Hindawi Publishing Corporation
Journal of Electrical and Computer Engineering
Volume 2012, Article ID 691864, 24 pages
doi:10.1155/2012/691864

Research Article

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

Jason Cong, Karthik Gururaj, Peng Zhang, and Yi Zou

Computer Science Department, University of California, Los Angeles, Los Angeles, CA 90095, USA

Correspondence should be addressed to Karthik Gururaj, karthikg@cs.ucla.edu

Received 17 October 2011; Revised 30 December 2011; Accepted 11 January 2012

Academic Editor: Yuan Xie

The ever-increasing design complexity of modern digital systems makes it necessary to develop electronic system-level (ESL) methodologies with automation and optimization in the higher abstraction level. How the concurrency is modeled in the application specification plays a significant role in ESL design frameworks. The state-of-art concurrent specification models are not suitable for modeling task-level concurrent behavior for the hardware synthesis design flow. Based on the concurrent collection (CnC) model, which provides the maximum freedom of task rescheduling, we propose task-level data model (TLDM), targeted at the task-level optimization in hardware synthesis for data processing applications. Polyhedral models are embedded in TLDM for concise expression of task instances, array accesses, and dependencies. Examples are shown to illustrate the advantages of our TLDM specification compared to other widely used concurrency specifications.

1. Introduction

As electronic systems become increasingly complex, the motivation for raising the level of design abstraction to the electronic system level (ESL) increases. One of the biggest challenges in ESL design and optimization is that of efficiently exploiting and managing the concurrency of a large amount of parallel tasks and design components in the system. In most ESL methodologies, task-level concurrency specifications, as the starting point of the optimization flow, play a vital role in the final implementation quality of results (QoR). The concurrency specification encapsulates detailed behavior within the tasks, and explicitly specifies the coarse-grained parallelism and the communications between the tasks. System-level implementation and optimization can be performed directly from the system-level information of the application. Several ESL methodologies have been proposed previously. We refer the reader to [1, 2] for a comprehensive survey of the state-of-art ESL design flows.

High-level synthesis (HLS) is a driving force behind the ESL design automation. Modern HLS systems can generate register transaction-level (RTL) hardware specifications that come quite close to hand-generated designs [3] for synthesizing computation-intensive modules into a form of hardware accelerators with bus interfaces. At this time, it is not easy for the current HLS tools to handle task-level optimizations such as data transfer between accelerators, programmable cores, and memory hierarchies. The sequential C/C++ programming language has inherent limitations in specifying the task-level parallelism, while SystemC requires many implementation details, such as explicitly defined port/module structures. Both languages impose significant constraints on optimization flows and heavy burdens for algorithm/software designers. A tool-friendly and designer-friendly concurrency specification model is vital to the successful application of the automated ESL methodology in practical designs.

The topic of concurrency specification has been researched for several decades—from the very early days of computer science to today's research in parallel programming. From the ESL point of view, some of the results were too general and led to high implementation costs for general hardware structures [4, 5], and some results were too restricted and could only model a small set of applications [6, 7]. In addition, most of the previous models focused only on the description of the behavior or computation, but introduced redundant constraints unintentional for

implementation—such as the restrictive execution order of iterative task instances. CnC [8] proposed the concept of decoupling the algorithm specification with the implementation optimization; this provides the larger freedom of task scheduling optimization and a larger design space for potentially better implementation QoR. But CnC was originally designed for multicore processor-based platforms that contain a number of dynamic syntax elements in the specification—such as dynamic task instance generation, dynamic data allocation, and unbounded array index. A hardware-synthesis-oriented concurrency specification for both behavior correctness and optimization opportunities is needed by ESL methodologies to automatically optimize the design QoR.

In this paper we propose a task-level concurrency specification (TLDM) targeted at ESL hardware synthesis based on CnC. It has the following advantages.

(1) Allowing maximum degree of concurrency for task scheduling.

(2) Support for the integration of different module-level specifications.

(3) Support for mapping to heterogeneous computation platforms, including multicore CPUs, GPUs, and FPGAs (e.g., as described in [9]).

(4) Static and concise syntax for hardware synthesis.

The remainder of our paper is organized as follows. Section 2 briefly reviews previous work on concurrency specifications. Section 3 overviews the basic concepts in CnC. Section 4 presents the details of our TLDM specification. In Section 5 we illustrate the benefits of our TLDM specification with some concrete examples. Finally, we conclude our paper in Section 6 with a discussion of ongoing work.

2. Concurrent Specification Models

A variety of task-level concurrent specification models exist, and each concurrent specification model has its underlying model of computation (MoC) [10]. One class of these models is derived from precisely defined MoCs. Another class is derived from extensions of sequential programming languages (like SystemC [11]) or hardware description languages (like Bluespec System Verilog [12]) in which the underlying MoC has no precise or explicit definitions. These languages always have the ability to specify different MoCs at multiple abstraction levels. In this section we focus on the underlying MoCs in the concurrent specification models and ignore the specific languages that are used to textually express these MoCs.

The analyzability and expressibility of the concurrent specification model is determined by the underlying MoC [10]. Different MoCs define different aspects of task concurrency and implementation constraints for the applications. The intrinsic characteristics of each specific MoC are used to build the efficient synthesizer and optimizer for the MoC. The choice of the MoC greatly influences the applicable optimizations and the final implementation results as well. The key considerations in MoC selection are the following.

(i) *Application scope*: the range of applications that the MoC can model or efficiently model.

(ii) *Ease of use*: the effort required for a designer to specify the application using the MoC.

(iii) *Suitability for automated optimization*: while a highly generic MoC might be able to model a large class of applications with minimal user changes, it might be very difficult to develop efficient automatic synthesis flows for such models.

(iv) *Suitability for the target platform*: for example, a MoC, which implicitly assumes a shared memory architecture (such as CnC [8]), may not be well suited for synthesis onto an FPGA platform where support for an efficient shared memory system may not exist.

While most of these choices appear highly subjective, we list some characteristics that we believe are essential to an MoC under consideration for automated synthesis.

(i) *Deterministic execution*: unless the application domain/system being modeled is nondeterministic, the MoC should guarantee that, for a given input, execution proceeds in a deterministic manner. This makes it more convenient for the designer (and ESL tools) to verify correctness when generating different implementations.

(ii) *Hierarchy*: in general, applications are broken down into subtasks, and different users/teams could be involved in designing/implementing each subtask. The MoC should be powerful enough to model such applications in a hierarchical fashion. An MoC that supports only a flat specification would be difficult to work with because of the large design space available.

(iii) *Support of heterogeneous target platforms and refinement:* modern SoC platforms consist of a variety of components—general-purpose processor cores, custom hardware accelerators (implemented on ASICs or FPGAs), graphics processing units (GPUs), memory blocks, and interconnection fabric. While it may not be possible for a single MoC specification to efficiently map to different platforms, the MoC should provide directives to refine the application specification so that it can be mapped to the target platform. This also emphasizes the need for hierarchy because different subtasks might be suited to different components (e.g., FPGA versus GPUs); hence, the refinements could be specific to a subtask.

2.1. General Models of Computation. We start our review of previous work with the most general models of computation. These models impose minimum limitations in the specification and hence can be broadly applied to describe a large variety of applications.

Communicating sequential process (*CSP*) [4] allows the description of systems in terms of component processes that operate independently and interact with each other solely through message-passing communication. The relationships between different processes, and the way each process

communicates with its environment, are described using various process algebraic operators.

Hierarchy is supported in CSP where each individual process can itself be composed of subprocesses (whose interaction is modeled by the available operators). CSP allows processes to interact in a nondeterministic fashion with the environment; for example, the nondeterministic choice operator in a CSP specification allows a process to read a pair of events from the environment and decide its behavior based on the choice of one of the two events in a nondeterministic fashion.

One of the key applications of the CSP specification is the verification of large-scale parallel applications. It can be applied to detect deadlocks and livelocks between the concurrent processes. Examples of tools that use CSP to perform such verification include FDR2 [13] and ARC [14]. Verification is performed through a combination of CSP model refinement and CSP simulation. CSP is also used for software architecture description in a Wright ADL project [15] to check system consistency; this approach is similar to FDR2 and ARC.

Petri nets [16] consist of places that hold tokens (tokens represent input or output data) and transitions that describe the process of consuming and producing tokens (transitions are similar to processes in CSP). A transition is enabled when the number of tokens on each input arc is greater than or equal to the required number of input tokens. When multiple transitions are enabled at the same time, any one of them can fire in a nondeterministic way; also, a transition need not fire even if it is enabled. Extensions were proposed to the Petri net model to support hierarchy [17]. Petri nets are used for modeling distributed systems—the main aim being to determine whether a given system can reach any one of the user-specified erroneous states (starting from some initial state).

Event-Driven Model (EDM). The execution of concurrent processes in EDM is triggered by a series of events. The events could be generated by the environment (system inputs) or processes within the system. This is an extremely general model for specifying concurrent computation and can, in fact, represent many specific models of computation [18]. This general model can easily support hierarchical specification but cannot provide deterministic execution in general.

