not the PCA architecture itself. Once the compiler has these fundamental components, it attempts to assign them to architectural nodes and optimizes the result. Most of this process is not particular to any particular architecture, so the complex code that will be required for these steps should be applicable to virtually any PCA compiler. The whole process consists of the following steps:
1. **Find Code Components:** The first analysis that must be performed is to use the source language’s rules to determine how the code may be split into independent threads and kernels — the “code components.” For some languages, these may be explicitly marked. Others may be more implicitly created through means such as a `fork()` call buried within the code. Threads of this latter nature will probably have to be dynamically created and scheduled at runtime, but at least the potential for their creation should be noted during compilation. Finally, any notations that indicate how these threads should be scheduled on the architecture should be recorded. Meanwhile, kernel code (which will probably be fairly explicit) for streaming applications should be pulled out and divided up into individual, separate kernels for later compilation stages.

2. **Preliminary Code Compilation:** Within each code component, the upper-level compiler should try compiling the original source code down to intermediate VM code. In the process, it should attempt to gauge the size of the various threads and kernels, and also start building memory maps to be used by each component. This task will allow the compiler to determine approximate memory usage by each component. Simultaneously, data-intensive portions of each component should be analyzed to determine whether or not they may be automatically parallelized (notations in the source may provide hints as to how this might be done). These should be noted for potential parallelization in later stages.

3. **Map Computation Usage:** Source hints, heuristics, and/or profiling information (preferred, if available) should be used to try mapping the collection of threads and thread kernels to the architecture’s processing nodes. Ideally, the compiler will try to consider the need for some code components to execute simultaneously, balance the computation load, and minimize communication overheads in its rough layout. If extra nodes are available (or some are extremely underutilized), then the compiler may attempt to parallelize some of the heavily data parallel code components to use the available resources. If too few nodes are available, some code components will have to share nodes sequentially (one after another), or by using a low-level OS kernel to switch from one to another as they execute. Alternately, if the workload forms into a sort of data pipeline, then the system can just morph over time to handle individual pipe stages one by one.

This stage will be the toughest to design, and could easily be a key location for future research, particularly in profiling compiler design. In the meantime, we will probably have to use source hints (especially at first), heuristics, and feedback from later stages of the compiler to generate a multitude of “reasonable” mappings and then compile them all. Profiling can then be used to select the fastest solution — or provide the programmer with enough information to choose the best solution for his or her problem, after other constraints like resource usage are also considered.

4. **Map Memory Usage:** Once an attempt at the computation layout has been made, the compiler should lay out the memory required by each of the code components in the available memories in an effort to get them as close to their home node (or nodes) as
is possible. Re-execution of the previous stage may be necessary if the data memories turn out to all be very far from their associated computation nodes. Data-intensive portions of the program should be blocked and stripmined at this point in order to use the local memories (caches and SRFs) as efficiently as possible.

5. **Map Network Usage:** It will now be possible to estimate memory usage over the various network links in the model to determine bandwidth-limiting network links and to determine where long-latency accesses will cause significant delays to be encountered within the code components. Information about “hot” network links with major traffic jams should be recorded and fed back into the previous two stages of the compiler for re-analysis there, as these can only be avoided by remapping computation or data blocks in memory. The performance impact of long-latency accesses can be minimized by inserting prefetching before the accesses, marking loops for software pipelining, or other conventional tricks. The impact may also be reduced by scheduling multiple code components, all with memory latency problems, to timeshare with each other on a node.

6. **Generate Streamed VM API Code and Compile:** Stream kernels should now be parallelized using the results from the previous optimization stages and compiled using the low-level stream compiler for the architecture. The resulting object code map should be fed back into the upper-level compiler to permit it to generate kernel code loads. If there is kernel code memory thrashing due to an unexpectedly large static kernel code size, then re-optimization of steps 3-5 may be necessary.

7. **Generate Threaded VM API Code and Compile:** Once all of the complex optimization tasks have been performed, the upper-level compiler may finally generate code separately for each code component. The intermediate code from step 2 should be processed to tie it to the appropriate processor, memory, and network environment. The most significant part of this final transformation is allocating specific addresses to memory references, now that mapping of data to memory and within any SRFs has been performed. Streaming kernel loads should also be inserted into the code based on the actual kernel sizes from step 6. Another major function that must be performed here is inserting appropriate synchronization primitives based on the communication necessary between the threads. When these transformations are complete, the lower-level compiler may be invoked to generate object code for each thread.

