MIT-CSAIL-TR-2010-054 November 23, 2010

LEAP Scratchpads: Automatic Memory and Cache Management for Reconfigurable Logic [Extended Version]

Michael Adler, Kermin E. Fleming, Angshuman Parashar, Michael Pellauer, and Joel Emer

LEAP Scratchpads: Automatic Memory and Cache Management for Reconfigurable Logic [Extended Version] [‡]

Michael Adler[†] Kermin E. Fleming[¶] Angshuman Parashar[†] Michael Pellauer[¶] Joel Emer[†¶]

[†]Intel Corporation
VSSAD Group
{michael.adler, angshuman.parashar, joel.emer}@intel.com

[¶]Massachusetts Institute of Technology
Computer Science and A.I. Laboratory
Computation Structures Group
{kfleming, pellauer, emer}@csail.mit.edu

ABSTRACT

Developers accelerating applications on FPGAs or other reconfigurable logic have nothing but raw memory devices in their standard toolkits. Each project typically includes tedious development of single-use memory management. Software developers expect a programming environment to include automatic memory management. Virtual memory provides the illusion of very large arrays and processor caches reduce access latency without explicit programmer instructions.

LEAP scratchpads for reconfigurable logic dynamically allocate and manage multiple, independent, memory arrays in a large backing store. Scratchpad accesses are cached automatically in multiple levels, ranging from shared on-board, RAM-based, set-associative caches to private caches stored in FPGA RAM blocks. In the LEAP framework, scratchpads share the same interface as on-die RAM blocks and are plug-in replacements. Additional libraries support heap management within a storage set. Like software developers, accelerator authors using scratchpads may focus more on core algorithms and less on memory management.

Two uses of FPGA scratchpads are analyzed: buffer management in an H.264 decoder and memory management within a processor microarchitecture timing model.

Categories and Subject Descriptors

C.5.m [**Computer System Implementation**]: Miscellaneous

General Terms

Algorithms, Performance

Keywords

FPGA, memory management, caches

[‡]This is an extended version of a paper presented at FPGA 2011: Proceedings of the 19th Annual ACM/SIGDA International Symposium on Field Programmable Gate Arrays (ISFPGA).

1. INTRODUCTION

FPGAs are increasingly employed as coprocessors alongside general purpose CPUs. The combination of large memory and ease of programming a general purpose machine along with the abundant parallelism and low communication latency in an FPGA make the pair attractive for *hybrid* algorithms that split computation across both engines.

Memory management in software development is supported by a rich set of OS and library features. Describing overlays [6, 16], a method of swapping regions of code or data to fit in limited physical memory, elicits pitying chuckles from even well-educated computer scientists who began their careers after virtual memory became pervasive. Software designers targeting general purpose hardware long ago accepted that the gain in programmer efficiency from using compilers, support libraries and operating systems outweighs possible performance gains of hand-coding raw instructions.

The memory subsystem in general purpose hardware offers a hierarchy of storage, ranging from fast but small caches embedded in the processor to large external RAM arrays on memory buses, and to swap files on disks. Management of cache state is controlled by fixed hardware algorithms chosen for their overall performance. Explicit, hand-tuned cache management instructions are typically added only to the most performance-sensitive programs. Tremendous effort has been spent building compilers capable of automatic cache-management, e.g. [12, 13]. As general purpose processors add more parallel processing, language designers continue to add abstract memory management to design tools in order to split algorithmic design from the grunt work of memory management [3].

The gap between the programming environment on the general purpose half and the reconfigurable half of a hybrid machine is stark. Most FPGA developers still code in low level languages equivalent to assembly language on general purpose machines. Those optimizing a set of loop kernels may use C or Java-like languages [8, 10, 11, 14] and a handful are beginning to use languages such as Bluespec [2, 19] that support language-based static elaboration and polymorphic module definitions.

The state of memory management on reconfigurable logic is similarly primitive. FPGA synthesis tools support relatively easy management of on-die memory arrays. The interface to on-die RAM blocks is simple: a method for writing a value to an address and a two-phase pair of read request and response methods. This interface may be made timing insensitive by predicating the methods with *ready* and *enable* flags and buffering state on pipeline stalls [5].

1.1 Scratchpad memory hierarchies

What if an algorithm needs more memory than is available on-die? At best, designers are offered low-level device drivers for embedded memory controllers, PCIe DMA controllers or some other bus. Building an FPGA-side memory hierarchy is treated as an application-specific problem. Even methods for mapping memory management as basic as malloc and free to on-die RAM for C-like synthesis languages are a very recent innovation [22]. On general purpose hardware the memory hierarchy is invisible to an application, except for timing. A similar memory abstraction, identical to the interface to on-die RAM blocks but implementing a full storage hierarchy, is equally useful for a range of FPGA-based applications.

Our project began as an effort to accelerate processor microarchitecture timing models using FPGAs. We quickly realized that some effort writing a general programming framework would make our task more tractable. The resulting platform is in active use for timing models and has been adopted for other algorithmic accelerators, such as an H.264 decoder. Both of these applications are considered in this paper.

We have written LEAP *(Logic-based Environment for Application Programming)* [17], a platform for application development on reconfigurable logic. LEAP runs on any set of reconfigurable logic connected to general purpose machines. Like an operating system, LEAP is layered on top of device-specific drivers. It presents a consistent virtual platform on any hardware. Application writers may then target the virtual platform, rendering their code portable across communication fabrics. LEAP presents the same interface over connections as diverse as FPGAs plugged directly into Intel Front Side Bus sockets and FPGAs connected to a host over a JTAG cable. The virtual platform provides a rich set of services, including streaming I/O devices, application control primitives, and an asynchronous hybrid procedural interface similar to remote procedure calls [18]. The platform also provides automatic instantiation of processor-like memory hierarchies, ranging from private caches, through shared caches and down to host memory. In this paper we focus on the automatically constructed memory stack.

LEAP defines a single, timing insensitive, interface to scratchpad memory hierarchies. The same write, read request and read response interface methods are used for any memory implementation defined by the platform, along with the predicates governing whether the methods may be invoked in a given FPGA cycle. The simplest memory device allocates an on-die RAM block. However, LEAP memory stacks sharing the same interface can be configured for a variety of hierarchies. The most complicated has three levels: a large storage region such as virtual memory in a host system, a medium sized intermediate latency memory such as SDRAM controlled by an FPGA, and fast, small memories such as on-FPGA RAM blocks. Converting a client from using on-die memory to a complex memory hierarchy is simply a matter of instantiating a different memory module with identical connections.

For a given set of hardware, low-level device drivers must be provided for each level in a physical hierarchy. Virtual devices and services are layered on top of these physical device drivers, thus providing a consistent programming model independent of the underlying physical devices. Our goal is to make programming an FPGA more like software development on general purpose hardware. Programmers target an abstract set of virtual services similar to general purpose kernel and user-space libraries. Like general purpose hardware, programmers may get an algorithm working with generic code and then, optionally, tune their application for specific hardware latencies and sizes.

1.2 Related work

Many researchers have considered the problem of cache hierarchies in reconfigurable logic and embedded systems. Automatic generators build a variety of cache types and sizes, treating caches as building blocks [25]. Panda et al. presented an algoritm for computing application-specific cache hierarchies designed to minimize off-chip references [15].

