Towards Scalable Utilization of Embedded Manycores in Throughput-Sensitive Applications

Matin Hashemi and Soheil Ghiasi
University of California, Davis
Department of Electrical and Computer Engineering
Davis, CA 95616
{hashemi,ghiasi}@ucdavis.edu

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, has limited 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 model for 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.

1. INTRODUCTION

Actor-oriented application 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, 2, 3, 4, 5]. 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 implementations[1] when one tries either to port the application to different platforms, or to explore implementation design space on a range of platform choices [6]. 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 [7, 8].

As an example, consider the merge sort dataflow network, which is composed of actors for splitting the data into segments, sorting of data 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 functional behavior. 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 set*. Assignment of values to forming set 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.

2. BACKGROUND AND PRELIMINARIES

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 [9, 10], and form the focus of our work. In the SDF model, an actor (task) is a tuple (In, Out, F), where $In \subseteq InputPorts$ is the set of input ports, $Out \subseteq$

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

978-1-4577-1743-7/11/$26.00 ©2011 IEEE

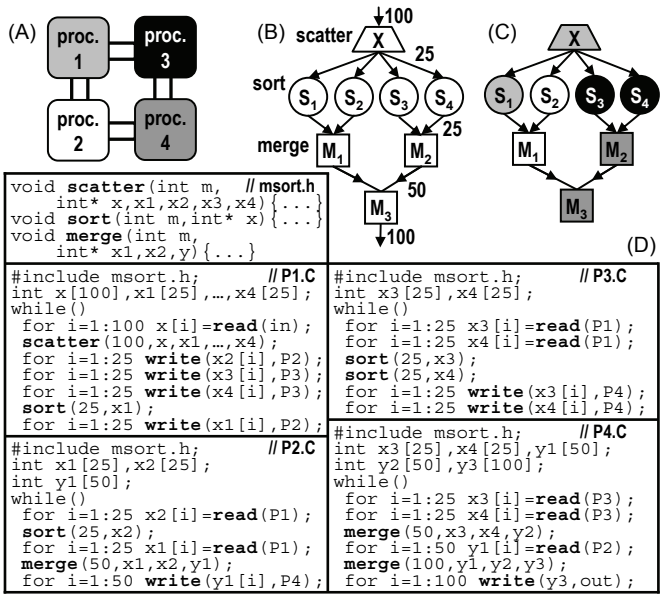

Figure 1: A) Example platform. B) Sort application modeled as a SDF. C) Tasks are assigned to processors (color coded). D) Synthesized software modules. Outputs of tasks S_i and M_i are implemented with arrays `xi` and `yi`, respectively.

Outputports is the set of output ports, and F denotes the transformation function of the actor. $Ports = InputPorts \cup Outputports$, and $InputPorts \cap Outputports = \varnothing$. 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 $G(A, C)$, where vertices (A) represent actors, and directed edges (C) is a subset of $Ports^2$, which represent data communication channels. Each port is connected to exactly one channel, and each channel is connected to ports of some actor [11].

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, which can be connected to input ports of other tasks or possibly output streams of the application. The execution is meant to continue indefinitely. Figure 1.B shows an example.

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 distributed-memory parallel processors communicating via inter-processor FIFO channels. Many existing manycores conform to this abstraction [7, 8]. 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.

Synthesis of parallel software modules deals with several key issues, such as scheduling and allocation of tasks, that are fairly well researched [2]. We present our work in subsequent sections with reference to a baseline software synthesis scheme that is summarized below using a simple example.

Figure 1.A shows an example abstract target platform with four processors. Figure 1.B illustrates the SDF graph for an example streaming sort application, which sorts 100 data tokens per invocation. The `scatter` task reads 100 tokens from the input stream, and divides them into segments of 25 tokens that are passed onto the four `sort` tasks. After the four segments are sorted by the `sort` tasks, two `merge` tasks combine the four segments into two larger sorted data segments of size 50. Finally, another `merge` task combines the two segments and generates the sorted output stream.

In the baseline synthesis scheme, tasks are statically scheduled to allow infinite periodic repetitions [2]. For example, $1(XS_1S_2S_3S_4M_1M_2M_3)$ is a valid periodic schedule. Then, tasks are assigned to processors. Figure 1.C illustrates an example task assignment in which tasks X and S_1, S_2 and M_1, S_3 and S_4, M_2 and M_3 are assigned to processors 1, 2, 3, and 4, respectively.