**The Lower-Level Compiler**

Once we have VM code, the “lower” portion of the compiler may be invoked to perform the final, architecture-specific stages of compilation:

1. **Expand API Primitives:** API-level synchronization and memory primitives must be expanded into their appropriate architecture-specific form for each event or access.
This could be as simple as performing macro expansions, or could easily be a somewhat involved step for some architectures.

2. **Compile to Object Code:** It should now be a relatively simple task to compile the VM code to the actual object code for the machine. This portion of the compilation framework will be very similar to a conventional uniprocessor compiler, since the VM code will have been pre-decomposed into single threads. Fine-grained optimizations (most involving the register files in the computation nodes) that require intimate knowledge of the architecture, such as register scheduling, loop unrolling, and peephole optimizations, should be performed at this point in time, much as they are in conventional compilers today.

3. **Link in Low-Level Libraries:** Some applications will require prebuilt low-level support such as thread-scheduling kernels, virtual memory managers, software cache coherency control programs, or I/O libraries. These will need to be selectively linked to the appropriate nodes at this point in time.

As it must be rebuilt for every architecture, the lower portion of the compiler is intentionally much simpler and easy to assemble from existing software components. As a result, we should be able to focus most of our future research efforts on the difficult stages in the “common” upper portion of the compilation framework.
Extensions for More Reconfigurable Hardware

The basic VM specification defined earlier in this document adequately describes the most common hardware configurations that will be encountered within a typical PCA architecture. As a result, we may choose to just stop the VM definition there, in order to keep it as simple as possible. However, it is likely that a variety of somewhat non-standard hardware configurations may also be possible on some architectures — and reconfiguring to these may be desirable for some applications. Unfortunately, if the VM API does not allow these somewhat more complex types of hardware to be “virtualized,” then it will never be possible for the “upper half” compilers to ever target these configurations.

Therefore, it may be desirable to add some additional reconfiguration hooks to the API, even if only a few source languages require or even facilitate the use of such hardware environments. To give a concrete example, DSP programming techniques often call for threaded processors with SRF-like local memories allocated and controlled explicitly by the compiler. With the basic API it is not possible to “virtualize” and then target hardware that supports this programming model. At the expense of more complexity, this section offers a more comprehensive VM environment that adds more flexibility that we may desire in the final PCA VM.

VM Extensions

The initial part of a virtual machine specification must allow for more options for configuring the hardware at any point in time if these extensions are implemented. Two types of descriptors handle this additional configuration. First, we must describe the ways that each tile, memory range, or network segment within the architecture can morph. Each of the possible configurations is described by the API extensions and then given a name or number that can be used to reconfigure the architecture over time.

Processor Control: Each processor can be configured to execute code in different ways. While we could keep this as simple as “threaded” and “streamed” mode, we might also want to provide a wider spectrum of selections (such as many light threads vs. single heavy thread). Some architectures may have options that convert each PCA tile into different numbers of effective “processors” depending upon how the node is reconfigured. A predefined metadata file associated with each architecture would be responsible for specifying all legal configurations of the processing nodes, and then the upper-level compiler could choose appropriate configurations from the list depending upon the kind of source material it was compiling.

The basic configuration may also need to define precisely what a processor will be doing. If required by the architecture and application, one or more tiles may need to be allocated to predefined low-level tasks such as cache coherence directory control.
Other nodes may be configured to run single threads statically, to execute tasks off of a run queue (probably the default for streaming nodes), or be allocated a thread-scheduling kernel (the node-by-node “OS” for the system).

**Local Memory Control:** It may also be desirable to allow the local memory (or multiple, separate local memories) associated with each node to be configured more flexibly, if one wants to be able to use more than just cache memory or an SRF. Metadata provided by the architecture to the upper-level compiler would have to specify not only how much local memory existed, but also how configurable it is. Some architectures might require that memory within a processing node be all-cache or all-directly addressable memory, but not both. Other architectures might allow more flexible mixing and matching.