CoRAM, a current research effort, is an investigation of application interfaces to memory hierarchies within reconfigurable logic [4]. CoRAM defines both a memory API and a control thread model for managing traffic between on-die and off-die memory. The CoRAM model could be implemented within the LEAP framework. LEAP scratchpads are a structured, hierarchical cache topology connected to applications through a relatively simple interface. In LEAP, the specific hierarchy instantiated is configurable from building blocks of both direct mapped and set associative caches. Clients may either accept the default private caches or may plug in their own, taking advantage of the sorts of access pattern optimizations considered in [15]. A CoRAM implementation within LEAP would replace the default, private, on-die cache with a CoRAM memory and control thread.

Other projects, such as hthreads [1], have also built hybrid computation frameworks. Hthreads offers a hybrid pthreads-style thread management environment layered on coherent memory. The automatic instantiation of cached memory hierarchies we describe for LEAP scratchpads could be inserted into an hthreads stack as well.

2. SCRATCHPAD ARCHITECTURE

2.1 FPGA On-Die RAM Blocks

On-die FPGA RAM blocks can be configured quite flexibly. Xilinx RAM blocks are organized as 18Kb or 36Kb blocks in data widths of 1, 2, 4, 9, 18 or 36 bits [24]. Altera RAM blocks have similar widths. Synthesis tools automatically provide the illusion of arbitrary size and width by grouping multiple blocks into a single logical block and mapping into the closest available bit width. A large Xilinx Virtex 6 FPGA has about 32Mb of RAM.

Access to RAM blocks is simple: a single cycle write operation and a two phase read request / read response protocol. Even a naïve implementation can be dual ported, permitting simultaneous reads and writes. RAM blocks are fast, flexible and easy to access as private storage within a module. Unfortunately, they are finite. What are we to do for algorithms with memory footprints too large for on-FPGA RAM?

2.2 On-Board RAM

Many FPGA platforms have on-board RAM and have memory controllers available as logic blocks. Compared to an FPGA's internal RAM blocks, on-board memory is plentiful: typically measured in megabytes or gigabytes. Unlike FPGA RAM blocks, on-board memory is a monolithic resource. At most only a few banks are available, managed by individual

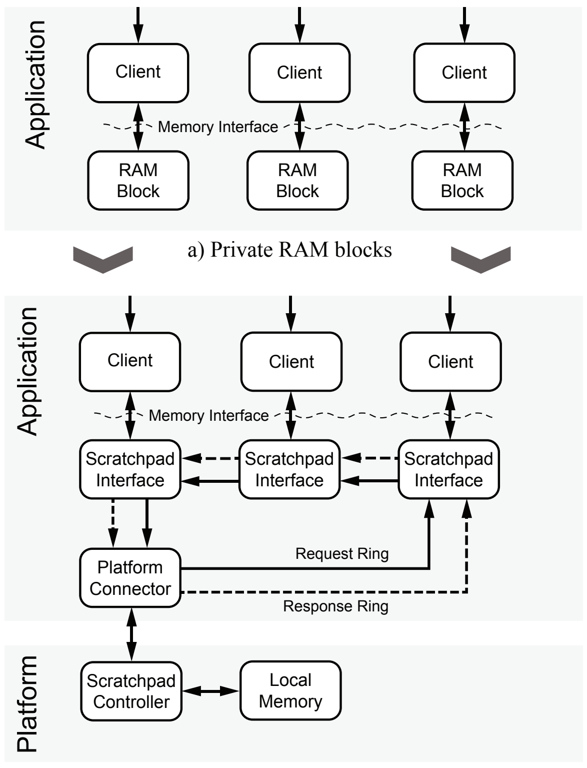

Figure 1: Transforming private RAM blocks to scratchpads. The memory interface between clients and storage is unchanged following the transformation. Only timing is different.

controllers. In order to share on-board RAM among multiple clients the memory must be partitioned and managed by a central controller. We call this service the *scratchpad controller*. The controller is responsible for partitioning a large memory into individual *scratchpads*, corresponding to private memories requested by clients. The controller then routes requests from scratchpads to their unique memory segments. This is implemented using an indirection table, mapping scratchpads to memory base offsets.

Except for latency, moving storage to a different level in the memory hierarchy is invisible to software written for general purpose hardware. The change could range from missing in an L1 cache to suffering a page fault and swapping data in from a disk. While not an absolute requirement for FPGA-based scratchpads, having the ability to express memory I/O operations independent of their underlying implementation and latency is equally convenient on reconfigurable logic. In our implementation, the difference between a client using a private on-die RAM block and a scratchpad in a shared memory is only a single source line (see Section 2.7). The client using a RAM block invokes a module that instantiates on-die memory. To use a scratchpad instead, the client replaces this instantiation with a module that connects itself to the scratchpad controller.

Each client requesting a scratchpad memory instantiates a *scratchpad interface*. This interface is private to a single client, transforming client-side references to requests in the scratchpad controller. The scratchpad controller is a shared resource. Connecting multiple clients to the controller requires an interconnect and arbitration. For a small number of scratchpads, a set of point-to-point connections from scratchpad interfaces to the controller along with a round-robin arbiter works perfectly well. As the number of clients grows, the burden on FPGA routing becomes too great and a more sophisticated network is required. We have built a pair of token rings, using self-assembling rings described in [19]. The transformation from private RAM blocks to scratchpad memories is illustrated in Figure 1. Deadlocks are avoided by assigning requests to one ring and responses to the other. A pair of rings was chosen instead of a single ring with virtual request and response channels both to increase network bandwidth and because the FPGA overheads of channel buffering and multiplexing are similar to the simpler, multi-ring, solution. One ring stop is responsible for forwarding messages between the rings and the scratchpad controller.

2.2.1 *Identifying scratchpads*

The scratchpad controller must have a way of identifying individual scratchpads. Each scratchpad interface must be assigned a unique identifier that we call a *scratchpad ID*. Each instantiation of a scratchpad interface module takes a compile-time constant argument specifying a unique scratchpad ID. Every request from an interface module to the scratchpad controller is tagged with an ID.

Reconfigurable logic and table sizes are minimized if the scratchpad ID space is dense. LEAP provides a namespace management tool for generating unique identifiers. This dictionary tool was originally conceived for mapping integer identifiers to strings in order to trigger printing of messages on a host from an FPGA without having to specify hardware logic for passing variable length strings. We have extended it to solve the general problem of managing identifier spaces, including syntax for managing numerically dense subspaces.

Using LEAP dictionaries and conventions, an implementor allocating scratchpad IDs would specify:

```
def VDEV.SCRATCH.FBUF_Y  "Frame buffer Y";
def VDEV.SCRATCH.FBUF_U  "Frame buffer U";
def VDEV.SCRATCH.FBUF_V  "Frame buffer V";
```

in order to allocate a group of scratchpads named *FBUF_Y*, *FBUF_U* and *FBUF_V*. The dotted notation represents numerically dense subregions.