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.D illustrates the generated software modules for the example.

3. MOTIVATING EXAMPLE

To motivate the underlying idea of FORMLESS, we consider the sort example of Figure 1.A, 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.

An immediate observation is that the example task graph cannot readily utilize many (more than 8 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 scatter 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 inter-task communication 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 2 sketches the idea for the example sort application in which fanout degree of the scatter task and fanin degree of the merge task are parametrically specified. The number of tasks, their type and composition, as well as their data production rates are immediate functions of the two scatter-fanout and merge-fanin parameters. Three example

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

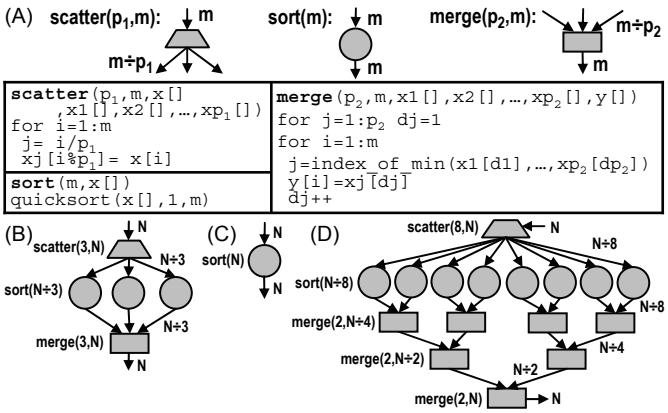

Figure 2: A) FORMLESS specification of the sort example. B-D) Example instantiations.

instances of the FORMLESS graphs are shown in Figure 2.

4. FORMLESS SDF

We make the key observation that SDF specifications are structurally rigid. Such task graphs do not 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 set. Let p be a vector of integer parameters, and R be a vector of their corresponding valid ranges. That is, $\forall i, p_i \in R_i$, and $R_i \subset \mathbb{N}$. The forming set includes all elements of p: $FS = \bigcap_i p_i$. Furthermore, let S be a vector of sets, whose elements are tuples defined on selected elements of R. For example, $S_1 = R_2 \times R_5$.

We extend the definition of actor i (A_i^*), to the tuple $(In(S_i), Out(S_i), F(S_i))$, where $In(S_i) \subset InputPorts \times S_i)$, $Out(S_i) \subset OutputPorts \times S_i)$, and $F(S_i)$ is the new data transformation function of the actor, which is specified as a function of the underlying parameters in S_i. The definition of ports is naturally extended to include all input and output ports: $Ports(A_i) = In(S_i) \cup Out(S_i)$, $Ports(FS) = \bigcup_i In(S_i) \cup \bigcup_i Out(S_i)$, and $\forall i, j : In(S_i) \cap Out(S_j) = \varnothing$. The static data rate mapping of actor ports is extended to be a function of the actor parameters: $Rate^* : Ports(A_i) \rightarrow \mathbb{N}$. Given vectors p, R, S and the forming set FS, the SDF graph is extended to $G(A^*, C^*)$, where A^* denotes the set of parametrized actors, and $C^* \subset Ports(FS)^2$.

In the sort example of Figure 2, parameters p_1 and p_2 control the fanout and fanin degree of the scatter and merge tasks, respectively. In our experiments (Section 5), we worked with $R_1 = \{1, 2, 3, 4, 8, 9, 16, 27\}$ and $R_2 = \{2, 3\}$. S sets of the `scatter` and `sort` tasks include R_1, and S set of the `merge` task includes R_2. Note that p_1 and p_2 cannot be selected independently.

We would like to stress that our primary objective in this work is to demonstrate the merit of the idea and scalability of malleable specifications, rather than development of a formal higher-order programming language de-emphasizing implementation aspects [12, 13, 14]. In our scheme, the onus is on the programmer to define the ports, actor computations and graph composition based on the parameters. Furthermore, he has to ensure that every assignment of values from the specified range to parameters 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. scattering, sorting or merging in the example of Figure 2).

4.1 Exploration of Forming Set Space

To examine the merits of FORMLESS, we developed a design space exploration (DSE) scheme whose block diagram is depicted in Figure 3. The DSE instantiates a platform-driven task graph from a given FORMLESS specification by optimizing the forming set parameters. 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.

