FORMLESS: Scalable Utilization of Embedded Manycores in Streaming Applications

Matin Hashemi
Sharif University of Technology
mhashemi@ee.sharif.edu

Mohammad H. Foroozannejad
Soheil Ghiasi
University of California, Davis
{mhforoozan,ghiasi}@ucdavis.edu

Christoph Etzel
University of Augsburg
christoph.martin.etzel@student.uni-augsburg.de

Abstract

Variants of dataflow specification models are widely used to synthesize streaming applications for distributed-memory parallel processors. We argue that current practice of specifying streaming applications using *rigid* dataflow models, implicitly prohibits a number of platform oriented optimizations and hence limits portability and scalability with respect to number of processors. We motivate *Functionally-cOnsistent stRucturally-MalLEabe Streaming Specification*, dubbed FORMLESS, which refers to raising the abstraction level beyond fixed-structure dataflow to address its portability and scalability limitations. To demonstrate the potential of the idea, we develop a design space exploration scheme to customize the application specification to better fit the target platform. Experiments with several common streaming case studies demonstrate improved portability and scalability over conventional dataflow specification models, and confirm the effectiveness of our approach.

Keywords Dataflow Graph, Stream Application, Embedded Manycore Processor

1. Introduction

Actor-oriented specification models, such as task graphs and other dataflow-based representations, have yielded promising results for synthesis and optimization of streaming applications on distributed memory parallel processors [1–4]. Parallel software synthesis from such models is especially favorable due to the explicit specification of concurrency, which allows straight-forward synthesis of parallel implementations by proper allocation and scheduling of computation and communication.

In principle, specifying the application as a set of tasks and their dependencies is meant to only model the functional aspects of an application, which should enable seamless portability to new platforms by fresh platform-driven allocation and scheduling of tasks and their executions. However, such specifications are rather *rigid* in that some non-behavioral aspects of the application are *implicitly* hard coded into the model at design time. Consequently, allocation and scheduling processes are likely to generate poor [1] implementations when one tries either to port the application to different platforms, or to explore implementation design space on a range of platform choices [5]. The limitations of conventional dataflow-based models with portability, scalability and subsequently the ability to explore implementation tradeoffs (e.g., with respect to number of cores) have become especially critical with availability of platforms with a large number of processor cores, which can dedicate a wide range of resources to an application [6, 7].

As an example, consider merge sort dataflow network, which is composed of actors for splitting the data into segments, sorting of segments using a given algorithm (e.g., quicksort), and merging of the sorted segments into a unified output stream. A specific instance of the sort network would have rigid structural properties, such as number of sort actors or fanin degree of merge actors. The choice of structure, although implicitly hard coded into the specification, is orthogonal to application's end-to-end functionality. It is intuitively clear that the optimal network structure would depend on the target platform, and automatic software synthesis from a rigid specification is bound to generate poor implementations over a range of platforms.

Our driving observation is that the scalability limitation of software synthesis from rigid dataflow models could be addressed if the specifications were sufficiently malleable at compile time, while maintaining functional consistency. We present an example manifestation of the idea, dubbed FORMLESS, which extends the classic notion of dataflow by abstracting away some of the unnecessary structural rigidity in the model. In particular, malleable aspects of the dataflow structure are modeled using a set of parameters, referred to as the *forming vector*. Assignment of values to the parameters instantiates a particular structure of the model, while all such assignments lead to the same end-to-end functional behavior. A simple example of a forming set parameter is the fanin degree of merge actors in the sort example.

Our approach opens the door to design space exploration methodologies that can *hammer out* a FORMLESS specification to form an optimized version of the model for the target platform. The "formed" model can be subsequently passed onto conventional allocation and scheduling processes to generate a quality parallel implementation. We also present such a design space exploration scheme that determines the forming set using platform-driven profiles of application tasks. Experimental results demonstrate that FORMLESS yields substantially improved portability and scalability over conventional dataflow modeling. Note that the primary objective in this paper is to demonstrate the merit of malleable specifications in terms of scalability. Development of a formal higher-order programming language or a sophisticated design space exploration are beyond the scope of this paper.

[1] We focus on throughput as the quality metric.

LCTES 2012 June 12–13, 2012, Beijing, China

2. Background and Preliminaries

The notion of dataflow is a natural fit to modeling of data-intensive streaming applications [1–4]. Synchronous dataflow (SDF) is a special dataflow model of computation in which data rates are specified statically. SDF-compliant kernels are at the heart of many streaming applications [8, 9], and form the focus of our work. In the SDF model, a task (actor) is a tuple (In, Out, F), where $In \subseteq InputPorts$ is the set of input ports, $Out \subseteq Outputports$ is the set of output ports, and F denotes the transformation function of the task. Each port has a statically-defined rate, which is the mapping $Rate : Ports \rightarrow \mathbb{N}$. A streaming application can be modeled as a directed graph $G(V, E)$, where vertices (V) represent tasks, and directed edges (E) is a subset of $InputPorts \times OutputPorts$, which represent data communication channels. Each port is connected to exactly one channel, and each channel is connected to ports of some task. A task can be fired upon availability of sufficient data on all its input ports. Firing of a task consumes data from its input ports, and produces data on its output ports. The execution is meant to continue indefinitely. Figure 2.B shows an example SDF.

2.1 Software Synthesis

In synthesizing streaming software, we target execution platforms whose abstract model exposed to the synthesis process can be viewed as a number of identical distributed-memory parallel processors which are arranged in a mesh and communicate via point-to-point FIFO channels. Many existing manycores conform to this abstraction [6, 7]. Moreover, the model is reasonably accurate at high-level for other platforms that implement the abstract view using different underlying architecture. For instance, network-based inter-processor communication coupled with proper system software can implement virtual inter-processor FIFO channels.

Automated parallel software synthesis for stream applications normally involves several key steps that are fairly well researched [1]. Figure 1 shows the most key steps, namely, assignment of tasks to processors, scheduling of tasks for periodic execution on the processors, layout of the processors on the manycore chip, and code generation. As discussed in later sections, we present our method as an improvement on top of the baseline software synthesis.