2.2.2 *Initialization and addressing*

An initialization step is required in order to subdivide on-board memory into individual scratchpads. At start-up, each scratchpad interface computes the size of its scratchpad array. The interfaces then send allocation requests to the scratchpad controller. The controller receives allocation requests and builds a partition table, mapping individual scratchpads to unique regions of memory. An error is signaled if the size of all allocation requests exceeds available memory. Each scratchpad interface operates in its own private, zero-based address space. Within the common controller, addresses are computed as the sum of the private address and a scratchpad's on-board memory offset from the partition table.

The mapping from private scratchpad array indices to on-board memory addresses is fully contained within the scratchpad controller. Although our current controller implementation maps all scratchpads dynamically at start-up and has no protocol for releasing regions, it would be relatively easy to extend the private protocol between scratchpad interfaces and the controller to permit dynamic release and reallocation of memory. Only the partition table must be updated. An ambitious implementation could even rebase region mappings in order to combine fragmented free memory blocks into larger chunks.

2.3 Marshaling

Astute readers will have noticed a problem in our transformation of RAM block clients to scratchpad clients. Synthesis tools permit FPGA RAM allocation in any bit width. While the underlying hardware does not support arbitrary width, it is sufficiently flexible that memory is allocated relatively efficiently. In contrast, on-board memory is presented in chunks of words and lines, with some hardware adding write masks to support byte-sized writes.

An easy, but unacceptably inefficient solution would be fixed mapping of RAM block addresses to word-sized on-board memory chunks. The fixed mapping would not support data widths larger than a memory word. It would also waste nearly the entire word for small data widths, turning a dense 1024 x 1-bit RAM block into a 64KB chunk, assuming a 64 bit word!

To solve this mapping problem, the scratchpad interface interposes a marshaling layer between the client and requests to the platform interface. When objects are smaller than the memory word size, multiple objects are grouped into a single memory word. When objects are larger than the memory word size, the marshaling layer spreads objects across multiple words. In the first case the marshaler is forced to request read-modify-write operations in order to update an entry. In the second case the marshaler must emit multiple read or write requests in order to reference all memory words corresponding to a scratchpad location. From the client's perspective, the word size remains the size originally requested.

The LEAP platform provides a marshaling library module. Compile-time parameters declare the memory word size along with the desired scratchpad width and number of elements. The marshaler computes the dimensions of an on-board-memory-sized container for holding the equivalent data and determines whether read-modify-write or group reads and writes are required. It also exports read and write methods that act on the requested array's data type. The methods automatically trigger either read-modify-write or group reads and writes when needed.

2.4 Private Caches

With the addition of marshaling we now have an architectural description for replacing RAM blocks with on-board memory scratchpads that is fully functional. Unfortunately, it will perform terribly. RAM block references that were formerly single cycle references and parallel for each block have been converted into a shared, high contention, higher latency resource. A cache is needed, both to provide lower latency and to reduce the number of requests that reach on-board memory. LEAP provides low latency, direct mapped, caches, though developers may specify their own cache implementations optimized for particular access patterns.

The position of the cache, above or below the marshaler, is a compromise. Choosing to insert the cache between the client and the marshaler would eliminate many read-modify-write operations in the marshaler. However, read-modify-write operations are required because the data width above the marshaler is small. Consider a scratchpad of boolean values. Caching above the marshaler would require tag sizes to cover the address space but would have only one bit data buckets. This ratio of meta-data to actual cache data is unacceptable.

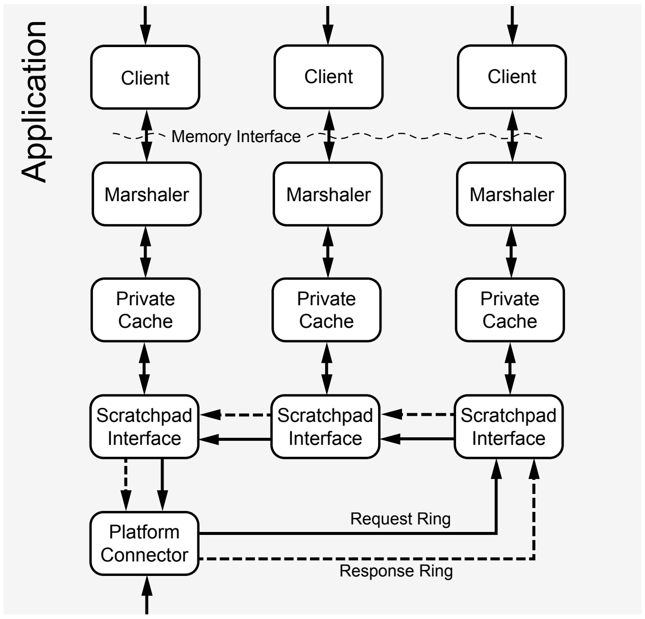

Figure 2: Client-facing model hierarchy, including marshaling from client data to on-board memory word-sized accesses, private L1 caching and a token ring networking scratchpad interfaces. The platform interface at the bottom of the stack forwards messages between the ring network and the scratchpad manager (not shown).

In our implementation, both the private cache and the marshaler present the same interface to the client. The relative order of the marshaler and a cache is invisible to the scratchpad client. A compile-time heuristic could choose a locally optimal topology based on a scratchpad's size and data type, placing the cache either above or below the marshaler. In our current implementation the cache is always inserted below the marshaler. The full hierarchy is shown in Figure 2.

2.5 Host Memory

The hierarchy has now expanded available FPGA-side memory from the capacity of on-die RAM blocks to the capacity of on-board RAM. This solution is fully functional on both stand-alone FPGAs and on FPGAs connected to a host computer. For scratchpad memories, on-die RAM block usage is reduced to fixed sized caches. Now we face the same question asked at the end of Section 2.1: What are we to do for algorithms with memory footprints too large for on-board RAM?

If the FPGA is connected via a high speed bus to a host computer, the solution is the same as when we ran out of on-die memory: push the backing storage one level down in the hierarchy, using host memory as the home for scratchpad data. The method is essentially identical to the method for the on-board memory scratchpad controller in Section 2.2.2. An indirection table must map scratchpad memories to host