Metropolis [18] is a modeling and simulation environment for platform-based designs that uses the event-driven execution model for functionally describing application/computation. Implementation platform modeling is also provided by users as an input to Metropolis, from which the tool can perform synthesis, simulation, and design refinement.

Transaction-Level Models (TLMs). In [19] the authors define six kinds of TLMs; however, the common point of all TLMs is the separation of communication and computation. The kinds of TLMs differ in the level of detail specified for the different computation and communication components. The specification of computation/communication could be cycle-accurate, approximately timed or untimed, purely functional, or implementation specific (e.g., a bus for communication).

The main concern with using general models for hardware synthesis is that the models may not be suitable for analysis and optimization by the synthesis tool. This could lead to conservative implementations that may be inefficient.

2.2. Process Networks. A process network is the abstract model in most graphical programming environments, where the nodes of the graph can be viewed as processes that run concurrently and exchange data over the arcs of the graph. Processes and their interactions in process networks are much more constrained than those of CSP. Determinism is achieved by two restrictions: (1) each process is specified as a deterministic program, and (2) the quantity and the order of data transfers for each process are statically defined. Process networks are widely used to model the data flow characteristics in data-intensive applications, such as signal processing.

We address three representative process network MoCs (KPN, DPN, and SDF). They differ in the way that they specify the execution/firing and data transfer; this brings differences in determining the scheduling and communication channel size.

2.2.1. Kahn Process Network (KPN). KPN [20] defines a set of sequential processes communicating through unbounded first-in-first-out (FIFO) channels. Writes to FIFOs are nonblocking (since the FIFOs are unbounded), and reads from FIFOs are blocking—which means the process will be blocked when it reads an empty FIFO channel. Peeking into FIFOs is not allowed under the classic KPN model. Applications modeled as KPN are deterministic: for a given input sequence, the output sequence is independent of the timing of the individual processes.

Data communication is specified in the process program in terms of channel FIFO reads and writes. The access patterns of data transfers can, in general, be data dependent and dynamically determined in runtime. It is hard to statically analyze the access patterns and optimize the process scheduling based on them. A FIFO-based self-timed dynamic scheduling is always adopted in KPN-based design flows, where the timing of the process execution is determined by the FIFO status.

Daedulus [21] provides a rich framework for exploration and synthesis of MPSoC systems that use KPNs as a model of computation. In addition to downstream synthesis and exploration tools, Daedulus provides a tool called KPNGen [21], which takes as input a sequential C program consisting of affine loop nests and generates the KPN representation of the application.

2.2.2. Data Flow Process Network (DPN). The dataflow process network [5] is a special case of KPN. In dataflow process networks, each process consists of repeated "firings" of a dataflow "actor." An actor defines a (often functional)

quantum of computation. Actors are assumed to fire (execute atomically) when a certain finite number of input tokens are available on each input edge (arc). A firing is defined as consuming a certain number of input tokens and producing a certain number of output tokens. The firing condition for each actor and the tokens consumed/produced during the firing are specified by a set of firing rules that can be tested in a predefined order using only blocking read. The mechanism of actor firings helps to reduce the overhead of context switching in multicore platforms because the synchronization occurs only at the boundary of the firing, not within the firings. But compared to KPN, this benefit does not help much in hardware synthesis.

2.2.3. Synchronous Data Flow Graph (SDF). SDF [7] is a more restricted MoC than DPN, in which the number of tokens that can be consumed/produced by each firing of a node is fixed statically. The fixed data rate feature can be used to efficiently synthesize the cyclic scheduling of the firings at the compile time. Algorithms have been developed to statically schedule SDFs (such as [7]). An additional benefit is that for certain kinds of SDFs (satisfying a mathematical condition based on the number of tokens produced/consumed by each node), the maximum buffer size needed can be determined statically [7].

StreamIt [22] is a programming language and a compilation infrastructure that uses the SDF MoC for modeling real streaming applications. An overall implementation and optimization framework is built to map from SDF-based specifications of large streaming applications to various general-purpose architectures such as uniprocessors, multicore architectures, and clusters of workstations. Enhancements proposed to the classic SDF model include split and join nodes that model data parallelism of the actors in the implementations. In general, StreamIt has the same application expressiveness with SDF.

KPN, DPN, and SDF are suitable for modeling the concurrency in data processing applications, in which execution is determined by the data availability. The possibility of data streaming (pipelining) is intrinsically modeled by the FIFO-based communication. However, these models are not user-friendly enough as a preferred concurrency specification for data processing applications, because users need to change the original shared-memory-based coding style into a FIFO-based one. In addition, these models are not tool-friendly enough in the sense that (1) data reuse and buffer space reuse are relatively harder to perform in this overconstrained FIFO-based communication, (2) no concise expressions for iterative task instances (which perform identical behavior on different data sets) are embedded in one sequential process with a fixed and overconstrained execution order, and (3) it is hard to model access conflicts in shared resources, such as off-chip memory, which is an essential common problem in ESL design.

2.3. Parallel Finite State Machines (FSMs). FSMs are mainly used to describe applications/computations that are control-intensive. Extensions have been proposed to the classic FSM model to support concurrent and communicating FSMs, such as StateChart [6] and Codesign FSM (CFSM) [23]. Figure 1 shows that, in a hierarchical FSM, separate states and their transitions can be grouped into a hierarchical state D, and state D encapsulates the details of state B and C. Similarly, Figure 2 shows a concurrent FSM (Figure 2(a)) and one transition of its combined FSM (Figure 2(b)). A state of combined FSM is a combination of the states of the concurrent FSMs. The Koski synthesis flow [24] for ESL synthesis of MPSoC platforms uses the StateChart model for describing functionality of the system. All the state changes in the StateChart are assumed to be synchronous, which means, in the cases of transition from AX to BY, conditions p and q are validated simultaneously, and there is no intermediate state like AY or BX during the transition. However, in real systems the components being modeled as FSMs can make state changes at different times (asynchronously). CFSM breaks the limitation and introduces asynchronous communication between multiple FSMs. The Polis system [25] uses the CFSM model to represent applications for hardware-software codesign, synthesis, and simulation.

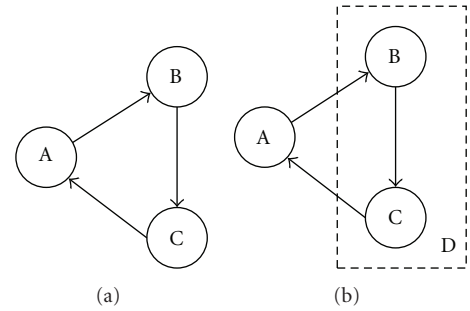

Figure 1: Hierarchical FSMs.

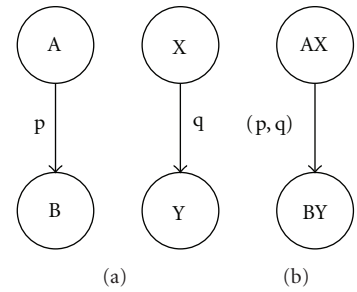

Figure 2: Parallel FSMs.

For a real application with both control and data flow, it is possible to embed the control branches into the module/actor specification of a data streaming model. For example, as mentioned in [26], additional tokens can be inserted into the SDF specification to represent global state and irregular (conditional) actor behavior. However, this would lose the opportunities to optimize the control path at system level, which may lead to a larger buffer size and more write operations to the buffer.

There is a series of hybrid specifications containing both FSM and data flow. *Finite state machine with datapath (FSMD)* models the synchronized data paths and FSM transitions at clock cycle level. *Synchronous piggyback dataflow networks* [26] introduce a global state table into the conventional SDF to support conditional execution. This model of computation was used by the PeaCE ESL synthesis framework [27], which, in turn, is an extension of the Ptolemy framework [19].

The *FunState* model [28] combines dynamic data flow graphs with a hierarchical and parallel FSM similar to StateCharts. Each transition in the FSM may be equivalent to the firing of a process (or function) in the network. The condition for a transition is a Boolean function of the number of tokens in the channels connecting the processes in the network. The System Co-Designer tool for mapping applications to MPSoC platforms uses the FunState model; the mapped implementation result is represented as a TLM (transaction-level model).

Program state machines (PSMs) [29] use the concept of hierarchical and concurrent FSMDs and extend it by replacing the cycle-based timing model with a general discrete timing model where the program can execute for arbitrary amount of time, and state transitions occur when the execution of the program is completed. In the SCE (system-on-chip environment) design flow [30], the SpecC [31] language specifies this particular model of computation. SpecC also provides primitives to specify FSM, parallelism, pipelining, and hierarchy of components. Using a target platform database (bus-based MPSoCs and custom IPs), the SCE flow generates TLMs of the target system, as well as hardware-software implementation for deployment.

