CoRAM: An In-Fabric Memory Architecture
for FPGA-based Computing

Eric S. Chung, James C. Hoe, and Ken Mai

Carnegie Mellon University
5000 Forbes Ave.
Pittsburgh, PA 15213
{echung, jhoe, kenmai}@ece.cmu.edu

ABSTRACT

FPGAs have been used in many applications to achieve orders-of-magnitude improvement in absolute performance and energy efficiency relative to conventional microprocessors. Despite their promise in both processing performance and efficiency, FPGAs have not yet gained widespread acceptance as mainstream computing devices. A fundamental obstacle to FPGA-based computing today is the FPGA's lack of a common, scalable memory architecture. When developing applications for FPGAs, designers are often directly responsible for crafting the application-specific infrastructure logic that manages and transports data to and from the processing kernels. This infrastructure not only increases design time and effort but will frequently lock a design to a particular FPGA product line, hindering scalability and portability. We propose a new FPGA memory architecture called Connected RAM (CoRAM) to serve as a portable bridge between the distributed computation kernels and the external memory interfaces. In addition to improving performance and efficiency, the CoRAM architecture provides a virtualized memory environment as seen by the hardware kernels to simplify development and to improve an application's portability and scalability.

Categories and Subject Descriptors

C.0 [**Computer System Organization**]: [System architectures]

General Terms

Design, standardization

Keywords

FPGA, abstraction, memory, reconfigurable computing

1. INTRODUCTION

With power becoming a first-class architectural constraint, future computing devices will need to look beyond

general-purpose processors. Among the computing alternatives today, Field Programmable Gate Arrays (FPGA) have been applied to many applications to achieve orders-of-magnitude improvement in absolute performance and energy efficiency relative to conventional microprocessors (e.g., [11, 6, 5]). A recent study [6] further showed that FPGA fabrics can be an effective computing substrate for floating-point intensive numerical applications.

While the accumulated VLSI advances have steadily improved the FPGA fabric's processing capability, FPGAs have not yet gained widespread acceptance as mainstream computing devices. A commonly cited obstacle is the difficulty in programming FPGAs using low-level hardware development flows. A fundamental problem lies in the FPGA's lack of a common, scalable memory architecture for application designers. When developing for an FPGA, a designer has to create from bare fabric not only the application kernel itself but also the application-specific infrastructure logic to support and optimize the transfer of data to and from external memory interfaces. Very often, creating or using this infrastructure logic not only increases design time and effort but will frequently lock a design to a particular FPGA product line, hindering scalability and portability. Further, the support mechanisms which users are directly responsible for will be increasingly difficult to manage in the future as: (1) memory resources (both on- and off-chip) increase in number and become more distributed across the fabric, and (2) long-distance interconnect delays become more difficult to tolerate in larger fabric designs.

Current FPGAs lack essential abstractions that one comes to expect in a general purpose computer—i.e., an Instruction Set Architecture (ISA) that defines a standard agreement between hardware and software. From a computing perspective, a common architectural definition is a critical ingredient for programmability and for application portability. To specifically address the challenges related to memory on FPGAs, the central goal of this work is to create a shared, scalable memory architecture suitable for future FPGA-based computing devices. Such a memory architecture would be used in a way that is analogous to how general purpose programs universally access main memory through standard "loads" and "stores" as defined by an ISA—without any knowledge of hierarchy details such as caches, memory controllers, etc. At the same time, the FPGA memory architecture definition cannot simply adopt what exists for general purpose processors and instead, should reflect the spatially distributed nature of today's FPGAs—consisting
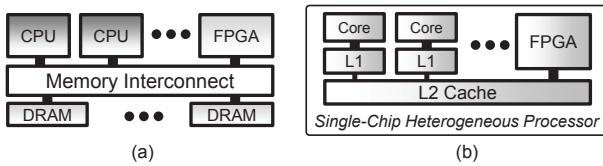
Figure 1: Assumed System Context.

of up to millions of interconnected LUTs and thousands of embedded SRAMs [24]. Working under the above premises, the guiding principles for the desired FPGA memory architecture are:

- *The architecture should present to the user a common, virtualized appearance of the FPGA fabric, which encompasses reconfigurable logic, its external memory interfaces, and the multitude of SRAMs—while freeing designers from details irrelevant to the application itself.*

- *The architecture should provide a standard, easy-to-use mechanism for controlling the transport of data between memory interfaces and the SRAMs used by the application throughout the course of computation.*

- *The abstraction should be amenable to scalable FPGA microarchitectures without affecting the architectural view presented to existing applications.*

Outline. Section 2 presents an overview of the CoRAM memory architecture. Section 3 discusses a possible microarchitectural space for implementing CoRAM. Section 4 demonstrates concrete usage of CoRAM for three example application kernels—Black-Scholes, Matrix-Matrix Multiplication and Sparse Matrix-Vector Multiplication. Section 5 presents an evaluation of various microarchitectural approaches. We discuss related work in Section 6 and offer conclusions in Section 7.

2. CORAM ARCHITECTURE

2.1 System Context

The CoRAM memory architecture assumes the co-existence of FPGA-based computing devices along with general-purpose processors in the context of a shared memory multiprocessor system (see Figure 1). The CoRAM architecture assumes that reconfigurable logic resources will exist either as stand-alone FPGAs on a multiprocessor memory bus or integrated as fabric into a single-chip heterogeneous multicore. Regardless of the configuration, it is assumed that memory interfaces for loading from and storing to a linear address space will exist at the boundaries of the reconfigurable logic (referred to as edge memory in this paper). These implementation-specific edge memory interfaces could be realized as dedicated memory/bus controllers or even coherent cache interfaces. Like commercial systems available today (e.g., Convey Computer [8]), reconfigurable logic devices can directly access the same virtual address space of general purpose processors (e.g., by introducing MMUs at the boundaries of fabric). The combined integration of virtual memory and direct access to the memory bus allows applications to be efficiently and easily partitioned across general-purpose processors and FPGAs, while leveraging the unique strengths of each respective device. A nearby processor is useful for handling tasks not well-suited to FPGAs— e.g., providing the OS environment, executing system calls,
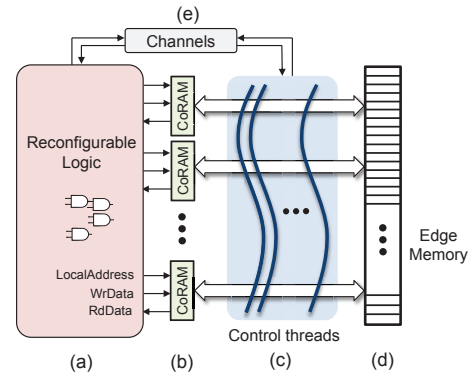

Figure 2: CoRAM Memory Architecture.

and initializing the memory contents of an application prior to its execution on the FPGA.

