Task-Level Data Model for Hardware Synthesis Based on Concurrent Collection

Mr.J.Kotaiah
Asst.Professor,EEE,SBIT
Khammam,TS,India
jupellikotaiah@gmail.com

Mrs.T.Samatha
Asst.Professor,EEE,SBIT
Khammam,TS,India
samsai1791@gmail.com

Mrs.P.Gayatridevi
Asst.Professor,EEE,SBIT
Khammam,TS,India
gaya3dava@gmail.com

Abstract

Growing complexity in the design of today's digital systems necessitates the creation of electronic system-level (ESL) approaches that include automation and optimization at a higher level of abstraction. ESL design frameworks heavily depend on the specifics of the application's concept of concurrency. When it comes to describing task-level concurrent behaviour in the hardware synthesis design cycle, state-of-the-art concurrent specification models fall short. We present a task-level data model (TLDM) for hardware synthesis of data-processing applications, based on the concurrent collection (CnC) paradigm, which allows for maximal task rescheduling flexibility. In order to represent task instances, array accesses, and dependencies succinctly, TLDM incorporates polyhedral models. We demonstrate the benefits of our TLDM definition over other popular concurrency specifications using examples.

Introduction

Increasing the degree of design abstraction to the ESL (electronic system level) is motivated by the growing complexity of electronic systems. Concurrently exploiting and managing a large number of parallel jobs and design components is a major problem in ESL design and optimization. As the first step in most ESL optimization flows, task-level concurrency requirements are crucial to the overall quality of results (QoR) of the implemented solution. Task-specific behaviour is encapsulated by the concurrency definition, whereas coarse-grained parallelism and task-to-task interactions are explicitly specified. The application's system-level data may be used for direct implementation and optimization. There are a variety of ESL approaches that have been presented. For an in-depth look at contemporary approaches to ESL workflow design, we recommend the works cited in [1, 2]. The automation of the ESL design process is driven by high-level synthesis (HLS). These days' HLS tools may build register transaction-level (RTL) hardware specifications that are very near to hand-generated designs [3] for synthesizing computation-intensive modules into hardware accelerators with bus interfaces. Task-level optimizations, such as data transmission across accelerators, programmable cores, and memory hierarchies, might be challenging for the present HLS tools to manage. System needs various implementation details, such as explicitly stated port/module structures, whereas the sequential C/C++ programming language has inherent restrictions in specifying the task-level parallelism. Both languages have severe limitations on optimization processes and add extra work for

developers of algorithms and computer programs. The actual implementation of the automated ESL technique relies heavily on the availability of a concurrent definition model that is both tool- and designer-friendly. From the earliest days of computing to present day studies of parallel programming, the issue of concurrency definition has been studied extensively. From an ESL perspective, some findings were too broad and incurred significant implementation costs for broad hardware architectures [4, 5], while other results were too narrow and could simulate just a select few applications [6, 7]. In addition, most of the earlier models primarily concerned themselves with describing the behaviour or computation, but unintentionally created superfluous requirements for implementation, including the exact execution order of iterative task instances. CNBC [8] suggested the idea of divorcing the algorithm definition from the impel mentation optimization, which allows for more leeway in scheduling tasks and a broader design space, both of which might lead to improved implementation quality of result (QoR). However, CnC was first developed for systems based on multicore processors, which need a number of dynamic syntactic features (such as dynamic task instance creation, dynamic data allocation, and unbounded array index) in the specification. For ESL techniques to automatically optimize the design's QoR, a concurrent definition that accounts for both the accuracy of behavior and possibilities for optimization is required.