4.1.1 Task Profiling

The workload associated with a task is composed of two components: computation 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. For example, the runtime of the quicksort algorithm on a list depends on the ordering of the numbers in the list. 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 characterization of 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. In addition, the communication-induced workload is analytically characterized as the latency of platform communication operations multiplied by task's communication volume to another processor. Tasks' latency estimates are compiled into a lookup-table that is available to the DSE tool.

4.1.2 Task Graph Formation

Formation of a task graph is essentially assignment of valid values to forming set 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 set parameters by enumeration, due to the manageable size of the solution space in our testcases, and quickness of subsequent solution quality estimation. In principle, however, high-level quality estimations can analyze performance bottlenecks to provide feedback and to guide the process of value assignment to forming set parameters.

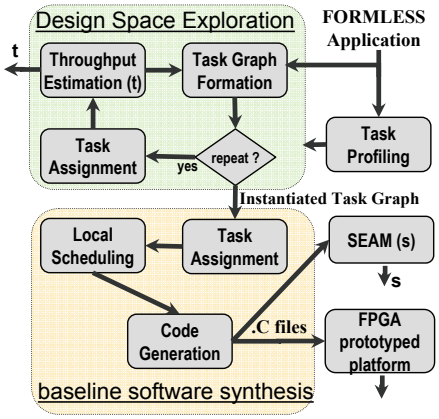

Figure 3: Design space exploration for platform-driven instantiation of a FORMLESS specification.

4.1.3 Task Assignment

Task assignment is a pre-requisite to application throughput estimation, and quantifying the suitability of a candidate task graph for a target platform. To maximize throughput, tasks' computations have to be distributed among processors as evenly as possible while inter-processor communication is judiciously minimized. 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:

$$W_p = \sum_{v \in V_p} W_v + \; In \sum_{u \notin V_p, v \in V_p} N_{e(u,v)} + \; Out \sum_{u \in V_p, v \notin V_p} N_{e(u,v)}$$

where V_p denotes the set of tasks assigned to processor p, W_v the computation workload of task v, and N_e the number of data tokens transferred over edge e from task u to task v. In and Out denote the latency of platform read and write system calls. The last two terms indicate communication-induced workload on p.

The task assignment can be viewed as packing objects of various size into a certain number of bins to minimize the largest bin size, with the unique requirement that object sizes partially depend on the bin assignments (due to assignment-dependent communication overhead). We use several fast packing heuristic algorithms, and select the solution that produces the highest throughput according to high-level estimates. Our primary focus is to quickly generate solutions to enable integration within the iterative DSE flow.

Our first heuristic implements the best-fit principle: tasks are sorted according to their computation workload. Visiting in the sorted order, each task is assigned to the processor that has the least total computation workload so far. Note that the communication-induced workload of tasks and processor workloads are resolved as tasks are assigned to processors. The second heuristic randomly assigns tasks to processors, and is repeated a constant number of times, and the best result is reported. Our third algorithm applies the first method on the first P tasks, and then switches to the second algorithm for the remainder of the tasks, where P is the number of processors.

4.1.4 Throughput Estimation

The throughput of a candidate solution depends on the buffer sizes of the platform FIFO channels [5]. FIFO channels with large buffers disentangle the steady state execution of processors at which point, the throughput of the system is determined by the slowest processor [], i.e., the processor with maximum workload assuming identical processors. The following theorem, which formalizes the result, forms the basis for our throughput estimation algorithm:

THEOREM 4.1. *Assuming large inter-processor buffers, for any task assignment of dataflow graph G to P processors, there exists an ordering of tasks on processors such that every precedence constraints is met (possibly by overlapping iterations), and the steady state execution period (inverse of throughput) of the application is* $\max_{1 \le p \le P} W_p$.

In our DSE high-level estimations, we assume the interconnection network has enough buffering capacity to disentangle steady state execution of processors. However, the communication-induced workload is accurately modeled as discussed above. Note that we accurately simulate the impact of interconnect buffer capacity in our final evaluations, which are performed using synthesized software from FORMLESS models (Section 5). The buffers are only assumed to be large during DSE to enable fast iterative exploration.