Figure 2 illustrates an example. Figure 2.B shows the SDF graph for an example streaming sort application, which sorts 100 data tokens per invocation. The `split` task reads 100 tokens from the input stream, and divides them into two segments of 50 tokens that are passed onto the two `sort` tasks. After the segments are sorted, the `merge` task combines them into the final sorted output stream. Figure 2.C shows an example task assignment, and 2.D the corresponding generated code.

Task functionalities are provided as sequential computations that are kept intact throughout the synthesis process. The software code for each processor is synthesized by stitching together the set of tasks that are assigned to that processor according to their schedule. For tasks that are assigned to the same processor, inter-task communication is implemented using arrays. That is, the producer task writes its data to an array, which is then read by the consumer task. Inter-processor communication is implemented using `read` and `write` system calls.

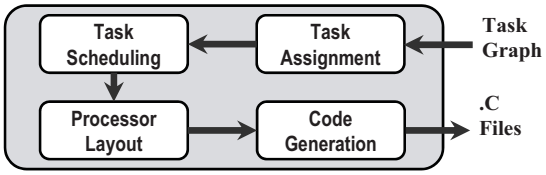

Figure 1. Baseline software synthesis.

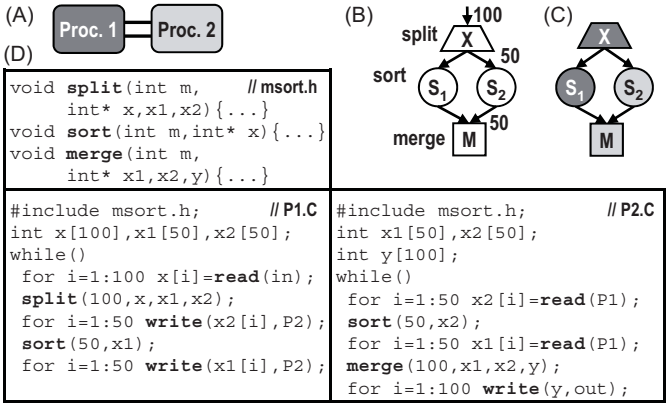

Figure 2. A) Example platform. B) Sort application modeled as a SDF. C) Tasks are assigned to processors (color coded). D) Synthesized software.

3. FORMLESS SDF

3.1 Motivating Example

To motivate the underlying idea of FORMLESS, we consider the sort example of Figure 2, and investigate the scaling of throughput when platforms with different number of processors are targeted. Let us assume that the `sort` task implements the quicksort algorithm, and the `merge` task merges two sorted data segments into one stream using the mergesort algorithm.

An immediate observation is that the example task graph cannot readily utilize many (more than 4 in the case of depicted task graph) processors due to the limited concurrency in the specification. At the other extreme, the throughput of the synthesized software is going to be poor when one processor is targeted, compared to eliminating the split and merge tasks and running a single sort task (i.e., the quicksort algorithm) on the entire input stream [2]. This is partly because the overhead of coordination among multiple parallel tasks is only justified if sufficient amount of parallelism exists in the platform. Intuitively, increasing concurrency in the task graph specification facilitates utilization of more parallel resources and potentially increases the potential for improving performance via load balancing between processors, however, it comes at the cost of degraded performance when platforms with fewer processors are targeted.

Having made this observation, our idea is to specify the tasks and their composition using a number of parameters. Adjustment of parameters enables "massaging" the structure of the task graph to fit the target architecture, while all candidate task graphs deliver the same end to end functionality.

Figure 3 sketches the idea for the example sort application in which fanout degree of the split task and fanin degree of the merge task are parametrically specified. The number of tasks, their types and compositions, as well as their data production rates are immediate functions of the two split-fanout and merge-fanin parameters. Three example instances of the FORMLESS graphs are shown in Figure 3.

[2] The discussion does not pertain to sorting of large databases which does not entirely fit in the memory.

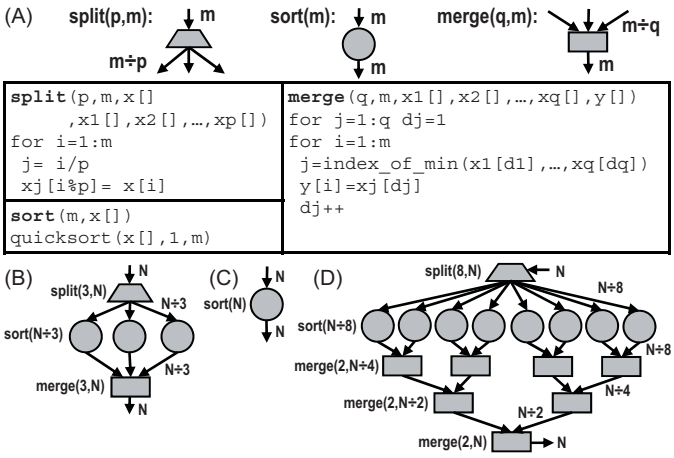

Figure 3. FORMLESS specification of the sort example: A) Actor specifications. B-D) Example instantiations.

3.2 Formalism

We make the key observation that SDF specifications are structurally rigid. Such task graphs do not fully live up to the intended promise of separating functional aspects of the application from implementation platform, and thus, fail to deliver efficient portability and scalability with respect to number of processors in the platform. To address the portability and scalability limitations, not only application specification has to be sufficiently separated from implementation platform, but it also has to admit platform-driven transformations and optimizations.