2.2 Architectural Overview

Within the boundaries of reconfigurable logic, the CoRAM architecture defines a portable application environment that enforces a separation of concerns between computation and on-chip memory management. Figure 2 offers a conceptual view of how applications are decomposed when mapped into reconfigurable logic with CoRAM support. The reconfigurable logic component shown in Figure 2a is a collection of LUT-based resources used to host the algorithmic kernels of a user application. It is important to note that the CoRAM architecture places no restriction on the synthesis language used or the internal complexity of the user application. For portability reasons, the only requirement is that user logic is never permitted to directly interface with off-chip I/O pins, access memory interfaces, or be aware of platform-specific details. Instead, applications can only interact with the external environment through a collection of specialized, distributed SRAMs called CoRAMs that provide on-chip storage for application data (see Figure 2b).

CoRAMs. Much like current FPGA memory architectures, CoRAMs preserve the desirable characteristics of conventional fabric-embedded SRAM [16]—they present a simple, wire-level SRAM interface to the user logic with local address spaces and deterministic access times (see Figure 2b), are spatially distributed, and provide high aggregate on-chip bandwidth. They can be further composed and configured with flexible aspect ratios. CoRAMs, however, deviate drastically from conventional embedded SRAMs in the sense that the data contents of individual CoRAMs are actively managed by finite state machines called "control threads" as shown in Figure 2c.

Control threads. Control threads form a distributed collection of logical, asynchronous finite state machines for mediating data transfers between CoRAMs and the edge memory interface. Each CoRAM is managed by at most a single control thread, although a control thread could manage multiple CoRAMs. Under the CoRAM architectural paradigm, user logic relies solely on control threads to access external main memory over the course of computation. Control threads and user logic are peer entities that interact over predefined, two-way asynchronous channels (see Figure 2e). A control thread maintains local state to facilitate its activities and will typically issue address requests to the edge memory interface on behalf of the application; upon comple-

tion, the control thread informs the user logic through channels when the data within specific CoRAMs are ready to be accessed through their locally-addressed SRAM interfaces. Conversely, the user logic can also write its computational results into CoRAMs and issue "commands" to the control threads via channels to write the results to edge memory.

Control actions. To express the memory access requirements of an FPGA application, control threads can only invoke a predefined set of memory and communication primitives called control actions. Control actions describe logical memory transfer commands between specific embedded CoRAMs and the edge memory interface. A control thread at the most basic level comprises a sequence of control actions to be executed over the course of a program. In general, a control thread issues control actions along a dynamic sequence that can include cycles and conditional paths.

Software versus RTL. An important issue that merits early discussion is in determining what the "proper" level of abstraction should be for expressing control threads and control actions. The most straightforward approach to exposing control actions to user logic is to distribute standard, wire-level interfaces throughout the fabric. In this case, the application designer would be directly responsible for constructing hardware control threads (i.e., FSM) that generate memory address requests on behalf of the user logic and issue control actions through the standard interfaces.

In this paper, we make the key observation that from a performance perspective, expressing control threads in a low-level abstraction such as RTL is not a critical requirement. In many cases, the process of generating address requests to main memory is not a limiting factor to FPGA application performance since most time is either spent waiting for memory responses or for computation to progress. For the remainder of this paper, it is assumed that control threads are expressed in a high-level C-based language to facilitate the dynamic sequencing of control actions.

Our selection of a C-based language affords an application developer not only simpler but also more natural expressions of control flow and memory pointer manipulations. The control threads implemented in software would be limited to a subset of the C language and would exclude "software" features such as dynamic memory allocation. Any high-level language used must be synthesizable to finite state machines or even compiled to hard microcontrollers that can execute control thread programs directly if available in an FPGA. Generally, the overall inefficiencies of executing a high-level language would not directly impede the overall computation throughput because the control threads do not "compute" in any usual sense and are used only to generate and sequence the control actions required by an application.

2.3 CoRAM Architecture in Detail

In this section, we describe in detail the standard memory management interface exported by control actions, and how they are invoked within control threads. Figure 3 illustrates the set of control actions available to an application developer. The control actions shown have the appearance of a memory management API, and abstract away the details of the underlying hardware support—similar to the role served by the Instruction Set Architecture (ISA) between software and evolving hardware implementations. As will be demonstrated later in Section 4, the basic set of con-

```
/*** CoRAM handle definition and acquisition ***/
struct {int n; int width; int depth; ...} coh;
coh  get_coram(instance_name, ...);
coh  append_coram(coh coram, bool interleave, ...);

/*** Singleton control actions ***/
void coram_read(coh coram, void *offset,
                void *memaddr, int bytes);
tag  coram_read_nb(coh coram, ...);
void coram_write(coh coram, void *offset,
                 void *memaddr, int bytes);
tag  coram_write_nb(coh coram, ...);
void coram_copy(coh src, coh dst, void *srcoffset,
                void *dstoffset, int bytes);
tag  coram_copy_nb(coh src, coh dst, ...);
bool check_coram_done(coh coram, tag, ...);
void coram_membar();

/*** Collective control actions ***/
void collective_write(coh coram, void *offset,
                      void *memaddr, int bytes);
void collective_read(coh coram, void *offset,
                     void *memaddr, int bytes);

/*** Channel control actions ***/
void fifo_write(fifo f, Data din);
Data fifo_read(fifo f);
void ioreg_write(reg r, Data din);
Data ioreg_read(reg r);
```

Figure 3: Control action definitions.

trol actions defined can be used to compose more sophisticated memory "personalities" such as scratchpads, caches, and FIFOs—each of which are tailored to the memory patterns and desired interfaces of specific applications.

The first argument to a control action is typically a program-level identifier (called co-handle) for an individual CoRAM or for a collection of CoRAMs that are functioning as a single logical structure (in the same way embedded SRAMs can be composed). The co-handle encapsulates both static and runtime information of the basic width and depth of the logical CoRAM structure and the binding to physical CoRAM blocks in the fabric. (A co-handle can also be used to represent a channel resource such as a FIFO or I/O register, although only the appropriate communication control actions are compatible with them.) The CoRAM definitions in Figure 3 comprise basic memory block transfer operations (*coram_read*, *coram_write*) as well as on-chip CoRAM-to-CoRAM data transfers (*coram_copy*). In addition to memory operations, control actions support asynchronous two-way communication between user logic and control threads via FIFOs (e.g., *fifo_write*) and I/O registers (e.g., *ioreg_write*). Although not listed in the definitions, control threads can also communicate to each other through ordered message-passing primitives.