2.4. Parallel Programming Languages. OpenMP (Open Multi-Processing) is an extension of C/C++ and Fortran languages to support multithread parallelism on a shared memory platform. A thread is a series of instructions executed consecutively. The OpenMP program starts execution from a master thread. The code segments that are to be run in parallel are marked with preprocessor directives (such as #pragma omp parallel). When the master thread comes to a parallel directive, it forks a specific number of slave threads. After the execution of the parallelized code, the slave threads join back the master thread. Both task parallelism and data parallelism can be specified in OpenMP. A global memory space is shared by all the threads, and synchronization mechanisms between threads are supported to avoid race conditions.

MPI is a set of APIs standardized for programmers to write portable message-passing programs in C and Fortran. Instead of preprocessing directives, MPI uses explicit API calling to start and stop the parallelization of the threads. Data transfers between parallelized threads are through message-passing using API function calls. Blocking access is supported in MPI to perform synchronization between threads.

Cilk [32] is another multithreaded language for parallel programming that proposes extensions to the C language for parallel processing. Cilk introduces the *spawn* construct to launch computations that can run in parallel with the thread that invokes spawn and the *sync* construct which makes the invoking thread wait for all the spawned computations to complete and return. The Cilk implementation also involves a runtime manager that decides how the computations generated by spawn operations are assigned to different threads.

Habanero-Java/C [33, 34] includes a set of task parallel programming constructs, in a form of the extensions to standard Java/C programs, to take advantage of today's multicore and heterogeneous architectures. Habanero-Java/C has two basic primitives: async and finish. The async statement, async <stmt>, causes the parent task to fork a new child task that executes <stmt> (<stmt> can be a signal statement or a basic block). Execution of the async statement returns immediately. The finish statement, finish <stmt>, performs a join operation that causes the parent task to execute <stmt> and then wait until all the tasks created within <stmt> have terminated (including transitively spawned tasks). Compared to previous languages (like Cilk), more flexible structures of task forking and joining are supported in Habanero-Java/C because the fork and join can happen in arbitrary function hierarchies. Habanero-Java/C also defines specific programming structures such as *phasers* for synchronization and *hierarchical place trees* (HPTs) for hardware placement locality.

However, all these parallel programming language extensions were originally designed for high-performance software development on multicore processors or distributed computers. They have some intrinsic obstacles when specifying a synthesized hardware system in ESL design: (1) general runtime routines performing task creation, synchronization, and dynamic scheduling are hard to implement in hardware; (2) the dynamic feature of task creation makes it hard to analyze the hardware resource utilization at synthesis time.

3. Review of Concurrent Collections (CnCs)

The CnC [8] was developed for the purpose of separating the implementation details for implementation tuning experts from the behavior details for application domain experts—which provides both tool-friendly and user-friendly concurrency specifications. The iterations of iterative tasks are defined explicitly and concisely, while the model-level details are encapsulated within the task body. Although most of these concurrent specifications target general-purpose multicore platforms, the concepts can also be used for the task-level behavior specification of hardware systems.

3.1. Basic Concepts and Definitions. A behavior specification of the application for hardware synthesis can be considered as a logical or algorithmic mapping function from the input data to the output data. The purpose of hardware synthesis is to map the computation and storage in the behavior specification into temporal (by scheduling) and spatial (by binding/allocation) implementation design spaces for performance, cost, and power optimization.

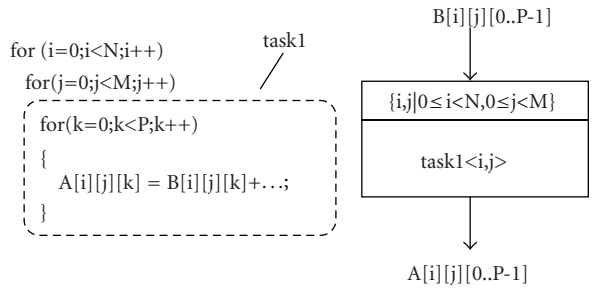

Figure 3: Concurrent task modeling.

In general, higher-level optimization has a larger design space and may lead to better potential results. To raise the abstraction level of the optimization flow, CnC uses steps as the basic units to specify the system-level behavior of an application. A step is defined as a statically determined mapping function from the values of an input data set to those of an output data set, in which the same input data will generate the same output data regardless of the timing of input and output data. A step is supposed to execute multiple times to process the different sets of data. Each execution of the step is defined as a step instance and is invoked by a control tag. Each control tag has an integer vector that is used to identify or index the instance of the step. An iteration domain is the set of all the iterator vectors corresponding to the step, representing all the step instances. The step instance of step t indexed by iterators (i, j) is notated as t<i,j> or t:i,j. As shown in Figure 3, we encapsulate the loop k into the task1, and task1 will execute N×M times according to the loop iterations indexed by variables i and j. Task1 has an iteration domain of {i, j | 0≤i<N, 0≤j<M}. Each iterator vector (i, j) in the iteration domain represents a step instance task1<i,j>. Compared to the explicit order of loops and loop iterations imposed in the sequential programming languages, no overconstrained order is defined between steps and step instances if there is no data dependence. In the concurrent specification, an application includes a collection of steps instead of a sequence of steps, and each step consists of a collection of step instances instead of a sequence of step instances.

A data item is, in general, defined as a multidimensional array used for the communication between steps and step instances. The array elements are indexed by data tags, each data tag represents the subscript vector of an array element. The data domain of an array defines the set of available subscript vectors. For an access reference of a multidimensional array A, its data tag is notated as $(A_0, A_1, A_2, \ldots)$. For example, the subscript vector of A[i][j][k] is (i, j, k), where A_0= i, A_1= j and A_2= k. Each step instance will access one or more of the elements in the arrays. A data access is defined as a statically determined mapping from the step instance iterator vector to the subscript vectors of the array elements that the task instance reads from or writes to. The input data access of task1, B[i][j][0...(P − 1)] in Figure 3, is the mapping {<i,j> → B[B_0][B_1][B_2] | B_0=i, B_1=j}. In this access, no constraints are placed on the subscript B_2, which means any data elements with the same B_0 and B_1 but different B_2 will be accessed in one step instance, task1<B_0, B_1 >. From the iteration domain and I/O accesses of a task, we can derive the set of data elements accessed by all the instances of the step.

Dynamic single assignment (DSA) is a constraint of the specification on data accesses. DSA requires that each data element only be written once during the execution of the application. Under the DSA constraint, an array element can hold only one data value, and memory reuse for multiple liveness-nonoverlap data is forbidden. CnC adopts the DSA constraint in its specification to avoid the conflicts of the concurrent accesses into one array element and thus provide the intrinsic determinism of the execution model.

Dependence represents the execution precedence constraints between step instances. If one step generates data that are used in another step, the dependence is implicitly specified by the I/O access functions of the same data object in the two steps. Dependence can also be used to describe any kind of step instance precedence constraints that may guide the synthesizer to generate correct and efficient implementations. For example, to explicitly constrain that the outmost loop i in Figure 3 is to be scheduled sequentially, we can specify the dependence like {task1<i, *> → task1<i+1, *>}, a concise form for $\forall j_1, j_2$, task1<i,j_1 > → task1<i+1,j_2 >. From the iteration domains and dependence mapping of two steps (the two steps are the same for self-dependence), we can derive the set of precedence constraints related to the instances of the two steps.

The execution of the concurrent steps is enabled when the two conditions are satisfied: (1) the control tag is generated (by primary input or by the execution of steps); (2) all the input data corresponding to the step instance is generated (by primary input or by the execution of steps). In Intel CnC, the iteration domains (control tag collections) of the steps are not statically defined and control tags are generated dynamically during the execution. An enabled step instance is not necessary in order to execute immediately. A runtime scheduler can dynamically schedule an enabled task instance at any time to optimize different implementation metrics.

3.2. Benefits and Limitations of CnC. There are some properties that the other MoCs have in common with CnC—support for hierarchy, deterministic execution, and specification of both control and data flow. However, there are a few points where CnC is quite distinct from the other concurrent specification models.

While most specification models expect the user to explicitly specify parallelism, CnC allows users to specify dependences, and the synthesizer or the runtime decides when and which step instances to schedule in parallel. Thus, parallelism is implicit and dynamic in the description. The dynamic nature of parallelism has the benefit of platform independency. If an application specified in CnC has to run on two different systems—an 8-core system (multicore CPU) and a 256-core system (GPU-like)—then the specification need not change to take into account the difference in the number of cores. The runtime would decide how many step

instances should be executed in parallel on the two systems. However, for the other MoCs, the specification may need to be changed in order to efficiently use the two systems.

Another benefit of using CnC is that since the dependence between the step and item collections is explicit in the model, it allows the compiler/synthesis tool/runtime manager to decide the best schedule. An example could be rescheduling the order of the step instances to optimize for data locality. Other MoCs do not contain such dependence information in an explicit fashion; the ordering of different node executions/firings is decided by the user and could be hidden from any downstream compiler/synthesis tool/runtime manager.

