System-Level Performance Estimation for Application-Specific MPSoC Interconnect Synthesis

Po-Kuan Huang, Matin Hashemi, Soheil Ghiasi
Electrical and Computer Engineering
University of California, Davis, CA 95616, USA
{pohuang,hashemi,ghiasi}@ucdavis.edu

Abstract— **We present a framework for development of streaming applications as concurrent software modules running on multi-processors system-on-chips (MPSoC). We propose an iterative design space exploration mechanism to customize MPSoC architecture for given applications. Central to the exploration engine is our system-level performance estimation methodology, that both quickly and accurately determine quality of candidate architectures. We implemented a number of streaming applications on candidate architectures that were emulated on an FPGA. Hardware measurements show that our system-level performance estimation method incurs only 15% error in predicting application throughput. More importantly, it always correctly guides design space exploration by acheiving 100% fidelity in quality-ranking candidate architectures. Compared to behavioral simulation of compiled code, our system-level estimator runs more than 12 times faster, and requires 7 times less memory.**

I. Introduction

Advancements of technology and continuation of Moore's law have enabled integration of many processing elements on the same chip. Multi-processor system-on-chip (MPSoC) platforms exhibit substantial performance and energy improvements over conventional uni-processors, and yet, provide the flexibility of general-purpose computing systems [1]. Nevertheless, lack of efficient application-specific MPSoC architecture synthesis tools, have impeded proliferation of MPSoC architectures in embedded applications, where performance and energy consumption are the primary design concerns [2].

To address this issue, we present a library-based framework for MPSoC architecture synthesis in which, both processors and their interconnect architecture are composed using a number of available implementations [3], [4]. Our target system is a *soft* multi-processor in that its malleable architecture is implemented on FPGA-like programmable fabric. We aim at customizing the architecture to improve performance (throughput) of given streaming applications.

The synthesis process explores the design space by iterative generation of candidate architectures followed by system-level estimation of their performance. Consequently, accuracy and runtime of system-level performance estimation technique are instrumental to quality and practicality of the synthesis process.

We focus on system-level performance estimation during interconnect synthesis [5], [6], where a number of candidate interconnect architectures have to be compared. Examples of candidate architectures are point-to-point, bus-based or network-on-chip primitives with different parameters such as word width and buffer size. Our estimation methodology takes as input the specification of a candidate architecture, high-level application specification, potential task assignment and task schedule. It quickly generates accurate performance estimations without compiling high-level estimations.

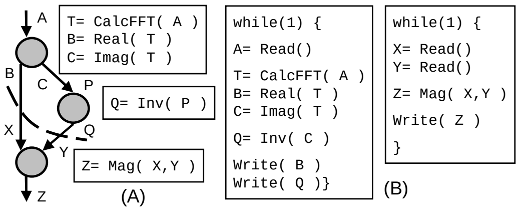
We implemented candidate designs on an FPGA, and mapped several streaming applications onto them. We measured throughput of applications running on hardware, and compared the results with system-level early estimations. Throughput estimations are only about 15% lower than actual measurements. More importantly, our estimation method demonstrated 100% fidelity in our experiments. That is, it always ranked candidate architectures correctly, which is the primary requirement in efficient exploration of design space. Our technique runs about 12 times faster than behavioral simulation while requiring 7 times less memory.

II. Background

Application Model: Many researchers have investigated appropriate abstractions for modeling of streaming applications that are meant to be implemented as concurrent software modules. While the purpose of this paper is not delving into model of computation and programming languages research, we briefly discuss several outstanding choices for modeling streaming applications.

It is a widely-accepted assertion that coarse grain parallelism-extraction is a very difficult problem and hence, sequential single-threaded languages, such as standard `C`, are not appropriate choices for modeling concurrent tasks. A number of leading experts believe that thread-based application development in general, is not a productive and reliable method of developing concurrent software [7], [8]. A possible alternative would be to represent the applications in an *actor-oriented* model, where coarse grain parallelism is explicit, and tasks co-exist and communicate with one another using a governing protocol [9], [10].