Example. To illustrate how control actions are used, the example in Figure 4 shows how a user would (1) instantiate a CoRAM as a black-box module in their application, and (2) program a corresponding control thread to read a single data word from edge memory into the CoRAM. The control thread program shown in Figure 4 (right) first acquires a co-handle (L2), and passes it into a *coram_write* control action (L3), which performs a 4-byte memory transfer from the edge memory address space to the CoRAM blocks named by the co-handle. To inform the application when the data is ready to be accessed for computation, the control thread
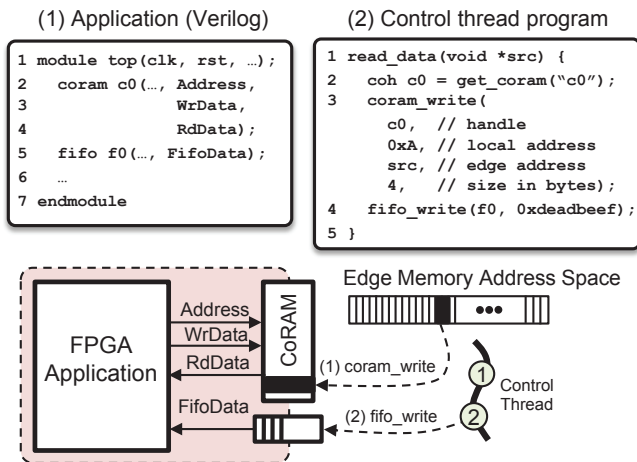
Figure 4: Example Usage of CoRAMs.

passes a token to the user logic using the *fifo_write* control action (the acquisition of the channel FIFO's co-handle is omitted for brevity).

Advanced control actions. Control threads can employ more advanced control actions to either increase memory level parallelism or to customize data partitioning. The non-blocking control actions (e.g., *coram_write_nb*) explicitly allow for concurrent memory operations and return a tag that must be monitored using *check_coram_done*. A control thread can also invoke *coram_membar*, which serializes on all previously executed non-blocking control actions by that control thread. For parallel transfers to a large number of CoRAMs, a collective form of read and write control actions is also supported. In the collective form, *append_handle* is a helper function that can be used to compose a static list of CoRAMs. The newly returned co-handle can then be used with collective control actions to perform transfers to the aggregated CoRAMs as a single logical unit. When operating upon the composed handle, sequential data arriving from memory can either be striped across the CoRAMs' local addresses in a concatenated or word-interleaved pattern. Such features allow the user to customize the partitioning of application data across the multiple distributed CoRAMs within the reconfigurable logic.

Future enhancements. It is not difficult to imagine that many variants of the above control actions could be added to support more sophisticated patterns or optimizations (e.g., broadcast from one CoRAM to many, prefetch, strided access, etc.). In a commercial production setting, control actions—like instructions in an ISA—must be carefully defined and preserved to achieve the value of portability and compatibility. Although beyond the scope of this work, compilers could play a significant role in static optimization of control thread programs. Analysis could be used, for example, to identify non-conflicting control actions that are logically executed in sequence but can actually be executed concurrently without affecting correctness.

3. CORAM MICROARCHITECTURE

The CoRAM architecture presented thus far has deliberately omitted the details of how control threads are actually executed and how data is physically transported between the
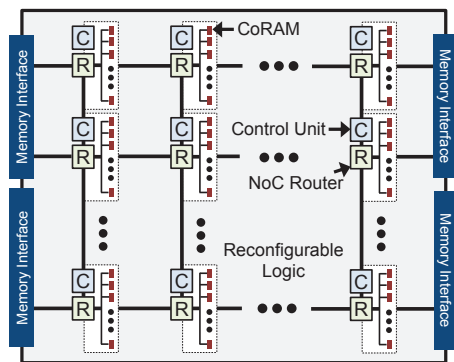

Figure 5: Conceptual Microarchitecture Sketch of Reconfigurable Logic with CoRAM Support.

CoRAMs and the edge memory interfaces. The CoRAM architecture definitions (i.e., control actions) form a contractual agreement between applications and hardware implementations. In the ideal case, a good implementation of CoRAM should provide robust hardware performance across a range of applications without affecting correctness and without significant tuning required by the user.

Hardware overview. By construction, the CoRAM architecture naturally lends itself to highly distributed hardware designs. The microarchitectural "sketch" shown in Figure 5 is architected in mind to scale up to thousands of embedded CoRAMs based on current FPGA design trends [24]. CoRAMs, like embedded SRAMs in modern FPGAs, are arranged into vertical columns [26] and organized into localized clusters. Each cluster is managed by an attached Control Unit, which is a physical host responsible for executing the control programs that run within the cluster. Control programs can be realized by direct synthesis into reconfigurable logic (e.g., using high-level synthesis flows) or can be compiled and executed on dedicated multithreaded microcontrollers (e.g., a multithreaded RISC pipeline). Control Units must also maintain internal queues and scoreboarding logic to track multiple outstanding control actions, and to allow querying of internal state (e.g., *check_coram_done*).

Data distribution. An integral component to the Control Unit is the network-on-chip, which is responsible for routing memory address requests on behalf of Control Units to a multitude of distributed memory interfaces and delivering data responses accordingly. Within each cluster, multiple CoRAMs share a single network-on-chip router for communication and data transport. At the macroscale level, multiple routers are arranged in a 2D mesh to provide global connectivity between clusters and memory interfaces. Internal to each cluster, queues and local interconnect provides connectivity between the CoRAMs and the shared router interface. The local interconnect internal to the cluster also contains marshalling logic to break large data transfers from the network into individual words and to steer them accordingly to the constituent CoRAMs based on the data partitioning a user desires (e.g., a *collective_write*).

Soft versus hard logic. To implement the CoRAM architecture, the most convenient approach in the short term would be to layer all the required CoRAM functionality on top of conventional FPGAs. In the long term, FPGAs developed in mind with dedicated CoRAM architectural sup-
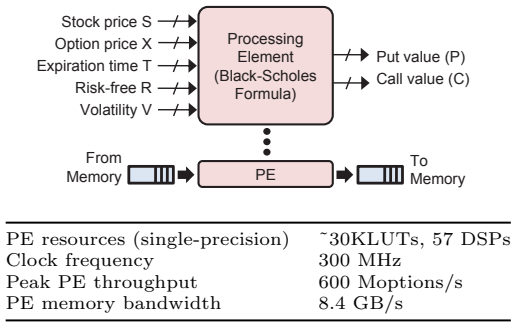
Figure 6: Black-Scholes Processing Element.

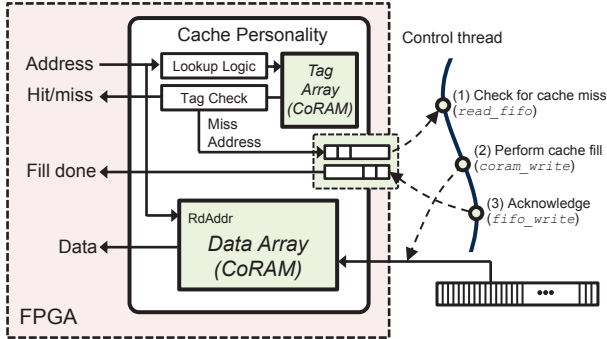| PE resources (single-precision)	~30KLUTs, 57 DSPs
Clock frequency	300 MHz
Peak PE throughput	600 Moptions/s
PE memory bandwidth	8.4 GB/s

Figure 7: Stream FIFO memory personality.

port can become more economical if certain features become popularly used. From the perspective of a fabric designed to support computing, we contend that a hardwired network-on-chip (NoC) offers significant advantages, especially if it reduces or eliminates the need for long-distance routing tracks in today's fabrics. Under the CoRAM architectural paradigm, global bulk communications are restricted to between CoRAM-to-CoRAM or CoRAM-to-edge. Such a usage model would be better served by the high performance (bandwidth and latency) and the reduced power and energy from a dedicated hardwired NoC that connects the CoRAMs and the edge memory interfaces. With a hardwired NoC, it is also more cost-effective in area and energy to over-provision network bandwidth and latency to deliver robust performance across different applications. Similarly, the control units used to host control threads could also support "hardened" control actions most commonly used by applications. The microarchitecture shown in Figure 5, with its range of parameters, will be the subject of a quantitative evaluation in Section 5. We next present three case studies that demonstrate concrete usage of CoRAM.

