CMOST: A System-Level FPGA Compilation Framework

Peng Zhang[1*], Muhuan Huang[1,2*], Bingjun Xiao[2], Hui Huang[2], Jason Cong[2]
[1] Falcon Computing Solutions, Los Angeles, USA
[2] Computer Science Department, University of California, Los Angeles, USA
pengzhang@falcon-computing.com, {mhhuang, xiao, huihuang, cong}@cs.ucla.edu

ABSTRACT
Programming difficulty is a key challenge to the adoption of FPGAs as a general high-performance computing platform. In this paper we present CMOST, an open-source automated compilation flow that maps C-code to FPGAs for acceleration. CMOST establishes a unified framework for the integration of various system-level optimizations and for different hardware platforms. We also present several novel techniques on integrating optimizations in CMOST, including task-level dependence analysis, block-based data streaming, and automated SDF generation. Experimental results show that automatically generated FPGA accelerators can achieve over 8x speedup and 120x energy gain on average compared to the multi-core CPU results from similar input C programs. CMOST results are comparable to those obtained after extensive manual source-code transformations followed by high-level synthesis.

Categories and Subject Descriptors
B.5.2 [**Hardware**]: Design Aids – *automatic synthesis*

General Terms
Algorithms, Design, Experimentation

Keywords
System-Level Optimization, High-Level Synthesis, FPGA

1. INTRODUCTION
The performance improvement from traditional frequency and multi-core scaling has significantly slowed down due to power consumption issues. FPGAs provide the opportunity to exploit customization and specialization for energy-efficient computing. However, the adoption of FPGA as a computing platform is currently limited by the design productivity issues, such as exploration of large design space, and time-consuming and error-prone design environment. There is an urgent need for design automation tools to tackle these issues to enable customized computing.

High-level synthesis (HLS) tools, such as [1], establish an automated design path from C to RTL, and this enables the design of FPGA hardware using a high-level programming language for module-level designs. But there is little support for system-level design automation, which requires many microarchitecture considerations, e.g., proper memory and communication architectures to connect various RTL modules, and the integration of the hardware and software modules into the entire system.

There are several research efforts that focus on automating system generation for FPGAs, such as BORPH [2], CoRAM [3] and LEAP [4]. Using these platforms, designers do not need to be concerned with the implementation details of hardware architecture, software middleware, and HW/SW interface. In BORPH, hardware and software modules run as communicating processes in UNIX, and hardware design is based on Simulink where design options can be explored at system level. CoRAM models the system as hardware kernels, on-chip and off-chip memories and the control threads with a set of standard APIs to access these models in a high-level program. LEAP automatically instantiates caches on multi-layer memory hierarchy where designers do not need to care about tedious memory management for the hardware accelerators. A later work [5] establishes an automated compilation flow from perfectly nested loops in C-code into CoRAM models. These frameworks can generate executable systems rapidly from high-level abstraction, but do not provide automation in system-level optimizations.

State-of-the-art FPGA devices are large enough to support applications with many hardware kernels and embedded processors. The design complexity and design space at system level requires FPGA design flows to follow the platform-based design paradigm [6]. Early research on platform-based methodologies at the electronic system-level (ESL) is summarized in [7] where automated or manually guided design space exploration (DSE) is the main approach to finding good designs. Recently, the response surface model (RSM) [8] and machine learning [9] approaches are proposed to address the scalability problem. However, these general DSE-based flows do not have prior knowledge of the analytic models of the microarchitecture optimizations, and hence suffer from the scalability problem for larger applications.

Microarchitecture optimizations play a vital role in the results of FPGA designs. For example, better data reuse with available on-chip buffers can significantly reduce off-chip memory access [10]; loop transformations are applied to improve data locality in order to exploit parallelism in execution or reduce memory footprint [11]; intelligent system resources allocation among different modules can greatly improve system performance [12]. Whether to apply these optimizations and how to balance the tradeoffs between different optimizations become a significant challenge in automating the compilation process. The polyhedral model provides a unified framework for the scheduling of the repeated task instances. Some combined optimizations have been proposed based on the polyhedral model [10, 13, 14]. However, it is still a big challenge to integrate and combine all these optimization options into a fully automated implementation framework.

By tackling these challenges, our system CMOST targets at enabling software developers to work on FPGAs with a fully automated compilation flow—not only on push-button implementation but also on intelligent optimizations. The contributions of this paper are:

DAC '15, June 07 - 11, 2015, San Francisco, CA, USA
Copyright is held by the owner/author(s). Publication rights licensed to ACM.
ACM 978-1-4503-3520-1/15/06…$15.00
http://dx.doi.org/10.1145/2744769.2744807

* This work was mainly done at Computer Science Department in UCLA.

1. The first push-button compilation flow mapping general C programs into full system designs on different FPGA platforms.

2. A unified abstraction model for combinations of different microarchitecture optimization schemes using customization, mapping, scheduling and transformations.

3. Several novel techniques integrated into CMOST, including task-level dependency analysis, block-based data streaming, and automated SDF generation.

The rest of this paper is organized as follows: Section 2 describes the overall structure of CMOST. Section 3 introduces a unified model for the integration and combination of different system-level optimization schemes. Section 4 presents several novel techniques used in CMOST, followed by the experimental results and conclusion in Sections 5 and 6.