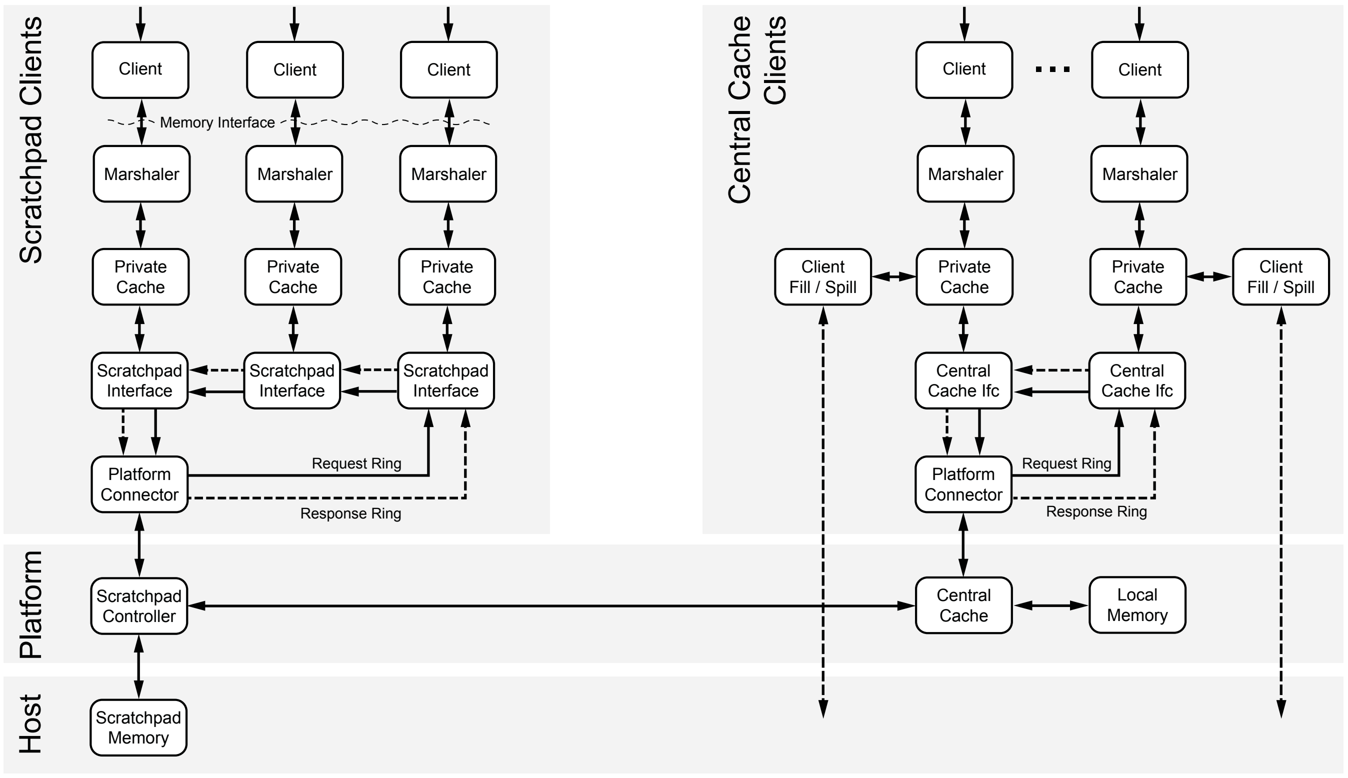

Figure 3: Full client-facing model hierarchy. Scratchpad clients (top-left) marshal requests to memory-sized chunks and are filtered by private caches. The scratchpad controller handles central cache spill and fill requests, transforming them into operations on the host scratchpad memory. Central cache clients (top-right) must provide their own spill and fill drivers, which likely connect to drivers on the host. The central cache protocol tags each request and cached line with a specific client, enabling proper routing of responses and spill or fill requests.

addresses. Instead of reading and writing data from on-board memory, the scratchpad controller reads and writes host memory using either direct memory access or a protocol over an I/O channel.

2.6 Central Cache

Moving the backing storage from on-board RAM to host memory offers more space at the expense of access time. Configuring the now unused on-board RAM as a last-level cache can reduce this penalty. Because only one central cache controller is instantiated we can afford a more complicated controller. The platform's central cache controller is set associative with LRU replacement.

Clients connecting to the central cache identify themselves using a dictionary-based mechanism similar to the scratchpad ID allocation scheme described in Section 2.2.1. Like the scratchpad controller, the central cache constructs a unique address space for each client by concatenating client IDs and address requests from clients. This internal address space enables the central cache to associate entries with specific clients.

Clients connecting to the central cache must provide functions for spilling and filling memory lines. Pushing the details of spills and fills out of the central cache allows a variety of clients to connect, all sharing the same on-board RAM, each with unique methods of reading and writing their backing storage. The LRU central cache policy automatically optimizes the footprint of each client in the central cache based on the global access pattern of all clients.

Figure 3 shows the full caching hierarchy. The scratchpad controller, shown at the left in the platform, is one client of the central cache. Misses from all L1 scratchpad clients are passed by the scratchpad controller to the central cache. The scratchpad controller provides spill and fill functions to the central cache that write and read lines from the scratchpad memory stored on the host. Other clients of the central cache are shown in the top right of the figure. In this example configuration, each client instantiates a private L1 cache and provides its own spill and fill methods that communicate with the host. While not currently implemented in the platform, the spill and fill methods could be extended with coherence messages. Along with coherence messages, the central cache tag structure could be enhanced to represent line ownership. Hybrid algorithms sharing data between the FPGA and the host could then communicate and force synchronization between FPGA-side caches and host memory.

2.7 Language

LEAP is written in Bluespec SystemVerilog [2]. While the platform framework is language independent, some Bluespec features simplified our implementation. The language offers powerful static elaboration and type algebra, enabling predicate functions that govern whether operations that invoke predicated methods will fire within an FPGA cycle. As long as the user adopts a latency insensitive programming style, predicates defined within our implementation of scratchpad read request and response methods make it simple to replace a RAM block with a scratchpad by changing only a single line of source code. The method interfaces for RAM blocks and scratchpads are identical. Externally, they differ only in their timing and, consequently, their public methods' predicates. All LEAP RAM block and scratchpad classes share the following interface:

```
interface MEMORY_IFC#(type t_ADDR,
                      type t_DATA);
  method Action readReq(t_ADDR addr);
  method ActionValue#(t_DATA) readRsp();

  method Action write(t_ADDR addr,
                      t_DATA val);
endinterface
```

The definition above declares the interface to the class *MEMORY_IFC*. The class uses two abstract type names, *t_ADDR* and *t_DATA*, that will be bound to explicit types when instantiated. Three methods are defined: *readReq* requests a memory read, *readRsp* receives the read response and *write* updates the value at a location. Consider a trivial pipeline that reads a RAM block:

```
 1  typedef Bit#(8) ADDR_T;
 2  typedef Bit#(4) DATA_T;
 3
 4  // Allocate 256 4-bit elements
 5  MEMORY_IFC#(ADDR_T, DATA_T) mem
 6    <- mkBRAM();
 7
 8  FIFO#(ADDR_T) inQ <- mkFIFO();
 9  FIFO#(DATA_T) outQ <- mkFIFO();
10
11  rule recvInReq;
12    let addr = inQ.first();
13    inQ.deq();
14    mem.readReq(addr);
15  endrule
16
17  rule sendOutRsp;
18    let val <- mem.readRsp();
19    outQ.enq(val);
20  endrule
```

This pipeline defines a 256 entry RAM block of four bit objects on line 5. An input FIFO is defined on line 8 and an output FIFO on line 9. *recvInReq* consumes incoming requests on *inQ* and initiates a memory read request. *sendOutRsp* consumes the memory value response and forwards it to *outQ*, the output FIFO.

Rules in Bluespec are like always blocks in *Verilog*: they fire as a group in a single cycle, predicated by a boolean expression. Unlike Verilog, the predicate may be a function of the methods invoked inside the rule. The *recvInReq* rule works because the FIFO *first* method allows an enclosing rule to fire only when the FIFO is not empty and the memory *readReq* method may fire only when the request queue is not full. Similarly, the outbound FIFO permits its *enq* method to fire only when the FIFO is not full and the memory *readRsp* method requires that a response be available.

