Data Reorganization Engines for the Next Generation of System-On-a-Chip FPGAs*

Pedro C. Diniz and Joonseok Park
University of Southern California / Information Sciences Institute
4676 Admiralty Way, Suite 1001
Marina del Rey, California 90292
{pedro, joonseok}@isi.edu

ABSTRACT
Field-Programmable-Core-Arrays (FPCA) will include various computing cores for a wide variety of applications ranging from DSP to general purpose computing. With the increasing gap between core computing speeds and memory access latency, managing and orchestrating the movement of data across multiple cores will become increasingly important. In this paper we propose data reorganization engines that allow a wide variety of data reorganizations intra- as well as inter-memory modules for future FPCAs. We have experimented with a suite of data reorganizations pervasive in DSP applications. Our limited set of experiments reveals that the proposed designs for these engines are flexile and use little design area in current FPGA fabrics, making them amenable to be easily integrated in future FPCAs as either soft- or hard- macros.

Categories and Subject Descriptors
B.5.1 [**Register-Transfer-Level Implementation**]: Design Aids – *automatic synthesis*; B.7.1 [**Integrated Circuits**]: Types and Design Styles – *algorithm implementation in hardware*; B.6.2 [**Logic Design**]: Design Styles – *memory control and access*; D.3.4 [**Programming Languages**]: Processors – *compilers*.

Keywords
Data Reorganization; High-level Synthesis; Field-Programmable-Gate-Arrays (FPGAs).

1. INTRODUCTION
With the increasing number of available transistors on a die it will soon be possible to embed various computing cores and memory modules on a single chip in what we call a Field-Programmable-Core-Array (FPCA). These cores will possibly include DSPs, microprocessor and other IP cores to perform very specialized functions such as FFTs or digital image encoding/decoding.

While the logic functions of each of the cores should be well defined and understood by designers, we foresee the need to define several memory interfacing abstractions to adequately channel data between functional blocks and the memory modules in an FPCA. These memory interfaces should provide access to internal buses as well as customized support for specific memory access patterns and formats. For example a particular FFT IP core might want to access data in row-wise and column-wise format performing bit-reversal addressing as part of its data access pattern. On the other hand a DSP IP core might want to pack and unpack 13-bit fix-point formatted data into a 32-bit word to increase its memory bandwidth.

Given the increasing processing speeds of the various cores we believe the orchestration of data across multiple memory modules will become an increasingly important issue. As with conventional processing, data locality will be a standing issue and reorganizing data for locality is and will continue to be an important technique for increasing data locality across computations.

The presence of a reconfigurable logic FPGA-like fabric in future generations of FPCAs enables the existence of customizable data engines that can autonomously perform the copying and reorganization of data between the memories associated with multiple cores or even within a single memory. This opportunity relieves each of the computing cores of the burden of performing such data management functions, thereby increasing not only the performance of the cores, but also the overall system performance as it exposes the ability of mapping tools to exploit more aggressively pipelining execution techniques across cores.

In this paper we focus on the design and evaluation of data engines that can be programmable to perform a wide variety of data reorganizations both in-memory as well as across memories. This work focuses on the abstractions for memory interfaces and generalized controllers for various data access patterns and data reorganization operations. This paper does not focus on the issues that arise in the mapping of applications to these FPCA devices although we briefly discuss them in section 3.

This paper makes the following contributions:

- It presents the internal architecture for a family of data reorganization engines for modifying the layout of array variables either on a single memory or across multiple memories.

* Funded by the Defense Advanced Research Project Agency under contract number F30603-98-2-0113

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
FPGA'02, February 24-26, 2002, Monterey, California, USA.
Copyright 2002 ACM 1-58113-542-5/02/0002…$5.00.

- It evaluates these data reorganization engines for a variety of design parameters and compares the performance attained for a small set of data reorganization operations.
- It presents a compiler analysis algorithm for automatically determining opportunities for data reorganizations using the transpose operation in applications written in C.

The preliminary experimental results reveal that the data reorganization engines proposed in this paper are small using only 5% of the area of current FPGA devices (Virtex® V1000 BG560) and deliver up to 80Mbytes per second of sustained data rate conversions between memory modules. This experience suggests that in future FPCAs, with larger transistor counts, it will be feasible to dynamically configure several of these data engines for data copying and reorganization across multiple memory modules.