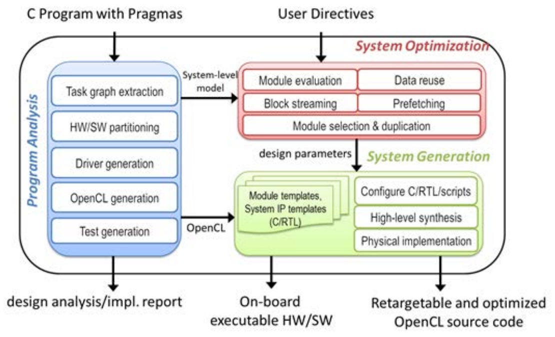
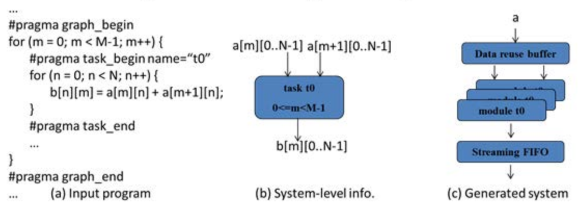


Figure 1. CMOST system diagram



```
...
#pragma graph_begin
for (m = 0; m < M-1; m++) {
    #pragma task_begin name="t0"
    for (n = 0; n < N; n++) {
        b[n][m] = a[m][n] + a[m+1][n];
    }
    #pragma task_end
    ...
}
#pragma graph_end
...
```

Figure 2. A simple example demonstrating the design flow

2. CMOST DESIGN FLOW

2.1 Overall design flow

CMOST provides a push-button design flow to generate the executable system on FPGAs from user programs, as shown in Figure 1. Programmers only need to mark the regions of the program (called tasks) for acceleration by pragmas as Figure 2 (a). Data accesses between SW and HW modules are coded directly using array references, and arbitrary loop structures are supported including imperfectly nested loops. The system-level information for the tasks such as the iteration domain and data access patterns is extracted statically and automatically as Figure 2 (b). System-level optimizations such as data reuse and module duplication are performed automatically based on the extracted high-level information as Figure 2 (c). Moreover, optimization results are encoded as parameters to instantiate the templates of the implementation files, including the software programs executed in host processors, and HLS C-code and RTLs for hardware modules.

2.2 Platform virtualization

At this point, CMOST adopts a bus-based architecture template (Figure 3(a)) to abstract away details of the hardware platform and provide portability for different platforms. The standard bus interface of HW cores makes it easy to integrate with the platform peripherals from different vendors. The template supports two acceleration scenarios (i) servers are connected to FPGA via PCIe and (ii) processor(s) are embedded in FPGA. In the automation flow

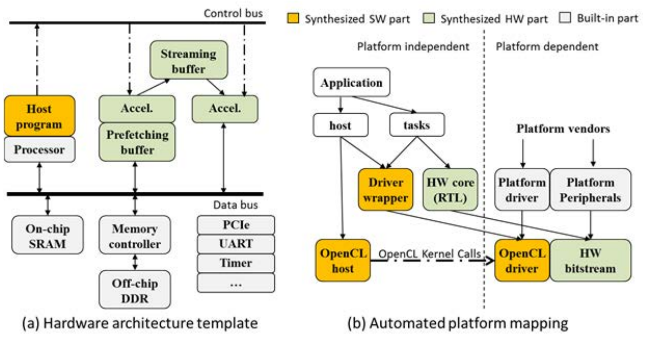


Figure 3. CMOST platform virtualization

(Figure 3(b)), the platform-dependent and platform-independent parts are separated to maximize the design reusability between platforms. CMOST generates the implementation files in the OpenCL format with standard HW/SW interfaces. The OpenCL host program is totally platform independent. OpenCL APIs invoked in the host program are implemented by the driver wrappers in CMOST where the effort to support different platform drivers is minimized.

3. ABSTRACTION FRAMEWORK

FPGA provides the opportunities to exploit high performance and energy efficiency by customization and specialization of the accelerators. The large design space results in design complexity in all aspects of computation, communication and storage subsystems. In typical designs, a sequence of optimization schemes is applied for different objectives, and the system bottleneck may switch from one aspect to another during the process. For example, in the stencil application shown in Figure 4, *data reuse* is first applied to solve the off-chip bandwidth bottleneck by allocating local reuse buffers; *data blocking* (loop tiling) reduces the reuse buffer sizes; *data prefetching* overlaps communication with the computation to increase performance of one module; then *dataflow streaming* enables data-dependent modules to execute simultaneously in a pipeline fashion; and finally, *module selection and parallelization* optimize the area/performance trade-offs among multiple modules in the streaming system. As a result, a unified modeling is required to integrate all these steps into an automated flow, and to boost the research on how to order/combine these steps efficiently.

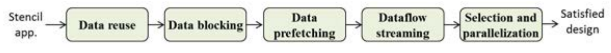
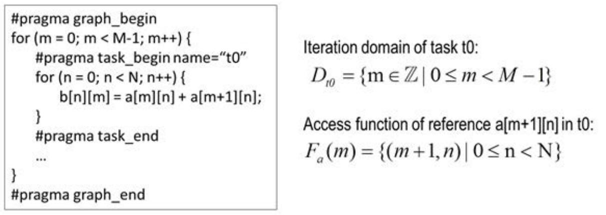


Figure 4. The optimization steps for a stencil application

3.1 Task-level application model

The feasibility and profitability of system-level optimizations are determined by system-level features of the applications. A unified application model is required to support various optimizations. The polyhedral model is used to represent the application as a set of repeatedly executed statements, a set of data arrays that the statements produce or consume, and a set of necessary constraints on the execution order of the statements to keep the semantics of the input program [14]. This abstract representation provides the opportunity for the compiler to find the proper scheduling of the statement instances for the specific optimizations instead of the original order in the sequential program.