Transforming the pipeline to use scratchpads requires only a change to the instantiation of *mem*:

```
4  // Allocate 256 4-bit elements
5  MEMORY_IFC#(ADDR_T, DATA_T) mem
6    <- mkScratchpad(VDEV.SCRATCH.FBUF_Y,
7                    SCRATCHPAD_CACHED);
```

In this case, *mem* is instantiated as a scratchpad with a private L1 cache, stored in on-die RAM blocks, and a shared L2, stored in on-board RAM. The rules that invoke the *readReq* and *readRsp* methods are unchanged because the predicates that enable them are encoded in the *mkScratchpad* implementation. Like software on general purpose hardware, the timing may change when moving from one memory hierarchy to another without requiring a changes to source code. Developers may choose to optimize for a particular memory timing, but changes are not required for accuracy.

3. USING SCRATCHPADS

Scratchpads are intended to be useful for a wide variety of client algorithms and access patterns. LEAP provides a number of configuration options. Clients may request automatic instantiation of private caches or may, instead, connect directly with the central cache. Streaming clients, where caching would not improve performance, may bypass the central cache and communicate directly with host memory.

Our work with FPGAs began with the goal of building high-performance processor timing models. In the process, we recognized the potential from generalizing the platform interface. As a test of scratchpad performance, we analyzed a pair of distinct algorithms: an H.264 decoder and a multi-core processor timing model. We implemented both applications for a Virtex-5 LX330 on a Nallatech Intel Xeon FSB Accelerated Computing Platform. This module has 8MB of on-board DDR SDRAM and is plugged into a CPU socket on a host computer. The central cache is stored in the 8MB on-board RAM.

3.1 H.264 Decoding

Some implementation of H.264 [9, 21] is deployed on nearly every new device capable of displaying a video, including smart phones, graphics cards, and, of course, televisions. The computational requirements of decoding H.264 video vary depending on video resolution, frame rate, and level of compression used. At the low end, mobile phone applications favor videos encoded in low resolution formats at low frame rates, as small as QCIF (176×144) at 15 frames per second. At the high end of the spectrum, Blu-ray discs are encoded at 1080p (1920×1080) at 24 frames per second. At the low end, decoding may be handled by software. However, decoding HD streams typically requires specialized hardware as even the most recent processors strain under the burden of 1080p.

H.264 reconstructs video at the granularity of 16×16 pixel macroblocks, which may be further subdivided in some decoding steps. Although H.264 employs several other compression techniques, the highest levels of compression are derived from pixel prediction and correction. This technique involves the decoder predicting future pixels based on pixels it has already decoded. Because the decoder is deterministic, the encoder knows the behavior of the decoder at encoding time, and can include corrections if the decoder prediction will be incorrect. These corrective factors tend to have small magnitude relative to a whole pixel, yielding a high compression ratio.

Stream	QCIF	720p
Input	0.01	2.75
Output	0.50	26.5
Interprediction	1.00	67.0

Figure 4: H.264 Bandwidth Requirements For Sample Streams (MB/s)

H.264 can source pixels for prediction in two ways: *Intraprediction* predicts macroblocks in a frame from other previously decoded, spatially local, macroblocks in the same frame. *Interprediction* predicts macroblocks from motion-indexed pixels in previously decoded frames. Intraprediction takes advantage of spatial locality within a frame – colors and textures are similar across a frame. Interprediction takes advantage of temporal locality across a frames – adjacent frames tend to depict similar images. In the context of memory management, this second kind of prediction is of the greatest interest, since it necessarily involves retrieving data from previously computed, stored frames.

3.1.1 Memory Requirements

H.264 has three distinct memory streams: input, output, and interprediction. As shown in Figure 4, the interprediction stream dominates memory bandwidth usage. This dominance arises from the method by which blocks are interpredicted in H.264. Previous standards permitted only whole pixel motion vectors: interprediction of a macro block only involved copying some other macroblock and then applying a corrective factor to it. The bandwidth requirements of this style of interprediction must be strictly less than the output bandwidth, since not all blocks are interpredicted. However, H.264 permits sub-pixel motion vectors. To provide better image reconstruction in this case, H.264 applies blurring filters to pixels from as many as five different adjacent macroblocks to produce a single predicted pixel. In the worst case pixel dependency, thirty-six pixels must be accessed to compute a single output pixel. Interprediction requires high memory bandwidth, and, unlike the input and output streams, its access pattern is dynamic. Whole, half, and quarter pixel motion vectors may be present at any point in the frame.

3.1.2 Design Exploration

Fortunately, H.264 exhibits a number of properties that can be exploited to develop an efficient memory system. First, the memory streams for the intensity (luminance) and coloration (chrominance) fields are independent. Second, motion vectors tend to be spatially homogeneous, yielding a high degree of regional locality in pixel accesses. Third, interprediction feedback occurs at the granularity of a frame, thus simplifying management of coherence across multiple caches.

These properties result in an interesting design exploration space for memory hierarchies in H.264, both because H.264 is present on platforms with diverse performance and power characteristics and because of its significant memory requirements. A number of architectures are feasible, depending on the desired resolution, throughput, area, and power requirements of a design. For example, a cheap but power-hungry decoder might use a minimally sized cache, while a more expensive, but power efficient decoder might use multiple large caches to cut down on off-chip memory accesses. It is beyond the scope of our present study to find optimal solutions to these problems. Rather, we demonstrate, by producing a number of disparate designs, that the scratchpad abstraction is an effective means of exploring the memory hierarchy design space without substantial coding overhead.

For our memory hierarchy explorations, we use an existing H.264 implementation [7]. This codec originally targeted an ASIC implementation, but performance increases in FPGAs permit us to reuse the codec without significant modifications. The original design assumed a request-response memory hierarchy similar to scratchpads, but only implemented a simple SRAM-based L1 cache even though it partitioned its memory requests into three separate streams: one for the luminance field and for each of the two chrominance fields. As a result, the original hierarchy suffered from performance issues in the presence of slower backing storage, like DRAM, and could only achieve sufficient memory for HD decoding if the frame buffer was stored in a high-bandwidth, low-latency SRAM.

Because the original algorithm connects to a standard, timing-independent, memory interface, we can easily integrate a scratchpad-based hierarchy. We implemented three distinct memory hierarchies and attached them to the H.264 decoder, illustrated in Figure 5. First, we implemented a simple, shared, on-die, RAM-based frame buffer. This buffer is constrained by both the size of on-die RAM available on the FPGA and offered bandwidth, which must be multiplexed among the three field streams. It is worth noting that this memory architecture is chosen by many published H.264 implementations. Second, we implemented a single scratchpad memory hierarchy backed by the central cache. While the architecture makes use of the automatic mid-level cache provided by scratchpads, the lowest-level cache is still multiplexed among many users. Third, we used separate scratchpads to implement the storage for the luma and chroma components. Although this architecture requires a larger number of caches, the bandwidth offered by the hierarchy is substantially higher due to the distribution of requests across caches. Additionally, individual caches experience better performance due to the removal of conflict misses. In configuring the designs, we attempted to size memory parameters to achieve similar levels of on-die memory usage in each design.