This paper is organized as follows. In the next section we outline our vision for the Field-Programmable-Core-Array (FPCA) and what we believe will be the biggest application mapping challenges. We highlight the growing importance of the data management and data reorganizations. In section 3 we illustrate the utilization of a transpose data reorganization for increasing data locality across multiple loops in an application and where the loops are executed by distinct IP cores. Section 4 describes the internal architecture of the proposed data reorganization engines. In section 5 we present a compiler algorithm that automatically extracts opportunities for the transpose data reorganization operation across loops. In section 6 we present experimental results for the implementation of the proposed data engines using today's FPGA mapping technology and devices. We survey related work in section 7 and conclude in section 8.

2. THE FPCA ARCHITECTURE

With the growing number of available transistors on a die it is very likely that we will see in the future an FPGA-like device that incorporates hard- or soft- macro cores to implement specialize functionalities[1]. These cores can be either existing IP cores such as microprocessors, memories and other specialized units (*e.g.*, FFT or compression core). These SoC FPGAs, also named here Field-Programmable-Core-Arrays (FPCA) could have as elementary section as depicted in Figure 1.

Under this possibly heterogeneous target architecture programmers and/or tools will have to perform a wide variety of tasks, namely:

- **Computation Partitioning**. This task is responsible for partitioning the computation and the corresponding data between the various computing units in the FPCA taking into account the various features of each of the IP cores.
- **Data Mapping**. This task is responsible to partitioning the computation data among the various storage resources (*e.g.*, RAM modules, tapped-delay lines or direct input sensor-like devices). While it is possible to define a static notion of binding of data to storage, FPCAs will allow a more loose association between data and storage over time.

[1] We define a soft macro as one that uses the underlying FPGA fabric to implement a particular functionality whereas a hard macro is defined as implemented directly as transistors and therefore imprinted on the FPGA die.

- **Communication Scheduling**. This task is responsible for mapping communication channels to routing resources. While the current FPGA devices allow for a static interconnection network, we believe in the future we will see organizations closer to internal packeted network with the possibilities for programmable routing mechanisms as a way to defeat congestion and deadlock issues.

While most of these efforts have been developed in various contexts, most of them have been confined to a particular system or architecture and have, therefore, not revealed the level of integration that heterogeneous target architectures require.

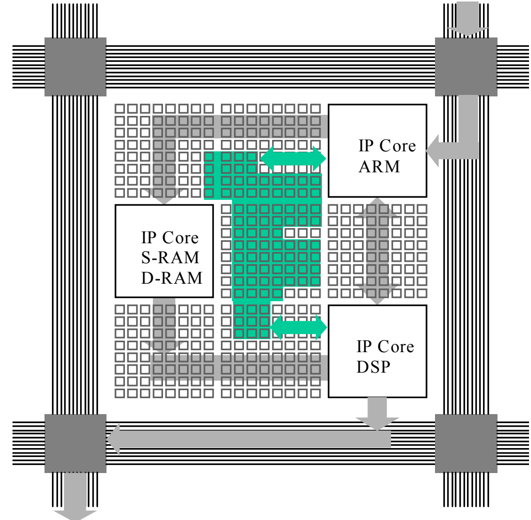

Figure 1. Section of possible System-On-a-Chip FPGA.

In this work we have drawn on our previous experience with the development of memory controllers for FPGAs [10] to build parameterizable data engines capable of autonomously performing data transfers between memory blocks. FPCAs presents a natural target for the proposed data engines as these engines can be embedded in the FPGA fabric and dynamically reconfigured to deliver a customizable interconnect between the IP blocks.

3. EXAMPLE

We now illustrate the application of the data copying and reorganization in the context of mapping an application to an FPCA. The example code, depicted in Figure 2, consists of two loop nests. The first loop nest scales the two two-dimensional array variables *a* and *b* by two scalar factors, *factor1* and *factor2*, respectively. The second loop nest performs a matrix multiplication depositing the result in the *out* array variable.

Because the second loop uses floating point intensively, a compiler might map that loop nest to a DSP code whereas the remaining of the computation executes on a RISC core. As a result the data in the array *a* and array *b* may need to be copied from the memory associated with the RISC core to the memory associated with the DSP core. Under this scenario a data engine can also transpose the data in array *b*, so that the dot product computation of the second loop nest can exhibit a smaller stride and hence enhanced cache locality. Furthermore, and because the data has been truncated in loop 1 to be 8 bits wide, this reorganization also has the added advantage of increasing the

number of elements which the DSP can fetch per memory access operation, hence increasing its effective bandwidth.

```
/* loop 1 – a floating point computation with truncation */
for(i = 0; i < N; i++){
  for(j = 0; j < M; j++){
    a[i][j] = (short)(a[i][j] / factor1);
    b[i][j] = (short)(b[i][j] / factor2);
  }
}
/* loop 2 – A simple matrix multiply */
for(i = 0; i < N; i++){
  for(j = 0; j < N; j++){
    sum = 0;
    for(k = 0; k < N; k++){
       sum += a[i][k] * b[k][j];
    }
    out[i][j] = sum;
  }
}
```

Figure 2. Example Candidate Code for Data Reorganization.