We propose raising the level of abstraction in specifications to eliminate the rigid structure of the task graph, while preserving its functional behavior. Our approach is to require application designers to specify the tasks and the structure of the task graph using a number of parameters, referred to as the *forming vector*. Specifically, a forming vector Φ is defines as

$$\Phi = (\phi_1, \phi_2, \ldots, \phi_{|\Phi|})$$

where ϕ_j is a *forming parameter* whose possible set of values are a subset of domain δ_j. Hence, domain of the forming vector Φ is equal to

$$\Delta = \delta_1 \times \delta_2 \times \ldots \times \delta_{|\Phi|}$$

We extend the definition of a task α such that input ports, output ports and data transformation function of α are all specified as functions of the underlying parameters in Φ. In other words, task α is defined as the tuple

$$\forall \Phi \in \Delta_\alpha : \ \alpha(\Phi) = \big(In_\alpha(\Phi), Out_\alpha(\Phi), F_\alpha(\Phi)\big)$$

For example, the merge task in Figure 3.A is defined based on the forming vector $\Phi = \{q, m\}$. The function $In_{merge}(q, m)$ specifies q input ports of rate $\frac{m}{q}$, and function $Out_{merge}(q, m)$ specifies one output port of rate m. The data transformation function $F_{merge}(q, m)$ specifies a mergesort algorithm which combines q sorted input arrays of size $\frac{m}{q}$ into a single sorted output array of size m. In this example, $\Delta_{merge} = \{(q, m) \mid m \geq 2, q \geq 2, m \bmod q = 0\}$.

We also extend the definition of task graph $G(V, E)$ such that tasks (V) and channels (E) are specified as functions of the underlying parameters in Φ. Formally, task graph G is defined as the tuple

$$\forall \Phi \in \Delta_G : \ G(\Phi) = \big(V_G(\Phi), E_G(\Phi)\big)$$

$V_G(\Phi)$ is a function which specifies the set of tasks in G based on forming vector Φ, and is formally defined as

$$V_G(\Phi) = \big\{\alpha_1(\Phi_1), \alpha_2(\Phi_2), \ldots, \alpha_{|V|}(\Phi_{|V|})\big\}$$

where $\alpha_i(\Phi_i)$ is an instance of task α_i which is formed based on forming vector Φ_i, and both α_i and Φ_i are determined based on the given Φ. For instance, the task graph in Figure 3.B is specified based on forming vector $\Phi = \{p, q, m\} = \{3, 3, N\}$, and function V_G specifies the set of tasks as

$$V_G(3, 3, N) = \big\{\text{split}(3, N), \text{sort}(\frac{N}{3}), \text{sort}(\frac{N}{3}), \text{sort}(\frac{N}{3}), \text{merge}(3, N)\big\}$$

in which, for example, task merge$(3, N)$ is instance of merge(q, m) $= \big(In_{\text{merge}}(q, m), Out_{\text{merge}}(q, m), F_{\text{merge}}(q, m)\big)$, where $\{q, m\} = \{3, N\}$.

Similarly, $E_G(\Phi)$ is a function which specifies the set of channels in G based on the forming vector Φ, and is formally defined as

$$E_G(\Phi) = \big\{(prd, cns) \mid prd \in Out_{\alpha_i}(\Phi_i), cns \in In_{\alpha_j}(\Phi_j)\big\}$$

where (prd, cns) denotes a channel from an output port prd of some task α_i to an input port cns of some task α_j.

We would like to stress that our primary objective in this paper is to demonstrate the merit of the idea and scalability of malleable specifications. In our scheme, it is the programmer's duty to define the ports, task computations and graph composition based on the parameters. Furthermore, he has to ensure that every assignment of values from the specified domain Δ_G to the forming vector Φ results in the same functional behavior. This tends to be straight forward since tasks perform the same high-level function under different parameters (e.g. splitting, sorting or merging in the example of Figure 3).

3.3 Higher-Order Language

Development of a formal higher-order programming language involves many considerations that are beyond the scope of this paper [10–12]. However, in this section we present an example realization of the general idea that we have developed.

Figure 4.A presents the prototype for specifying task and application task graph based on a set of parameters. The task specification starts with a list of forming parameters and their type. The `interface` section specifies the set of input and output ports of the task, and the `function` section specifies its data transformation function, all based on the given parameters.

Similarly, application specification also starts with a list of forming parameters. The `interface` section is the same as task interface. In a `composition` section, the tasks are instantiated by assigning the corresponding parameters using the `instantiate` construct. The channels are instantiated using the `connect` construct which connect ports of two tasks.

Figure 4.B shows the code for our previously mentioned sort application. For example, the merge task is specified with two parameters m and q. As we see the number and rate of input ports in this task is defined using a `for` loop. In general we allow a rich set of programming constructs such as `for` and `if-else` in order to provide enough flexibility in specifying the tasks based on the given forming parameters.

3.4 Related Work

Unfolding an SDF graph [13] is to construct a larger SDF which consists of multiple copies of the original graph. The unfolded SDF has the same functional behavior while expressing more parallelism. This technique can be employed to scale the throughput

(A)

```
task ActorName ( //list of parameters
                 Type1 ParamName1,
                 Type2 ParamName2,
                 ... ){
  interface {
    //list of input and output ports
    input InputPortName1 ( PortRate );
    input InputPortName2 ( PortRate );
    ...
    output OutputPortName1 ( PortRate );
    ...
  }
  function {
    //data transformation function
  }
}
```

```
application AppName ( //list of parameters
                      ...
                    ){
  interface {
    //list of input and output ports
    ...
  }
  composition {
    //actors:
    instantiate ActorName ActorID (ParamValue1, ...);
    ...
    //channels:
    connect ( ActorID.PortName, ActorID.PortName);
    ...
  }
}
```