We adopt the task graph application model that is commonly used in embedded systems community [11], [12]. Task graphs conform to the general notion of actor-oriented computing. In this model, applications are represented with a directed graphs whose nodes represent tasks. Edges of the graph represent inter-task data or control dependencies. Tasks



```
T= CalcFFT( A )
B= Real( T )
C= Imag( T )
```

```
Q= Inv( P )
```

```
Z= Mag( X,Y )
```

```
while(1) {
A= Read()
T= CalcFFT( A )
B= Real( T )
C= Imag( T )
Q= Inv( C )
Write( B )
Write( Q )}
```

```
while(1) {
X= Read()
Y= Read()
Z= Mag( X,Y )
Write( Z )
}
```

Fig. 1. A) An example task graph. B) Generated software when the top two nodes are mapped to one processor and the bottom node to another.

are atomic in that their internal computation is specified in sequential semantics, in C or higher-level languages, that cannot be automatically parallelized. Our loosely-defined notion of task graph is very similar to Kahn process networks [13] in which, tasks communicate asynchronously using unidirectional FIFO links, and receiving tasks block on empty input links.

Task graphs and Kahn process networks have a dataflow nature, which fits streaming applications very well. Several other dataflow-based models such synchronous dataflow (SDF) [14], [15] can be represented with task graphs. In case of SDF, for example, a node in the task graph would correspond to its associated node in SDF repeated by the number of node's appearances in SDF static schedule.

Target Architecture: We aim at implementing a given application as software modules running on processors of a MPSoC platform. To depart from the problems associated with shared states among threads, and to move closer to a robust actor-oriented implementation [7], we focus on processors that work with isolated memory spaces. Therefore, synthesized software processes would need to directly send and receive messages to synchronize.

The target multi-processor architecture is soft in that it is going to be built out of configurable logic, plus its architecture specifics can be customized to better serve the application. For example, the number of processors, processors' functional units or memory size, and interconnect architecture are among the specifics that have to be determined during synthesis. We refer to such architectures as *soft multi-processor*.

Our synthesis flow aims to generate an optimized soft multi-processor for a given (possibly set of) streaming application(s) under the area constraint of the configurable substrate. Quick and accurate performance estimation for candidate architectures at system-level, is of prime significance to quality and practicality of soft multi-processor synthesis. This process will be further discussed in Section III.

Application Implementation: To implement a given application on a given multi-processor architecture, application tasks have to be assigned to processing resources. Subsequently, tasks assigned to the same processor have to be scheduled to share the processor in time. Specifically, given a set of tasks assigned to processor p and having a specific schedule in mind, we synthesize the software running on p by combining computations of all such tasks according to the schedule. Inter-processor communication is implemented by calling appropriate communication libraries that realize necessary `read` and `write` primitives.

Figure 1 shows an example, in which the target architecture is composed of two processors connected using a uni-directional FIFO channel. Let us assume that the top two nodes (tasks) of the task graph are assigned to the first processor and the bottom node to the second processor. Let us examine the generated code for processor 1. We first add a `read` function to read the input data and initialize A. Then we insert code to perform the computation of `CaclFFT` node followed by `Inv` node. Finally `write` functions are inserted in the code to output the results B and Q to the second processor.

Notice that we implicitly assume that a known schedule dictates the ordering of the tasks assigned to a processor. The schedule ensure that tasks are combined considering their dependencies. For example, `Inv` node has to come only after `CalcFFT` node because it has data dependency to `CalcFFT` node. In this paper, we work with dataflow-based streaming applications that can be scheduled statically.

III. Soft Multi-Processor Synthesis

Our ultimate goal is to efficiently realize a given streaming application as concurrent software modules running on a customized multi-processor. Two main challenges have to be addressed to achieve this goal. Firstly, the architecture of soft multi-processor has to be synthesized such that it better handles the application. Secondly, the application has to be compiled into software modules running on processors.

Figure 2 illustrates a high-level view of soft multi-processor synthesis framework. The process is divided into two architecture customization and application compilation thrusts, where architecture customization naturally precedes compilation. This section discusses these two procedures, and highlights the significance of high-level performance and area estimation in the process.