The transformed code illustrating the application of this data copying and data reorganization is depicted in Figure 3 below. Here we have represented the data reorganization using an additional loop nest. The computation represented by this loop is, however, executed by a data engine. Obviously any future references *b[i][j]* in the original program after loop 2 must now be changed to *b[j][i]* or a new reorganization step must be insert to enforce correctness of the computation.

This example illustrates the opportunities for data reorganization the data engines proposed for future FPCAs aim to explore. First, there is a need to map different data arrays to possibly distinct memory modules. Second, there is a clear advantage of reorganizing the data either for simple access modes or by exploiting packing/padding opportunities in memory access word operations.

```
/* loop 1 – a floating point computation with truncation */
for(i = 0; i < N; i++){
  /* loop unchanged */
}

/* reorganization executed in data engine */
for(i = 0; i < N; i++){
 for(j = 0; j < N; j++){
  copy(a[i]][j],a[i][j]); /* copy to new memory */
  copy(b[i]][j],b[j][i]); /* copy to new memory */
  /* with transpose */
  }
}

/* loop 2 – a simple matrix multiply */
for(i = 0; i < N; i++){
  for(j = 0; j < N; j++){
    sum = 0;
    for(k = 0; k < N; k++){
       sum += a[i][k] * b[j][k]; /* reference modified */
    }
    out[i][j] = sum;
  }
}
```

Figure 3. Example Code with Reorganization.

4. DATA ENGINE ARCHITECTURE AND DATA REORGANIZATIONS

4.1 Internal Architecture

Figure 4 below depicts the internal architecture of the data reorganization engines proposed in this paper. It consists of a set of memory controllers (to be described in the next subsection), a simple control interface with internal registers to control its internal operations and a programmable switching network

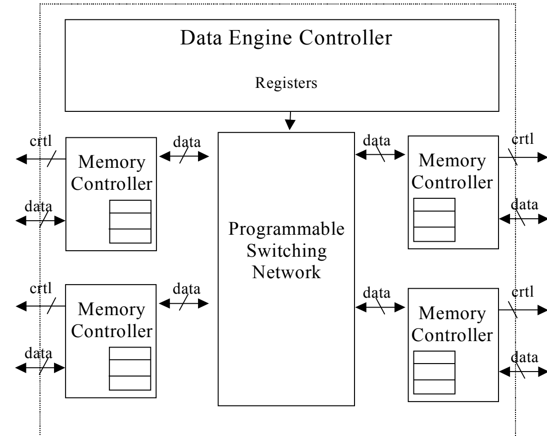

Figure 4. Data Engine Internal Architecture.

The controller interface has a set of externally visible registers that allow for other units to program the individual memory controllers as well as the overall status of the data engine.

4.2 Programmable Switching Network

The switching network is customized for the needs of a set of computations at hand. Rather than having a general-purpose switching network with its potentially very large area cost, we have opted for a design strategy, where multiple switching functions configurations can be merged into a single configuration for the switching network and selected by internal registers. This is clearly a trade-off in the design space for the switching network and we will explore in the future the validity of this approach in the light of whole application mapping where a wide range of switching patterns may exist.

4.3 Memory Controllers

We now describe the basic architectural features of the family of memory controllers we have designed and evaluated for FPGA-based machines. The basic structure is depicted in Figure 5 where we have illustrated the application of the memory controller directly feeding a datapath core.

Associated with each data port there are FIFO queues. FIFO queues are an integral part of the design to help tolerate the latency of memory operations as well as isolate the issues of scheduling in the datapath from the implementation of the memory operations, in particular when dealing with strict timing constraints.