(B)

```
task Merge ( int M, //length of output
             int Q, //fan-in degree
           ){
  interface {
    output merged_array ( M );
    for ( i=0; i < Q; i++ )
      input sub_array[i] ( M/Q );
  }
  function {
    //the mergesort algorithm
  }
}
```

```
task Sort ( int M //length of the array
          ){
  interface {
    input  unsorted_array ( M );
    output sorted_array ( M );
  }
  function {
    //the quicksort algorithm
  }
}
```

```
task Split( int M, //length of input
            int P, //fan-out degree
          ){
  ...
}
```

```
application MergeSort ( int P, //split fan-out degree
                        int Q  //merge fan-in degree
                      ){
  interface {
    input input_array ( N );
    output output_array ( N );
  }
  composition {
    if (P==1) ...
    else {
      //tasks:
      instantiate Split split ( N, P );
      for (i=0; i < P; i++)
        instantiate Sort sort[i] ( N/P );
      int D = log(P,Q);
      for (d=D-1; d >=0; d--) for (i=0; i < Q^d; i++)
        instantiate Merge merge[d][i] ( N/Q^d, Q );
      //channels:
      connect ( input_array, split.input_array );
      for (i=0; i < P; i++)
        connect ( split.output_array[i], sort[i].unsorted_array );
        connect ( sort[i].sorted_array,  merge[D-1][i/Q].sub_array[i%Q] );
      for (d=D-1; d > 0; d--) for (i=0; i < Q^d; i++)
        connect ( merge[d][i].merged_array, merge[d-1][i/Q].sub_array[i%Q] );
      connect ( merge[0][0].merged_array, output_array);
    }
  }
}
```

Figure 4. A) Prototype for specifying task and application. B) An example malleable specification for the sort application in Figure 3.

to platforms with larger number of cores. Another technique which also preserves the functional behavior and expresses more parallelism is to convert the input SDF graph to HSDF [13]. The FORMLESS approach introduced in this paper is orthogonal to such techniques and can be applied in parallel with unfolding or conversion to HSDF.

A number of dataflow extensions such as parameterized dataflow [14], scenario-aware dataflow [15], variable-rate dataflow [16] and schedulable parametric dataflow [17] primarily focus on specifications which enable different static and/or dynamic dataflow behaviors based on the parameters. Our focus, however, is to specify different possible implementations for the same application behavior, in order to achieve scaling of performance with respect to the number of processors. We are able to employ a rich set of programming constructs to specify many aspects of the task graph based on the forming parameters (Section 3.3). For example, not only the production/consumption port rates but also the number of ports for each task can be specified based on the parameters.

StreamIt compiler [3] automatically detects stateless filters (data-parallel tasks) and judicially parallelizes them in order to achieve better workload balance and hence scaling of performance. This approach provides some level of malleability, but it is limited to data-parallel tasks because it fully relies on the *compiler's* ability to detect malleable sections in the application.

In CUDA, scaling of performance is achieved by specifying the application with as much parallelism as practically possible. At runtime, an online scheduler has access to a pool of threads from which the non-blocked threads are selected and executed on available cores [18]. This enables the scaling of performance to newer devices with larger number of processors. However, performance optimization for a specific target GPU device fully relies on the *programmer* to optimally specify the application, e.g., the number of blocks per grid and the number of threads per block [19].

In MPI, the programmer may describe the amount of parallelism based on a set of parameters such as the number of available or idle cores. However, since the data rate of communications among MPI processes are not necessarily known at compile time, the allocation and scheduling of the processes are performed by the operating system at runtime.

We require the *programmer* to provide a malleable specification, and also, employ *complier* optimizations to select the best task graph based on the malleable specification at compile time.

4. Exploration of Forming Parameter Space

To examine the merits of FORMLESS, we developed a design space exploration (DSE) scheme whose block diagram is depicted in Figure 5. The DSE instantiates a platform-driven task graph $G(\Phi_{opt})$ from a given FORMLESS specification by optimizing the forming vector Φ. Central to the quality of the DSE are high-level estimation algorithms for fast assessment of the throughput of a specific instance of the task graph.

Task Profiling: The workload associated with a task is composed of two components: computation workload and communication induced workload. Since tasks are defined parametrically, their computation workload depends on the values of the relevant forming parameters. In addition, computation workload is inherently input-dependent, due to the strong dependency of the tasks' control flow with their input data. The communication-induced workload exists if some of the producers (consumers) of the data consumed (produced) by the task are assigned to a different processor. We take an empirical approach to characterize the computation workload. We measure the execution latency of several instances of the tasks (based on the forming parameters) on the target processor. For each case, we profile the runtime for several randomly generated input streams to average out the impact of input-dependent execution times. The data is processed via regression testing to obtain latency estimates for all parameter values. Hence, for a task $\alpha(\Phi)$, the profiling data provides DSE with a computation workload W_α.

In addition, for a channel (α, α') with communication volume $N_{(\alpha,\alpha')}$, the communication-induced workload of producer and consumer tasks are analytically characterized as $W_{write} \times N_{(\alpha,\alpha')}$ and $W_{read} \times N_{(\alpha,\alpha')}$, respectively. W_{write} and W_{read} are the profiled execution latency of platform communication operations.

Task Graph Formation: Formation of a task graph is essentially assignment of valid values to the forming parameters. Any such assignment implies a specific instantiation, which can be passed onto subsequent stages for quality estimation. Our current DSE implementation exhausts the space of forming vector parameters by enumeration, due to the manageable size of the solution space in our testcases, and quickness of subsequent solution quality estimation. In principle, high-level quality estimations can analyze performance bottlenecks to provide feedback and to guide the process of value assignment to forming set parameters. Note that our primary objective in this paper is to demonstrate the scalability of malleable specifications, and not development of a sophisticated DSE.

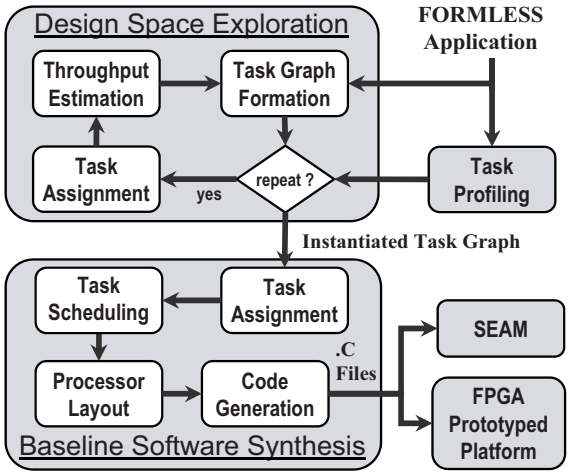

Figure 5. Design space exploration for platform-driven instantiation of a FORMLESS specification.