The traditional polyhedral model used in compiler optimizations is at either statement level or loop level, and only applicable to *static control programs* [15], which require the for-loop bounds and access indexes to be affine. CMOST proposes a *task-level polyhedral model*, where the basic unit is a task that may contain a segment of code in the program. For example, in Figure 5 task *t0* contains a for-loop (*n*) inside, which is not modeled in the iteration domain; and the access function does not map iterators to a data element, but to a set of data elements accessed in the task body. The benefits of the proposed model are twofold: 1) the complexity of the model becomes flexible according to the granularity of the tasks; and 2) the program inside the task is not required to be affine, as in the traditional polyhedral model. Only the loops within the graph scope but outside the task scope need to be affine, and loop transformation are applied on them to perform task scheduling.



```
#pragma graph_begin
for (m = 0; m < M-1; m++) {
    #pragma task_begin name="t0"
    for (n = 0; n < N; n++) {
        b[n][m] = a[m][n] + a[m+1][n];
    }
    #pragma task_end
    ...
}
#pragma graph_end
```

Iteration domain of task t0:

$$D_{t0} = \{\mathrm{m} \in \mathbb{Z} \mid 0 \le m < M-1\}$$

Access function of reference a[m+1][n] in t0:

$$F_a(m) = \{(m+1, n) \mid 0 \le \mathrm{n} < \mathrm{N}\}$$

Figure 5. Task-level polyhedral representation example

3.2 Unified optimization model

By analyzing the similarity and differences of the optimization schemes, we group the schemes into four basic dimensions: *Customization*, *Mapping*, *Scheduling* and *Transformation*. With the target *Optimization* in the center, we therefore arrive at CMOST as the name for our framework. Customization models the design spaces at application level using the parameterized source code. Mapping and scheduling determine the spatial resource allocation and temporal execution for each component in the application model. Microarchitecture optimizations are represented as a set of semantic-preserving transformations of the application model. Table I shows how different system optimizations are projected into the four basic dimensions.

Table I. Generalization of the microarchitecture optimizations

	Customization	Mapping	Scheduling	Transformation
Data reuse	-	allocate SRAM for buffer	-	create local buffer and fetcher
Data blocking	-	-	be in the order of blocks	-
Prefetching	-	allocate SRAM for buffer	overlap prefetch w/ computation	create local buffer and fetcher
Streaming	-	allocate SRAM for buffer	pipeline the different stages	-
Module selection	module design space	select the options to map	-	-
Parallelization	-	determine # of duplication	parallel execution	-

Customization models the application-level design space using the parameterized programs written by users. This is inspired by Genesis2 [16], which used a model-based methodology where design templates and their configurations are separated, and exploration of the detailed module implementations can be done at system level. While Genesis only supports only Verilog/SystemVerilog, CMOST extends the template representation to support C/C++, Tcl, Perl and any textual source code. This creates a unified mechanism for separating different design considerations in the whole design flow, such as platform-dependent vs. independent, and application-dependent vs. independent constraints. Another improvement over Genesis2 is the support of describing the design space in templates, which are the ranges of the parameters. Automated design exploration can benefit from this because a joint exploration of architecture parameters and task parameters can be performed. The design space of task t can be modeled as a set of Pareto-optimal points in the design metrics space:

$$Metrics_t = \{(res_{\mathrm{ti}}, lat_{\mathrm{ti}}, thrpt_{\mathrm{ti}}) \mid 0 \le \mathrm{i} < S_t\} \qquad (1)$$

where res_{ti}, lat_{ti} and $thrpt_{\mathrm{ti}}$ are resource utilization, latency and throughput of the i-th options of the task t, and S_t is the number of design options for task t.

Mapping determines the resource allocation of the tasks and data in the application model. Binary selection variables b_{ti} indicate whether a task t is implemented as its design option i.

$$res\_s_t = \sum_i res_{ti} \cdot b_{ti} \qquad (2) \quad \text{and} \quad \sum_i b_{ti} = 1 \qquad (3)$$

where res_s_t is the resource utilization of the selected option for task t. Integer duplication factor d_t indicates the number of parallel hardware units allocated for task t.

$$res\_d_t = res\_s_t \cdot d_t; \quad lat\_d_t = lat\_s_t; \quad thrpt\_d_t = thrpt\_s_t \cdot d_t \qquad (4)$$

To model data reuse and streaming buffers, binary variables r_{aj} and s_{aj} indicate whether the reuse or prefetching scheme is applied to the access reference j of array a.

$$res\_r_{aj}[sram] = r_{aj} \cdot sram\_r_{aj}; \quad res\_r_{aj}[BW] = (1 - r_{aj}) \cdot BW_{aj} \qquad (5)$$

The constraints for mapping are the total resource for each type, e.g. LUT, FF, DSP, SRAM and BW (for off-chip bandwidth) :[1]

$$\sum_t res\_d_t[lut] \le total\_lut; \ \ldots; \ \sum_t res\_d_t[BW] \le total\_BW \qquad (6)$$