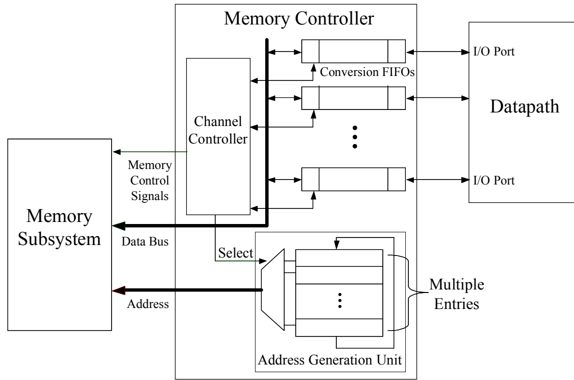

Figure 5. Basic Memory Controller Architecture.

While this is not the only design possibility for these data engines, it is a very modular approach that has been tested successfully in the context of a compiler that maps applications directly to FPGAs [10].

4.4 Data Reorganizations

We foresee the need of hardware support for data management and data reorganizations in the orchestration of data across the various memories in the next generation of FPCA chips. The basic operations include:

- **Splitting and Striping** – In this operation the array is splited between two or more memories in a regular fashion, for example by columns or by rows.
- **Data Padding** – In this operation each of the elements of an array is expanded into a set of K words in memory. This operation is typically used for alignment purposes.
- **Data Packing** – In this operation several elements of an array are packed into multi-element words to match the word width of the underlying architecture. Typically, K elements are packed to make a 32-bit word.
- **Merging** – This operation is essentially the dual of the splitting and striping and it reconstructs a full array out of disperse sections of the array.
- **Transpose** – In this operation a 2 dimensional array section is transposed to increase the locality of future data access.
- **Multicast** – In this operation an array is broadcast to a small set of memories. This replication might be the result of compiler analysis aimed at reducing memory contention.
- **Gather/Scatter** of array blocks using an index array. Typically this operation is used in sparse-matrix computations and requires some form of support for index array accesses.

While some of these data reorganizations can, in general, be applied to any array data structure, some require advanced indexed addressing support as is the case with the gather/scatter operation. At present we have not developed data engines with this indexing capability and therefore have focused on data array reorganizations that can be implemented using affine data access patterns only.

4.5 Finite Affine Data Streams

This work focuses on finite data streams (hereafter simply referred as streams) that are generated through *affine* data access patterns. These finite streams represent a natural match for a wide variety of description of sections of arrays commonly found in practical applications. For finite *affine* data streams the address generator sequence is defined by the tuple <base,elem_size>. The physical address generated by this tuple is defined by address = base + (*count* * elem_size) where *count* is initialized to zero and automatically incremented at each memory access. The basic memory operations simply define which tuple to use to generate the address. The memory operations are defined as *read*(*n*) and *write*(*n*) where *n* defines which tuple to use to generate the corresponding physical address.

5. COMPILER ANALYSIS

We now present a compiler algorithm that finds the opportunities for applying the operation of data transpose to multidimensional array variables.

5.1 Basic Definitions

The compiler analysis focuses on the set of loop nests *LN* in the program. Each loop nest consists of a number of loops *L* that may not be perfectly nested. For each loop *L* in a nest, we defined the nesting depth *nd(L)* as 1 for the outermost loop in the nest and *n* for the innermost nesting depth. We also define *index(L)* as the index variable for the loop *L*. For each loop *L* we define the set of array references in *L* as *AR(L)*. For each array variable *A*, we define the number of dimensions as *ndims(A)*. The algorithm focuses on array references in the affine domain.[2] The algorithm represents the affine access functions as an (*ndims(A)* x (*ndims(A)* + 1) matrix with the coefficients being the coefficients of the loop index variables (see for example [15]). For example, the array reference *R* = *a[-i+1][2*j]*, where *i* and *j* are loop index variables, would have as the row vectors of its access matrix vector, the vector v_1 = (0 2 0) for the first array dimension (the rightmost syntactically) and v_2 = (-1 0 1) for the second dimension of the array. For each array variable *A* we also defined the function *loc(A)* as being the memory to which the compiler has mapped the array variable in its entirety.[3] Given an array reference *A*, we also define the function *proj(R,i)* that yields a vector with 1 for every array dimension where the loop index *i* coefficient is non-zero. In the example above *proj(R,i) = (1 0)* whereas *proj(R,j) = (0 1)*. This projection function will be useful in understanding which dimensions of a given array carry a possible temporal reuse in the innermost loop of a nest.

---

[2] For the transpose algorithm, however, the precision of the affine restriction in unnecessary and the algorithm described in the next section can effectively deal with non-affine references. For the remaining transformations an array with at least one non-affine access function implies that the corresponding dimension cannot be subject to any affine data reorganization operation.