4. CoRAM IN USAGE

The CoRAM architecture offers a rich abstraction for expressing the memory access requirements of a broad range of applications while hiding memory system details unrelated to the application itself. This section presents our experiences in developing three non-trivial application kernels using the CoRAM architecture. Below, we explain each kernel's unique memory access pattern requirements and discuss key insights learned during our development efforts. The control thread examples presented in this section are excerpts from actual applications running in our CoRAM

```
1   coh ram = get_coram(...); // handle to FIFO buffer
2   char *src = ...; // initialized to Black-Scholes data
3   int src_word = 0, head = 0, words_left = ...; // size
4   while(words_left > 0) {
5     int tail        = ioreg_read(ram);
6     int free_words  = ram->depth - (head - tail);
7     int bsize_words = MIN(free_words, words_left);
8
9     if(bsize_words != 0) {
10      coram_write(ram, head, src +
11                  src_word * ram->wdsize, bsize_words);
12      ioreg_write(ram, head + bsize_words);
13      src_word   += bsize_words;
14      words_left -= bsize_words;
15      head       += bsize_words;
16    }
17  }
```

Figure 8: Control program for Input Stream FIFO.

simulator, which models a microarchitecture based on Figure 5. For our applications, the compute portions of the designs were placed-and-routed on a Virtex-6 LX760 FPGA to determine the peak fabric processing throughput.

4.1 Black-Scholes

The first FPGA application example, Black-Scholes, is widely used in the field of quantitative finance for option pricing [21]. Black-Scholes employs a rich mixture of arithmetic floating-point operators but exhibits a very simple memory access pattern. The fully pipelined processing element (PE) shown in Figure 6 consumes a sequential input data stream from memory and produces its output stream similarly. The application's performance is highly scalable; one could increase performance by instantiating multiple PEs that consume and produce independent input and output data streams. Performance continues to scale until either the reconfigurable logic capacity is exhausted or the available external memory bandwidth is saturated. The characteristics of our Black-Scholes PE are shown in Figure 6 (bottom).

Supporting Streams with CoRAM. To support the sequential memory access requirements of Black-Scholes, we develop the concept of a stream FIFO "memory personality", which presents a simple FIFO interface to user logic (i.e., *data, ready, pop*). The stream FIFO employs CoRAMs and a control thread to bridge a single Black-Scholes pipeline to the edge memory interface. Figure 7 illustrates the stream FIFO module, which instantiates a single CoRAM to be used as a circular buffer, with nearby head and tail registers instantiated within reconfigurable logic to track the buffer occupancy. Unlike a typical FIFO implemented within reconfigurable logic, the producer of the stream FIFO is not an entity hosted in logic but is managed by an associated control thread.

Figure 7 (right) illustrates a simple control thread used to fulfill the FIFO producer role (the corresponding code is shown in Figure 8). The event highlighted in Step 1 of Figure 7 first initializes a source pointer to the location in memory where the Black-Scholes data resides (L2 in Figure 8). In Step 2, the control thread samples the head and tail pointers to compute how much available space is left within the FIFO (L5-L7 in Figure 8). If sufficient space exists, the event in step 3 performs a multi-word byte transfer from the edge memory interface into the CoRAM using the *coram_write* control action (L10 in Figure 8). The event in

```
1  void mmm(Data* A, Data* B, Data *C, int N, int NB) {
2    int j, i, k;
3    for (j = 0; j < N; j += NB) {
4      for (i = 0; i < N; i += NB) {
5        for (k = 0; k < N; k += NB) {
6          block_mmm_kernel(A + i*N + k,
7                           B + k*N + j,
8                           C + i*N + j, N, NB);
9        }
10     }
11   }
12 }
```
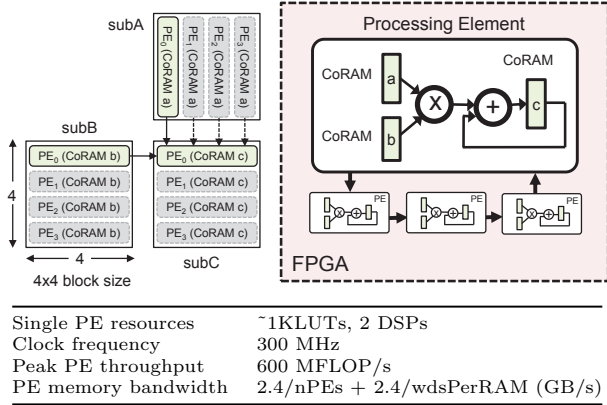
Figure 9: Reference C code for Blocked MMM [4].

Single PE resources	~1KLUTs, 2 DSPs
Clock frequency	300 MHz
Peak PE throughput	600 MFLOP/s
PE memory bandwidth	2.4/nPEs + 2.4/wdsPerRAM (GB/s)

Figure 10: FPGA design for Blocked MMM.

Step 4 completes the FIFO production by having the control thread update the head pointer using the *ioreg_write* control action to inform the reconfigurable logic within the stream FIFO module when new data has arrived (L12 in Figure 8). Finally, L13-L15 show updates to internal state maintained by the control thread. (Note: we do not show the control thread program for the corresponding output stream FIFO.)

Discussion. The use of CoRAM simplified the overall development efforts for Black-Scholes by allowing us to express the application's memory access pattern at a high level of abstraction relative to conventional RTL flows. Our control thread program described the sequential memory transactions required by Black-Scholes as a sequence of untimed steps using a simple, C-based language. This abstraction was simple to work with and allowed us to iterate on changes quickly and conveniently. An important idea that emerged during our development efforts was the memory personality concept. A memory personality "wraps" CoRAMs and control threads within reconfigurable logic to provide an even higher level of abstraction (i.e., interface and memory semantics) best suited to the application at hand. As will be discussed later, many other kinds of memory personalities can also be constructed and further combined to form a re-usable shared library for CoRAM.