Scheduling determines the execution start time of each task instance. In the polyhedral model, scheduling functions are used to specify the execution order via an affine mapping from the task iteration domain to the space of order vectors. To simplify the discussion, we only address 1-D order/scheduling vectors.

$$\theta(\vec{x}) = \mathrm{T} \cdot \vec{x} + \vec{c} \qquad (7)$$

where $\vec{x}$ is the task instance index, and $\theta(\vec{x})$ is the order vector. However, the polyhedral model is originally used for loop transformation where only the relative order of the statement instances is of interest. Task scheduling in FPGA optimizations needs an extended model to support the execution of the pipelined and parallel task instances. In CMOST, data dependency constraints, which need to be preserved for the program semantics, consider execution latency of the task instances.

$$\phi_s(\vec{x}) + lat_s \le \phi_t(\vec{y}), \ \forall \ s[\vec{x}] \to t[\vec{y}] \qquad (8)$$

where ϕ_s is the start time vector of task s in time domain, and $s[\vec{x}] \to t[\vec{y}]$ means task instance $t[\vec{y}]$ is dependent on $s[\vec{x}]$. FPGA hardware modules are typically running in a pipelined way, where the initiation interval is the reciprocal of the throughput.

$$\phi_t(\vec{x}) + 1/thrpt_t \le \phi_t(\vec{y}), \ \forall \ t, \vec{x} + d_t \le \vec{y} \qquad (9)$$

Duplicated hardware units of the same task allow multiple task instances to start simultaneously:

$$\phi_t(\vec{x}) = (\mathrm{T} \cdot \vec{x} + \vec{c}) / d_t \qquad (10)$$

For task-level pipelining, additional constraints for the streaming buffers are required (in the case of double buffering):

$$D_s = D_t \ \wedge \ \phi_t(\vec{x} - 2) + lat_t \le \phi_s(\vec{x}) \ \wedge \ \phi_s(\vec{x}) + lat_s \le \phi_t(\vec{x})$$
$$\forall \ s \Rightarrow t, \vec{x} \in D_t \qquad (11)$$

where D_t is the iteration domain of task t, and $s \Rightarrow t$ means there is a stream from task s to task t. Finally system performance can be expressed as:

$$sys\_lat = \max_{\vec{x}}(\phi_t(\vec{x})), \ sys\_thrpt = 1 \Big/ \max_{\vec{x}}(\phi_t(\vec{x}) - \phi_t(\vec{x} - 1)) \qquad (12)$$

where t is the output task we use to measure performance. Figure 6 provides some examples of the scheduling modeling.

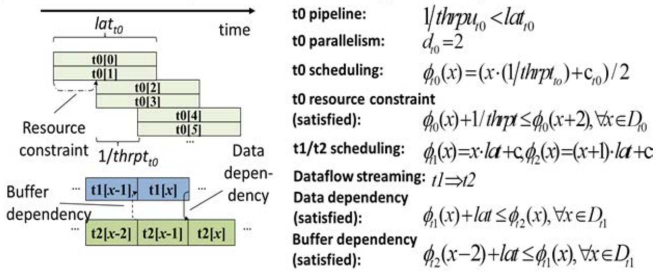

lat_{t0}
time
t0[0]
t0[1]
t0[2]
t0[3]
t0[4]
t0[5]
Resource constraint
$1/thrpt_{t0}$
Data depen-dency
Buffer dependency
t1[x-1]
t1[x]
t2[x-2]
t2[x-1]
t2[x]
t0 pipeline: $1/thrpu_{t0} < lat_{t0}$
t0 parallelism: $d_{t0} = 2$
t0 scheduling: $\phi_{t0}(x) = (x \cdot (1/thrpt_{t0}) + c_{t0})/2$
t0 resource constraint (satisfied): $\phi_{t0}(x) + 1/thrpt \le \phi_{t0}(x+2), \forall x \in D_{t0}$
t1/t2 scheduling: $\phi_{t1}(x) = x \cdot lat + c, \phi_{t2}(x) = (x+1) \cdot lat + c$
Dataflow streaming: $t1 \Rightarrow t2$
Data dependency (satisfied): $\phi_{t1}(x) + lat \le \phi_{t2}(x), \forall x \in D_{t1}$
Buffer dependency (satisfied): $\phi_{t2}(x-2) + lat \le \phi_{t1}(x), \forall x \in D_{t1}$

Figure 6. Examples of the scheduling modeling

[1] We assume all modules continue running, so the total BW is the sum of the module BWs. More complex cases are beyond the scope of this paper.

Transformation changes the application model into another semantic-equivalent form so that better mapping and scheduling results can be achieved. With this abstraction, a common scheduling and mapping engine can be separated from the microarchitecture optimizations. In Figure 7, the transformation for data reuse and data prefetching can be unified by adding a buffer *data0_t* and a task *fetcher* into the application model. Design options, such as whether to apply reuse, prefetching or the combination of reuse and prefetching, are explored transparently in the mapping and scheduling steps in a general way.