[3] The algorithm operates under the assumption that another compiler algorithm has performed the binding of arrays to storage and possibly applied data layout transformations such as striping with the corresponding array renaming. The compiler analysis algorithm presented here is orthogonal to those transformations.

5.2 Transpose Analysis Algorithm

The transpose algorithm starts by building a *CFG* where the nodes of the graph represent loop nests that have been entirely mapped to the same processing element. Next the algorithm examines every edge of this *CFG* to understand which array variables, and using which references, are accessed by the two loop nests, the loop nest associated with the source of the *CFG* edge L_{src} and the loop nest associated with the destination of the *CFG* edge, L_{dst}. For each of these loop nests the algorithm next extracts the array references for a given array for which the mapping to memory modules for the source loop nest and destination loop nest differ. These arrays are the potential candidates for being subject to a transpose operation during copying. The algorithm, however, can be trivially modified to include data arrays that can be transposed in-place in the same memory. Finally the algorithm determines for each of the array variables whether or not they exhibit similar temporal reuse patterns in the two loop nests associated with an edge of the *CFG*.

The algorithm computes a **reuse vector** that aims at capturing numeric relative temporal reuses between the array dimensions. Because there can be multiple array references inside a loop nest or even in multiple statements of an imperfectly nested loop, the algorithm uses a simple weighted metric to decide which of the dimensions of the array are the most likely to exhibit temporal reuses. This weighting metrics uses the projected dimension vector *proj(R,i)* for each index array reference. The algorithm adds up the product of the projection vectors by a numerical value that is the number of references in the loop nest plus one. This approach ensures that regardless of the number of references of the array variable the algorithm will give more weight to references located in inner loops versus references in outer loops. The algorithm uses the common approach to control flow by considering both branches of conditional statements as always taken and are therefore given the same weight.

Once the algorithm computes the reuse vector for each array variables and for each of the loops, it determines which dimensions an array variables should be transposed by sorting the indices of the reuse vectors. A transpose operation that reorders the array dimensions such that the corresponding reordered metric vectors exhibit values with increasing values aims at maximizing the temporal reuse in the reordered array. Figure 7 illustrates the application of the algorithm in Figure 6 to the code example in section 3. Notice that although the numeric value for each array reference can have varying values, the important aspect is the relative value of the reuse vector elements. As expected the algorithm determines that only array *b* should be transposed during copying. The algorithm derives this information from the two sorted index vectors **(0,1)** and **(1,0)** corresponding to the two loop nests as the maximum of the reuse vector in the two loop nests occurs in distinct dimensions.

5.3 Remarks

Although the current algorithm implementation focuses on array variables that are mapped to distinct memories, the algorithm can be trivially adapted to determine opportunities for data reorganization inside the same loop nest. Of particular interest is the scenario where a given loop (typically an outermost loop) has a set of sequential loop nests that could have resulted from the application of high-level loop transformations such as unrolling.

```
int maxIndex(vector vec){
  returns the index with highest numerical value;
}

vector sortingIndex(vector vec, int dir){
  returns the sorting indexing of the elements of vec;
}

Refset  sameArrayReferences(A,LN1, LN2){
  for all Ref in AR(LN1) and AR(, LN2) do
    if(A(Ref) == A)
      add Ref to return set;
  end for
}

vector computeProjWeight(Ref,LN){
  int M = numberRefs(A(Ref),LN);
  vector res;
  for all loops L in LN do
    i = nd(L); // nesting depth of L
    res += (M+1)^i * proj(Ref,index(L));
  end for
  return res;
}

/* Transpose Analysis Algorithm */
for all edges (Lsrc,Ldst) in the Loop Nest CFG do
  for all Arrays A in both Lsrc + Ldst do
   v1 = v2 = 0;
    for all Ref in sameArrayReferences(A,Lsrc, Ldst) do
      v1 += computeProjWeight(Ref,Lsrc);
      v2 += computeProjWeight(Ref,Ldst);
    end for
    if(ndims(Array(Ref)) == 2) then /* the simple case */
      if((m1 = maxIndex(v1)) != (m2 = maxIndex(v2)))
        if(maxIndex(v2) == 0)
          transpose between m1 and m2 for A;
    else /* the general case */
    transpose(sortingIndex(v1,up),sortingIndex(v2,up));
    end
  end for
end for
```

Figure 6. Transpose Analysis Algorithm.