4.2 Matrix-Matrix Multiplication

The next example, Matrix Matrix Multiplication (MMM), is a widely-used computation kernel that multiplies two matrices encoded in dense, row-major format [18]. For multiplications where the input and output matrices are too large to fit in on-chip SRAMs, a commonly used blocked algorithm decomposes the large calculation into repeated multiplica-

```
1.  void mmm_control(Data *A, Data *B, ...) {
2.    coh ramsA = ..; // 'a' CoRAMs, word-interleaved
3.    coh ramsB = ..; // 'b' CoRAMs, concatenated
4.    for (j = 0; j < N; j += NB) {
5.      for (i = 0; i < N; i += NB) {
6.        for (k = 0; k < N; k += NB) {
7.          fifo_read(...);
8.          for (m = 0; m < NB; m++) {
9.            collective_write(ramsA, m*NB, A + i*N+k +
                                m*N, NB*dsz);
10.           collective_write(ramsB, m*NB, B + k*N+j +
                                m*N, NB*dsz);
11.         }
12.         fifo_write(...);
13.       }
14.     }
15.   }
```

Figure 11: MMM control thread code example.

tions of sub-matrices sized to fit within the on-chip SRAMs (see reference C code in Figure 9). This strategy improves the arithmetic intensity of MMM by increasing the average number of floating-point operations performed for each external memory byte transferred.

Figure 10 illustrates a parameterized hardware kernel developed for single-precision blocked MMM. The design employs p identical dot-product processing elements (PE) and assumes that the large input matrices A, B, and the output matrix C are stored in external memory. In each iteration: (1) different sub-matrices $subA$, $subB$, and $subC$ are read in from external memory, and (2) $subA$ and $subB$ are multiplied to produce intermediate sums accumulated to sub-matrix $subC$. The sub-matrices are sized to utilize available SRAM storage. (Square sub-matrices of size 4x4 are assumed in this explanation for the sake of simplicity.) The row slices of $subB$ and $subC$ are divided evenly among the p PEs and held in per-PE local CoRAM buffers. The column slices of $subA$ are also divided evenly and stored similarly. A complete iteration repeats the following steps p times: (1) each PE performs dot-products of its local slices of $subA$ and $subB$ to calculate intermediates sum to be accumulated into $subC$, and (2) each PE passes its local column slice of $subA$ to its right neighbor cyclically (note: as an optimization, step 2 can be overlapped with step 1 in the background).

MMM Control thread. To populate the CoRAM buffers of each PE, a single control thread is used to access all of the necessary per-row and per-column slices of $subA$, $subB$, and $subC$. The pseudo-code of the control thread program used to implement the required memory operations is shown in Figure 11 (for brevity, the reading and writing of $subC$ is omitted). In L2, $ramsA$ is a co-handle that represents an aggregation of all the 'a' CoRAMs belonging to the PEs in word-interleaved order ($ramsA$ can be constructed from multiple CoRAMs using *append_handle*). If passed into a *collective_write* control action, matrix data arriving sequentially from edge memory would be striped and written across multiple 'a' CoRAMs in a word-by-word interleaved fashion. This operation is necessary because the 'a' CoRAMs expect the data slices in column-major format whereas all matrix data is encoded in row-major. The co-handle $ramsB$ expects data in a row-major format and is simply a concatenation of all of the 'b' CoRAMs. Within the body of the inner loop, the control thread first waits for a token from the user logic (L7) before executing the *collective_write* con-
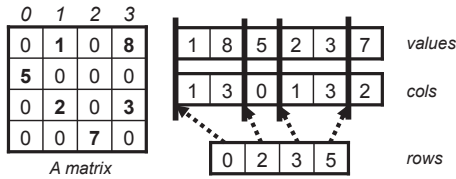
```
1. void spmv_csr (int n_rows, int *cols,
2.                Data *rows, Data *x, Data *y) {
3.   for(int r = 0; r < n_rows; r++) {
4.     int sum = 0;
5.     for(i = rows[r]; i <= rows[r+1]; i++)
6.       sum += vals[i] * x[cols[i]];
7.     y[r] = sum;
8.   }
9. }
```

Figure 12: Sparse Matrix-Vector Multiplication.

trol actions used to populate the CoRAMs (L9-L10). Upon completion, the control thread informs the user logic when the data is ready to be accessed (L12). The control thread terminates after iterating over all the blocks of matrix C.

Discussion. Our overall experience with developing MMM highlights more sophisticated uses of CoRAM. In particular, the collective control actions allowed us to precisely express the data transfers for a large collection of CoRAMs in very few lines of code. Control actions also allowed us to customize the partitioning of data to meet the on-chip memory layout requirements of our MMM design. It is worth noting how the code in Figure 11 appears similar to the C reference code, with the exception that the inner-most loop now consists of memory control actions rather than computation. One insight we developed from this example is that control threads can allow us to easily express the re-assignment of FPGA kernels to different regions of the external memory over the course of a large computation. This feature of the CoRAM architecture could potentially be used to simplify the task of building out-of-core FPGA-based applications that support inputs much larger than the total on-chip memory capacity.

4.3 Sparse Matrix-Vector Multiplication

Our last example, Sparse Matrix-Vector Multiplication (SpMV), is another widely-used scientific kernel that multiplies a sparse matrix A by a dense vector x [18]. Figure 12 (top) gives an example of a sparse matrix A in Compressed Sparse Row (CSR) format. The non-zero values in A are stored in row-order as a linear array of *values* in external memory. The column number of each entry in *values* is stored in a corresponding entry in the column array (*cols*). The i'th entry of another array (*rows*) holds the index to the first entry in *values* (and *cols*) belonging to row i of A. Figure 12 (bottom) gives the reference C code for computing $A \times x$. Of all the kernels we studied, SpMV presented the most difficult design challenge for us due to a large external memory footprint and an irregular memory access pattern.

Figure 13 illustrates an FPGA design for SpMV, where multiple processing elements (PEs) operate concurrently on distinct rows of A. The contents of the *rows* array are streamed in from edge memory to a centralized work scheduler that assigns rows to different PEs. For each assigned row, a PE employs two stream FIFOs to sequentially read in data blocks from the *values* and *cols* arrays, respectively. To
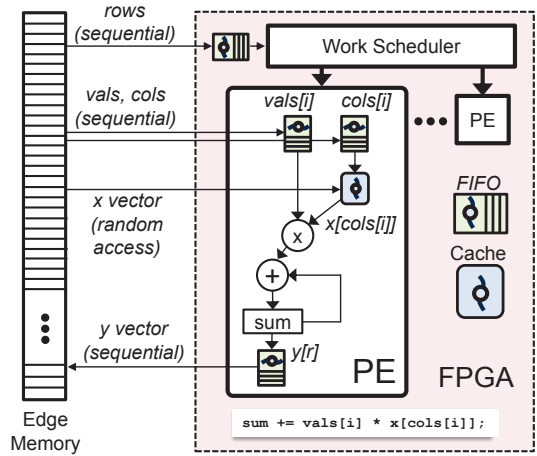
Single PE resources	~2KLUTs
Clock frequency	300 MHz
Peak PE throughput	600 MFLOP/s
PE memory bandwidth	≥3.6GB/s

Figure 13: FPGA design for SpMV.

configure the memory pointers for each of the two stream FIFOs, the PE logic must continuously pass row assignment information (via channels) to the control threads belonging to the stream FIFOs. To calculate each term of a dot-product, a PE must make an indirect reference to vector x (L6). This type of memory access poses a particularly difficult challenge because the memory addresses (i.e., *cols*) are not necessarily sequential and could be accessed randomly. Furthermore, x can be a very large data structure that cannot fit into aggregate on-chip memory. Unlike the MMM example from earlier, the performance of SpMV is highly dependent on efficient bandwidth utilization due to its low arithmetic intensity. An optimization to reduce memory bandwidth is to exploit any reuse of the elements of x across different rows through caching.