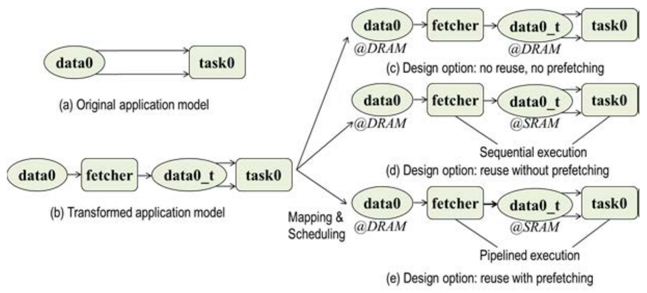


Figure 7. Unification of optimizations by transformation

3.3 Design space exploration

Under CMOST optimization modeling, the whole design space can be explored in four increasing scopes: resource-time space, semantics-reserving transformations, application alternatives coded as a template specification, and, finally, user-interactive space for different input specifications (as shown in Figure 8). The inner design space is relatively easier to model and explore automatically than the outer spaces. The resource-time space is analytically modeled where optimal designs can be found by solving mathematical programming problems. Many useful transformations are related to the specific characteristics of the hardware platforms, and they may need to be explored iteratively because of the algorithm complexity and inaccuracy in modeling the platform details. But efficient exploration schemes can be developed by considering the specific features of the transformations. With the ranges of the parameters pre-defined in the user source code, the application template space can be explored automatically. Given a pre-defined system target, the exploration is done from inner iteration to outer iteration. If inner iteration fails to find a feasible solution, another round of the outer loop iteration will be triggered until the design objectives are satisfied.

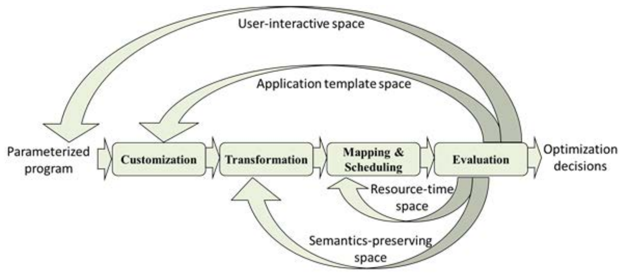


Figure 8. Multi-layer design space exploration

4. DESIGN FLOW INTEGRATION

The CMOST framework also proposes a number of novel techniques that are necessary for integrating the system-level optimizations into a fully automated design flow. We highlight a few in this section.

4.1 Task-level dependence analysis

In the task-level polyhedral model, the access function is extended to support mapping from a task instance to a set of data elements, instead of to one data element in the traditional polyhedral model. This requires the underlying dependence analysis tool be extended as well. The dependency is calculated between data regions instead of data elements. The work in [17] proposed a formulation to calculate the reusable data regions by intersecting the polytopes of the data elements that successive loop iterations access. But the reuse across two loop iterations is not considered in the formulation. Thus, it is necessary to design a general dependence distance calculation pass for the task-level polyhedral model representation to integrate data reuse into the system-level automation.

Access functions in the task-level polyhedral model can be expressed as $F_t(\vec{y}) = \{ f_t(\vec{y}, \vec{\alpha}_t) \mid \vec{y} \in D_t, \vec{\alpha}_t \in D_\alpha \}$ where $\vec{y}$ is the instance index of task *t*, $\vec{\alpha}_t$ is the iterator variable inside the task body for the array reference, and *f* is the a linear combination of components in $\vec{y}$ and $\vec{\alpha}_t$.

We define two task instances as dependent if any data element produced by one instance is used by the other instance. A dependency polytope can be used to represent the set of index pairs of the dependent task instances.

$$P_{st} = \{(\vec{x}, \vec{y}) \mid F_s(\vec{x}) \cap F_t(\vec{y}) \neq \varnothing, \vec{x} \in D_s, \vec{y} \in D_t\} \quad (13)$$

This appears to be different from the basic polyhedral model [14] in terms of the mathematic form, but if we substitute the access functions in Equation (13), P_{st} is actually in a perfectly linear form in terms of iterator variables.

$$P_{st} = \{(\vec{x}, \vec{y}) \mid f_s(\vec{x}, \vec{\alpha}) = f_t(\vec{y}, \vec{\beta}), \vec{x} \in D_s, \vec{\alpha} \in D_\alpha, \vec{y} \in D_t, \vec{\beta} \in D_\beta\} \quad (14)$$

Hence all the general polyhedral analysis methods can be applied in this task-level model. For example, the reuse buffer size is determined by the reuse distance. Reuse distance is the difference of task instance indexes between the source and reused access references, which can be conservatively calculated as

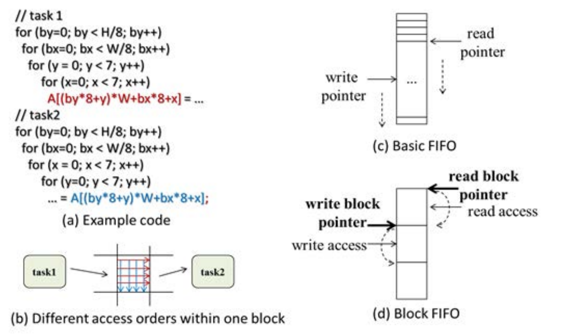
$$\vec{r} = \text{lexmax}(\vec{y} - \vec{x}) \quad \text{s.t. } (\vec{x}, \vec{y}) \in P_{st} \quad (15)$$

where *lexmax* is calculating the lexicographically maximum vector. This optimization problem can be solved by integer linear programming. The reuse distance we obtain can be used to calculate the reuse buffer size by the approach in [10].