5. EXPERIMENTAL EVALUATION

To demonstrate the merits of our idea, we implemented both FORMLESS design space exploration and baseline software synthesis schemes (Figure 3). We were challenged with accessing scaled distributed-memory manycores, as neither they nor their cycle-accurate simulators were not accessible to evaluate the synthesized software. Consequently, we develop an abstract model to accurately estimate application throughput. We confirm the accuracy of the model by comparing it with smaller scale multicores that we could prototype on an FPGA. Note that typical FPGAs have insufficient resources to accommodate prototyping platforms with many processors.

5.1 SEAM: Sequential Execution Abstraction Model

The basic observation is that the local execution phase of every processor, in which no communication with the other processors occurs, can be abstracted as deterministic latencies without compromising much accuracy. Specifically, we replace the sequential computation of the tasks with a function `wait(w)`, where w is the computation workload of that task according to profiling results. As an example, SEAM replaces the function calls `sort(25,x2)` and `merge(50,x1,x2,y1)` with corresponding wait functions on processor 2 in Figure 1.D.

This approach simplifies software modules into a number of `wait` functions that represent the task computations, and `read/write` system calls that represent inter-processor communications. Subsequently, we generate a behavioral Verilog model, which essentially captures the behavior of wait and read/write operations in that processor. The generated models are interfaced to the Verilog model of the interconnection network to accurately model the impact of buffers, and simulated using commercial Verilog simulators

to obtain application throughput. Note that SEAM is far more scalable than cycle-accurate simulation of a manycore system.

To confirm the accuracy of SEAM, we emulated several multiprocessors of small complexity on Altera DE2 board using NiosII soft processors. We mapped and executed synthesized software to the emulated processors, and measured the steady state throughput on hardware. We compared the results with SEAM estimations. Figure 4 shows the error percentage between SEAM estimations (s column) and the actual measured throughput, which are quite small. The "D" entries refer to the cases in which, small buffers caused the application to deadlock, which are always accurately predicted by SEAM.

Architecture		Application							
		sort{2,2}		sort{9,3}		mmul{3,1}		mmul{3,3}	
# of cores	fifo depth	s	t	s	t	s	t	s	t
2	1024	2.7	2.7	15	15	1.2	1.2	13	13
3	1024	1.1	1.1	1.1	1.1	0.3	0.3	1.4	1.4
4	1024	1.1	1.1	19	17	0.3	0.3	0.2	0.2
2	32	2.7	2.7	D	D	4.4	9.1	12	45
3	32	D	D	0.4	50	D	D	D	D
4	32	1.1	0.6	D	D	0.3	12	D	D
Geo. Mean		1.6	1.4	3.4	11	0.68	1.6	2.6	3.6

Figure 4: %error in estimation of application throughput by SEAM (s) and high-level task assignment estimates (t) vs. FPGA emulated systems. Deadlocks (D) are accurately predicted by SEAM.

5.2 Application Case Studies

We experimented with merge sort, fast fourier transform (FFT), matrix multiplication and advanced encryption standard (AES) applications. As described previously, in the sort application parameters p_1 and p_2 control the fanout and fanin degree of the scatter and merge tasks, respectively. FFT application is constructed as the well-known butterfly network and a parameter p_1 controls the butterfly radix.

Our third case study is matrix multiplication ($A \times B = C$). As illustrated in Figure 5.A, a block (submatrix) of C, e.g., C_{21}, can be calculated by multiplying the corresponding blocks of matrix A and B, e.g., $A_2 \times B_1$. Adjusting the block size in C trades off the degree of concurrency among operations with the required amount of data replication and movement. Therefore, we construct a FORMLESS task graph with two parameters p_1 and p_2 that control the number of row and column blocks that matrices A and B are divided into. The task graph of Figure 5.B is formed by $p = \{3, 2\}$.

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 6.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,

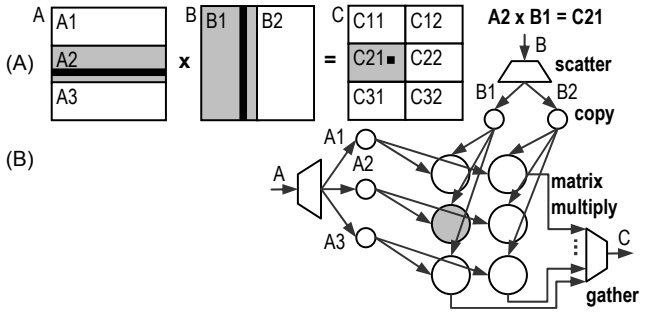

Figure 5: Matrix multiply: A) Block operations for $p = \{3, 2\}$. B) Task graph formed with $p = \{3, 2\}$.