Models of Concurrent Specification

Every model of concurrent specification has its own model of computing (MoC) [10], and there are many different models at the task level. Models with well-defined MoCs are one category. Extensions of sequential programming languages (such as System [11]) or hardware description languages (such as Blue spec System Verilog [12]) that do not provide exact or explicit specifications of the underlying MoC form a second type. All MoCs may be specified in these languages at a variety of abstraction levels. The precise languages utilized to textually represent these MoCs will be overlooked in this section as attention is instead directed at the underlying MoCs in the concurrent specification models. The underlying MoC [10] determines the analysability and expressibility of the concurrent specification model. Task concurrency and implementation limitations for applications are two areas that are defined in detail by the various MoCs. Each MoC's efficient synthesizer and optimizer is custom-made according to the MoC's unique set of features. The available optimizations and the final implementation outcomes are also heavily impacted by the MoC selection. The following are the most important factors to think about while choosing a MoC. (i) Application scope refers to the variety of use cases that may be modeled successfully or effectively by the MoC. (ii) Usability, or how simple it is for a designer to utilize the MoC to define an application. (iii) Whether or not it's conducive to automated optimization: although a very generic MoC may be able to represent a wide variety of applications with little edits from the user, it may be exceedingly challenging to create effective automatic synthesis processes for such models. (iv) Compatibility with the target platform; for instance, an FPGA platform may not be optimal for synthesizing a MoC that expects a shared memory architecture (like CnC [8]) since such a platform may lack support for an efficient shared memory system. We present certain features that we think are necessary for a MoC to be considered for automated synthesis, despite the fact that most of them seem to be very subjective. (i) Deterministic execution: the MoC should ensure that, for a given input, execution occurs in a deterministic way, unless the application domain/system being represented is nondeterministic. Because of this, the designer (and ESL tools) will have an easier time ensuring that their generated implementations are valid. (ii) Organizing by hierarchy: apps often consist of smaller tasks that may be designed and implemented by separate users or groups of developers. The MoC need to be robust enough to simulate such applications in a tree structure. Due of the vast design area, it would be challenging to work with a MoC that only supports a flat specification. (iii) Heterogeneous target platform support and refinement: current SoC platforms have several different types of hardware, including general-purpose processor cores, custom hardware accelerators (implemented on ASICs or FPGAs), GPUs, memory blocks, and connective fabric. While it may be unrealistic to expect a single MoC definition to easily transfer to several platforms, the MoC should nonetheless give guidance for optimizing the application specification for the chosen platform. This further highlights the need of a hierarchical structure, since the refinements may be task-specific, as distinct subtasks may be better suited to different components (e.g., FPGA vs GPUs).

Analysing CnCs (Concurrent Collections)

The CnC [8] enables both tool-friendly and user-friendly con currency specifications by decoupling implementation details for implementation tuning specialists from specifics of behaviour for application domain experts. While the model-level information is included inside the task body, the iterations of iterative tasks are stated simply and succinctly. These notions may be applied to the design of hardware systems' behaviour at the task level, even though most of these concurrent specifications are intended for general-purpose multi-core platforms.

Fundamental Principles and Terms

An application's behaviour definition for hardware synthesis may be thought of as a mapping function, either logical or algorithmic, from input data to output data. In order to optimize performance, cost, and power consumption, hardware syn thesis maps the compute and storage in the behaviour specification to temporal (by scheduling) and spatial (by binding/allocation) implementation design spaces.

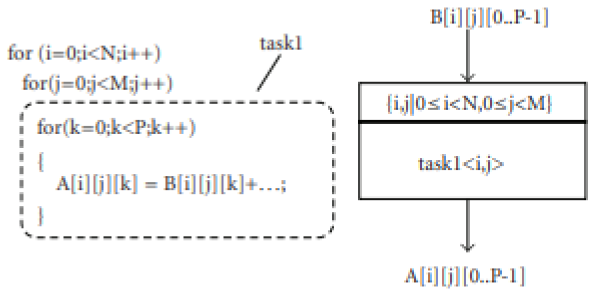


Figure 3: Concurrent task modelling.