Within the resolution allowed by the architecture, the upper-level compiler may allocate local memory within each node to several possible modes:

<table>
<thead>
<tr>
<th>Mode Name</th>
<th>Characteristics</th>
</tr>
</thead>
<tbody>
<tr>
<td>Directly addressed</td>
<td>Only the local processor(s) can access this local memory</td>
</tr>
<tr>
<td>Directly addressed, visible remotely</td>
<td>Same as above, but (some) other processors can access the memory via the network (R-only W-only, or R/W)</td>
</tr>
<tr>
<td>Local-only FIFO</td>
<td>Memory (or controlling software) automatically maintains a FIFO R/W pattern to memory</td>
</tr>
<tr>
<td>FIFO, visible remotely</td>
<td>Same as above, but visible by (some) other processors remotely</td>
</tr>
<tr>
<td>Incoherent cache</td>
<td>Has no allocation in the global address space. Instead, just caches remote, incoherently cacheable memory. This may be “upgraded” to coherent cache memory if necessary.</td>
</tr>
<tr>
<td>Coherent cache</td>
<td>Same as above, but uses hardware or low-level software to enforce coherency.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Mode Name</th>
<th>Manual Usage</th>
<th>Automatic Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>Direct access</td>
<td>Variable designated as a local, direct-access variable in its type definition</td>
<td>Compiler may use this type of memory as an SRF or other sort of DSP or stream-optimized prefetch buffer, especially with streaming languages; code for small, kernel-like programs</td>
</tr>
<tr>
<td>Direct, visible remotely</td>
<td>Same as above</td>
<td>Same as above, but when another node needs to read the data, too</td>
</tr>
<tr>
<td>Local-only FIFO</td>
<td>Needed at all?</td>
<td>—</td>
</tr>
<tr>
<td>FIFO, visible remotely</td>
<td>Special, “active” datatype or object</td>
<td>Message-passing communications buffer</td>
</tr>
<tr>
<td>Incoherently cacheable remote</td>
<td>Special type designation for MP threads</td>
<td>All code other than small kernels; local-only data (i.e. stacks) or data for non-multithreaded software</td>
</tr>
<tr>
<td>Cache-coherent remote</td>
<td>Special type designation for single thread/cache programs</td>
<td>Most data for software supporting multithreaded-on-a-multiprocessor programming models</td>
</tr>
<tr>
<td>Remote Only</td>
<td>I/O and manual synchronization</td>
<td>—</td>
</tr>
</tbody>
</table>
The wide variety of possible memory mappings puts a fair amount of pressure on the upper-level compiler to choose the “correct” type of memory for each variable, and to allocate the “correct” amount of local memory to each type of variable (if multiple types may be allocated). The previous table gives some guidelines that compiler writers might use to target the various memory types.

**Network:** Each network segment within the morphable chip could be a configurable object, too. By default, each network segment is controlled dynamically as accesses are sent over it. However, it may be desirable to have additional configurations that can be applied to each network segment to statically devote a percentage of the bandwidth over that network segment to specialized communication “channels.” The configuration information would specify what percentage of each network link is to be dedicated to each channel, and then remote memory accesses in code that use channels must have an extra field in the memory R/W operation to specify the transmission channel number (such as `STORE(region, address, #bytes, data, channel#)`).

On architectures such as RAW that have an optimized form of static network communications, these additional configurations would be able to take advantage of the special capabilities. On other architectures, the additional network configuration information could safely be ignored.

**API Extensions: Hardware Configuration Descriptors**

With the addition of so much flexibility for the underlying node hardware, the API for the VM must be extended to allow specification of the specific underlying node configurations that are actually being used by a particular program at any point in time. While this is a straightforward enough process, with a suitable API (which is beyond the scope of this document), there is one major problem. While switching a node from one thread or kernel to another is a fairly straightforward task, and should have a fairly low overhead, switching a node from one computation model and/or local memory configuration to another will most likely be a high-overhead task. As a result, we will want to minimize or eliminate this overhead as much as possible.