A. Processor and Interconnect Architecture Customization

The goal of architecture customization is to determine the appropriate configuration for processors and their intercon-

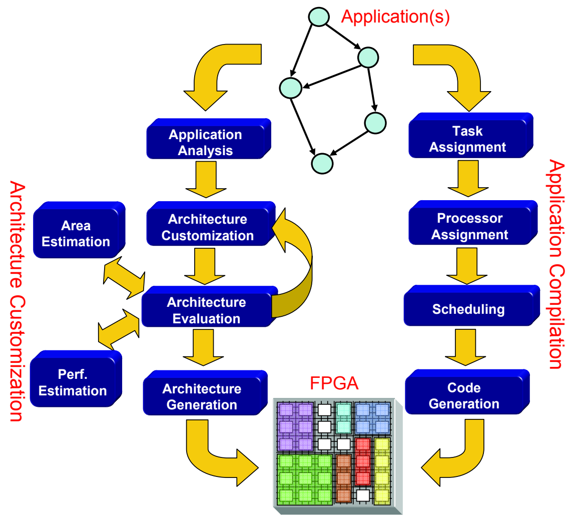


Fig. 2. Architecture and application design flow for customized soft multi-processor.

nect. This problem, a special case of the well-known high-level synthesis problem, can be very difficult in the general case: The synthesis engine has to adjust many interacting *knobs*, such as processors architecture, instruction set, functional units and memory hierarchy. As a result, efficient design space exploration is prohibitive.

We focus on a constrained, yet interesting, version of this problem by resorting to a library-based approach in which, processors, their functional units and interconnect primitives have to be selected from a number of choices available in the library. this limits the degree of freedom on some of customization knobs. Specifically, we assume that processors configuration is already determined, and we focus on interconnect synthesis.

Interconnect architecture is synthesized using point-to-point, bus or NoC interconnect primitives that exist in the library. Architecture parameters of interconnect primitives, such as buffer size of router switches or FIFO channels, are among the controllable knobs. Note that interconnect and processor are generally competing over the silicon area. For example, memory elements on programmable logic can be allocated as either processor cache or interconnect buffers.

The left part of the flow in Figure 2 illustrates architecture customization process. Customization is performed by iterative identification of candidate architectures, followed by their quality and feasibility evaluation using system-level performance and area estimation techniques. Quality and practicality of architecture customization directly depend on accuracy and runtime of estimation methods, respectively.

Estimation methods have to function at the system-level to run fast, which is a requirement for their integration in the iterative candidate architecture evaluation process. On the other hand, they have to be accurate-enough to correctly guide the customization engine. In Section IV, we discuss our approach toward development of such estimation methodologies.

In this paper, we only focus on interconnect synthesis problem. That is, we assume that the number of required processors and their configuration are already determined. The interconnect synthesis stage does not have control over processors configuration, although, it might provide feedback and re-invoke processor configuration if the best interconnect does not meet application performance requirement.

B. Application Compilation

Once the target architecture is determined, the application has to be compiled onto the processors to generate the executable code for each processor (right part of the flow in Figure 2). Recall that applications are specified using task graphs. To compile the application, first, constituting tasks are assigned to virtual processors, and then, virtual processors are assigned to physical processors. Subsequently, tasks' computation and communication are scheduled, and executable code for each processor is generated accordingly. For our target application domain (streaming), computation and communication can be statically scheduled. The compilation pass is out of the scope of this paper, but interested readers may refer to our recent paper on the topic [16].

IV. System-Level Estimation

This section overviews our methodology in estimating performance of a candidate interconnect architecture at system-level. The input to our performance estimation is a candidate architecture, including both processor and interconnect configuration, and associated task assignment and schedule of the application. The objective is to estimate throughput, without compiling the application.

Performance Model Reduction: In order to accurately estimate the performance of an application, candidate multiprocessor architecture has to be simulated using parallel software modules. The simulation requires cycle-accurate monitoring of instruction execution at each processor, complemented by cycle-accurate simulation of interconnect architecture. Although such simulation would result in accurate performance estimations, it is not fast enough for utilization in iterative evaluation of candidate architectures.

We introduce a simplified workload and communication characterization method that eliminates the need to cycle-accurately simulate instructions execution and data communication. Our technique preserves coarse-grain temporal behavior of the application, and is both fast and accurate. Therefore, it can be utilized in iterative design space exploration framework.

Our approach is to reduce the complexity of performance model by coarsening temporal behavior of concurrent software modules. We view a software module (running on one of the processors) as a series of computation and communication phases, and hence, we attempt to characterize latency and volume of computation and communication phases at system-level. Subsequently, a substantially-reduced verilog model is generated for the system in which, characterized temporal phases replace software module instruction sequence. Simulating the reduced model is considerably faster, while giving reasonable performance accuracy.