Task Assignment: Task assignment is a prerequisite to application throughput estimation, and quantifying the suitability of a candidate task graph for a target platform. Tasks' computations should be distributed among processors as evenly as possible while interprocessor communication is judiciously minimized. This can be modeled as a graph partitioning problem, in which a graph $G(V, E)$ is cut into a number of subgraphs $G_p(V_p, E_p)$, one for each processor. We employ METIS graph partitioning package [20] for this purpose because our primary focus is to quickly generate solutions to enable integration within the iterative DSE flow. Every vertex (task) $\alpha \in V$ is assigned a weight W_α which denotes its computation workload, and every edge (α, α') is assigned a weight of $N_{(\alpha,\alpha')}$ which denotes its communication volume.

Throughput Estimation: For typical FIFO channels with small latency (relative to processors' execution period), the communication overhead only appears as communication-induced workload on processors (Section 2). That is, the workload of a processor can be estimated as:

$$W_p = \sum_{\alpha \in V_p} W_\alpha \quad + W_{read} \times \sum_{\alpha \notin V_p, \alpha' \in V_p} N_{(\alpha,\alpha')} + W_{write} \times \sum_{\alpha \in V_p, \alpha' \notin V_p} N_{(\alpha,\alpha')}$$

where W_{read} and W_{write} denote the execution latency of platform read and write system calls. The last two terms indicate communication-induced workload on p. We use workload of the slowest processor to estimate the throughput. Formally

$$Throughput = 1 \div \max_{1 \le p \le P} W_p$$

For a given task assignment, throughput of a candidate solution depends on the buffer sizes of the platform FIFO channels [2], as well as the firing schedule of the tasks that are assigned to the same processor. The above equation merely serves to provide a rough throughput estimate for guiding the DSE. Note that we accurately simulate the impact of interconnect limited buffer size in our final experimental evaluations, which are performed using synthesized software from FORMLESS models (Section 5).

5. Experimental Evaluation

5.1 Application Case Studies

To demonstrate the merits of our idea, we experiment with low-density parity check (LDPC), advanced encryption standard (AES), fast fourier transform (FFT), parallel merge sort (SORT) and matrix multiplication (MMUL).

Low-Density Parity Check: A regular LDPC code is characterized by an $M \times N$ parity check matrix, called the H matrix. N defines the code length and M is the number of parity-check constraints on the code (Figure 6.A). Based on matrix H, a Tanner graph is defined which has M check nodes and N variable nodes. Each check node C_i corresponds to row i in H and each variable node V_j corresponds to column j. A check node C_i is connected to V_j if H_{ij} is one (Figure 6.B). The input data is fed to the variable nodes, and after processing goes to the check nodes and again back to the variable nodes. This process repeats R times, where R depends on the specific application of the LDPC code. In practice, the H matrix has hundreds or thousands of rows and columns, and therefore, given the complexity of edges in the Tanner graph, we decided not to use this graph as the task graph for software implementation. In fact, direct hardware implementation of the Tanner graph is also not desired because a huge portion of the chip area would be wasted for routing resources [21].

We construct the task graph in the following manner. The variable and check nodes are collapsed into single nodes, and subse-

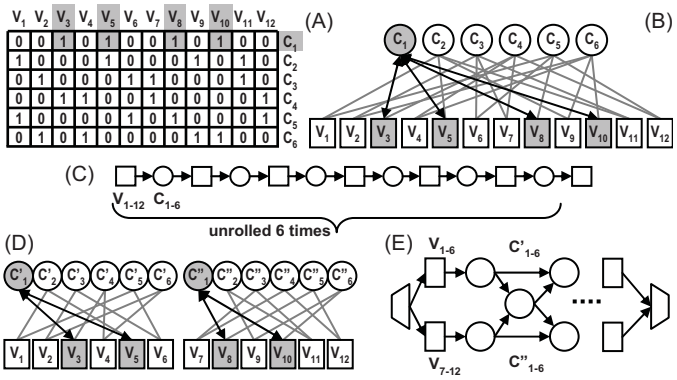

Figure 6. LDPC application: A) Sample H matrix. B) Tanner graph. C) Task graph. Row-Split LDPC based on [21] : D) Tanner graph. E) Task graph.

quently, the graph is unrolled R times (Figure 6.C). We experiment with the LDPC code used in 10GBASE-T standard, where the matrix size is 384×2048 and $R = 6$.

In order to have a malleable specification, we decided to employ the Row-Split method which is a low-complexity message passing LDPC algorithm and is originally developed for hardware implementation [21]. In this method, in order to reduce the complexity of the edges, the Tanner graph is generated while the rows are split by a factor of $\phi = 2, 4, 8$ or 16. As shown in Figure 6.D for $\phi = 2$, the variable nodes are divided into $\phi = 2$ groups, $V_1, \ldots, V_{\frac{N}{2}}$ and $V_{\frac{N}{2}}, \ldots, V_N$, and each check node C_i is split into $\phi = 2$ nodes C_i' and C_i''. The corresponding task graph is shown in Figure 6.E, where additional synchronization nodes are required for the check nodes. Interested readers may refer to [21] for further details.

Advanced Encryption Standard: The AES is a symmetric encryption/decryption application which performs a few rounds of transformations on an stream of 128-bit data (4×4 array of bytes). The number of rounds depends on the length of the key which is 10 for 128-bit keys. As shown in Figure 7.A, the task graph for the AES cipher consists of four basic tasks called `sub`, `shf`, `mix` and `ark`. Task `sub` is a nonlinear byte substitution which replaces each byte with another byte according to a precomputed substitution box. In `shf`, every row r in the 4×4 array is cyclically shifted by r bytes to the left. Task `mix` views each column as a polynomial x, and calculates modulo $x^4 + 1$. Task `ark` adds a round key to all bytes in the array using XOR operation. The round keys are precomputed and are different for each of the 10 rounds.