To implement caching within each PE, Figure 14 illustrates a read-only cache memory personality built using the CoRAM architecture. Within the cache, CoRAMs are composed to form data and tag arrays while conventional reconfigurable logic implements the bulk of the cache controller logic. A single control thread is used to implement cache fills to the CoRAM data arrays. When a miss is detected, the address is enqueued to the control thread through an asynchronous FIFO (step 1 in Figure 14). Upon a pending request, step 2 of the control thread transfers a cache block's worth of data to the data array using the *coram_write* con-
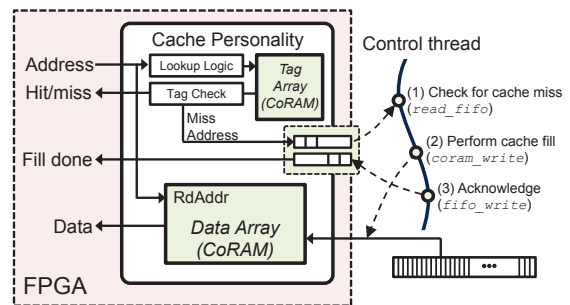

Figure 14: SpMV cache personality.

trol action. In step 3, the control thread acknowledges the cache controller using the *fifo_write* control action.

Discussion. The SpMV example illustrates how different memory personalities built out of CoRAM can be employed in a single design to support multiple memory access patterns. Caches, in particular, were used to support the random access pattern of the x vector, whereas the stream FIFOs from Black-Scholes were re-used for the remaining sequential accesses. In our development efforts of SpMV, the instantiation of CoRAM-based memory personalities along with spatially-distributed PEs allowed us to quickly instantiate "virtual taps" to external memory wherever needed. This level of abstraction was especially convenient as it allowed us to concentrate our efforts on developing only the processing components of SpMV.

4.4 Case Study Remarks

The overall experience in developing applications with CoRAM reveals significant promise in improving the programmability of FPGA-based applications on the whole. While the CoRAM architecture does not eliminate the effort needed to develop optimized stand-alone processing kernels, it does free designers from having to explicitly manage memory and data distribution in a low-level RTL abstraction. Specifically, control threads gave us a general method for dynamically orchestrating the memory management of applications, but did not force us to over-specify the sequencing details at the RTL level. Control threads also did not limit our ability to support fine-grained interactions between processing components and memory. Fundamentally, the high-level abstraction of CoRAM is what would enable portability across multiple hardware implementations.

The memory personalities developed in our examples (stream FIFO, read-only cache) are by no means sufficient for all possible applications and only highlighted a subset of the features possible. For example, for all of the memory personalities described, one could replace the *coram_write* control action with *coram_copy*, which would permit transfers from other CoRAMs within the fabric and not necessarily from the memory edge. Such control actions could, for instance, be used to efficiently compose multi-level cache hierarchies out of CoRAMs (e.g., providing the SpMV PEs with a shared L2 cache) or be used to setup streams between multiple application kernels. It is conceived that in the future, soft libraries consisting of many types of memory personalities would be a valuable addition to the CoRAM architecture. This concept is attractive because hardware vendors supporting the CoRAM abstraction would not have to be aware of all the possible personality interfaces (e.g., caches, FIFOs) but would only be implementing the "low-level" control actions, which would automatically facilitate compatibility with existing, high-level libraries.

5. EVALUATION

This section presents our initial investigations to understand the performance, area, and power implications of introducing the CoRAM architecture to a conventional FPGA fabric. Given the large design space of CoRAM, the goal of this evaluation is to perform an early exploration of design points for future investigations.

Methodology. To model an FPGA with CoRAM support, a cycle-level software simulator was developed in Bluespec

Variables	Selected parameters
FPGA resources	474KLUTs, 720 CoRAMs, 180 threads
Clock rates	User Logic @ 300MHz, CoRAM @ 0.3-1.2GHz
Off-chip DRAM	128GB/sec, 60ns latency
Control Unit	8-thread, 5-stage MIPS core
Network-on-Chip	2D-mesh packet-switched NoC
	3-cycle hop latency, 300MHz-1.2GHz
Topology	{16,32,64} nodes x {8,16} memports
Router datapath	{32,64,128}-bit router datapaths

Table 1: FPGA Model Parameters.

System Verilog [3] to simulate instances of the design illustrated in Figure 5. Table 1 summarizes the key configuration parameters of the simulated fabric design. The simulator assumes a baseline reconfigurable fabric with 474 KLUTs and 720 CoRAMs (4KB each) to reflect the capacity of the Virtex-6 LX760, the largest FPGA from Xilinx today. In the performance simulations, we assume the user logic within the fabric can operate at 300MHz based on placed-and-routed results of the processing kernels from Section 4. For design space exploration, we considered different points based on CoRAMs assumed to operate at 300 MHz to 1.2 GHz to study the relative merits between soft- versus hard-logic implementations.

The simulator models a 2D-mesh packet-switched network-on-chip based on an existing full-system simulator [22]. The NoC simulator models the cycle-by-cycle traffic flow among the CoRAMs and the edge memory traffic resulting from the control actions issued by control threads. The bandwidth available at each network endpoint is shared by a concentrated cluster of locally connected CoRAMs. For design space exploration, we varied the NoC performance along three dimensions: (1) operating frequency between 300MHz to 1.2GHz (in sync with the CoRAM frequency); (2) network link and data width (32, 64, 128 bits per cycle); and (3) number of network end-points (16, 32, 64) and thus the number of CoRAMs sharing an end-point.

To incorporate memory bandwidth constraints, our simulator models an aggressive external memory interface with four GDDR5 controllers, providing an aggregate peak bandwidth of 128GB/sec at 60ns latency. This external memory interface performance is motivated by what can be achieved by GPUs and CPUs today. Even then, the performance simulations of the Sparse Matrix-Vector Multiplication (SpMV) kernels reported in this section are all memory bandwidth bound, never able to completely consume the reconfigurable logic resources available in the fabric.

