LEAP Scratchpads: Automatic Memory and Cache Management for Reconfigurable Logic *

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.

Categories and Subject Descriptors

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

General Terms

Algorithms, Performance

Keywords

FPGA, memory management, caches

*An extended version of this paper showing experimental results is available as MIT Technical Report MIT-CSAIL-TR-2010-054

FPGA'11, February 27–March 1, 2011, Monterey, California, USA.

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. 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. [6, 7]. 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 [2].

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 [4, 5, 8] and a handful are beginning to use languages such as Bluespec [1, 11] 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 [3].

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 [12]. 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.

We have written LEAP *(Logic-based Environment for Application Programming)* [9], 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 [10]. 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. For a review of related work and analysis of workloads using LEAP scratchpads please see the extended version of this paper, released as MIT Technical Report MIT-CSAIL-TR-2010-054.

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 [13]. 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 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. 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.

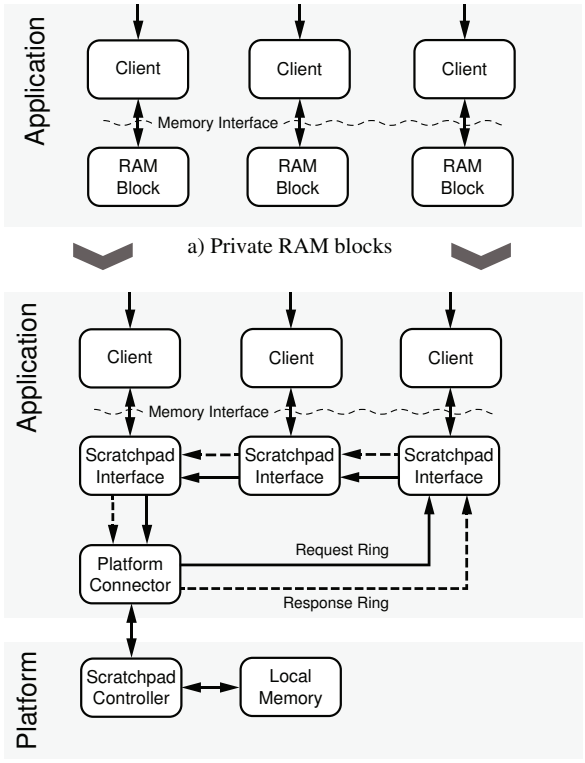

Figure 1: Transforming private RAM blocks to scratchpads. The memory interface between clients and storage is unchanged following the transformation. Only timing is different.

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 [11]. 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.

2.3 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.

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.

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. 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.

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.

3. 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.

4. REFERENCES

[1] 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.

[2] 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.

[3] 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.

[4] 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.

[5] 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.

[6] 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.

[7] 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.

[8] 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.

[9] 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.

[10] 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.

[11] 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.

[12] 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.

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