The design space for optimization at a higher level is bigger, which might lead to better outcomes. To increase the optimization flow's abstraction level, CnC employs steps as the fundamental units for defining the system-level behaviour of a program. A step is a mapping function from the values of an input data set to the values of an output data set that is calculated statically and does not depend on the order in which the input and output data occur. In order to process the various data sets, it is necessary to repeat a step numerous times. A step instance is a definable instance of the step that is called into action by a control tag. A numeric vector associated with each control tag serves to uniquely identify this particular execution of the procedure. The set of all iterator vectors for a given step, or all instances of the step, is called the iteration domain. t or t:i,j represents the instance of step t indexed by the iterators (I, j). As can be seen in Figure 3, we wrap the loop k inside of task1, and task1 will run NM times based on the number of times through the loop that was specified by the I and j variables. Assignment1's iteration domain is [I, j] | 0i. If there is no data dependency, there is no need to declare an over constrained order between steps and step instances, in contrast to the explicit order of loops and loop iterations enforced by sequential programming languages. In the concurrent specification, an application is represented by a set of steps rather than a series of them, and each step is represented by a set of step instances rather than a sequence of them. In the context of communication between steps and step instances, a data item is often specified as a multidimensional array. Data tags are used to identify individual array members by their subscript vectors. The set of accessible subscript vectors in an array is determined by the data domain of the array. The notation (A0, A1, A2,) for the data tag of a multidimensional array A indicates its access reference. For instance, if A[I][j][k] is a matrix, then its subscript vector would be (I, j, k) if A0= I, A1= j, and A2= k. One or more of the array items will be accessed by each occurrence of the step. Data access is the mapping between the iterator vector of a step instance and the subscript vectors of the array items that the task instance reads from or writes to, and this mapping is decided statically. In Figure 3, the mapping B[B0][B1][B2] | B0=I, B1=j provides access to the input data for task1. Any data components sharing the same B0 and B1 but a different B2 will be accessible in the same task1 instance since the subscript B2 is not constrained. The collection of data items accessible by all instances of the step may be determined from the task's iteration domain and I/O accesses. The specification on data accesses includes the constraint of dynamic single assignment (DSA). DSA mandates a single write operation per data element over application lifetime. Memory reuse for multiple liveness-nonoverlap data is prohibited, and the DSA constraint limits array elements to a single data value. To ensure the execution model is intrinsically deterministic, CnC uses the DSA constraint in its specification to prevent conflicts caused by several threads trying to access the same array element at the same time. The constraints of execution priority between step instances are represented by dependency. The I/O access functions of the same data object in the two phases automatically specify the dependency if one step creates data that are utilized in the other step. For the sake of guiding the synthesizer in producing proper and efficient implementations, the term "dependence" may also be used to represent any kind of step instance precedence restrictions. For instance, task1 task1> is a shorthand for j 1, j2, task1 task1>, which specifies that the outermost loop i in Figure 3 must be scheduled consecutively. The set of precedence constraints for two steps may be derived from their iteration domains and dependency mapping (the two steps are the same for self-dependence).

Data Model for Specific Tasks

Here, we propose a paradigm for describing concurrency at the task level, drawing inspiration from the Intel CnC. For the purpose of task-level hardware system synthesis, we provide a set of modifications to Intel CnC. We begin with a brief summary of the similarities and distinctions between TLDM and CnC. Then, we construct our TLDM specification in C++, modelling the high-level data needed for scheduling tasks using classes and fields. While C++ classes are utilized as the in-memory representation in the automation flow, we also provide a text-based format that allows users to specify TLDM without having to resort to any special libraries or frameworks. There is semantic parity between these two ways of specification. Our TLDM standard is shown on a sample application. To further prove that our suggested TLDM is superior for hardware synthesis, we also provide a comprehensive comparison between it and the CnC standard.

An Introduction to TLDM 4.1.

The strengths of CnC are carried over into our tailored TLDM specification paradigm for hardware synthesis. The parallels and divergences between TLDM and CnC are laid forth in Table 1. Most of TLDM's syntax and semantics come straight from CnC. Task, data, iterator vector, and subscript vector correspond to the CnC values of step, data, control tag, and data tag, respectively. Set tasks, data, access, and dependencies in a class TLDMApplication; Example 1 To prevent the unpredictable behavior during task instantiation, TLDM eliminates the CnC syntax for the control item. While in TLDM the task instance is activated after all input data is available, in CnC a step instance is enabled once its control item is produced. The precedence relationship between task instances may be modeled by converting control items into data items. Data (input/output) accesses, iteration domains, and data domains are all statically and clearly declared in TLDM. In CnC's step functions, expressions like "tag generation" and "data access" imply a dependency on previous stages. TLDM offers syntax for defining task-level dependencies between individual instances. The TLDM standard does not explicitly need DSA limitation. If the DSA limitation is breached, the programmer must impose dependent requirements on task instances to guarantee data coherence and program determinism.