Figure 3 visualizes an example of our model simplification. In the figure, a candidate interconnect architecture is being evaluated under a given task assignment and schedule. To perform behavioral simulation, the compilation path should be followed, which generates code similar to the snippet shown on Figure 3.a for each processor. Our technique views the processors as traffic generator/consumer with known delay elements, and simplifies temporal behavior of the code based on estimations (Figure 3.b). Details of our model reduction is the following.

Task Workload Characterization: The first step to estimate system performance, is to characterize the workload intensity associated with each task. Although compilers can perform various inter-task optimizations after several tasks are assigned to the same processor, it is reasonable to ignore such potential optimizations at system-level. Therefore, we analyze tasks in isolation and associate a deterministic number to each task that represents its workload latency on its allocated processor.

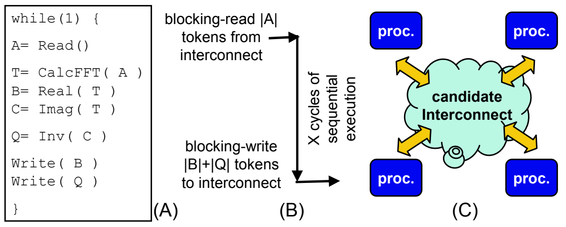

```
while(1) {
A= Read()
T= CalcFFT( A )
B= Real( T )
C= Imag( T )
Q= Inv( C )
Write( B )
Write( Q )
}
```
(A)
blocking-read |A| tokens from interconnect
X cycles of sequential execution
blocking-write |B|+|Q| tokens to interconnect
(B)
proc.
proc.
candidate Interconnect
proc.
proc.
(C)

Fig. 3. A) Sample code of a processor, B) and its simplified performance model for quick evaluation of C) candidate interconnect architectures.

Specifically, we profile processors in the library to estimate their cycle per instruction (CPI) distribution. We also analyze the impact of functional units available in the library on processor's CPI. We make the observation that from a system-level perspective, it is reasonable to assume that only instructions directly benefiting from the functional unit are affected. For example, we assume that only floating point operations are affected by inclusion (exclusion) of a dedicated floating point unit to (from) a soft processor.

Following library characterization, tasks internal computations are analyzed at high-level, without compiling to assembly, to derive a rough mapping between high-level language constructs and processor instructions. Furthermore, we use first order control-flow estimation such as average if-then-else path latencies, and expected number of loop iterations, to account for application control-flow behavior. Note that streaming applications are mostly data-flow intensive and their control-flow characteristics are minimal. This one-time analysis derives the estimated number of clock cycles needed for execution of a task on its allocated processor.

Similarly, inter-task communication cost is estimated from high-level application specification in which, parallelism is explicit. Given a specific task assignment and for a particular task, the number of data tokens that might have to be transmitted to other processors is readily calculated by checking the processors to which immediate descendant of the task are allocated. For applications modeled as synchronous dataflow graphs, each node appears a specific number of times in the steady state schedule. The number of appearances and data production and consumption rates are known statically, which enable quick one-time characterization of tasks data communication volume.

Processor Workload Characterization: Tasks assigned to the same processor are scheduled to generate the corresponding executable code (Section II). Scheduling impacts temporal behavior of code running on the processor, because temporal motion of inter-processor read/write operations might create or eliminate blockings on communication channels. Therefore, parallel application performance directly depends on task schedules, among other factors.

At system-level, the workload associated to a processor can be estimated to be a straight-forward combination of workloads of tasks that are assigned to that processor, according to the given schedule. Note that this estimation incurs inaccuracies because combined workload of several tasks (in terms of cycles) is not necessarily equal to sum total of tasks workloads that are characterized in isolation. Compiler optimizations, such as enhanced register allocation to reduce register spilling, typically improve upon sum total of tasks workloads. However, "sum total" estimation is a good approximation at system-level.

Consequently, we simplify temporal behavior of a processor workload to a sequence of computation, with estimated latency, and communication, with estimated volume. This simplification is used to generate a temporal behavior model for each interconnect port, which replaces processor and its executable binary in a cycle-accurate simulation. As mentioned before in Figure 2, we use this simplified model as part of the iterative approach.

V. Evaluation and Discussion

Experiments Setup: We utilized Xilinx Embedded Development Kit (EDK) to develop the aforementioned soft multiprocessor synthesis framework. Xilinx EDK provides a library of soft processors and interconnect primitives. Xilinx 32-bit soft processor, called Microblaze, can be customized in a number of ways. We characterized Microblaze processor to determine its clock per instruction (CPI) distribution.