However, CnC was originally developed for general-purpose multicore platforms. Some issues need to be solved in order for CnC to become a good specification model for task-level hardware synthesis. First, dynamic semantics, such as step instance generation, make it hard to manage the task scheduling without a complex runtime kernel. This leads to a large implementation overhead in hardware platforms such as FPGA. Second, memory space for data items is not specified, which may imply unbounded memory size because the data item tags are associated with the dynamically generated control tags. Third, design experience shows that DSA constraints cause many inconveniences in algorithm specifications, and this makes CnC user-unfriendly. Fourth, currently there is not a stable version of CnC that formally supports hierarchy in the specification. As for those limitations of CnC as a task-level concurrency specification model for hardware synthesis, we have proposed our TLDM based on CnC and adapted it to hardware platforms.

4. Task-Level Data Model

In this section a task-level concurrency specification model is proposed based on the Intel CnC. We introduce a series of adaptations of Intel CnC targeted at task-level hardware system synthesis. We first overview the correspondence and differences between TLDM and CnC. Then our TLDM specification is defined in detail in a C++ form, and classes and fields are defined to model the high-level information used for task scheduling. While the C++ classes are used as the in-memory representation in the automation flow, we also define a text-based format for users to specify TLDM directly. These two forms of specifications have the equivalent semantics. An example application is specified using our TLDM specification. We also make a detailed comparison of the proposed TLDM with the CnC specification to demonstrate that the proposed TLDM is more suitable for hardware synthesis.

4.1. Overview of TLDM. Our TLDM specification model inherits the benefits of CnC and is customized to hardware synthesis. Table 1 summarizes the similarities and differences between TLDM and CnC. TLDM inherits most of the syntax and semantics elements from CnC. The counterparts for step, data, control tag, and data tag in CnC are, in TLDM, task, data, iterator vector and subscript vector, respectively.

```
class tldm_app {
  set<tldm_task *>        task_set;
  set<tldm_data *>        data_set;
  set<tldm_access *>      access_set;
  set<tldm_dependence *>dependence_set;
};
```

Listing 1

TLDM removes the syntax of the control item in CnC to avoid the dynamic behavior in task instantiation. In CnC a step instance is enabled when its control item is generated and all its input data is ready; in TLDM a task instance is enabled when its input data is ready. Control items may be transformed into data items to model the precedence relation between corresponding task instances. In TLDM, iteration domain, data domain, and data (input/output) accesses are specified explicitly and statically. According to the execution model of CnC, the dependence between steps is implied by the tag generation and data access statements in the step functions. TLDM provides syntax to explicitly specify the dependence between task instances in the task-level specification. DSA restriction is not formally enforced in the TLDM specification. Programmers need to ensure data coherence and program determinism by imposing dependence constraints of task instances in the case that DSA restriction is broken.

4.2. A C++ Specification for TLDM. A TLDM application consists of a task set, a data set, an access set, and a dependence set (See Listing 1).

The task set specifies all the tasks and their instances in a compact form. One task describes the iterative executions of the same functionality with different input/output data. The instances of one task are indexed by the iterator vectors. Each component of the iterator vector is one dimension of the iterator space, which can be considered as one loop level surrounding the task body in C/C++ language. The iteration domain defines the range of the iteration vector, and each element in the iteration domain corresponds to an execution instance of the task. The input and output accesses in each task instance are specified by affine functions of the iterator vector of the task instance. The access functions are defined in the *tldm_access* class.

Class members *parent* and *children* are used to specify the hierarchy of the tasks. The task hierarchy supported by our TLDM can provide the flexibility to select the task granularity in our optimization flow. A coarse-grained low-complexity optimization flow can help to determine which parts are critical for some specific design target, and fine-grained optimization can further optimize the subtasks locally with a more precise local design target.

A pointer to the details of a task body (task functionality) is kept in our model to obtain certain information (such as module implementation candidates) during the task-level synthesis. We do not define the concrete form to specify the

Table 1: Syntax and semantics summary of CnC and TLDM.

	CnC	TLDM
Computation	Step	Task
Communications	Data item	Data
	Control item	Not Supported
Index of set	Control item tag	Iterator vector
	Data item tag	Subscript vector
Execution model	(1) Control item is generated	Input data are ready
	(2) Input data are ready	
Iteration and data domain	Unbounded and dynamically determined	Bounded and statically specified
Data accesses	Embedded in step body	Specified in task definition
Dependence	Implicitly specified	Can be explicitly specified
DSA	Enforced	Not enforced

```
class tldm_task {
    string                    task_id;
    tldm_iteration_domain*    domain;
    vector<tldm_access *>     io_access; // input and output data accesses
    tldm_task *               parent;
    vector<tldm_task *>       children;
    tldm_ task_body*          body;
};
```

Listing 2

task body in our TLDM specification. A task body can be explicitly specified as a C/C++ task function, or a pointer to the in-memory object in an intermediate representation such as a basic block or a statement, or even the RTL implementations (See Listing 2).

An iteration domain specifies the range of the iterator vectors for a task. We consider the boundaries of iterators in four cases. In the first simple case, the boundaries of all the iterators are determined by constants or precalculated parameters that are independent of the execution of the current task. In the second case, boundaries of the inner iterator are in the form of a linear combination of the outer iterators and parameters (such as a triangular loop structure). The iteration domain of the first two cases can be modeled directly by a polyhedral model in a linear matrix form as *affine_iterator_range*. In the third case, the boundaries of the inner iterators are in a nonlinear form of the outer iterators and parameters. By considering the nonlinear term of outer iterators as pseudoparameters for the inner iterators, we can also handle the third case in a linear matrix form by introducing separate pseudoparameters in the linear matrix form. In the fourth and most complex case, the iterator boundary is determined by some local data-dependent variables varying in each instance of the task. For example, in an iterative algorithm, a data-dependent convergence condition needs to be checked in each iteration. In TLDM, we separate the iterators into data independent (first three cases) and data-dependent (the fourth case). We model data-independent iterators in a polyhedral matrix form (class *polyhedral_set* is described below in this subsection); we model data dependent iterators in a general form as a TLDM expression, which specifies the execution condition for the task instance. The execution conditions of multiple data-dependent iterators are merged into one TLDM expression in a binary-tree form (See Listing 3).

The data set specifies all the data storage elements in the dataflow between different tasks or different instances of the same tasks. Each *tldm_data* object is a multidimensional array variable or just a scalar variable in the application. Each data object has its member *scope* to specify in which task hierarchy the data object is available. In other words, the data object is accessible only by the tasks within its scope hierarchy. If the data object is global, its scope is set to be NULL. The boundaries (or sizes) of a multidimensional array are predefined constants and modeled by a polyhedral matrix *subscript_range*. The detailed form of *subscript_range* is similar to that of *affine_iterator_range* in the iteration domain. Accesses out of the array bound are forbidden in

```
class tldm_iteration_domain{
    vector<tldm_data *>     iterators;
    polyhedral_set          affine_iterator_range;
    tldm_expression*        execute_condition;
};
class tldm_expression {
    tldm_data*              iterator; // the iterator to check the expression
    int                     n_operator;
    tldm_expression*        left_operand;
    tldm_expression*        right_operand;
    tldm_data*              leaf;
};
```

Listing 3

```
class tldm_data {
    string                  data_id;
    tldm_task*              scope;
    int                     n_dimension; // number of dimensions
    polyhedral_set          subscript_range; // ranges in each dimensions
    tldm_data_body*         body;
};
```

Listing 4

our TLDM execution model. A body pointer for data objects is also kept to refer to the array definition in the detailed specification (See Listing 4).

The access set specifies all the data access mappings from task instances to data elements. Array data accesses are modeled as a mapping from the iteration domain to the data domain. If the mapping is in an affine form, it can be modeled by a polyhedral matrix. Otherwise, we assume that possible ranges of the nonaffine accesses (such as indirect access) are bounded. We model the possible range of each nonaffine access by its affine (or rectangular) hull in the multidimensional array space, which can also be expressed as a polyhedral matrix (class *polyhedral_set*). A body pointer (*tldm_access_body* *) for an access object is kept to refer to the access reference in the detailed specification (See Listing 5).

The dependence set specifies timing precedence relations between task instances of the same or different tasks. A dependence relation from (task0, instance0) to (task1, instance1) imposes a constraint in task scheduling that (task0, instance0) must be executed before (task1, instance1). The TLDM specification supports explicit dependence imposed by specified *tldm_dependence* objects and implicit dependence embedded in the data access objects. Implicit dependence can be analyzed statically by a compiler or optimizer to generate derived *tldm_dependence* objects. The explicit dependence specification provides the designer with the flexibility to add user-specified dependence to help the compiler deal with complex array indices. User-specified dependence is also a key factor in relieving designers from the limitations of dynamic single assignment, while maintaining the program semantics. The *access0* and *access*1 fields point to the corresponding *tldm_access* objects and can be optional for the user-specified dependence. In most practical cases, the dependence between task instances can be modeled by affine constraints of corresponding iterators of the two dependent tasks as *dependent_relation*. If the dependence relation between the two iterators is not affine, either a segmented affine form or an affine hull can be used to specify the dependence in an approximate way (See Listing 6).

A polyhedral_set object specifies a series of linear constraints of scalar variables in the data set of the application. The scalar variables can be the iterators (including loop iterators and data subscripts) and the loop-independent parameters. Each constraint is a linear inequality or equality of these scalar variables, and is modeled as an integer vector consisting of the linear coefficients for the scalar variables and a constant term and an inequality/equality flag. Multiple constraints form an integer matrix (See Listing 7).