Specification of TLDM in C++

See Listing 1 for a breakdown of the components that make up a TLDM application: tasks, data, access, and dependencies. The task set provides a concise description of all tasks and their instances. The repetition of a function's execution with new input and output data is characterized as a single job. Iterator vectors are used to locate specific instances of a given job. Each element of the iterator vector represents a level of the iterator space, analogous to a loop around the main body of a job in C/C++. The iteration domain specifies the bounds of the iteration vector, with each element representing a unique instance of the task's execution. Each task instance's input and output accesses are defined as affine functions of the task instance's iterator vector. Tldm access class defines the access functionalities. Class parents and offspring define the order of operations. Our TLDM's ability to handle a task hierarchy allows for granularity in task selection inside our optimization loop. In order to discover which portions are crucial for a given design objective, a coarse-grained low complexity optimization flow may be useful, and fine-grained optimization can then further optimize the subtasks locally with a more precise local design target.

benefits of TLDM

For task-level hardware synthesis, it is important to have a concurrency specification model that (i) can model all applications in the domain, (ii) is easy enough to use by domain experts regardless of implementation details, (iii) can integrate diverse com putations (possibly in different languages or stages of refinement), and (iv) can generate high-quality results efficiently. This section provides examples to show how the proposed TLDM specification model is tailored to meet these needs.

Optimizations at the Task Level

The advantage of implicit parallelism is shared by both our TLDM model and CnC, from which it is developed. Data flow networks and other models, on the other hand, do not account for dynamic parallelism. Figure 8 is a straightforward illustration of parallelizable loops in sequential programming. Since processes in process networks are created sequentially, the only method to specify concurrency is to start new processes at the same time. In Figure 8, the user statically specifies the number of parallel jobs, which are represented as a number of processes in the network. However, there are a variety of possible target systems, therefore a universally applicable static value may not exist. In order to take use of a GPU's hundreds of processing units in parallel, the application must be divided into smaller and smaller chunks than it would need to be on a multicore CPU. The maximum parallelism between task instances is stated explicitly in our TLDM, and users just need to declare a collection of tasks together with the iterator domain and data accesses. Depending on the target platform, the runtime/synthesis tool will decide how many and how finely-grained segments may execute in parallel.

In addition, the collection-based design does not impose any unnecessary limits on the scheduling of tasks in terms of execution order. The KPN specification must explicitly describe the loop order in each job process, as seen in the data streaming application in Figure 9. It is feasible to reduce the number of steps required for synthesis and optimization.

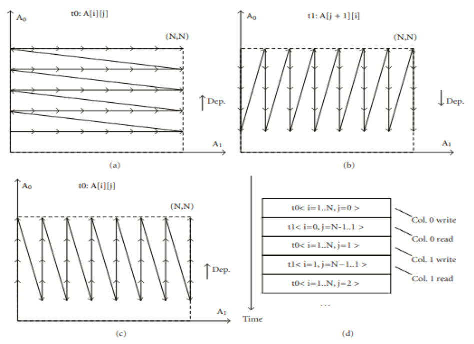
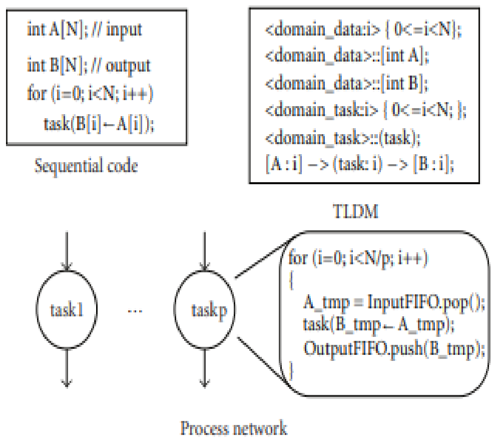


Figure 7: (a) Initial access order of t0 writes. (b) Initial access order of t1 reads. (c) Optimized access order of t0 writes. (d) Optimized task instance scheduling for buffer reduction.



```
int A[N]; // input
int B[N]; // output
for (i=0; i<N; i++)
    task(B[i]←A[i]);
```

Sequential code

```
<domain_data:i> { 0<=i<N};
<domain_data>::[int A];
<domain_data>::[int B];
<domain_task:i> { 0<=i<N; };
<domain_task>::(task);
[A : i] −> (task: i) −> [B : i];
```

TLDM

```
for (i=0; i<N/p; i++)
{
  A_tmp = InputFIFO.pop();
  task(B_tmp← A_tmp);
  OutputFIFO.push(B_tmp);
}
```

Figure 8: Implicit parallelism (TLDM)

opposed to the process network that makes explicit parallelism. simply switching around the sequence of the loop's iterations, the buffer size between jobs may be altered. However, the optimizer for a KPN-based flow must go deep into the process specification and interact with the module-level implementation, making these improvements more challenging. While sequential languages impose unnecessary order limitations, our TLDM definition defines all dependencies between tasks at the instance level. This allows optimizations to be made at the task level without requiring changes to the implementation at the module level, which is both possible and convenient.