Microblazes can be interconnected using Xilinx Fast Simplex Links (FSL), which essentially implement point-to-point FIFO channels with configurable buffer size and width. We also implemented a standard network-on-chip router element with 5 bi-directional ports. Typically, one port connects to a processor and the remaining four ports are connected to neighboring routers. Hence, routers can be cascaded to interconnect an arbitrary-sized mesh of processors.

Figure 4 illustrates the flow of our experiments. We utilize MIT StreamIt [17] compiler to analyze and implement high-level application specifications. StreamIt is a programming language whose semantics are closely related to synchronous dataflow model of computation [14] with a few enhancement. Specifically, standard SDF model is enhanced to allow application initialization phase and utilization of limited control flow in program specification. In addition, StreamIt provides an open-source compilation framework for stream programs specified in its language. StreamIt compiler takes as input an application specified in enhanced synchronous dataflow (SDF) semantics with StreamIt syntax, and after partitioning of the task graph, generates parallel `C` codes for parallel

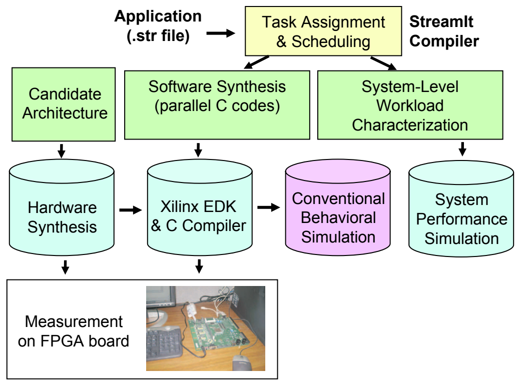

Application (.str file)
Task Assignment & Scheduling
StreamIt Compiler
Candidate Architecture
Software Synthesis (parallel C codes)
System-Level Workload Characterization
Hardware Synthesis
Xilinx EDK & C Compiler
Conventional Behavioral Simulation
System Performance Simulation
Measurement on FPGA board

Fig. 4. Experiments flow

processors. Parallel codes should be compiled for the target uni-processor to generate executable binary.

We implemented our task and processor workload characterization method in StreamIt before generating C codes. Processor workloads are estimated using high-level specification of the applications in StreamIt language and Microblaze CPI distribution, as discussed in Section IV. In addition, StreamIt candidate task assignment and schedule are used during processors workload characterization.

The characterized workload is spelled out as a simple verilog code that breaks up estimated temporal behavior of the processors into computation, with known latency, and communication, with known volume, phases. This models is integrated with the interconnect architecture model, and simulated using Modelsim simulator. To compare our reduced model with exact simulation, we generated parallel C codes using StreamIt and compiled them for processors. The executable binary is plugged into behavioral model of the processors, integrated with the interconnect architecture model, and simulated using Modelsim.

In order to evaluate the accuracy of our results, we implemented candidate architectures on Xilinx ML310 board that has a Virtex II Pro FPGA. StreamIt and Microblaze C compiler (mb-gcc) were used to generate executable binaries from high-level application task graph. Application throughput measurements from operating hardware are used as baseline for estimation accuracy.

Performance and Area Estimation Results: We designed ten different candidate architectures, which are summarized in Figure 5. Candidate architectures contain different number of Microblaze processors that are interconnected using either packet-switched on-chip network or point-to-point FIFO channels (FSL). The buffer size in both packet-switching routers and FIFO channels are configured to explore different design points.

We selected 5 representative streaming applications from StreamIt benchmarks: Bitonic Sort, FFT, Filter Bank, Blocked Matrix Multiplication, and Time Devision Equalization. We estimated system throughput for all applications on the aforementioned 10 candidate architectures. Candidate architectures were also implemented on FPGA board and applications were mapped to measure performance.

Figure 6 illustrates the average value of throughput measured on emulated candidate architectures. The last entry on X-axis shows the geometric mean value for all applications. The left vertical axis in the figure compares the runtime of our performance estimation technique with that of behavioral simulation of the system. In both cases, simulations were run to generate 100 output tokens.