Figure 4 shows examples of polyhedral matrices for iteration domain, data domain, data access, and task dependence, respectively. The first row of the matrix represents the list of iterators and parameters (*variables*), where $A0$ and $A1$ are the dimensions of array A and the # and $ columns are the constant terms and inequality flags, respectively. The following rows of the matrices represent linear constraints (*polyhedral_matrix*), where the elements of the matrices are the linear coefficients of the variables. For example, the second constraint of the iteration domain case is -i+0j+N-1≥0 (i≤N-1).

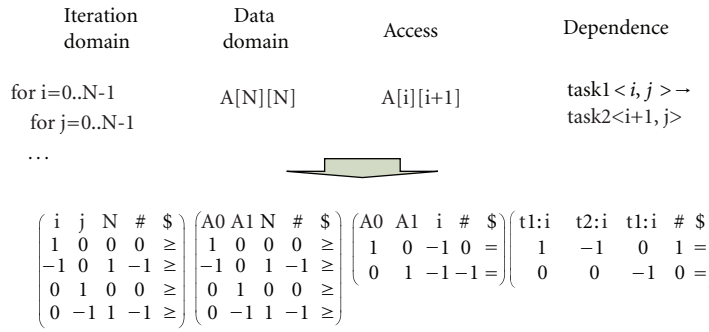

Figure 4: Polyhedral matrix representations.

```
class tldm_access {
    tldm_data*          data_ref; // the data accessed
    bool                is_write; // write or read access
    polyhedral_set      iterators_to_subscripts; // access range
    tldm_access_body*   body;
};
```

Listing 5

```
class tldm_dependence {
    tldm_task*          task0;
    tldm_task*          task1;
    tldm_access*        access0; //optional
    tldm_access*        access1; //optional
    polyhedral_set      iterator_relation;
};
```

Listing 6

```
class polyhedral_set {
    int                         n_iterator;
    int                         n_parameter;
    vector<tldm_data *>         variables;
    vector<vector<int≫          polyhedral_matrix;
};
```

Listing 7

4.3. A Textual Specification for TLDM. Similar to CnC, we use round, square, and angle brackets to specify tasks, data, and domains, respectively. Domain items can be iterator domains for tasks, or subscript domains for data items. Colons are used to specify an instance of these collections (See Listing 8).

Domains define the range constraints of iterator vectors and subscript vectors. For task instances that have variable iteration boundaries, some data items can be used as parameters in specifying the iteration range by arrow notations. Conditional iteration such as convergence testing can be specified by the keyword *cond*(…). Domains are associated with the corresponding data objects or tasks by double-colons (See Listing 9).

```
(task)
[data]
<domain>
(task_instance : iterator_vector)
[data_instance : subscript_vector]
<domain_instance: iterator_vector or subscript_vector>

e.g.
(task1)                 // A task named "task1"
<data0 : i, j, k>       // data element data0[i][j][k]
```

Listing 8

Input and output data accesses of task instance are defined by arrow operators. A range of data elements can be specified in a concise form by double-dot marks (See Listing 10).

Explicit dependence can also be specified by arrow operators. Dependence defines the relation between task instances (See Listing 11).

The body of a task can be defined in the statement where the iterator domain is associated with the task by using a brace bracket. A keyword *body_id*(…) is used to link to the detailed module-level information for the task. Task hierarchy can be defined by embedding the subtask

```
[parameter_list] -> <data_domain : subscript_vector>
{subscript_vector range};
[parameter_list] -> <iterator_domain : iterator_vector> {iterator_vector range};

<data_domain> :: [type data_name];
<iterator_domain> :: (task_name);

e.g.
// A[100][50]
<A_dom : A0, A1> {0<=A0; A0<100; 0<=A1; A1<50;};
<A_dom> : [double A];

// for (i=0; i<p; i++) task1(i);
[p] -><task1_dom : i> {0<=i; i<p;};
<task1_dom> : (task1);

// while (res > 0.1) {res = task2();}
[res] -> <task2_dom : t>  {cond(res > 0.1)};
<task2_dom> : (task2);
```

Listing 9

```
input_data_list -> (task_name : iterator_vector) -> output_data_list;

e.g.
// A[i][j], B[i][j] -> task1<i,j> -> C[j][i]
[A : i, j], [B : i, j] -> (task1 : i, j) -> [C : i, j]

//A[i][0],. . .,A[i][N]-> task2<i> -> B[2* i + 1]
[A : i, 0..N] -> (task1 : i) -> [B : 2* i + 1]
```

Listing 10

definitions into the body of the parent task. Data objects can also be defined in the task bodies to become local data objects (See Listing 12).

4.4. Examples of TLDM Modeling. Tiled Cholesky is an example that is provided by Intel CnC distribution [35]. Listings 13, 14 and 15 show the sequential C-code specification and TLDM modeling of the tiled Cholesky example. In this program we assume that tiling factor p is a constant.

The TLDM built from the tiled Cholesky example is shown in Listing 14. The data set, task set, iteration domain, and access set are modeled in both C++ and textual TLDM specifications.

Listings 16 and 17 show how our TLDM models the nonaffine iteration boundaries. The nonaffine form of outer loop iterators and loop-independent parameters are modeled as a new pseudo-parameter. The pseudo-parameter nonAffine(i) in Listing 17 is embedded in the polyhedral model of the iteration domain. The outer loop iterators (i) are associated with the pseudo-parameter as an input variable. In this way we retain the possibility of parallelizing the loops with nonaffined bounded iterators and keep the overall specification as a linear form.

For the convergence-based algorithms shown in Listings 18 and 19, loop iteration instance is not originally described as a range of iterators. An additional iterator (t) is introduced to the "while" loop to distinguish the iterative instances of the task. If we do not want to impose an upper bound for the iterator t, an unbounded iterator t is supported as in Listing 19. A *tldm_expression* exe_condition is built to model the execution condition of the "while" loop, which is a logic expression of *tldm_data* convergence_data. Conceptually, the instances of the "while" loop are possible to execute simultaneously in the concurrent specification. CnC imposes dynamic single-assignment (DSA) restrictions to avoid nondeterministic results. But DSA restriction requires multiple duplications of the iteratively updated data in the convergence iterations, which leads to a large or even unbounded array capacity. Instead of the DSA restriction in the specification, we support the user-specified dependence on reasonably constrained scheduling of the "while" loop instances to avoid the data access conflicts. Because the convergence condition will be updated and checked in each "while" loop iteration, the execution of the "while" loop must be done in a sequential way. So we add dependence from task1<t> to task1<t+1> for each t as in Listing 17. By removing the DSA restriction in the specification, we do not need multiple duplications of the iterative data, such as convergence or any other internal updating arrays.

```
(task_name0 : iterator_vector) -> (task_name1 : iterator_vector

e.g.
// task1<i,j> -> task2<j,i>
(task1 : i, j) -> (task2 : j, i)

// task1<i-1, 0>,...,task1<i-1, N> -> task1<i, 0>,...,task1<i, N>
(task1 : i-1, 0..N) -> (task1 : i, 0..N)
```

Listing 11

```
<iterator_domain0> :: (task_name)
{
  <data_domain> :: [type local_data_item];

  <iterator_domain1> :: (sub_task_name1)
  {
    // leaf node
    body_id(task_id1);
  };
  <iterator_domain2> :: (sub_task_name2)
  {
    // leaf node
    body_id(task_id2);
  };
  // access and dependence specification
};
```

Listing 12

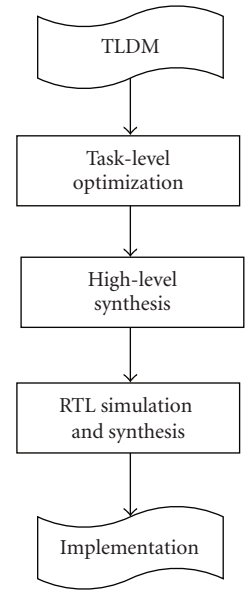

Figure 5: TLDM-based design flow.

Listings 20 and 21 show an indirect access example. Many nonaffine array accesses have their ranges of possible data units. For example, in the case of Listing 20, the indirect access has a preknown range (x<= idx[x] <= x+M). We can conservatively model the affine or rectangular hull of the possible access range in *tldm_access* objects. In Listing 21, the first subscript of the array_A is not a specific value related to the iterators, but a specific affine range of the iterators (j <= A_0 < j+M).

4.5. Demonstrative Design Flow Using TLDM. The detailed design optimizations from TLDM specification are beyond the scope of this paper. We show a simple TLDM-based design flow in Figure 5 and demonstrate how buffer size is reduced by task instance rescheduling as an example of task-level optimizations using TLDM. A task-level optimization flow determines the task-level parallelism and pipelining between the task instances, order of the instances of a task, and the interconnection buffers between the tasks. After task-level optimization, parallel hardware modules are generated automatically by high-level synthesis tools. RTL codes for both the hardware modules and their interconnections are integrated into a system RTL specification and then synthesized into the implementation netlist.