Loop Nest 1

```
  Ref  a[i][j]   : reuse vec = (3,9)  sortingIndex = (0,1)
  Ref  b[i][j]   : reuse vec = (3,9)  sortingIndex = (0,1)
```

Loop Nest 2

```
  Ref  a[i][k]  : reuse vec = (2,8)  sortingIndex = (0,1)
  Ref  b[k][j]  : reuse vec = (8,4)  sortingIndex = (1,0)
  Ref  out[i][j]: reuse vec = (2,4)  sortingIndex = (0,1)
```

Figure 7. Transpose Analysis Algorithm for the Code Example in Section 3.

The algorithm described here is capable of handling non-perfectly nested loops as well as non-affine access functions by virtue of focusing only on which dimensions of each array are accessed by each array reference. However, and due to its simplicity the current algorithm implementation cannot capture notions such as sections of arrays (*e.g.*, a tile or a row) or the direction in which the data is accessed.

Finally, the algorithm handles correctly the cases where the programmer has explicitly performed the transpose operation. In this case the algorithm would find a reuse vector with equal values in all its elements, therefore not prompting any further transpose operation.

5.4 Implementation Status and Future Work

The compiler analysis algorithm described in this section has been implemented in approximately 500 lines of SUIF code and is being integrated in the DEFACTO compilation system [4]. As part of future work we intend complement the algorithm described in this paper with a more sophisticated algorithm capable of detecting opportunities for data reorganization when data layout transformations such as striping and blocking are applied. In this context an integrated compiler algorithm can decrease the data access latency or reduce the storage requirements for tasks in a producer-consumer relationship when combining data reorganization with tiling or pipelining execution techniques.

6. EXPERIMENTAL RESULTS

We have implemented code generation functions in C that emit templates in structural and behavioral VHDL for the data reorganization engines described in section 3. Each data engine is parameterizable in terms of the number of the memory modules to which it interfaces. Each memory controller is parameterizable in terms of the number of entries and sizes of the base address and offsets. In addition we have generated selected data patterns in the switching network module and integrated it with the complete data engine design.

6.1 Design Results

We begin our discussion with the experimental evaluation of the resources required for different units with distinct data access patterns. For each of the comparison we use figure of merit as being the maximum achievable clock rate, number of CLBs (Configurable Logic Blocks) the unit uses in an FPGA implementation and the maximum achievable bandwidth between the input and output ports.

6.2 Application Experience

In this application experience we have focused on a small set of kernels data patterns, namely, transposing, row-wise and column-wise accesses and data packing and unpacking. For the purpose of the experiments we have focused on particular implementations for these operations in terms, with different parameters, of the number of channels and bit widths used, namely:

- Merging (MG-4/8) – In this kernel we merge 4 8-bit input streams into a single 32-bit output stream by interleaving each of the 8-bit data elements in a single word.
- Transpose (TP) – In this kernel the output stream is a transpose of the input data stream for a known fixed stride for both the input and the output. Both streams have a 32-bit width format.
- Stripping (ST-4/8) – In this kernel a single 32-bit word input stream is striped across 4 8-bit outputs.
- Replication (RP-8/32) – In this kernel a single 32-bit data stream is replicated across 8 outputs.

6.3 Results

We have encoded these data access patterns and generated the VHDL descriptions for the data engines that implement them. We then used the Xilinx Foundations tool set [16] and commercially available synthesis tools such as Synplicty [13] to generate a real implementation and observed its metrics as presented in Table 1 and Table 2 below. We then tested the implementation of these kernel data reorganizations on an existing real FPGA-based board by applying the data reorganization to copy and reorganize data between two external memories of the same FPGA device on the WildStar™ board [14]. This board is fitted with Xilinx Virtex® 1000 BG560 FPGA devices each with 12,288 slices of CLBs.

Table 1 presents the size in terms of number of slices used in the FPGA for the memory controllers and network required for each of the data reorganization engines as well as the total for the data engine (along with the corresponding percentage occupancy of the FPGA). For each of the data reorganizations (except for the transpose) we have included results from different parameters. For example, we include results for the merging operation for 4 streams of 8 bits wide each (MG-4/8) and results for merging with 2 streams of 16-bit wide (MG-2/16).

Table 1. Size Breakdown for the Various Kernel Data Reorganization Engines.

Kernel	**Memory Controllers**	**Network**	**Total** (%)
MG-4/8	487	38	525 (4.2)
MG-2/16	325	36	361 (2.9)
TP-32	195	37	232 (1.9)
ST-4/8	477	39	516 (4.2)
ST-2/16	252	39	291 (2.4)
RP-8/32	475	103	578 (4.7)
RP-2/32	323	59	382 (3.1)

Table 2 addresses the performance of each of the variants of data engines by presenting the maximum clock rate and the sustained rate transfer in Mega Bytes per second (MB/sec).