Therefore, tasks `sub` and `ark` can be parallelized over all elements of the array, and task `shf` only over the four rows, and task

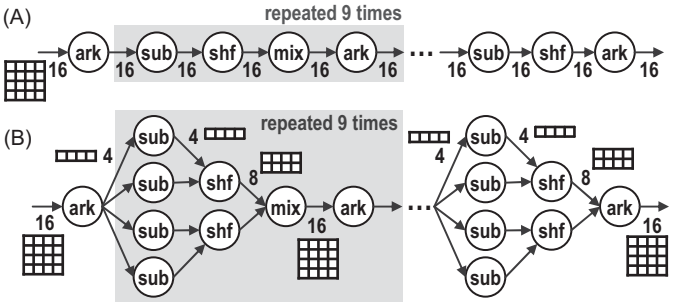

Figure 7. AES: A) $\Phi = (1, 1, 1, 1)$ B) $\Phi = (4, 2, 1, 1)$.

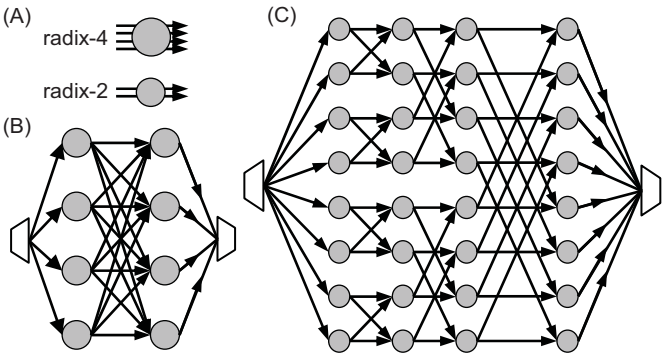

Figure 8. A) Radix-2 and radix-4 butterfly tasks. B) 16-point FFT application with radix-4 butterfly tasks. C) The same FFT computed with radix-2.

`mix` only over the four columns. We constructed the FORMLESS task graph with four parameters. ϕ_1, ϕ_2 and ϕ_4 control the number of rows that the array is divided into for the `sub`, `shf` and `ark` tasks. Parameter ϕ_3 controls the number of columns that the array is divided into for the `mix` task. For example, the task graph of Figure 7.B is formed by $\Phi = (4, 2, 1, 1)$.

Fast Fourier Transform: Fourier tansform of an input array is an array of the same size. Fast Fourier Transform (FFT), an efficient algorithm for this computation, is performed using a number of basic butterfly tasks connected in a dataflow network. The basic butterfly operation calculates Fourier of two inputs and is called a radix-2 butterfly. In general, however, FFT can be calculated using butterfly operations with radices other than 2, although typically powers of 2 are used.

An N-point radix-r FFT uses a dataflow network of radix-r butterfly tasks. This network is organized in $\log_r^N$ stages each containing $\frac{N}{r}$ butterfly tasks. Figure 8.B shows the structure of the dataflow network for a 16-point FFT application using radix-4 butterfly tasks. Figure 8.C shows the same computation performed using radix-2 butterflies. Since the computation of FFT is independent of the choice of radix, we define our FORMLESS model for FFT based on a forming parameter ϕ_1 which is the radix. The radix determines structure of the task graph as well as inter-task data communication rates.

Parallel Merge Sort: A forming parameter ϕ_1 controls the number of parallel sort actors, and a parameter ϕ_2 controls the fanout and fanin degree of the split and merge actors (Figure 9). The value of ϕ_1 should be an integer power of ϕ_2 to generate a valid task graph, i.e., $\phi_1 = \phi_2^n$, $n \geq 0$.

Matrix Multiply: The objective is to calculate $A \times B = C$. A block (submatrix) of C can be calculated by multiplying the corresponding blocks of matrix A and B. Adjusting the block size in C trades off the degree of concurrency among operations with

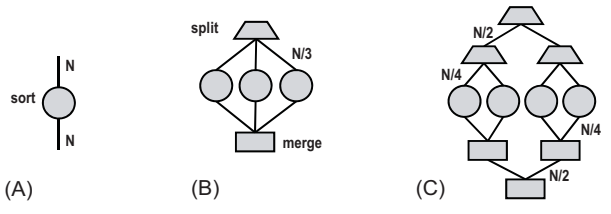

Figure 9. Parallel Merge Sort: A) $\Phi = (1, 2)$, B) $\Phi = (3, 3)$, C) $\Phi = (4, 2)$.

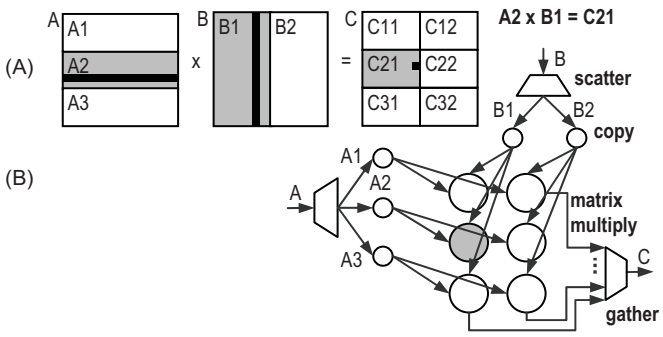

Figure 10. Matrix Multiply: A) Block operations for $\Phi = (3, 2)$. B) Task graph formed with $\Phi = (3, 2)$.

the required amount of data replication and movement. Therefore, we construct a FORMLESS task graph with two parameters ϕ_1 and ϕ_2 that control the number of row and column blocks that matrices A and B are divided into. The task graph of Figure 10.B is formed by $\Phi = (3, 2)$.

The domains of the forming vectors used in experimenting the above applications are shown in Figure 11. For example in the AES application, each of the four forming parameters can be 1, 2 or 4.

5.2 Experiment Setup

We implemented both FORMLESS design space exploration and baseline software synthesis schemes (Figure 5). For a given number of processors, P, within the range of 1 to 100, an optimized task graph $G(\Phi_{opt})$ is constructed, and subsequently, parallel software modules (separate `.C` files) are synthesized for this task graph.