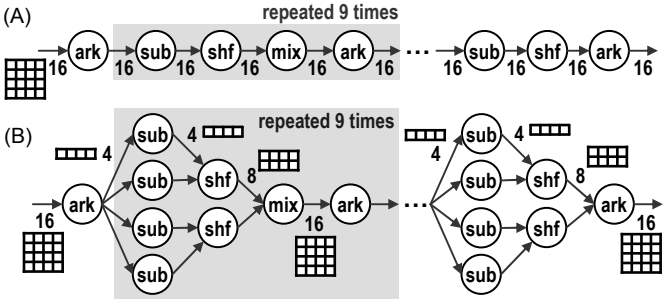

Figure 6: AES application: A) $p = \{1, 1, 1, 1\}$ B) $p = \{4, 2, 1, 1\}$.

and task `mix` only over the four columns. We decided to construct the FORMLESS task graph with four parameters. $p1$, $p2$ and $p4$ control the number of rows that the array is divided into for the `sub`, `shf` and `ark` tasks. Parameter $p3$ controls the number of columns that the array is divided into for the `mix` task. For example, the task graph of Figure 6.B is formed by $p = \{4, 2, 1, 1\}$.

As discussed in Section 4, every p_i has a valid range R_i. The R_i ranges that we used in experimenting the above four applications are shown in Figure 7.A. For example in the AES application, each of the four p_i's can be 1, 2 or 4.

5.3 Results

Figure 7.B presents the application throughput numbers normalized relative to single-core throughput. The results, obtained through SEAM simulations from synthesized parallel implementations, show that throughput of the FORMLESS applications consistently increases with increasing number of cores in all the case studies. Throughput of the best instantiated task graph (shown in black color) consistently beats the throughput of any rigid task graph. Note that rigid task graphs, some of which are shown in gray, have a limited scope of efficient portability and scalability with respect to number of cores. To better visualize the results in cases with small number of cores, we show the ratio of the two gray throughput curves with a dashed curve.

In the AES application, $p = \{4, 2, 1, 2\}$ is selected by the DSE tool for platforms with 80 to 90 cores. However, this specific task graph $G(\{4, 2, 1, 2\})$ does not yield the highest throughput on smaller or larger targets. For example on smaller targets (less than 30 cores), it has less throughput than task graph $G(\{1, 1, 1, 1\})$. This is better seen on the dashed ratio curve.

Similar scenarios happen for sort, matrix multiply and FFT case studies as well. Each forming set yields the highest throughput only for a range of targets. This result validates

(A)

Merge Sort	{ p_1,p_2 }	R1={1,2,3,4,8,9,16,27} R2={2,3}
FFT	{ p_1}	R1={2,4,16}
Matrix Multiply	{ p_1,p_2 }	R1=R2={1,2,3,4,5,6}
AES	{ p_1,p_2,p_3,p_4 }	R1=R2=R3=R4={1,2,4}

(B)