Table2. Implementation Metrics for Selected Set of Data Reorganization Patterns.

Kernel	**Rate** MHz	**Bandwidth** MB/sec
MG-4/8	40.3	80
MG-2/16	40.1	80
TP-32	40.2	80
ST-4/8	40.4	80
ST-2/16	43.4	86
RP-4/32	40.3	20 to 80
RP-2/32	40.8	20 to 80

For the replication operation the sustained rate varies between 20 to 80 MB/sec depending on the number of distinct target memory modules. A 20MB/sec rate is imposed on single memory reorganization operations due to contention on the particular FPGA memory bus interface used in these experiments [14].

6.4 Discussion

As the results in Table 1 and Table 2 reveal, for all of the implementations (all excluding the external vendor provided memory interface) the area consumed is fairly small – less than 5% of the whole FPGA target capacity (Virtex® V1000 BG560). This percentage follows a simple trend in terms of the resources used. For the versions where only 2 streams are used (hence smaller memory controllers) the percentage space used is about 3%. This metric increases to about 5% when the number of streams doubles to 4.

The attained clock rates are fairly high for designs that do not exploit any FPGA-specific characteristics and had no hand tuning for either placement or routing of the data engine components. As a result of these clock rates the attained bandwidth for copying and reorganization of data is respectable. We believe these results are very encouraging given that all of the designs are generated automatically using compiler-extracted parameters.

7. RELATED WORK

7.1 Data Layout and Mapping for Custom Computing and Embedded Machines

Other researchers have addressed the issues of mapping variables to memories and improving the performance of the memory subsystem for custom computing machines. Gokhale and Stone [6] proposed an automatic array allocation compile-time algorithm for multi-level memory subsystem. They attempt to allocate array variables to memories based on the memory latency, data access frequency and execution schedule. In the area of embedded systems researchers have also developed methodologies to automate the mapping and subsequent management of data across multiple memory modules [8,9]. Catthoor, Balasa *et. al*. developed and evaluated memory optimizations for embedded systems for a particular application set [1,3,7] but focused on optimizations to minimize memory area and power consumption.

7.2 Custom Address Generators

Data access modules and module generators have been used extensively in the area of adaptive computing, most notably for FPGA-based computing engines, as a way to manage the complexity of mapping computations written in high-level languages to FPGAs [5]. Miranda *et.al* [7] proposed a method to optimize address stream generations for a single computing task.

7.3 Data Reorganization

Other researchers have recognized the value of data reorganization for general purpose computing. In the context of multiprocessing Rivera and Tseng [11] have implemented a data reorganization compiler analysis and code generation for SUIF [12] to improve cache locality and memory bandwidth utilization through a combination of packing and padding for multidimensional arrays. In the embedded high-performance computing community there has been a substantial effort in providing a standard data reorganization APIs [2] with emphasis on programmer and application portability. The compiler analysis algorithm described here can easily be targeted to generate specific data reorganization functions in this API.

7.4 Discussion

The work described in this paper differs from these efforts in many aspects. First, it is orthogonal to the issues of mapping array variables to memories in SoC systems. An integrated approach of the mapping of variables to memories with the work described here would exploit different reorganizations for different sections of the arrays and possibly even at different points in the execution. The combination of data reorganization across memories has the potential benefit of allowing faster pipelining by reducing the latency with which a consumer needs to interface with a producer. Second, the work presented in this paper does not address data reorganization in-place but rather takes advantage of the fact that in future FPCAs it is very likely that some level of inter-memory module data copying will be required. During this copying the data engine can therefore perform some of these data reorganization “for free” rather than requiring one of the computing cores to perform that reorganization for subsequent performance enhancement. Unlike other approaches that require manual specification of streams and their parameters, we have developed a set of abstractions that can be generated automatically by a compilation/synthesis system.