The Control Unit from Figure 5 is modeled as a multi-threaded microcontroller that executes control thread programs directly. The model assumes that the fabric has 740kB of SRAM (beyond CoRAMs) needed to hold the 180 control thread contexts (code and data). The performance simulator also does not model the multithreaded pipelines explicitly. However, the control threads are compiled and executed concurrently as *Pthreads* within the simulation process and are throttled with synchronization each simulated cycle to mimic execution delay (varied between 300MHz and 1.2GHz for design space exploration).

RTL Synthesis. For each of the design points considered by the performance simulation, the power and area is also estimated by synthesizing RTL-level designs of the CoRAM mechanisms using Synopsys Design Compiler v2009 to target a commercial 65nm standard cell library. The power

	C = CoRAMs, T = control threads, N = nodes = clusters, M=mbanks
	$K = C/N$ = CoRAMs per cluster, $X = T/N$ = control threads/cluster
	P = watts, p = watts/GHz, $freq$ = CoRAM clock frequency

$A_{cluster} = A_{router} + X \times A_{costate} + \frac{X}{8} \times (A_{mips} + A_{other})$

$A_{total} = N \times A_{cluster} + M \times A_{router}$

$P_{cluster} = freq \times (p_{router} + X \times p_{costate} + \frac{X}{8} \times (p_{mips} + p_{other}))$

$P_{total} = N \times P_{cluster} + freq \times M \times p_{router}$

Table 2: Area and Power Overhead Formulas.

Component	mm^2	mW/GHz
MIPS core (8 threads)	.08	59
Per-thread state (SRAM)	.04	8
32-bit router	.07	38
64-bit router	.11	48
128-bit router	.30	64
Other (queues, logic, etc.) (est'd)	.40	-

Table 3: 65nm Synthesis Results.

and area of the NoC are estimated by synthesizing an open-sourced router design[1]. The power and area of the Control Unit is estimated by synthesizing an 8-way multithreaded 5-stage MIPS core automatically generated from the T-piper tool [17]. The power and area of the SRAMs for the control thread contexts are estimated using CACTI 4.0 [20]. Table 3 summarizes the resulting area and power characteristics of various components. The total power and area overhead for a particular design point is calculated using the formulas in the bottom of Table 2 by setting the parameters in the top portion of the table.

It is important to keep in mind that, despite the efforts taken, the reported power and area estimates are only approximate. However, by just comparing relative magnitudes, the estimates give an adequate indication that the total cost of adding even an aggressive, pure hard-logic CoRAM implementation is small relative to the inherent cost of a reconfigurable fabric like the Virtex-6 LX760. Our estimates also have a large conservative margin built-in since they are based on a 65nm standard cell library several years old.

5.1 Design Space Exploration Results

From the design explorations, we report results for the Sparse Matrix-Vector Multiplication kernel, which was our most memory-intensive application (results for Matrix Matrix Multiplication and Black-Scholes are discussed qualitatively at the end). We simulated the execution of the SpMV kernel running on design points generated from an exhaustive sweep of parameters given in Table 1. For each design point, we report the SpMV kernel's execution performance averaged over test input matrices from [9]. The graphs in Figure 15 plot the performance (GFLOP/sec) achieved by each design point (on the x-axis) against its area and power overheads (on the y-axis) from adding the CoRAM architecture support. The data points represented by the same markers correspond to design points with CoRAM mechanisms at the same frequency (300MHz, 600MHz, 900MHz or 1.2GHz). All of the design points incur a fixed $18mm^2$ from SRAM storage and the MIPS cores for the required 180 control thread contexts; this can be a very significant portion of the total area overhead for some design points. Nevertheless, the total area overhead is small in comparison to the hundreds of mm^2 typical of even small FPGAs today [1].
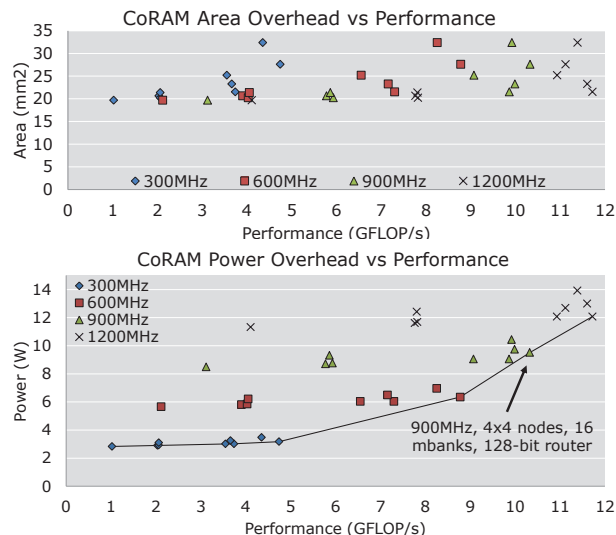
Figure 15: Estimated area and power overhead.

In Figure 15 (top), points in the lower-right corner correspond to higher performance and lower area overhead. For all of the frequencies, a nearly minimal area design point achieves almost the best performance possible at a given frequency. This suggests that the operating frequency of the CoRAM mechanisms has a first-order impact on the overall application performance, beyond the impact of microarchitectural choices. This result suggests that it may be difficult for soft implementations of the CoRAM mechanisms to perform comparably well as hard implementations in the future even when reconfigurable logic resources are made plentiful.

The power-overhead-vs-performance plot in Figure 15 (bottom) exhibits a cleaner Pareto front. In this plot, the higher performance design points tend to require a greater power overhead. It should be pointed out that the microarchitecture design point *4x4nodes-16mbanks-128bit* appears consistently on the Pareto front for all frequencies (also optimal in the area-overhead-vs-performance plot). This suggests that it is possible to select this point to achieve minimum area overhead and apply frequency scaling to span the different positions on the power-performance Pareto optimal front. However, this conclusion is based only on the results of the SpMV application kernel. Further study including a much greater range of application kernels is needed. Although performance results were not presented for MMM or BS, our experiments showed that less aggressive designs (e.g., *600MHz-4x4nodes-16mbanks-128bit*) were sufficient for these compute-bound applications to reach peak performance running on the CoRAM architecture.

6. RELATED WORK

A large body of work has explored specialized VLSI designs for reconfigurable computing. GARP [12] is an example that fabricates a MIPS core and cache hierarchy along with a collection of reconfigurable processing elements (PE). The PEs share access to the processor cache but only through a centralized access queue at the boundary of the reconfigurable logic. Tiled architectures (e.g., Tilera [23]) consist of a large array of simple von Neumann processors instead of fine-grained lookup tables. The memory accesses by the cores are supported through per-core private caches

interconnected by an on-chip network. Smart Memories [14] on the other hand employs reconfigurable memory tiles that selectively act as caches, scratchpad memory, or FIFOs.

The idea of decoupling memory management from computation in CoRAM has been explored previously for general-purpose processors [19, 7]. Existing work has also examined soft memory hierarchies for FPGAs (e.g., [25, 10, 13, 15]). The most closely related work to CoRAM is LEAP [2], which shares the objective of providing a standard abstraction. LEAP abstracts away the details of memory management by exporting a set of timing-insensitive, request-response interfaces to local client address spaces. Underlying details such as multi-level caching and data movement are hidden from the user. The CoRAM architecture differs from LEAP by allowing explicit user control over the lower-level details of data movement between global memory interfaces and the on-die embedded SRAMs; the CoRAM architecture could itself be used to support the data movement operations required in a LEAP abstraction.