3.1.3 Results

The performance results for the various memory hierarchy implementations are shown in Figure 6. Results are not shown for higher resolutions using on-die RAM blocks because they are not synthesizable on the FPGA, due to the size of the on-die RAM frame buffer. Unlike direct implementations employing on-die memory, the scratchpad versions require only caches on the FPGA and remain synthesizable as the problem size grows. As expected, the memory hierarchy in which only a single scratchpad is used offers less performance than the hierarchy in which the field memory streams are split across multiple platforms. However, the performance increase is much larger than a factor of three. This larger difference is a result of head-of-line blocking in the single scratchpad hierarchy. High-latency misses prevent faster, unrelated hits from exiting the in-order memory response queue, thereby stalling the processor pipeline. To determine the implementation area of the different memory hierarchies, shown in Figure 7, we combine the areas of the FPGA platform and H.264 memory control. Because of our complex synthesis process, it is difficult to separate individual components of the FPGA platform from one another, so an FPGA platform without any memory hierarchy is provided as a reference.

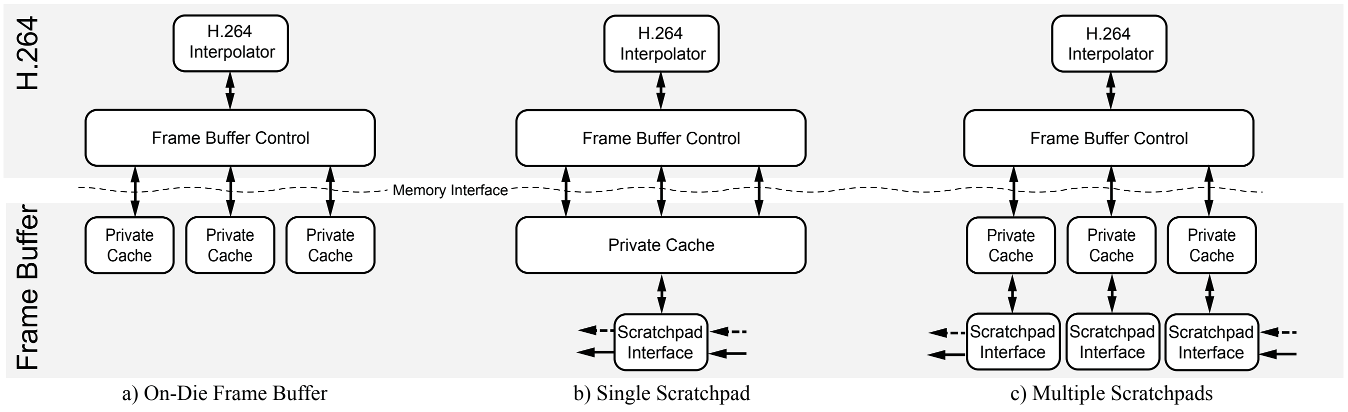

Figure 5: H.264 memory hierarchies explored. (a) Stores all interpolation data on the FPGA. (b) Stores the same data in a single scratchpad, allowing the decoder to work on larger frame resolutions. (c) Stores the data in multiple scratchpads, increasing memory I/O parallelism and reducing cache conflict misses and head-of-line blocking.

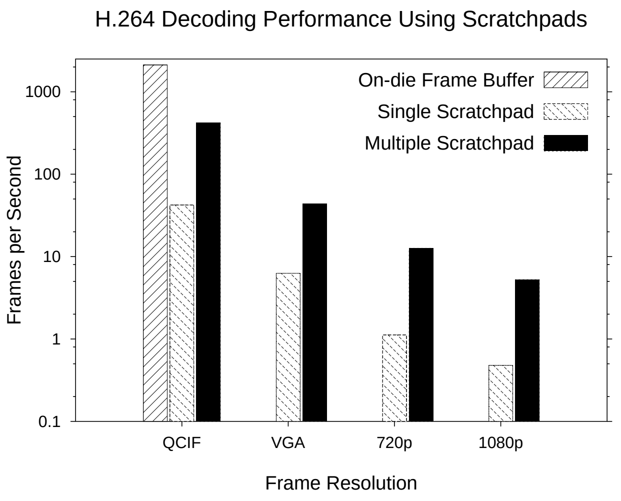

Figure 6: H.264 performance, plotted on a log scale.

3.2 Multi-Core Processor Timing Model

The second memory hungry workload we consider is a processor timing model. FPGAs are a well-known solution for emulating and prototyping designs after the RTL has been written. Recently, a new application has emerged: using FPGAs for accelerating architectural design-space exploration. This scenario presents two problems: not every circuit translates efficiently into an FPGA and FPGA capacity becomes a concern as it should not artificially limit architectural exploration. The solution to these problems is separation of the *FPGA clock* from the *model clock* [20]. This allows us to use FPGA-efficient circuits such as RAM blocks while simulating the timings of FPGA-inefficient circuits such as CAMs and multi-ported register files. RAM configured to model a CAM may be searched sequentially, and the *model clock cycle* is not incremented until the search concludes.

In separating the FPGA clock from the model clock we treat the FPGA as a highly parallel programming substrate. Like a software-based timing model, the FPGA is configured to compute model time algorithmically, independent of the FPGA's hardware clock. This programmatic computation of model time offers the model writer many opportunities for choosing space-efficient structures. For example, multiple instances of the same target model circuit may be represented by time-multiplexing a single physical instance in the model. A multi-core target processor may be modeled using a single physical instance of the core, with the multiplexed modeling algorithm stepping round robin through the logical target cores. Only the data for each core, such as the register file state, must be replicated. The modeling algorithm is typically implemented as many stages in an FPGA pipeline. Time-multiplexed instances then run in parallel along the pipeline, in the same fashion as instruction execution in standard processor pipelines [23].

3.2.1 Scratchpads in Timing Models

Time-multiplexing solves only one space problem: logic for many instances of the same target processor now fits on an FPGA. Another problem remains: the data representing the state for each multiplexed object is proportional to the number of instances. What if the required data does not fit on-board the FPGA? Once again, we take advantage of the separation between the FPGA clock and the model clock. There is no specific FPGA-clock timing required for simulator accesses to modeling data. A target model's register file and cache states may be stored anywhere in an FPGA or host memory hierarchy. High latency storage may adversely affect simulator performance, but long latencies will have no effect on the results of simulation.

Timing models of processor caches are obvious candidates for off-FPGA storage. There is insufficient on-die RAM to hold modeled cache state, especially when many simulated

Memory Hierarchy	Registers	LUTs	On-Die RAM KB
Platform Components	9599	11239	8
On-die Frame Buffer	22834	31880	244
Single Scratchpad	53941	66184	208
Multiple Scratchpads	37815	52748	208

Figure 7: H.264 synthesis results, targeting Nallatech ACP.

processors are multiplexed on a single FPGA. We employ scratchpad memories to store modeled cache states.

In our simulator we split the model into two components: *functional* and *timing*. Both run on the FPGA. The functional model is responsible for architecturally correct execution of instructions according to the target instruction set architecture. The timing model is solely responsible for computing the time required for instructions to complete on the target microarchitecture. Only data required to compute target model cycles flows through the timing model's FPGA pipelines. Addresses of memory references are present in the timing model because they are required to compute cache occupancy. The values actually loaded and stored by memory references do not flow through the timing model because they do not affect timing. The values loaded and stored flow only through the functional model. Consequently, a timing model of a cache need store only the tags associated with each cache line. The data for the line itself is unnecessary, since it is redundant with the functional model state. While this optimization significantly reduces the complexity and memory footprints of cache timing models, tag arrays for multi-core simulation are still relatively large compared to FPGA RAM block sizes.