8. CONCLUSION

The growing number of available transistors on a die will enable the integration on a single chip of multiple heterogeneous IP cores with a programmable interconnect network in what we call a Field-Programmable-Core-Array (FPCA). In this context orchestrating the data and performing data reorganization will become an increasingly important issue as the speeds of computation of the cores increase. In this paper we have proposed a programmable data engine architecture for data copying and reorganization on a single memory or across multiple memory modules. We have also described the implementation of a compiler algorithm capable of detecting opportunities for transposing of array dimensions aiming at increasing the temporal reuse in an application. With the current FPGA technologies, and for a specific sets of data reorganizations, these data engines use less than 5% of the a Xilinx-Virtex® device (V1000 BG560) showing that even with today’s technology these data engines use a small amount of device space for a sustained data transfer rate of about 80 MB/sec. This experience makes the implementation of such programmable data reorganization engines in future FPCA or SoC-like chips very appealing.

9. REFERENCES

[1] F. Balasa, F. Catthoor, and H. De Man "“Dataflow-driven Memory Allocation for Multi-dimensional Signal Processing Systems”, Proceedings of the IEEE International Conference on Computer Aided Design (ICCAD’94), Santa Jose, Calif., Nov. 1994, pp. 31-34.

[2] K. Cain, J. LeBak and A. Skjellum, “Data Reorganization and Future Embedded HPC Middleware”, In Proc. of the Fourth Annual High-Performance Embedded Computing Workshop (HPEC’2000) Workshop, M.I.T. Lincoln Labs, Mass. 2000.

[3] F. Catthoor, F. Franssen, S. Wuytack, L. Nachtergaele, and H. DeMan “Global communication and memory optimizing transformations for low power signal processing systems”, IEEE workshop on VLSI signal processing, La Jolla, Calif., Oct. 1994.

[4] P. Diniz, M. Hall, J. Park, B. So and H. Ziegler, “Bridging the gap between Compilation and Behavioral Synthesis in the DEFACTO System”, To appear in the Proc. of the workshop on Languages and Compilers for Parallel Computing, (LCPC’2001), Aug. 2001.

[5] P. Diniz and J. Park, “Automatic Synthesis of Data Storage and Control Structures for FPGA-based Computing Machines”, In Proc. of the IEEE Symp. on FPGAs for Custom Computing Machines (FCCM’00), IEEE Computer Society Press, Los Alamitos, Calif., Oct. 2000, pp. 91-100.

[6] M. Gokhale and J. Stone “Automatic Allocation of Arrays to Memories in FPGA Processors With Multiple Memory Banks”, In Proc. of IEEE Symp. on FPGAs for Custom Computing Machines (FCCM’99), IEEE Computer Society Press, Los Alamitos, Calif. Oct. 1999, pp. 63-69.

[7] M. Miranda, F.Catthoor, M. Janssen and H. DeMan, “High-level address optimization and synthesis techniques for data-transfer-intensive applications”, IEEE Transactions on Very Large Scale Integration (VLSI) Systems, 6(4), Dec. 1998, pp. 677 –686.

[8] P. Panda, F. Catthoor, N. Dutt, K. Danckaert, E. Brockmeyer, C. Kulkarni, A. Vandercappelle and P. Kjeldsberg. “Data and memory optimization techniques for embedded systems”, ACM Transactions on Design Automation of Electronic System, 6(2), Apr. 2001, pp. 149-206.

[9] P. Panda, N. Dutt and A. Nicolau, “Exploiting off-chip memory access modes in high-level synthesis” Proceedings of the 1997 IEEE/ACM International Conference on Computer-Aided Design (ICCAD’97), 1997, pp. 333 – 340.

[10] J. Park and P. Diniz, “Synthesis of Memory Access Controllers for Streamed Data Applications for FPGA-based Computing Engines”, In Proc. of the International Symposium on Systems Synthesis (ISSS’2001), IEEE Computer Society Press, Los Alamitos, Calif., Oct. 2001, pp. 221-226.

[11] G. Rivera and C-W Tseng, “Data Transformations for Eliminating Cache Misses”, In Proc. of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI’98), ACM Press, New York, June 1998, pp. 38-49.

[12] “The Stanford SUIF Compilation System”, version 1.1.2 Public domain software and documentation available at http://suif.stanford.edu.

[13] Synplify synthesis reference manual. Synplicity Inc, 2000.

[14] WildStar™ Reference Manual revision 4.0, Annapolis MicroSystems Inc., 1999.

[15] M. Wolf and M. Lam, “A Loop Transformation Theory and an Algorithm for Maximizing Parallelism”, IEEE Transactions on Parallel and Distributed Systems, Oct. 1991.

[16] XILINX, Inc 2100 Logic Drive San Jose, Calif. 95214. Virtex™ 2.5V Filed Programmable Gate Arrays Product Specification. DS003(v2.4), 2000.