Conveniently, many threads or kernels running on a single node will actually use the same underlying configuration, such as the default configurations presented earlier. As a result, no high-overhead reconfiguration will be necessary when switching from one to another the majority of the time. In order to minimize reconfiguration, the extended API separates the node configurations away from the threads or kernels themselves. Each configuration is given a name, and each thread or kernel names the (set of) configuration(s) it requires. Simply by comparing the names of the incoming and outgoing configurations, it is possible to determine whether or not a costly reconfiguration is actually necessary before starting the new thread or kernel.
Appendix: API Summary

This appendix briefly summarizes all of the “function calls” defined by the API earlier in this document that the upper-level compiler may insert into code in order to pass information to the lower-level one. It also notes where other information (mostly metadata descriptions) will also need to be inserted into the source code before it can be passed from the upper-level compiler to the lower-level one. The syntax for these descriptions will be described more fully in a future document.

**Memory Map (single):** The memory map used by this machine must be defined. All regions of memory that may be accessed by threads running on the machine must be marked out and have thread access permissions defined for them.

**Hardware Configurations (zero or more, if allowed at all):** If the optional hardware reconfiguration maps from the last section are used, these will need to be defined for any processor nodes or network segments that will be reconfigured to a configuration other than one of the defaults. With some source languages, like straight C, this will be rare, but others may often use several different mappings.

**Thread Setup (per-thread):** Several bits of information about each thread running on the system must be indicated. These include the processor (i.e. “tile”) assignment for the thread, how the thread may be started and stopped, thread ID number, message passing buffer configuration, and the memory layout of the code space and stack for the thread. As the VM API develops, it is likely that a few other pieces of information may be added to this list.

**Thread Headers (at each threaded code segment):** All threaded code must be marked to match it up with one or more of the thread setups. When a code segment is marked as being shared by multiple threads, it may be compiled into object code multiple times if their memory environments are too different (globals are not shared, etc.).

**Thread Code (within each threaded code segment):** Threaded code segments will need to have load/store operations (other than ones for purely local variables), and inter-thread communication explicitly marked with a family of simple functions.

```c
// General load/store operations
data = _LOAD(region, address, #bytes)_STORE(region, address, #bytes, data)

// Synchronization: Locks
VM_LOCKINIT(region, lock_ptr) // Set up lock
VM_LOCKFREE(region, lock_ptr)
VM_LOCKACQUIRE_B(region, lock_ptr) // Use lock
got_it = VM_LOCKACQUIRE_NB(region, lock_ptr)
state = VM_LOCKTEST(region, lock_ptr)
```
VM_LOCKRELEASE(region, lock_ptr)

// Synchronization: Barriers
VM_BARRIERINIT(region, bar_ptr)    // Set up barrier
VM_BARRIERFREE(region, bar_ptr)
VM_BARRIER(region, bar_ptr, num_threads)    // Use barrier
state = VM_BARRIERTEST(region, bar_ptr)

// Message sends
VM_SEND_BLIND(int target_node, data, int #bytes)
error_code = VM_SEND_B(int target_node, data, int #bytes)
VM_SEND_ASSURED(int target_node, data, int #bytes)

// Non-blocking send with receipt
DONE finished = VM_SEND_NB(int target_node, data, int #bytes)
bool state = VM_SEND_TESTDONE(DONE finished)
int error_code = VM_SEND_SYNCDONE(DONE finished)

// Receive data
data = VM_RECEIVE (int source_node, int #bytes)
bool present = VM_DATAPRESENT(int source_node)
int #bytes = VM_BUFFERSIZE(int source_node)
bool full = VM_BUFFERFULL(int source_node)

// Thread-blocking block move
error_code = VM_BLOCKMOVE(int start_region, void *start_ptr, int end_region, void *end_ptr, int #bytes)

// Non-blocking block move
DONE finished = VM_BLOCKMOVE_NB(int start_region, void *start_ptr, int end_region, void *end_ptr, int #bytes)
bool state = VM_BLOCKTESTDONE(DONE finished)
int error_code = VM_BLOCK_SYNCDONE(DONE finished)

// Memory barrier operations
VM_MEMSYNC()    // Blocking
state = VM_MEMSYNCTEST()    // Nonblocking

// Memory barrier for a particular address
VM_MEMFENCE(region, address)
state = VM_MEMFENCETEST(region, address)
Along with the routines on this list, we will need a few maintenance routines to handle the infrequent thread startup/shutdown events such as remote procedure calls (which would be very similar to the stream kernel execution startup model, below), thread forks and/or joins, and so on.