4.2 Block-based data streaming

In the task level streaming, tasks in different pipeline stages communicate via FIFO so that the synchronization between the tasks is minimized, and read and write accesses can be performed in parallel. Automated flows such as [18] have been established to generate the FIFO-based streaming from high-level programs. However, limited by the channel type, the data communicated between the streaming stages are required to be in the same order at producer and consumer sides. So when the orders do not match, additional memory is needed at either side to perform the reordering operation.

We propose an extension of the traditional streaming framework by introducing block FIFOs. A block FIFO consists of several memory blocks where data access within one block can be accessed in random address, and the order of the blocks accessed by both sides should be the same. For example, in the DCT case in Figure 9(a), data are produced in row order in the first stage and consumed in column order in the second stage. Figure 9(d) shows the overall structure of the block FIFO, which is similar to a traditional FIFO where each data element in the FIFO is replaced by a memory block in block FIFOs. Control signals like read, write, empty and full are all at block level, and they are used to switch the FIFO pointers of blocks at two sides in a cyclic way like basic FIFO. Data accesses are only allowed within the block that FIFO pointers are pointing to, and address signals are used for random access. This locally-random-globally-ordered mechanism of block FIFOs fits well in the CMOST's task-centric application model where a block of data is accessed by each task instance.



```
// task 1
for (by=0; by < H/8; by++)
  for (bx=0; bx < W/8; bx++)
    for (y = 0; y < 7; y++)
      for (x=0; x < 7; x++)
        A[(by*8+y)*W+bx*8+x] = ...
// task2
for (by=0; by < H/8; by++)
  for (bx=0; bx < W/8; bx++)
    for (x = 0; x < 7; x++)
      for (y=0; y < 7; y++)
        ... = A[(by*8+y)*W+bx*8+x];
```

Figure 9. Example code and hardware structure for block FIFO

To fully automatically generate the block FIFO-based design, the size of the buffer and the address mapping should be determined. We first merge the references to be mapped to the block FIFO into one union data set. Then the address mapping problem can be generalized as: Given a parameterized set $\{f_t(\vec{y},\vec{\alpha}) \mid \vec{\alpha} \in D_\alpha\}$ representing the virtual address in the program, find a one-to-one mapping from this set to a non-negative integer set representing the block FIFO addresses. The maximum value in the new set should be minimized to save memory space. Research has been conducted on address mapping for memory size reduction [19], which is quite complex. We propose a novel and simple method to generate the addresses for block FIFO.

For example, let the input set be $\{256i + 8j + 5 \mid 0 \le i \le 7, 0 \le j \le 7\}$. This set does not start from zero, so we first subtract a constant from the expression and get 256*i*+8*j*. The points are scattered in the integer space, so we can divide the expression by the GCD of all the coefficients and get 32*i*+*j*. In addition, since the variable j has a small range of 8, the coefficient 32 can be reduced to 8 where a one-to-one mapping is still satisfied. Then the mapped local address in block FIFO is 8*i*+*j*. As a result, the original scattered points are mapped into a dense range from 0 to 63.

The detailed algorithm for address mapping can be summarized as Algorithm 1. The buffer size is also obtained in the algorithm.

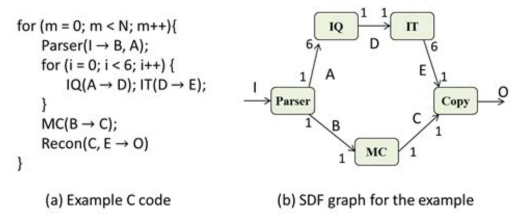
Algorithm 1 Address mapping for block FIFO

```
 1: input C; // list of the coefficients for the iterators $\vec{\alpha}$
 2: input R; // list of the ranges of the iterators $\vec{\alpha}$
 3: integer p; // current size for the mapped set
 4: sort C and R according the coefficient values (smaller first)
 5: divide all the values in C by the GCD of them; set p as 1
 6: for all the coefficients in C (indexed by i) do
 7:     if C[i] > p, C[i] = p and p = C[i]×R[i]; // coefficient reduced
 8:     else, p += C[i]×R[i];                    // already dense
 9: end for
10: Insert the constant into C to shift the staring address to zero
11: return the updated C for local address and p for buffer size
```

4.3 Automated SDF generation

System optimizations for streaming applications have been well studied in recent decades. To leverage this work, the current CMOST framework adopts the method in [12] for the scheduling and mapping optimizations. However, previous works rarely explored the issue of creating the synchronous dataflow (SDF) model from a sequential C program. Automated SDF generation is developed in CMOST to establish a fully automated design flow.

As shown in Figure 10(b), the SDF graph contains computation actors and communication edges. Data streams between actors via FIFOs in the edges. The numbers annotated on both sides of the edge represent the data rates, i.e., the number of units produced or consumed by each firing of the actor. If there are not enough data units on the input edge, the actor will not be fired.



```
for (m = 0; m < N; m++){
    Parser(I → B, A);
    for (i = 0; i < 6; i++) {
        IQ(A → D); IT(D → E);
    }
    MC(B → C);
    Recon(C, E → O)
}
```

Figure 10. MPEG example code and corresponding SDF

To generate a SDF graph from a C program, a task-level polyhedral model is first established. The challenge in generating the SDF graph is to ensure that the communication is in order between producers and consumers. By adopting block-based data streaming presented in the preceding subsection, we do not require each data element to be in order, but rather that each block be in order. The granularity of the block can be enlarged by merging multiple task instances into one block. So the main problem becomes determining a proper granularity of the actors that contain a group of task instances. If the actor is too fine-grained, the in-order requirement may not be met; and inversely, redundant memory resource will be consumed. We consider the granularity in terms of loop levels in the task iteration domain.