Different task instance scheduling will greatly affect the buffer size between the tasks. Instead of a fixed execution order of task instances in sequential program language, our TLDM can efficiently support selection of the proper execution order for task instance to reduce the buffer requirement between tasks. Listing 22 shows the example program to be optimized.

The TLDM specification for this example is shown in Figure 6. Two tasks are generated with their iteration domains. Four affine accesses to array A are notated in the TLDM graph. Three sets of data dependence are notated in the TLDM graph: two intratask dependences (dep0 and dep1), and one intertask dependence (dep2).

To evaluate the buffer size needed to transfer temporary data from task t0 to t1, we need to analyze the data element access pattern of the two tasks. As the execution order defined in the sequential C program, Figures 7(a) and 7(b) show the access order of the data t0 writes and t1 reads, respectively. A_0 and A_1 are array subscripts in the reference (like $A[A_0][A_1]$). The direction of the intratask data dependence is also shown in the two subgraphs. As we can see in the two illustrations, the access orders are not matched: one is row by row and the other is column by column. To maintain all the active data between the two tasks, an N×N buffer is needed.

```
1 int i, j, k;
2 data_type A[p][p][p+1];
3 for (k=0; k<p; k++) {
4     seqCholesky (A[k][k][k+1] ←A[k][k][k]);
5     for (j=k+1; j<p; j++) {
6         TriSolve(A[j][k][k+1] ←A[j][k][k], A[k][k][k+1]);
7         for (i=k+1; i<=j; i++) {
8             Update (A[j][i][k+1] ←A[j][k][k+1], A[i][k][k+1]);
9         }
10    }
11}
```

Listing 13: Example C-code of tiled Cholesky.

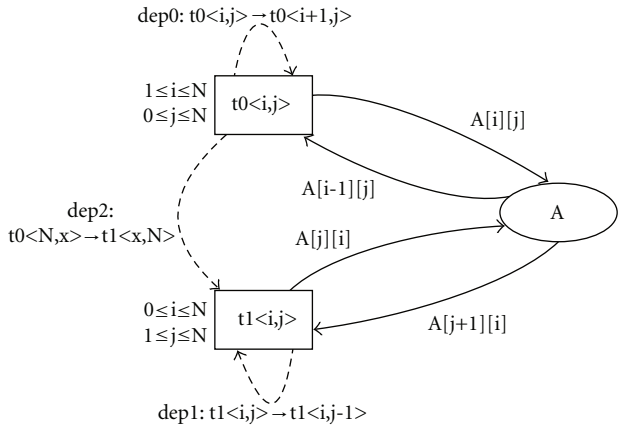

Figure 6: TLDM of the example in Listing 22.

In our TLDM specification, no redundant access order constraints are imposed, and optimization processes can select a proper execution order satisfying the dependence constraints to minimize the buffer requirement between the two tasks. The approach to reducing the buffer size is to match the access order of the array data between two tasks. A comprehensive task instance scheduling methodology is beyond the scope of this paper. Accordingly, we just evaluate two simple scheduling candidates: row-by-row and column by column for both tasks. For the row by row candidate, the intratask dependence for task t0 requires that task t0 start from row 1 to row N, while task t1 needs to start from row N to row 1, so the buffer size is not reduced. But for the column-by-column candidate, we can reorder the task instances of task t0 so that the array A is written column-by-column as shown in Figure 7(c). In this case, the required buffer size is reduced from N×N to N. The final scheduling of task instances is shown in Figure 7(d): each task processes one array column in turn.

Listing 23 shows the generated hardware module by TLDM-based task-level optimization, and it can be further synthesized by HLS tools. The order of the task instances and task-level pipeline are optimized to reduce the buffer size between the two tasks.

5. Advantages of TLDM

Task-level hardware synthesis requires a desired concurrency specification model that is (i) powerful enough to model all the applications in the domain, (ii) simple enough to be used by domain experts, independent of the implementation details, (iii) flexible enough for integrating diverse computations (potentially in different languages or stages of refinement), and (iv) efficient enough for generating a high quality of results in a hardware synthesis flow. This section presents concrete examples to illustrate that the proposed TLDM specification model is designed to satisfy these requirements.

5.1. Task-Level Optimizations. Since our TLDM model is derived from CnC, the benefit of implicit parallelism is common to both models. However, other models, like dataflow networks, do not model dynamic parallelism. The sequential code in Figure 8 is a simple example of parallelizable loops. For process networks, the processes are defined in a sequential way, so the concurrency can only be specified by initiating distinct processes. The number of parallel tasks (parameter p in Figure 8) is specified statically by the user—represented as multiple processes in the network. However, a single statically specified number may not be suitable for different target platforms. For example, on a multicore CPU, the program can be broken down into 8 to 16 segments that can be executed in parallel; however, on a GPU with hundreds of processing units, the program needs to be broken down into segments of finer granularity. In our TLDM, users only need to specify a task collection with its iterator domain and data accesses, and then the maximal parallelism between task instances is implicitly defined. The runtime/synthesis tool will determine the number and granularity of segments to run in parallel, depending on the target platform.

In addition, the collection-based specification does not introduce any redundant execution order constraints on the task scheduling. For example, in the data streaming application in Figure 9, the KPN specification needs to explicitly define the loop order in each task process. In the synthesis and optimization flow, it is possible to reduce

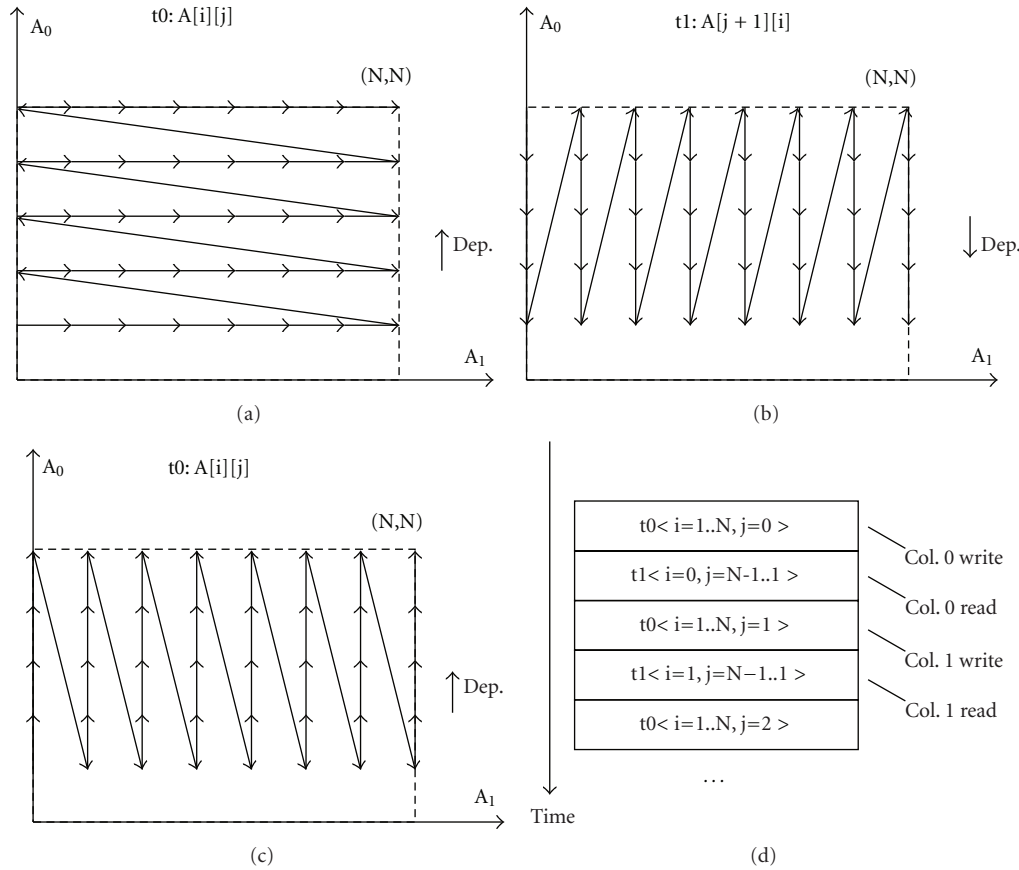

Figure 7: (a) Initial access order of t0 writes. (b) Initial access order of t1 reads. (c) Optimized access order of t0 writes. (d) Optimized task instance scheduling for buffer reduction.