3.2.2 Experimental Configuration

Our timing models are modular and may be configured into a variety of target topologies, ranging from simple single-model-cycle cores to detailed out-of-order cores. Cores may optionally have private instruction and data caches. Cores may be connected together using a variety of network topologies, including rings and meshes.

To understand the impact of scratchpads we are interested in the performance of the simulator itself, not the performance of any particular modeled microarchitecture. For this study we picked a simple target architecture: a set of independent cores. Each core is connected to a private memory controller. The performance of the architecture thus scales linearly with the number of cores in the target. This microarchitecture is well suited for measuring simulator performance as the problem size varies. For this target, a multiplexed simulator with access to infinite on-die memory and with fully utilized FPGA pipelines should have a constant simulation speed, measured in simulated instructions per wall-clock second, as the number of target cores varies. This is due do the multiplexing. While the available parallelism within the target varies relative to the number of target cores, the parallelism within the simulator implementation is limited by the single multiplexed core model. Of course there is not infinite on-die memory within the FPGA, which is why we employ scratchpads. With this simulator configuration the impact of varying FPGA memory hierarchies on run-time can be tested. By adjusting the number of target cores we vary the memory footprint.

We tested a model configuration in which each core has a simulated private, physically indexed, 128KB, direct mapped instruction and also a 128KB, direct mapped, data cache with 32 byte lines. Each cache is represented only by the address tags corresponding to references. For a 34 bit physical address space, each processor requires 2KB of storage for each cache model. The model is multiplexed, with only a single physical instance of a core described on the FPGA. Likewise, the cache tag arrays are multiplexed. A single multiplexed array holds the storage representing the cache tag array for all processor instruction caches and another holds the tags for all data caches. The arrays grow linearly with the number of simulated cores. We varied the problem size between 8 and 64 simulated cores, translating to total cache tag array footprints between 32KB and 256KB.

As a test of scratchpad performance we configured two flavors: one in which the cache tag arrays are stored directly in FPGA RAM blocks and another in which the arrays are stored in scratchpads. As noted in Section 2.7, these variations differ only in a single source line that either instantiates storage as on-die RAM or as a scratchpad. The algorithm accessing the storage is written deliberately to function correctly independent of timing. The scratchpads are fronted by 4KB of private L1 scratchpads caches, stored in RAM blocks on the FPGA. Note that these are physical caches on the FPGA to speed up scratchpad references, not part of the timing model target architecture.

The functional model also has data storage requirements. While both virtual to physical address translation and functional memory are managed by a connected software-based simulator, they are cached on the FPGA to improve simulator performance. Both the translation lookaside buffer (TLB) used for virtual to physical translation and functional memory are clients of the central cache. The TLB employs a 4KB private cache and functional memory is fronted by a 32KB cache to reduce pressure on the central cache. Synthesized FPGA resource requirements are shown in Figure 8.

Model	**Registers**	**LUTs**	**On-Die RAM KB**
16 Cores / On-die RAM	79729	97736	5670
16 Cores / Scratchpads	94341	114220	3582
32 Cores / Scratchpads	101230	126114	3978
64 Cores / Scratchpads	108491	148630	5130

Figure 8: Timing model synthesis results, targeting Nallatech ACP.

A workload consisting of eight reference programs from SPEC 2000 was run through the simulator: ammp, applu, gcc, mcf, mesa, parser, perlbmk and wupwise. Because we are interested in simulator performance and not the performance of the modeled target architecture, we simply need programs to generate scratchpad traffic. The simulator was run for 500 million instructions on each target core. The eight workloads were replicated for configurations with more than eight cores. Each core was allocated a private physical address space, so no memory was shared and the problem size grew linearly with the number of cores.

3.2.3 Results

The simulation rate, varying the number of cores in the target architecture, is shown in Figure 9. We measure simulator performance as the number of instructions simulated relative to wall clock time. Because the simulator is a single, multiplexed, instance, the simulation rate on unlimited memory is mostly constant and independent of the number of simulated cores. (In fact, because the simulator itself is pipelined on the FPGA, simulation rates measured in MIPS may improve as more cores are simulated. We see this in the speedup between 8 and 16 cores.)

For 8 and 16 core models it was possible to build equivalent simulators using both scratchpads and RAM blocks to hold the timing model's cache tag arrays. For those configurations,

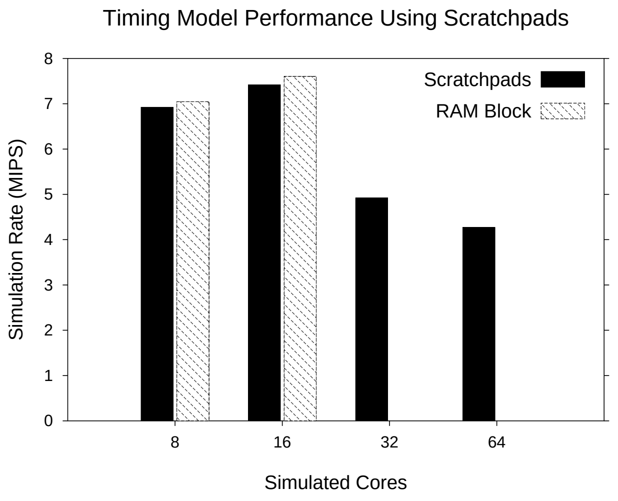

Figure 9: Timing model throughput, measured in simulation rate (millions of instructions per second) relative to wall clock time. For 8 and 16 core simulations it is physically possible to synthesize cache tag arrays in RAM blocks. Above 16 cores, the arrays no longer fit on the FPGA.

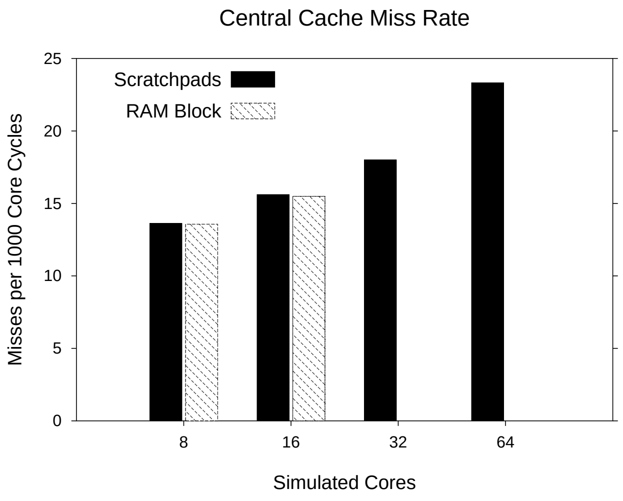

Figure 10: Central cache miss rate.

the performance with scratchpads was nearly identical to the direct implementation. The FPGA's RAM block area is simply not large enough for 32 and 64 core models. We see interesting results for those two configurations. Simulator performance drops for two reasons. First, central cache capacity is a problem, as seen in the miss rates in Figure 10. Second, the FPGA pipeline becomes saturated, leaving no room for additional parallelism. While performance clearly decreases as central cache pressure grows due to the sizes of scratchpads and functional memory, it should be noted that without these techniques the simulator would simply have failed to synthesize for the FPGA. Performance degradation due to cache footprint growth is similarly problematic on general purpose hardware. We have achieved our goal of an FPGA platform with a general purpose programming model.