Stream Control Code (within threaded code controlling streams): Threaded code segments that control streams will need to use the following family of routines to actually control the stream engines.

```
// SRF and code memory management
DONE finished = VM_STREAMLOAD(int source_region,
                           STREAM *address_from_main, int dest_srf_number,
                           STREAM *address_to_srf, int #records, int recordSize, int stride);
DONE finished = VM_STREAMSTORE_op(int
                           source_srf_number, STREAM *address_from_srf, int
                           dest_region, STREAM *address_to_main, int #records, int recordSize, int stride);
DONE finished = VM_STREAMGATHER(int dest_srf_number,
                           STREAM *address_to_srf, PTR_STREAM *record_ptrs_srf,
                           int source_region, int #records, int recordSize);
DONE finished = VM_STREAMSCATTER_op(int
                           source_srf_number, STREAM *address_from_srf,
                           PTR_STREAM *record_ptrs_srf, int dest_region, int
                           #records, int recordSize);
DONE finished = VM_STREAMMOVE(int source_srf_number,
                           STREAM *address_from, int dest_srf_number, STREAM
                           *address_to, int #records, int recordSize);
DONE finished = VM_KERNELLOAD(int code_region, byte *
                           address_from_main, int dest_stream_region, byte
                           *address_to_kmем, int kernelSize);

// Kernel execution
DONE finished = VM_KERNELSTART(int
                           target_stream_region, byte *kmем_start_address,
                           STREAM_DESCRIPTOR *ios);
// Where "ios" is an array of ...
typedef struct {
         STREAM *address_from;
         int stream_length;
} STREAM_DESCRIPTOR;
```
// Required nonblocking operation synchronization
int error = VM_DONESYNC(DONE tested);
VM_DONESYNC_N(int #dones, DONE rel1, DONE rel2, DONE rel3, ...);
VM_DONESYNC_NOCLEAR_N(int #dones, DONE rel1, DONE rel2, DONE rel3, ...);
bool complete = VM_DONEQUERY(DONE tested);

// Optional nonblocking operation synchronization
int error = VM_DONEERROR(DONE tested);
VM_DONERELEASE(DONE rel);
VM_DONERELEASE_N(int #dones, DONE rel1, DONE rel2, DONE rel3, ...);

**Stream Headers (at each stream kernel):** All stream kernels must be marked as such. On some architectures, some simple setup information (such as the target node for this stream kernel) may also be required. However, stream processor environments should be stable enough within an architecture that this may be minimized.

**Streaming Code (within each stream kernel):** As with threads, stream kernels will need to have memory operations involving the SRF (i.e. other than ones for purely local variables) explicitly marked.

```c
// Arbitrary LD/ST in kernel
data = _SRF_R(address, #bytes)
_SRF_W(address, #bytes, data)

// Indexed stream access in kernel
data_record = _SRF_RA(starting_address, record_size, record_number)
_SRF_WA(starting_address, record_size, record_number, data_record)

// Indexed kernel parameter stream access
data_record = _SRF_IOR(parameter_number, record_size, record_number)
_SRF_IOW(parameter_number, record_size, record_number, data_record)

// Basic stream I/O
data_record = _IN(parameter_number)
_OUT(parameter_number, data_record)
_OUT_INORDER(parameter_number, data_record)

// Peek at nearby stream elements, without advancing
data_record = _PEEK(parameter_number, const offset)
```
// Stream I/O that may conditionally occur (optional)
    data_record = _IN_CON(execute, parameter_number)
    _OUT_CON(execute, parameter_number, data_record)
    _OUT_CON_INORDER(execute, parameter_number,
                         data_record)

// Stream I/O control functions
    int numRecords = VM_SOS(int parameter_number);
    bool ended = VM_EOS(int parameter_number);
    bool full_buffer = VM_OF(int parameter_number);