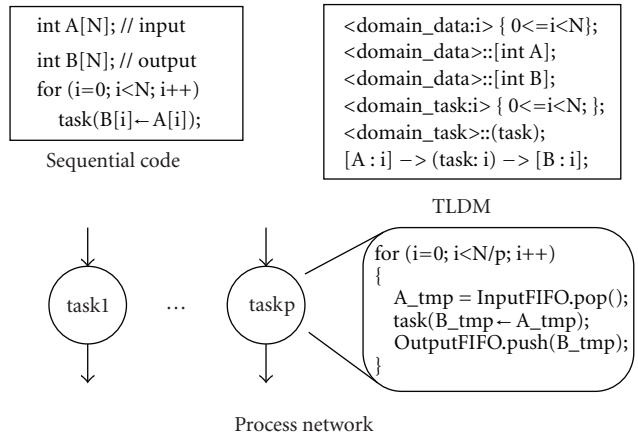

Figure 8: Implicit parallelism (TLDM) versus explicit parallelism (process network).

the buffer size between tasks by reordering the execution order of the loop iterations. But these optimizations are relatively hard for a KPN-based flow because the optimizer needs to analyze deep into the process specification and interact with module-level implementation. However, in our TLDM specification, necessary task-instance-level dependence information is explicitly defined without redundant order constraints in sequential languages. This offers the possibility and convenience of performing task-level optimizations without touching module-level implementation.

5.2. Heterogeneous Platform Supports. Heterogeneous platforms, such as the customized heterogeneous platform being developed by CDSC [9, 36], have gained increased attention

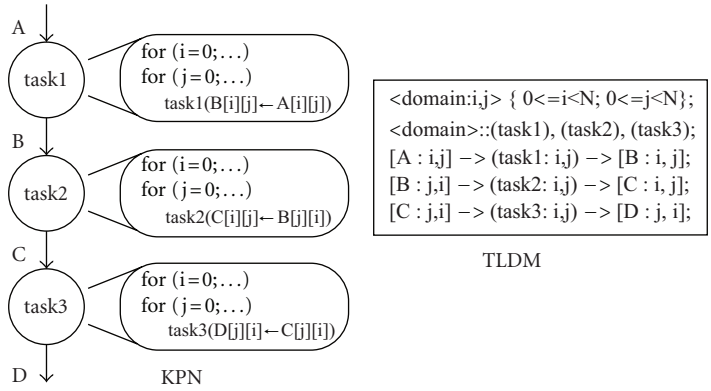

Figure 9: Support for task rescheduling.

in high-performance computation system design because of the benefits brought about by efficient parallelism and customization in a specific application domain. Orders-of-magnitude efficiency improvement for applications can be achieved in the domain using domain-specific computing platforms consisting of customizable processor cores, GPUs, and reconfigurable fabric.

CnC is essentially a coordination language. This means that the CnC specification is largely independent of the internal implementation of a step collection, and individual step collections could be implemented in different languages/models. This makes integrating diverse computational steps simpler for users, compared to rewriting each step into a common concurrent specification. The module-level steps could be implemented in C/C++/Java for GPPs, CUDA/OpenCL for GPUs, or even HDL such as VHDL and Verilog for hardware accelerator designs. In addition, because CnC is a coordination description supporting different implementations, it naturally supports an integrated flow to map domain-specific applications onto heterogeneous platforms by selecting and refining the module-level implementations. Our TLDM model inherits this feature directly from CnC.

We take medical image processing applications as an example. There are three major filters (denoise, registration, and segmentation) in the image processing pipeline. An image sequence with multiple images will be processed by the three filters in a pipelined way. At the task level, we model each filter as a task, and each task instance processes one image in our TLDM model. The task-level dataflow is specified in the TLDM specification, which provides enough information for system-level scheduling. Module-level implementation details are embedded in module-level specifications, which can support different languages and platforms. In the example shown in Figure 10, we are trying to map three filters onto a FPGA+GPU heterogeneous platform. One typical case is that denoise is implemented in an FPGA by a high-level synthesis flow from C++ specification, registration is mapped onto a GPU, and segmentation is specified as a hardware IP in the FPGA. Module specifications using C++ (for HLS-FPGA synthesis), CUDA (for GPU compiling), and RTL (for hardware design IPs) are coordinated with a common TLDM model. For each execution of task instance in TLDM, the module-level function is called once for C++ and CUDA cases, or a synchronization cycle by enabled and done signals is performed for the RTL case, which means the RTL IP has processed one image. The related input data and space for output data are allocated in a unified and shared memory space, so that the task-level data transfer is transparent to the module designers; this largely simplifies the module design in a heterogeneous platform.

To map the application in Figure 10 onto a heterogeneous platform, each task in the TLDM specification can have multiple implementation candidates—such as multi-core CPU and many-core GPU and FPGA versions. These module-level implementation candidates share the same TLDM specification. A synthesizer and simulator can be used to estimate or profile the physical metrics for each task and each implementation candidate. According to this module-level information and the task-level TLDM specification, efficient task-level allocation and scheduling can be performed to determine the architecture mapping and optimize the communication.

5.3. Suitability for Hardware Synthesis. Our TLDM is based on the Intel CnC model. Task, data, and iterator in TLDM are the counterparts of step, data item, and control tag in CnC, respectively. Regular tasks and data are specified in a compact and iterative form and indexed by iterators and array subscripts. Both task instances in TLDM and prescribed steps in CnC need to wait for all the input data to be available in order to start the execution and do not ensure the availability of the output data until the execution of the current instance is finished. The relative execution order of task and step instances is only constrained by true data dependence, not by textual positions in sequential programs. However, there are also great differences between TLDM and CnC. By restricting the general dynamic behavior allowed by CnC, TLDM is more suitable to hardware synthesis for most practical applications, and it differs from the general CnC in the following five aspects.

(1) CnC allows dynamically generating step instances; this is hard to implement in hardware synthesis. In addition,

```
tldm_app tiled_cholesky;

// iterator variables
tldm_data iterator_i (“i”); // scalar variable
tldm_data iterator_j (“j”);
tldm_data iterator_k (“k”);

// environment parameters
tldm_data param_p (“p”);

// array A[p][p][i + 1]
tldm_data array_A(“A”, 3); // n_dimension = 3
array_A.insert_index_constraint(0, “>=”, 0);
array_A.insert_index_constraint(0, “<”, p);
array_A.insert_index_constraint(1, “>=”, 0);
array_A.insert_index_constraint(1, “<”, p);
array_A.insert_index_constraint(2, “>=”, 0);
array_A.insert_index_constraint(2, “<”, p+1);

// attach data into tldm application
tiled_cholesky.attach_data(&iterator_i);
tiled_cholesky.attach_data(&iterator_j);
titled_cholesky.attach_data(&iterator_k);
tiled_cholesky.attach_data(&param_p);
tiled_cholesky.attach_data(&array_A);

// iteration domain of task seq_cholesky
tldm_iteration_domain id_k;
id_k.insert_iterator(iterator_k); // “k”
id_k.insert_affine_constraint(“k”, 1, “>=”, 0); // k*1 >= 0
id_k.insert_affine_constraint(“k”, -1, “p”, 1, “>”, 0); // -k+p > 0

// accesses A[k][k][k+1]
tldm_access acc_A0 (&array_A, WRITE);
acc_A0.insert_affine_constraint(“A(0)”, 1, “k”, -1, “=”, 0); // A0 = k
acc_A0.insert_affine_constraint(“A(1)”, 1, “k”, -1, “=”, 0); // A1 = k
acc_A0.insert_affine_constraint(“A(2)”, 1, “k”, -1, “=”, 1); // A2 = k+1
// accesses A[k][k][k]
tldm_access acc_A1 (&array_A, READ);
acc_A1.insert_affine_constraint(“A(0)”, 1, “k”, -1, “=”, 0);
acc_A1.insert_affine_constraint(“A(1)”, 1, “k”, -1, “=”, 0);
acc_A1.insert_affine_constraint(“A(2)”, 1, “k”, -1, “=”, 0);

// task seqCholesky
tldm_task seq_cholesky(“seqCholesky”);
seq_ cholesky.attach_id(&id_k);
seq_cholesky.attach_access(&acc_A0);
seq_cholesky.attach_access(&acc_A1);
seq_cholesky.attach_parent(NULL);
tiled_cholesky.attach_task(&seg_cholesky);


// iteration domain of task tri_solve
tldm_iteration_domain id_kj = id_k.copy(); // 0 <= k < p
id_kj.insert_iterator(iterator_j); // “j”
id_kj.insert_affine_constraint(“j”, 1, “k”, -1, “>=”, 1); // j-k >= 1
id_kj.insert_affine_constraint(“j”, -1, “p”, 1, “>”, 0); // -j+p > 0
// accesses A[j][k][k + 1]
```