4. CONCLUSION

Automatic cache instantiation frees the FPGA application developer to concentrate more on algorithmic design and less on memory management. A platform-provided memory hierarchy automatically partitions the on-board memory among competing clients. Without it, application writers would be forced to manage access to on-die RAM blocks and shared DDR RAM explicitly. Designers would most likely hard partition memory among clients. Like application development on general purpose machines, core algorithms may be written and then tuned for their particular memory access patterns.

LEAP scratchpads are in active use in projects as diverse as H.264 decoders and micro-architectural timing models. We have found them to be particularly useful for managing storage in applications that require multiple, large, random access buffers.

5. REFERENCES

[1] D. Andrews, R. Sass, E. Anderson, J. Agron, W. Peck, J. Stevens, F. Baijot, and E. Komp. Achieving Programming Model Abstractions for Reconfigurable Computing. *IEEE Trans. Very Large Scale Integr. Syst.*, 16(1):34–44, 2008.

[2] Arvind. Bluespec: A Language for Hardware Design, Simulation, Synthesis and Verification. In *MEMOCODE '03: Proceedings of the First ACM and IEEE International Conference on Formal Methods and Models for Co-Design*, page 249. IEEE Computer Society, 2003.

[3] Z. Budimlic, A. M. Chandramowlishwaran, K. Knobe, G. N. Lowney, V. Sarkar, and L. Treggiari. Declarative Aspects of Memory Management in the Concurrent Collections Parallel Programming Model. In *DAMP '09: Proceedings of the 4th Workshop on Declarative Aspects of Multicore Programming*, pages 47–58. ACM, 2008.

[4] H. J. C. Chung, Eric S. and K. Mai. CoRAM: An In-Fabric Memory Abstraction for FPGA-based Computing. In *FPGA '11: Proceedings of the 19th Annual ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, 2011.

[5] N. Dave, M. C. Ng, M. Pellauer, and Arvind. A design flow based on modular refinement. In *Formal Methods and Models for Codesign (MEMOCODE), 2010 8th IEEE/ACM International Conference on*, pages 11 –20, Jul. 2010.

[6] J. B. Dennis. Segmentation and the Design of Multiprogrammed Computer Systems. *J. ACM*, 12(4):589–602, 1965.

[7] K. Fleming, C.-C. Lin, N. Dave, Arvind, G. Raghavan, and J. Hicks. H.264 Decoder: A Case Study in Multiple Design Points. In *Formal Methods and Models for Co-Design, 2008. MEMOCODE 2008. 6th ACM/IEEE International Conference on*, pages 165 –174, Jun. 2008.

[8] M. B. Gokhale, J. M. Stone, J. Arnold, and M. Kalinowski. Stream-Oriented FPGA Computing in the Streams-C High Level Language. In *FCCM '00: Proceedings of the 2000 IEEE Symposium on Field-Programmable Custom Computing Machines*,

page 49. IEEE Computer Society, 2000.

[9] I.-T. V. C. E. Group. Draft ITU-T Recommendation and Final Draft International Standard of Joint Video Specification, May, 2003.

[10] S. Gupta, N. Dutt, R. Gupta, and A. Nicolau. SPARK: A High-Lev l Synthesis Framework for Applying Parallelizing Compiler Transformations. In *VLSID '03: Proceedings of the 16th International Conference on VLSI Design*, page 461. IEEE Computer Society, 2003.

[11] S. S. Huang, A. Hormati, D. F. Bacon, and R. Rabbah. Liquid Metal: Object-Oriented Programming Across the Hardware/Software Boundary. In *ECOOP '08: Proceedings of the 22nd European conference on Object-Oriented Programming*, pages 76–103. Springer-Verlag, 2008.

[12] W. W. Hwu and P. P. Chang. Achieving High Instruction Cache Performance with an Optimizing Compiler. *SIGARCH Comput. Archit. News*, 17(3):242–251, 1989.

[13] C.-K. Luk and T. C. Mowry. Cooperative Prefetching: Compiler and Hardware Support for Effective Instruction Prefetching in Modern Processors. In *MICRO 31: Proceedings of the 31st Annual ACM/IEEE International Symposium on Microarchitecture*, pages 182–194. IEEE Computer Society Press, 1998.

[14] W. A. Najjar, W. Böhm, B. A. Draper, J. Hammes, R. Rinker, J. R. Beveridge, M. Chawathe, and C. Ross. High-Level Language Abstraction for Reconfigurable Computing. *Computer*, 36(8):63–69, 2003.

[15] P. Panda, N. Dutt, and A. Nicolau. Local Memory Exploration and Optimization in Embedded Systems. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 18(1):3 –13, Jan. 1999.

[16] R. J. Pankhurst. Operating Systems: Program Overlay Techniques. *Commun. ACM*, 11(2):119–125, 1968.

[17] A. Parashar, M. Adler, K. Fleming, M. Pellauer, and J. Emer. LEAP: A Virtual Platform Architecture for FPGAs. In *CARL '10: The 1st Workshop on the Intersections of Computer Architecture and Reconfigurable Logic*, 2010.

[18] A. Parashar, M. Adler, M. Pellauer, and J. Emer. Hybrid CPU/FPGA Performance Models. In *WARP '08: The 3rd Workshop on Architectural Research Prototyping*, 2008.

[19] M. Pellauer, M. Adler, D. Chiou, and J. Emer. Soft Connections: Addressing the Hardware-Design Modularity Problem. In *DAC '09: Proceedings of the 46th Annual Design Automation Conference*, pages 276–281. ACM, 2009.

[20] M. Pellauer, M. Vijayaraghavan, M. Adler, Arvind, and J. Emer. A-Port Networks: Preserving the Timed Behavior of Synchronous Systems for Modeling on FPGAs. *ACM Trans. Reconfigurable Technol. Syst.*, 2(3):1–26, 2009.

[21] I. E. Richardson. *H.264 and MPEG-4 Video Compression*. John Willey & Sons, 2003.

[22] J. Simsa and S. Singh. Designing Hardware with Dynamic Memory Abstraction. In *FPGA '10: Proceedings of the 18th Annual ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, pages 69–72. ACM, 2010.

[23] Z. Tan, A. Waterman, R. Avizienis, Y. Lee, H. Cook, D. Patterson, and K. Asanović. RAMP Gold: an FPGA-Based Architecture Simulator for Multiprocessors. In *DAC '10: Proceedings of the 47th Design Automation Conference*, pages 463–468. ACM, 2010.

[24] Xilinx, Inc. UG363: Virtex-6 FPGA Memory Resources User Guide. 2010.

[25] P. Yiannacouras and J. Rose. A Parameterized Automatic Cache Generator for FPGAs. In *Proc. Field-Programmable Technology (FPT)*, pages 324–327, 2003.