Theorem 1. Given the dependency polytope P_{st}, two dependent references have in-order accesses at a common loop level *l* if

$$\exists \vec{e}_{0..(l-1)}, \forall(\vec{x},\vec{y}) \in P_{st}, \vec{y}_{0..(l-1)} - \vec{x}_{0..(l-1)} = \vec{e}_{0..(l-1)} \qquad (16)$$

where $\vec{e}_{0..(l-1)}$ is a constant *l*-dimensional vector, and $\vec{x}_{0..(l-1)}$ is the first *l* dimensions of the task instance index $\vec{x}$.

We omit the proof of Theorem 1 due to page limitations. Using Theorem 1, we can test and find the maximum loop level for each pair of dependent references. A task may connect to the others with multiple accesses, so the minimum in-order loop level for all the accesses of the task is the granularity for the actor. If no loop level can satisfy the condition (16), we consider that this edge cannot be streamed and the SDF generation returns with failure. Note that 1) Theorem 1 does not give a necessary condition for the in-order test, so the result is conservative which results in a larger buffer; 2) loop transformation can be applied to enforce the access order. The details of these two issues are not addressed here due to page limitation.

5. EXPERIMENTAL RESULTS

We select a set of real applications as our test bench: Medical Imaging contains several 3-D stencils; Black Scholes has a deep computation pipeline; MPEG is a typical streaming application; NAMD performs computation intensive molecular-level simulation; and Smith Waterman performs DNA sequence alignment using dynamic programming. We use the OpenMP implementation on a state-of-the-art 6-core CPU (Intel Xeon E5-2640@2.5GHz) as the reference. Xilinx VC707 contains a Xilinx Virtex-7 chip and Convey HC-1ex contains four Xilinx Virtex-6 FPGAs. Table II shows the comparison of the execution time results on these platforms. Both CMOST and the manual designs use Xilinx Vivado_HLS and Vivado as the implementation tool.

Table II. Comparison of implementation results[1]

Design	CPU (OpenMP)	HC1-ex (CMOST)	VC707 (CMOST)	VC707 (manual)	Speed-up	Energy Gain
Medical Imaging	14s	3.5s	7.9s	7.9s	1.7x	26x
Black Scholes	0.68s	0.06s	0.63s	0.63s	1.1x	17x
MPEG	2.2s	-	0.15s	0.14s	15x	220x
NAMD	4.8s	0.37	0.62s	0.25s	8x	120x
Smith Waterman	1.73s	-	0.09s	0.06s	18x	270x

[1] The speedup and energy comparison is between VC707 (CMOST) and CPU.

The results in Table II show that CMOST can obtain over 8x speedup for the last three cases. Although the speedup on first two designs is relative small, the results are comparable to those obtained with extensive manual source-code transformations. For the last two cases, dynamic scheduling is applied manually, which is not supported in the current flow. Overall, we can achieve over 8x speedup and 120x energy gain on average.

Table III. Comparison of programming efforts

Design	# of tasks in CMOST	original code line	OpenMP changes	CMOST changes	Manual changes
Medical Imaging	6	700+	6	28	800+
MPEG	5	6000+	10	35	1500+

Table III shows that CMOST achieves the speedup with a small number of source code changes (mainly for adding some pragmas to mark the hardware task regions). Compared to the manual HLS design, a great amount of effort is saved. In addition, once a design is ready for one FPGA platform, only one line of change in the directive file is needed for platform switching and working frequency change.

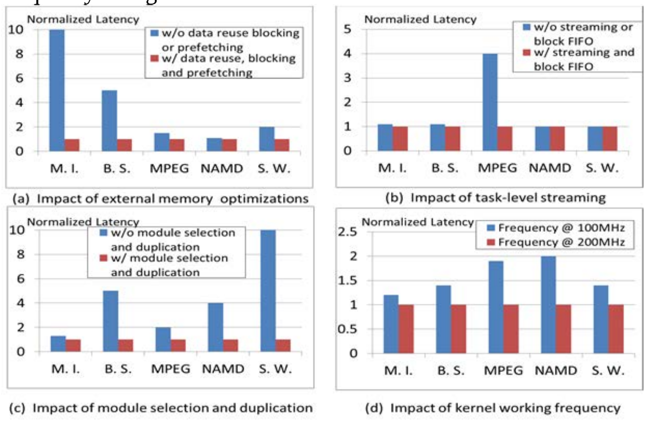


Figure 11. Impacts of optimizations on different applications

Figure 11 shows the detailed impacts of each optimization. Memory-bounded applications like medical imaging and Black Scholes benefit more from off-chip memory optimizations, while streaming applications like MPEG rely on task-level pipelining. Most of the cases benefit greatly from module duplication except that medical imaging has its bottleneck in off-chip memory accessing. Kernel frequency increase helps in all the designs, but only MPEG and NAMD can gain near-linear scaling, because these two are typical computation-bounded applications.