Conclusion

Work For the purpose of hardware synthesis and optimization in data processing applications, this study presents a high-level concurrent specification. Our TLDM definition explicitly models dependency restrictions and models parallelism between task instances. TLDM seeks to express applications in a static and limited manner with minimum over constraints for concurrency, as opposed to the preceding concurrent specification. In order to standardize and unify the representation of iteration domains, data domains, access patterns, and dependencies, the TLDM specification incorporates a polyhedral model. Nonaffine term extensions are also carefully explored in the specification to facilitate the study and synthesis of non-standard behaviour. Our TLDM specification's advantages in representing task-level parallelism for hardware synthesis on heterogeneous platforms are shown via illustrative case studies. The hardware synthesis pipeline based on TLDM is presently under development.

References

[1] A. Gerstlauer, C. Haubelt, A. D. Pimentel, T. P. Stefanov, D. D. Gajski, and J. Teich, "Electronic system-level synthesis methodologies," IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, vol. 28, no. 10, pp. 1517– 1530, 2009.

[2] A. Sangiovanni-Vincentelli, "Quo vadis, SLD? Reasoning about the trends and challenges of system level design," Proceedings of the IEEE, vol. 95, no. 3, Article ID 4167779, pp. 467– 506, 2007. [3] "An independent evaluation of the AutoESL autopilot highlevel synthesis tool," Tech. Rep., Berkeley Design Technology, 2010.

[4] C. A. R. Hoare, "Communicating sequential processes. Commun," Communications of the ACM, vol. 21, no. 8, pp. 666– 677, 1978.

[5] E. A. Lee and T. M. Parks, "Dataflow process networks," Proceedings of the IEEE, vol. 83, no. 5, pp. 773–801, 1995.

[6] D. Harel, "Statecharts: a visual formalism for complex systems," Science of Computer Programming, vol. 8, no. 3, pp. 231– 274, 1987.

[7] E. A. Lee and D. G. Messerschmitt, "Synchronous data flow," Proceedings of the IEEE, vol. 75, no. 19, pp. 1235–1245, 1987.

[8] Intel—Concurrent Collections for C/C++: User's Guide, 2010, http://software.intel.com/file/30235.

[9] J. Cong, G. Reinman, A. Bui, and V. Sarkar, "Customizable domain-specific computing," IEEE Design and Test of Computers, vol. 28, no. 2, pp. 6–14, 2011.

[10] E. A. Lee and A. Sangiovanni-Vincentelli, "A framework for comparing models of computation," IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, vol. 17, no. 12, pp. 1217–1229, 1998.

[11] SystemC, http://www.accellera.org/.

[12] BlueSpec, http://bluespec.com/. [13] FDR2 User Manual, 2010, http://fsel.com/documentation/ fdr2/html/fdr2manual 5.html.

[14] ARC CSP model checking environment, 2010, http://cs.adelaide.edu.au/∼esser/arc.html.

[15] R. Allen, A formal approach to software architecture, Ph.D. thesis, Carnegie Mellon, School of Computer Science, 1997, Issued as CMU Technical Report CMU-CS-97-144.

[16] T. Murata, "Petri nets: properties, analysis and applications," Proceedings of the IEEE, vol. 77, no. 4, pp. 541–580, 1989. [17] Petri net, 2010, http://en.wikipedia.org/wiki/Petri net.

[18] A. Davare, D. Densmore, T. Meyerowitz et al., "A next-generation design framework for platform-based design," DVCon, 2007.

[19] J. Buck, S. Ha, E. A. Lee, and D. G. Messerschmitt, Readings in Hardware/Software Co-Design. Ptolemy: A Framework for Simulating and Prototyping Heterogeneous Systems, Kluwer Academic, Norwell, Mass, USA, 2002.

[20] G. Kahn, "The semantics of a simple language for parallel programming," in Proceedings of the IFIP Congress, J. L. Rosenfeld, Ed., North-Holland, Stockholm, Sweden, August 1974.

[21] H. Nikolov, M. Thompson, T. Stefanov et al., "Daedalus: toward composable multimedia MP-SoC design," in Proceedings of the 45th Design Automation Conference (DAC '08), pp. 574–579, ACM, New York, NY, USA, June 2008.