We consider the following FPGA-prototyped multiprocessor system for throughput measurement of the synthesized software modules. Each processor is an Altera NiosII/f core with 8KB instruction cache and 64KB data cache. The communication network is a mesh which connects the neighbor processors with FIFO channels of depth 1024. The processors use a shared DDR2-800 memory, but they only access their own region in this memory, i.e., they communicate only through the FIFO channels. The compiler is `gcc` in Altera NiosII IDE with optimization flag `-O2`.

Due to limited FPGA capacity, we were able to implement the above architecture with up to 8 cores. For more number of cores, we employ our previously developed Sequential Execution Abstraction Model (SEAM), which is cycle-accurate in simulating the effect of inter-processor communication (e.g., blocks on empty or full buffers), and also accurately predicts any deadlock situation [22, 23].

SEAM, however, abstracts the local execution phases of every processor, in which no communication with the other processors occur, as deterministic `wait` periods. For example, SEAM replaces the function calls `sort(50,x2)` and `merge(100,x1,x2,y)` with corresponding wait functions on processor 2 in Figure 2.D. For a task α, the wait period is W_α which comes from the profiling (Section 4). Subsequently, a behavioral Verilog model is generated which captures the behavior of `wait` and `read/write` operations in every processor. The generated Verilog models are interfaced to the Verilog model of the interconnect network with exact buffer resources, and simulated using commercial Verilog simulators to obtain application throughput. Hence, SEAM relies on the accuracy of profiling data for modeling sequential phases of each processor, but it is far more scalable than cycle-accurate simulation of a manycore system in Verilog. We have confirmed SEAM's accuracy by comparing it with smaller scale multiprocessor systems that we could prototype in FPGA [22, 23].

Benchmark	Vector Φ	Domain of Φ
LDPC	(ϕ_1)	$\delta_1 = \{1, 2, 4, 8, 16\}$
AES	$(\phi_1, \ldots, \phi_4)$	$\delta_1 = \ldots = \delta_4 = \{1, 2, 4\}$
FFT	(ϕ_1)	$\delta_1 = \{2, 4\}$
SORT	(ϕ_1, ϕ_2)	$\delta_1 = \{1, 2, 3, 4, 8, 9, 16, 27\}, \delta_2 = \{2, 3\}$
MMUL	(ϕ_1, ϕ_2)	$\delta_1 = \delta_2 = \{1, \ldots, 10\}$

Figure 11. The domain of the forming parameters in our benchmark applications.

5.3 Measurement Results

We compare the throughput of the best instantiated task graph, i.e., $G(\Phi_{opt})$, with the throughput of rigid task graphs. Figure 12 presents the application throughput normalized relative to single-core throughput. The black curves show the throughput values obtained through SEAM simulations from synthesized parallel implementations, and the 8 black squares show the throughput measured on the FPGA prototype for systems up to 8 processors. The gray curves show throughput of a few rigid task graphs selected.

The experiments show that rigid task graphs have a limited scope of efficient portability and scalability with respect to number of processors. For example, an LDPC rigid task graph constructed with forming vector $\Phi = (4)$ does not scale beyond 40 processors. Note that a rigid task graph which scales to large number of processors does not necessarily yields the best throughput in smaller number of processors. For example, an AES rigid task graph constructed with $\Phi = (2, 2, 1, 2)$ only yields the highest throughput for 90 or more processors. For a single processor, this rigid task graph yields 74% of the best instantiated task graph.

Similar scenarios happen in all benchmark case studies. Each forming vector Φ yields the highest throughput only for a range of targets. In other words, throughput of the best instantiated task graph consistently beats the throughput of any rigid task graph. This result validates the effectiveness of FORMLESS in extending the scope of efficient portability and scalability with respect to number of cores.

It is interesting to see that, for example, in the matrix multiply application $\Phi = (5, 5)$ is not selected for the 25-core target. Instead, the DSE tool selected $\Phi = (6, 4)$ which has 24 multiply tasks. This forming set is not intuitive because one would normally split the multiplication workload into an array of $5 \times 5 = 25$ multiply tasks for 25 cores. The DSE tool considers the effect of smaller tasks (e.g., split tasks), and the communication-induced workloads as well. This again proves that an automated tool outperforms manual task graph formation. However, the DSE is able to scale performance only if the programmer has provided meaningful parallelism. For example in the SORT application, a larger value for the forming parameter ϕ_1 results in more parallel sort tasks, but the performance does not scale beyond 13 processors because the workload of the terminal merge stage, which is not parallelizable through FORMLESS, becomes the bottleneck.

Figure 12 can also be used to determine a reasonable target size, i.e., the number of processors, for each application. For example in the AES application, more than 40 processors does not yield a throughput gain unless we have at least 50 processors.

6. Concluding Remarks

We presented FORMLESS, a parametric extension to the static dataflow model, which enables portable and scalable development of streaming applications for manycore platforms. We demonstrated the applicability of the idea using several common streaming case studies. Experimental results confirmed the validity and applicability of the idea, while showcasing portability and scalability limitations of conventional task graphs.