6. CONCLUSION

We present an open-source C-to-FPGA automation flow, which can achieve over 8x speedup and 120x energy gain on average compared to multi-core CPU results using the similar input program. A unified optimization framework is proposed for the combination of various microarchitecture optimizations. Several novel techniques are introduced for the integration of the fully automated design flow. Further work will include 1) automating the task marking process to further minimize the design efforts and enable the optimization on task partitioning; 2) improving the design results by introducing more advanced microarchitecture optimizations such as dynamic scheduling; and 3) providing automation on system evaluation and debugging. CMOST is available for download at http://vast.cs.ucla.edu/software/cmost-system-level-fpga-synthesis.

7. ACKNOWLEDGEMENTS

The authors would like to thank Young-Kyu Choi, Hassan Kianinejad, Jie Lei, Peng Li, Jie Wang, and Yuxin Wang for the efforts in CMOST development and design case study. This research is partially supported by the NSF Expeditions in Computing Award CCF-0926127.

8. REFERENCES

[1] J. Cong, B. Liu, S. Neuendorffer, J. Noguera, K. Vissers, and Z. Zhang, “High-level synthesis for FPGA: From prototyping to deployment,” IEEE Trans. on CAD, vol. 30, no. 4, 2011.

[2] H. K. So and R. Brodersen, "A unified hardware/software runtime environment for FPGA-based reconfigurable computers using BORPH," *ACM TECS.* Jan. 2008, pp. 28.

[3] E. S. Chung, J. C. Hoe, and K. Mai, "CoRAM: an in-fabric memory architecture for FPGA-based computing," the 19th ACM/SIGDA intl. symp. FPGA. New York, 97-106.

[4] M. Adler, K. E. Fleming, A. Parashar, etc, "Leap scratchpads: automatic memory and cache management for reconfigurable logic," the 19th ACM/SIGDA intl. symp. FPGA, New York, NY, USA, 25-28.

[5] G. Weisz and J. C. Hoe, "C-to-CoRAM: compiling perfect loop nests to the portable CoRAM abstraction, " the 19th ACM/SIGDA intl. symp. FPGA, 2013, pp. 211-230.

[6] K. Keutzer, A.R. Newton, J.M. Rabaey, etc., "System-level design: orthogonalization of concerns and platform-based design," *IEEE TCAD.*, vol.19, no.12, pp.1523,1543, Dec 2000

[7] A. Gerstlauer, C. Haubelt, A. D. Pimentel,T. P. Stefanov, D. D. Gajski, and J. Teich, "Electronic system-level synthesis methodologies," *IEEE TCAD*, vol. 28, no. 10, pp. 1517–1530.

[8] S. Xydis, G. Palermo, V. Zaccaria, etc., "SPIRIT: Spectral-aware Pareto Iterative Refinement Optimization for supervised high-level synthesis," *IEEE TCAD*, vol. 34, no.1, pp.155, 2015

[9] H.-Y. Liu, and L.P. Carloni, "On learning-based methods for design-space exploration with High-Level Synthesis," ACM/EDAC/IEEE DAC, 2013, pp.1,7, May 29, 2013

[10] J. Cong, P. Zhang, and Y. Zou, "Optimizing memory hierarchy allocation with loop transformations for high-level synthesis," ACM/EDAC/IEEE DAC, 2012, pp.1229,1234, 3-7 June 2012

[11] P. Panda, F. Catthoor, etc., "Data and Memory Optimizations for Embedded Systems," ACM TODAES, 6(2):142–206,2001.

[12] J. Cong, M. Huang, B. Liu, P. Zhang, and Y. Zou, "Combining module selection and replication for throughput-driven streaming programs," ACM/EDAC/IEEE DATE, 2012, pp.1018,1023, 12-16 March 2012

[13] Q. Liu, G.A. Constantinides, K. Masselos, etc., "Combining data reuse with data-level parallelization for FPGA-targeted hardware compilation: a geometric programming framework," IEEE TCAD, vol.28, no.3, pp.305-315, 2009.

[14] W. Zuo, Y. Liang, P. Li, etc., "Improving high level synthesis optimization opportunity through polyhedral transformations, " the 19th ACM/SIGDA intl. symp. FPGA, 2013, New York, NY, USA, 9-18.

[15] C. Bastoul, "Code generation in the polyhedral model is easier than you think," the 13th International Conference on Parallel Architecture and Compilation Techniques, PACT 2014, vol., no., pp.7,16, 29 Sept.-3 Oct. 2004

[16] O. Shacham, S. Galal, S. Sankaranarayanan, etc., "Avoiding game over: Bringing design to the next level," Design Automation Conference (DAC), 2012 49th ACM/EDAC/IEEE , vol., no., pp.623,629, 3-7 June 2012

[17] L.-N. Pouchet, P. Zhang, P. Sadayappan, etc., "Polyhedral-based data reuse optimization for configurable computing," the ACM/SIGDA international symposium on FPGA, 2013, ACM, New York, NY, USA, 29-38.

[18] Sx Verdoolaege, Hx Nikolov, and Tx Stefanov, "pn: a tool for improved derivation of process networks." EURASIP J. Embedded Syst. 2007, 1 (January 2007), 19-19.

[19] A. Darte, R. Schreiber, and G. Villard, "Lattice-based memory allocation," Computers, IEEE Transactions on , vol.54, no.10, pp.1242,1257, Oct. 2005