Behavioral simulation requires about 150-450 seconds per iteration. Slow runtime makes it impractical to utilize behavioral simulation in iterative system-level design space exploration. System-level estimation runs about 10-12 times faster than behavioral simulation, because it eliminates the overhead of functional simulation of software instructions. Our method improves memory usage and simulation loading time of behavioral simulation more than 7 times. Note that behavioral simulation does not include intra-processor components, memory controller logic, etc. and is the simplest and fastest possible functional simulation.

Arch. name	# of processors	Interconnect arch.	Buffer Size
3x2-2	6	$3x2$ packet-switch	2
3x2-4	6	$3x2$ packet-switch	4
3x2-8	6	$3x2$ packet-switch	8
2x2-2	4	$2x2$ packet-switch	2
2x2-4	4	$2x2$ packet-switch	4
2x2-8	4	$2x2$ packet-switch	8
4-FSL-16	4	FSL cascade	16
4-FSL-32	4	FSL cascade	32
2-FSL-16	2	FSL cascade	16
4-FSL-32	2	FSL cascade	32

Fig. 5. Candidate interconnect architectures and configurations

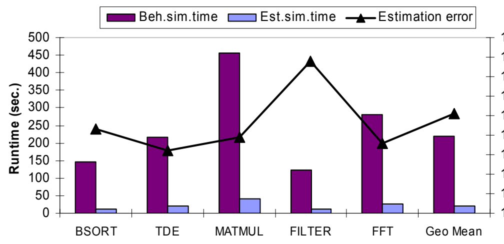


Fig. 6. Performance estimation accuracy and runtime comparison.

As shown on the right vertical axis of Figure 6, the error in system-level performance estimation is around 15%. This error is calculated as the relative difference between the actual throughput measured on FGPA, and estimated throughput using our model. The estimated throughput is consistenly lower than actual measurements, which suggests that our workload estimation is slightly pessimistic. Nevertheless as we show in next subsection, it can reliably guide design space exploration process.

Inaccuracy in throughput estimation is primarily due to two reasons. First, during high-level workload characterization a *rough* mapping between task specification and processor instructions are determined, which naturally is not perfectly accurate. Second, some compiler optimizations, such as improved register allocation, loop overhead reduction and memory access optimization, are enabled by combining multiple tasks that run on the same processor. At system-level, it is hard and runtime-expensive to consider such optimizations.

Quality-Ranking of Candidate Architectures: The primary function of system-level performance estimator, in an iterative design space exploration setting, is to correctly *rank* candidate architectures according to their quality. High-fidelity quality-ranking of candidate architectures correctly guides candidate architecture generation and selection, and ultimately, leads to success of the synthesis flow. High-fidelity quality-ranking is probably more useful than increasing absolute estimation accuracy.

To demonstrate fidelity of our estimation in quality ranking candidate architectures, we ranked all ten architectures according to their performance for a specific application. Figure 7 shows the data normalized to a single-processor performance. For each chart, the architectures on X-axis are