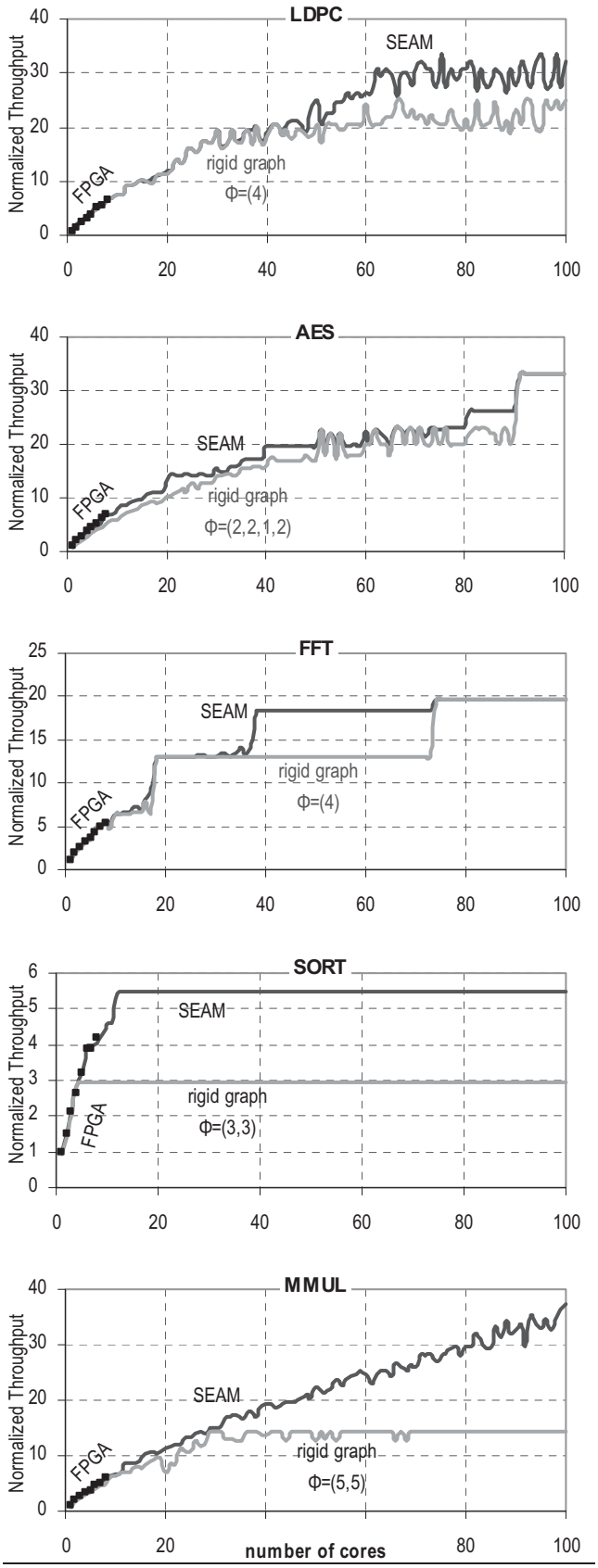

Figure 12. Application throughput on manycore platforms normalized with respect to single-core throughput. The black curve shows the throughput obtained from DSE instantiated task graphs. The gray curves show the throughput of sample rigid task graphs.

References

[1] S. Battacharyya, E. Lee, and P. Murthy. *Software synthesis from dataflow graphs*. Kluwer Academic Publishers, 1996.

[2] S. Stuijk, M. Geilen, and T. Basten. Throughput-buffering trade-off exploration for cyclo-static and synchronous dataflow graphs. *IEEE Transactions on Computers*, 57(10):1331–1345, 2008.

[3] M. Gordon. *Compiler techniques for scalable performance of stream programs on multicore architectures*. PhD thesis, Massachusetts Institute of Technology, 2010.

[4] Andy D. Pimentel et al. Exploring embedded-systems architectures with Artemis. *IEEE Computer*, 34(11):57–63, 2001.

[5] A. Sangiovanni-Vincentelli et al. Benefits and challenges for platform-based design. *Design Automation Conference (DAC)*, pages 409–414, 2004.

[6] D. Truong et al. A 167-processor 65 nm computational platform with per-processor dynamic supply voltage and dynamic clock frequency scaling. *Symposium on VLSI Circuits*, 2008.

[7] S. Bell et al. TILE64 processor: A 64-core SoC with mesh interconnect. *International Solid-State Circuits Conference (ISSCC)*, 2008.

[8] E. Lee and D. Messerschmitt. Synchronous data flow. *Proceedings of the IEEE*, 75(9):1235–1245, 1987.

[9] M. Geilen and T. Basten. Reactive process networks. *International Conference on Embedded Software (EMSOFT)*, pages 137–146, 2004.

[10] J. Colaço, A. Girault, G. Hamon, and M. Pouzet. Towards a higher-order synchronous data-flow language. *International Conference on Embedded Software (EMSOFT)*, pages 230–239, 2004.

[11] W. Taha. A gentle introduction to multi-stage programming. *Domain-Specific Program Generation, Lecture Notes in Computer Science (LNCS)*, 2004.

[12] J. Adam Cataldo. *The power of higher-order composition languages in system design*. PhD thesis, University of California, Berkeley, 2006.

[13] Marc Geilen. Reduction techniques for synchronous dataflow graphs. *Design Automation Conference (DAC)*, 2009.

[14] B. Bhattacharya and S. Bhattacharyya. Parameterized dataflow modeling for DSP systems. *IEEE Transactions on Signal Processing*, 49(10):2408–2421, 2001.

[15] B.D. Theelen et al. A scenario-aware data flow model for combined long-run average and worst-case performance analysis. *Formal Methods and Models in CoDesign*, 2006.

[16] Maarten H. Wiggers, Marco J.G. Bekooij, and Gerard J.M. Smit. Buffer capacity computation for throughput constrained streaming applications with data-dependent inter-task communication. *IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2008.

[17] Pascal Fradet, Alain Girault, and Peter Poplavko. A schedulable parametric data-flow MoC. *Design, Automation, and Test in Europe (DATE)*, 2012.

[18] J. Nickolls et al. Scalable parallel programming with CUDA. *ACM Queue*, 6:40–53, March 2008.

[19] *CUDA C best practices guide*, chapter 4.4. March 2011.

[20] G. Karypis and V. Kumar. METIS 4.0: Unstructured graph partitioning and sparse matrix ordering system. Technical report, CS Dept., University of Minnesota, Minneapolis, 1998.

[21] T. Mohsenin, D. Truong, and B. Baas. Multi-split-row threshold decoding implementations for LDPC codes. *International Symposium on Circuits and Systems (ISCAS)*, 2009.

[22] Po-Kuan Huang, Matin Hashemi, and Soheil Ghiasi. System-level performance estimation for application-specific mpsoc interconnect synthesis. *Symposium on Application Specific Processors (SASP)*, 2008.

[23] Matin Hashemi. *Automated Software Synthesis for Streaming Applications on Embedded Manycore Processors*. PhD thesis, University of California, Davis, 2011. Chapter 4.