7. CONCLUSIONS

Processing and memory are inseparable aspects of any real-world computing problems. A proper memory architecture is a critical requirement for FPGAs to succeed as a computing technology. In this paper, we investigated a new, portable memory architecture called CoRAM to provide deliberate support for memory accesses from within the fabric of a future FPGA engineered to be a computing device. CoRAM is designed to match the requirements of highly concurrent, spatially distributed processing kernels that consume and produce memory data from within the fabric. The paper demonstrated the ease-of-use in managing the memory access requirements of three non-trivial applications, while allowing the designers to focus exclusively on application development without sacrificing portability or performance. This paper also suggested a possible microarchitecture space for supporting the CoRAM architecture in future reconfigurable fabrics. An investigation of the trade-offs between performance, power, and area suggests that adding support for the CoRAM architecture in future devices only requires a modest overhead in power and area relative to the reconfigurable fabric.

8. ACKNOWLEDGEMENTS

9. REFERENCES

[1] Under the Hood: Intel has company at 65 nm. http://maltiel-consulting.com/Intel_leads_65-nanometer_technology_race_other-Texas_Instruments_Xilinx_AMD_catching_up.htm.

[2] M. Adler, K. E. Fleming, A. Parashar, M. Pellauer, and J. Emer. LEAP Scratchpads: Automatic Memory and Cache Management for Reconfigurable Logic. In *FPGA '11: Proceedings of the 2011 ACM/SIGDA 19th International Symposium on Field Programmable Gate Arrays*, 2011.

[3] Bluespec, Inc. http://www.bluespec.com/products/bsc.htm.

[4] F. Brewer and J. C. Hoe. MEMOCODE 2007 Co-Design Contest. In *Fifth ACM-IEEE International Conference on Formal Methods and Models for Codesign*, 2007.

[5] S. Che, J. Li, J. W. Sheaffer, K. Skadron, and J. Lach. Accelerating Compute-Intensive Applications with GPUs and FPGAs. In *SASP '08: Proceedings of the 2008 Symposium on Application Specific Processors*, pages 101–107, Washington, DC, USA, 2008. IEEE Computer Society.

[6] E. S. Chung, P. A. Milder, J. C. Hoe, and K. Mai. Single-Chip Heterogeneous Computing: Does the Future Include Custom Logic, FPGAs, and GPGPUs? In *MICRO-43: Proceedings of the 43th Annual IEEE/ACM International Symposium on Microarchitecture*, 2010.

[7] E. Cohler and J. Storer. Functionally parallel architecture for array processors. *Computer*, 14:28–36, 1981.

[8] Convey, Inc. http://www.convey.com.

[9] T. A. Davis. University of Florida Sparse Matrix Collection. *NA Digest*, 92, 1994.

[10] H. Devos, J. V. Campenhout, and D. Stroobandt. Building an Application-specific Memory Hierarchy on FPGA. *2nd HiPEAC Workshop on Reconfigurable Computing*, 2008.

[11] T. El-Ghazawi, E. El-Araby, M. Huang, K. Gaj, V. Kindratenko, and D. Buell. The Promise of High-Performance Reconfigurable Computing. *Computer*, 41(2):69 –76, feb. 2008.

[12] J. R. Hauser and J. Wawrzynek. Garp: a MIPS processor with a reconfigurable coprocessor. In *FCCM'97: Proceedings of the 5th IEEE Symposium on FPGA-Based Custom Computing Machines*, page 12, Washington, DC, USA, 1997. IEEE Computer Society.

[13] G. Kalokerinos, V. Papaefstathiou, G. Nikiforos, S. Kavadias, M. Katevenis, D. Pnevmatikatos, and X. Yang. FPGA implementation of a configurable cache/scratchpad memory with virtualized user-level RDMA capability. In *Proceedings of the 9th international conference on Systems, architectures, modeling and simulation*, SAMOS'09, pages 149–156, Piscataway, NJ, USA, 2009. IEEE Press.

[14] K. Mai, T. Paaske, N. Jayasena, R. Ho, W. J. Dally, and M. Horowitz. Smart Memories: A Modular Reconfigurable Architecture. In *ISCA'00: Proceedings of the 27th Annual International Symposium on Computer Architecture*, pages 161–171, New York, NY, USA, 2000. ACM.

[15] P. Nalabalapu and R. Sass. Bandwidth Management with a Reconfigurable Data Cache. In *Proceedings of the 19th IEEE International Parallel and Distributed Processing Symposium (IPDPS'05) - Workshop 3 - Volume 04*, IPDPS '05, pages 159.1–, Washington, DC, USA, 2005. IEEE Computer Society.

[16] T. Ngai, J. Rose, and S. Wilton. An SRAM-programmable field-configurable memory. In *Custom Integrated Circuits Conference, 1995., Proceedings of the IEEE 1995*, May 1995.

[17] E. Nurvitadhi, J. C. Hoe, T. Kam, and S.-L. Lu. Automatic Pipelining from Transactional Datapath Specifications. In *Design, Automation, and Test in Europe (DATE)*, 2010.

[18] W. H. Press, B. P. Flannery, S. A. Teukolsky, and W. T. Vetterling. *Numerical Recipes in C: the Art of Scientific Computing*. Cambridge University Press, 1988.

[19] J. E. Smith. Decoupled access/execute computer architectures. *SIGARCH Comput. Archit. News*, 10:112–119, April 1982.

[20] D. T. S. Thoziyoor, D. Tarjan, and S. Thoziyoor. Cacti 4.0. Technical Report HPL-2006-86, HP Labs, 2006.

[21] Victor Podlozhnyuk. Black-Scholes Option Pricing, 2007.

[22] T. F. Wenisch, R. E. Wunderlich, M. Ferdman, A. Ailamaki, B. Falsafi, and J. C. Hoe. SimFlex: Statistical Sampling of Computer System Simulation. *IEEE Micro*, July 2006.

[23] D. Wentzlaff, P. Griffin, H. Hoffmann, L. Bao, B. Edwards, C. Ramey, M. Mattina, C.-C. Miao, J. F. Brown III, and A. Agarwal. On-Chip Interconnection Architecture of the Tile Processor. *IEEE Micro*, 27(5):15–31, 2007.

[24] Xilinx, Inc. Virtex-7 Series Overview, 2010.

[25] P. Yiannacouras and J. Rose. A parameterized automatic cache generator for FPGAs. In *Proc. Field-Programmable Technology (FPT*, pages 324–327, 2003.

[26] S. P. Young. FPGA architecture having RAM blocks with programmable word length and width and dedicated address and data lines, United States Patent No. 5,933,023. 1996.