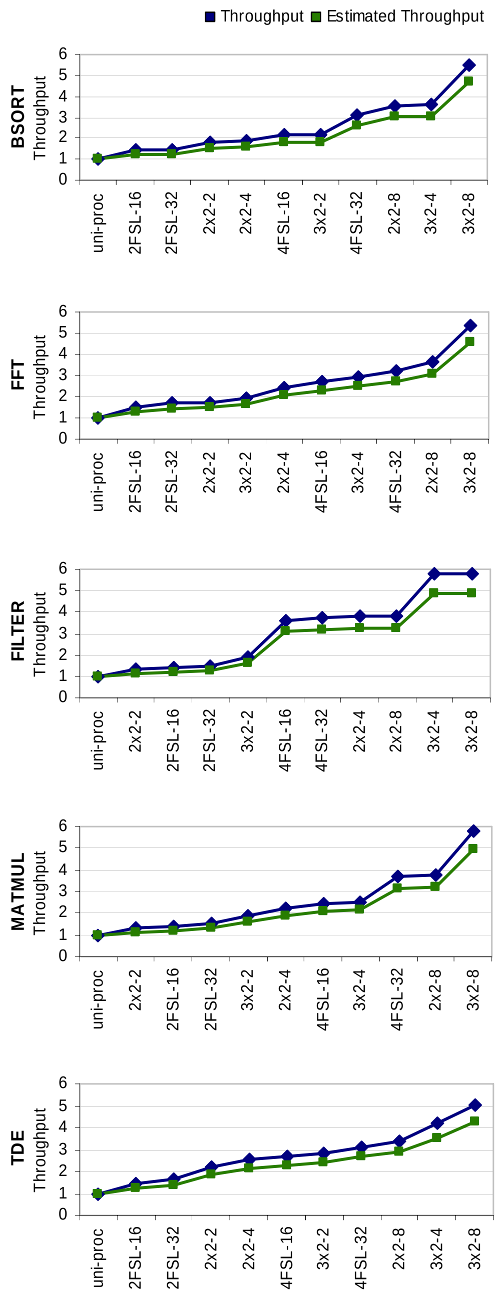


Fig. 7. Quality-ranking of candidate architectures for different applications. Estimation always correctly ranks candidate architectures.

reordered to have an ascending curve of "measured performance". Therefore, the ordering of architectures implies their ranking based on application performance. Then, "estimated performance" points are added to each chart, and connected to visualize their trend.

Interestingly, the estimated performance curve is also ascending in all cases, which implies that our estimation method correctly ranks candidate architectures according to their performance. For example, 2x2-8 configuration is estimated to have better throughput for all applications compared to 4FSL-32 configuration, which is verified by performance measurements. Fast runtimes and high-fidelity quality ranking of architectures enable integration of our system-level performance estimation in our iterative MPSoC customization and synthesis framework.

VI. Conclusions

We presented a MPSoC design exploration framework for efficient synthesis of streaming applications on customized soft multi-processors. We developed a fast and accurate system-level performance estimation technique that is very successful at guiding design space exploration engine by high-fidelity quality-ranking of candidate architectures. Extensive experiments including both simulation and emulation advocate the effectiveness of our approach. Future work will focus on accelerated design space exploration guided by enhanced application analysis and improved candidate architecture generation methods.

References

[1] Shekhar Borkar, Norman P. Jouppi, and Per Stenstrom. Microprocessors in the era of terascale integration. *DATE*, pages 237–242, 2007.

[2] Federico Angiolini et al. An integrated open framework for heterogeneous MPSoC design space exploration. *DATE*, pages 1145–1150, 2006.

[3] Nitin Deo et al. What happened to ASIC? go (recon)figure? *DAC*, 2004.

[4] Chris Rowen. *Engineering the Complex SOC: Fast, Flexible Design with Configurable Processors*. Prentice Hall, 2004.

[5] Alessandro Pinto, Luca P. Carloni, and Alberto L. Sangiovanni-Vincentelli. Constraint-driven communication synthesis. *DAC*, pages 783–788, 2002.

[6] Ilya Issenin and Nikil Dutt. Data reuse driven energy-aware MPSoC co-synthesis of memory and communication architecture for streaming applications. *CODES+ISSS*, pages 294–299, 2006.

[7] Edward A. Lee. The problem with threads. *IEEE Computer*, 39(5):33–42, 2006.

[8] Michael I. Gordon et al. A stream compiler for communication-exposed architectures. *ASPLOS*, 2002.

[9] Gul Agha. *Actors: A Model of Concurrent Computation in Distributed Systems*. PhD thesis, Massachusetts Institute of Technology, 1985.

[10] Gang Zhou, Man-Kit Leung, and Edward A. Lee. A code generation framework for actor-oriented models with partial evaluation. *International Conference on Embedded Software and Systems*, 2007.

[11] Ravindra Jejurikar and Rajesh Gupta. Optimized slowdown in real-time task systems. *IEEE Transactions on Computers*, 55(12):1588–1598, 2006.

[12] Keith S. Vallerio and Niraj K. Jha. Task graph extraction for embedded system synthesis. *VLSI Design*, 2003.

[13] Gilles Kahn and David B. MacQueen. Coroutines and networks of parallel processes. *IFIP*, pages 993–998, 1977.

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

[15] Shuvra S. Bhattacharyya, Praveen K. Murthy, and Edward A. Lee. Synthesis of embedded software from synchronous dataflow specifications. *Journal of VLSI Signal Processing Systems*, 21(2):151–166, 1999.

[16] Matin Hashemi and Soheil Ghiasi. Exact and approximate task assignment algorithms for pipelined software synthesis. *DATE*, 2008.

[17] William Thies, Michal Karczmarek, and Saman Amarasinghe. StreamIt: A language for streaming applications. *Proceedings of the International Conference on Compiler Construction*, 2002.