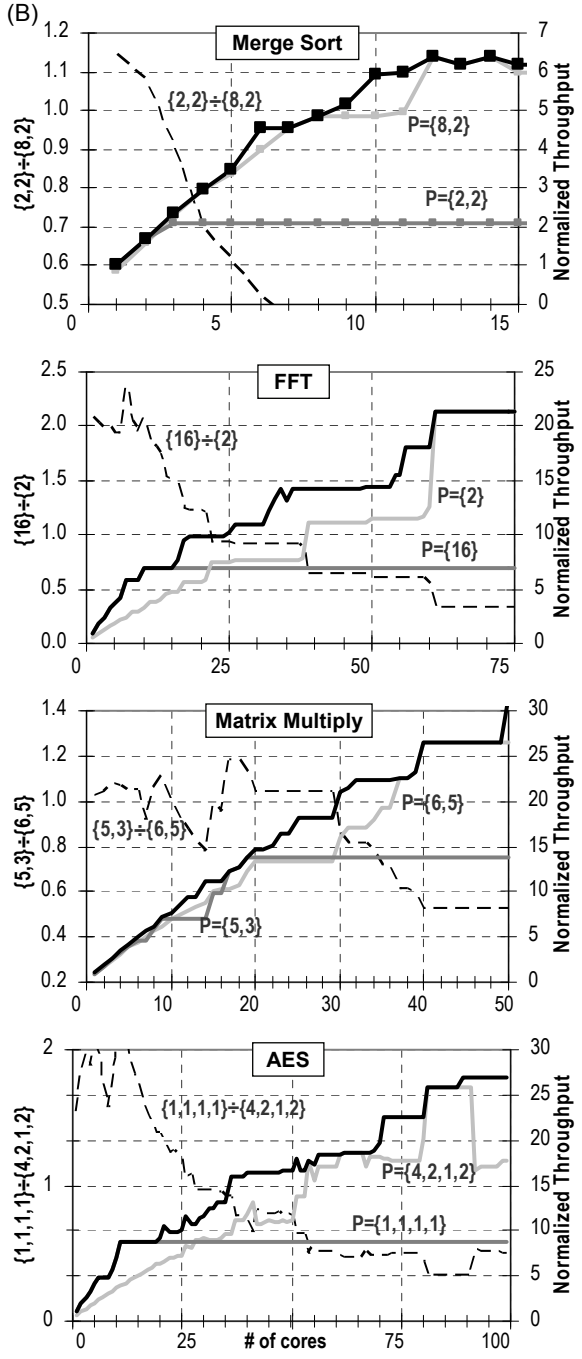

Figure 7: Application throughput on manycore platforms normalized with respect to single processor throughput. The black curve shows the throughput obtained from the DSE instantiated task graphs. The gray curves show the throughput of sample fixed task graphs. The dashed curve is the ratio of the two gray curves which is added for more visibility in small number of cores.

the effectiveness of FORMLESS in improving the portability and scalability with respect to number of cores.

It is interesting to see that, for example, in the matrix multiply application forming set $P = \{6, 6\}$ is not selected for the 36-core target. Instead, the DSE tool selected $P = \{6, 5\}$ which has 30 multiply tasks. This forming set is not intuitive because one would normally split the multiplication workload into an array of $6 \times 6 = 36$ multiply tasks for 36 cores. The DSE tool considers the effect of smaller tasks (e.g., copy tasks), and the communication-induced workloads as well. This again proves that an automated tool outperforms manual task graph formation.

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 demonstrate the validity and applicability of the idea, while showcasing limitations of conventional task graphs with respect to portability and scalability.

7. REFERENCES

[1] Gul Agha. *Actors: A Model of Concurrent Computation in Distributed Systems*. PhD thesis, MIT, 1985.

[2] Shuvra S. Battacharyya, Edward A. Lee, and Praveen K. Murthy. *Software Synthesis from Dataflow Graphs*. Kluwer Academic Publishers, 1996.

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

[4] Michael I. Gordon et al. Exploiting coarse-grained task, data, and pipeline parallelism in stream programs. *ASPLOS*, 2006.

[5] Sander Stuijk, Marc Geilen, and Twan Basten. Throughput-buffering trade-off exploration for cyclo-static and synchronous dataflow graphs. *IEEE Trans. Comput.*, 57(10):1331–1345, 2008.

[6] Alberto Sangiovanni-Vincentelli et al. Benefits and challenges for platform-based design. *DAC*, pages 409–414, 2004.

[7] Dean 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.

[8] Shane Bell et al. TILE64 processor: A 64-core SoC with mesh interconnect. *ISSCC*, 2008.

[9] Edward A. Lee and David G. Messerschmitt. Synchronous data flow. *Proceedings of the IEEE*, 75(9):1235–1245, 1987.

[10] Marc Geilen and Twan Basten. Reactive process networks. In *EMSOFT*, pages 137–146, 2004.

[11] Sander Stuijk, Marc Geilen, and Twan Basten. Exploring trade-offs in buffer requirements and throughput constraints for synchronous dataflow graphs. In *DAC*, pages 899–904, 2006.

[12] Jean-Louis Colaço, Alain Girault, Grégoire Hamon, and Marc Pouzet. Towards a higher-order synchronous data-flow language. In *EMSOFT*, pages 230–239, 2004.

[13] Walid Taha. A gentle introduction to multi-stage programming. In *Domain-Specific Program Generation*, pages 30–50, 2003.

[14] J. Adam Cataldo. *The Power of Higher-Order Composition Languages in System Design*. PhD thesis, University of California, Berkeley, 2006.