Listing 14: Continued.

```
tldm_access acc_A2 = acc_A0.copy();
acc_A2.replace_affine_constraint("A(0)", 1, "j", -1, "=", 0); // A0 = j
// accesses A[j][k][k]
tldm_access acc_A3 = acc_A1.copy();
acc_A3.replace_affine_constraint("A(0)", 1, "j", -1, "=", 0); // A0 = j
// accesses A[k][k][k + 1]
tldm_access acc_A4 = acc_A1.copy();
acc_A4.replace_affine_constraint("A(2)", 1, "k", -1, "=", 1); // A2 = k+1

// task TriSolve
tldm_task tri_solve("TriSolve");
tri_solve.attach_id(&id_kj);
tri_solve.attach_access(&acc_A2);
tri_solve.attach_access(&acc_A3);
tri_solve.attach_access(&acc_A4);
tri_solve.attach_parent(NULL);
tiled_cholesky.attach_task(&tri_solve);

// iteration domain of task update
tldm_iteration_domain id_kji = id_kj.copy();
id_kji.insert_iterator(iterator_i); // "i"
id_kji.insert_affine_constraint("i", 1, "k", -1, ">=", 1); // i-k >= 1
id_ kji.insert_affine_constraint("i", -1, "j", 1, ">=", 0); // -i+j >= 0

// accesses A[j][i][k + 1]
tldm_access acc_A5 = acc_A2.copy();
acc_A5.replace_affine_constraint("A(1)", 1, "i", -1, "=", 0); // A1 = i
// accesses A[j][k][k + 1]
tldm_access acc_A6 = acc_A4.copy();
acc_A6.replace_affine_constraint("A(0)", 1, "j", -1, "=", 0); // A0 = j
// accesses A[i][k][k + 1]
tldm_access acc_A7 = acc_A4.copy();
acc_A7.replace_affine_constraint("A(0)", 1, "i", -1, "=", 1); // A0 = i

// task Update
tldm_task update ("Update");
update.attach_id(&id_kji);
update.attach_access(&acc_A5);
update.attach_access(&acc_A6);
update.attach_access(&acc_A7);
update.attach_parent(NULL);
tiled_cholesky.attach_task(&update);

// dependence: TriSolve<k,j> -> Update<k,j, *>
tldm_dependence dept(&tri_solve, &update);
dept.insert_affine_constraint("k", 1, 0, "->", "k", 1, 0); // k0=k1
dept.insert_affine_constraint("j", 1, 0, "->", "j", 1, 0); // j0=j1
```

Listing 14: C++ TLDM specification of tiled Cholesky.

data collection in CnC is defined as unbounded: if a new data tag (like array subscripts) is used to access the data collection, a new data unit is generated in the data collection. In TLDM, iteration domain and data domain are statically defined, and the explicit information of these domains helps to estimate the corresponding resource costs in hardware synthesis. Data domain in TLDM is bounded: out-of-bounds access will be forbidden in the specification.

Consider the CnC specification for the vector addition example in Figure 11. The environment is supposed to generate control tags that prescribe the individual step instances that perform the computation. Synthesis tools would need to synthesize hardware to (i) generate the control tags in some sequence and (ii) store the control tags till the prescribed step instances are ready to execute. Such an implementation would be inefficient for such a simple program. TLDM works around this issue by introducing the concept of iteration domains. Iteration domains specify that all control tags are generated as a numeric sequence with a constant stride. This removes the need to synthesize hardware for explicitly generating and storing control tags.

(2) In CnC, input/output accesses to the data collections are embedded implicitly in the step specification. TLDM adopts explicit specification for inter-task data accesses. In

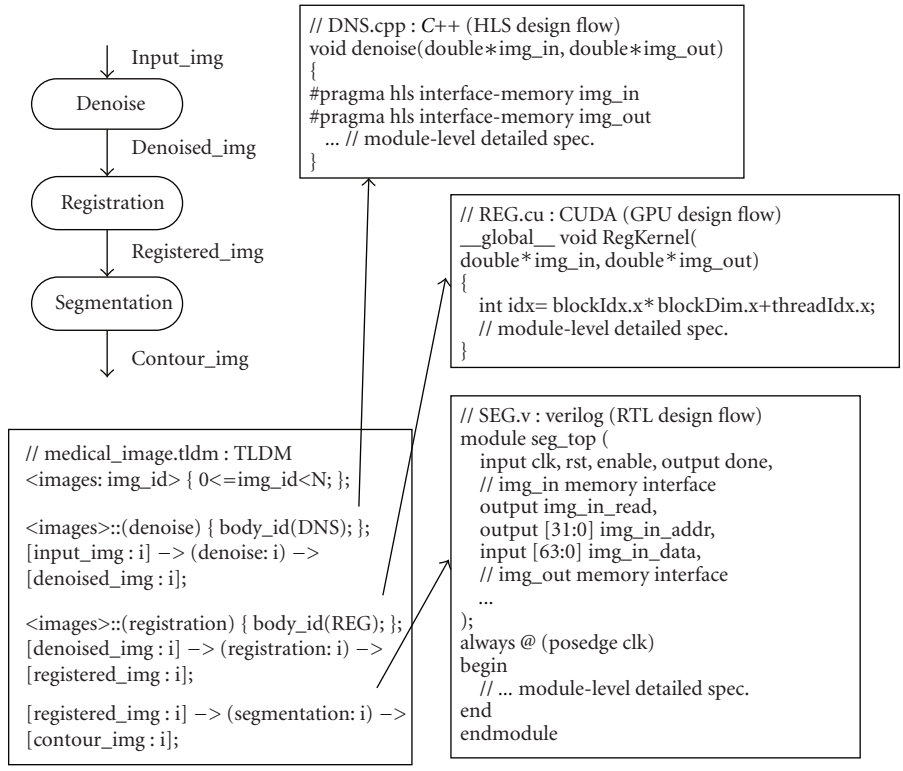

Figure 10: Heterogeneous computation integration using TLDM.

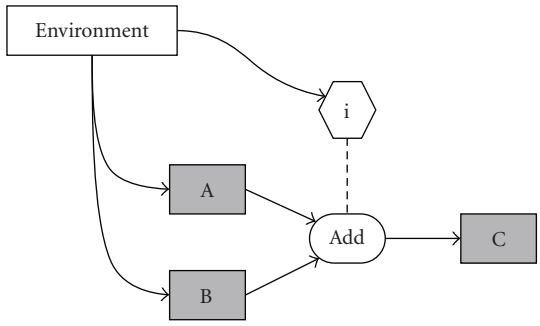

Figure 11: CnC representation for vector add.

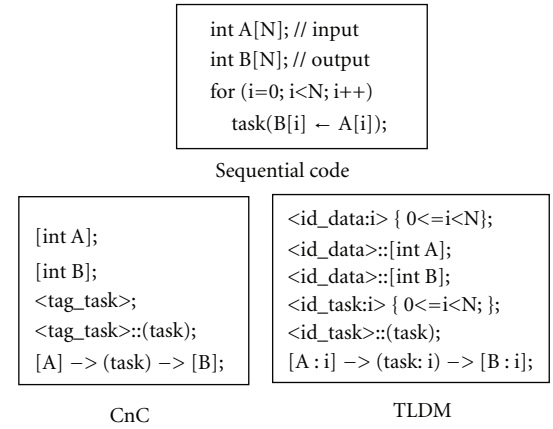

Figure 12: TLDM versus CnC: iterator domain and access pattern are explicitly specified in TLDM.

our simple example in Figure 12, the task *task* reads data from data collections A, but for a given step instance i, task-level CnC does not specify the exact data element to be read. This makes it hard for hardware synthesis to get an efficient implementation without the access information. TLDM specifies the access patterns directly, which can be used for dependence analysis, reuse analysis, memory partitioning, and various memory-related optimizations.

(3) In the static specification for the domain and mapping relations, a polyhedral model is adopted in TLDM to model the integer sets and mappings with linear inequality and equations. As shown in Figure 13, the range of the iterators can be defined as linear inequalities, which can be finally expressed concisely with integer matrices. Each row of the matrix is a linear constraint, and each column of the matrices is the coefficient associated with each variable. Standard polyhedral libraries can be used to perform transformation on these integer sets and to analyze specific metrics such as counting the size. The polyhedral model framework helps us to utilize linear/convex programming and linear algebra properties in our synthesis and optimization flow.

Note that TLDM introduces a hybrid specification to model the affine parts in the polyhedral model and non-affine parts in the polyhedral approximation or nonaffine execution conditions. Compared to the traditional linear

```
<> :: (top_level) // top level task, single instance
{
<> :: [data_type p]; // no domain for scalar variable

// data A[p][p][p+1]
[p] -> <A_dom : a0,a1,a2> {0<=a0<p; 0<=a1<p; 0<=a2<p+1;};
<A_dom> :: [array_A];

// task seqCholesky
[p] -> <task1_dom:k> {0<=k<p;};
<task1_dom> :: (task1) {body_id("seqCholesky");};
[A : k, k, k] -> (task1:k) -> [A : k, k, k + 1];
//task TriSolve
[p] -> <task2_dom : k,j> {0<=k<p; k+1<=j<p;};
<task2_dom> :: (task2) {body_id("TriSolve");};

// task Update
[p] -> <task3_dom:k,j,i> {0<=k<p; k+1<=j<p; k+1<=i<=j;};
<task3_dom> :: (task3) {body_id("Update");};
[A : j, k, k + 1],[A : i, k, k + 1] -> (task3 : k,j,i) -> [A : j, i, k + 1];

//dependence
(task2 : k,j) -> (task3 : k,j,(k+1)..j)
};
```

Listing 15: Textual TLDM specification of tiled Cholesky.

```
for (i=0; i<N; i++)
    for (j=0; j<i*i; j++) // nonaffine boundary i*i
        task0(...);
```

Listing 16: Example C-code of nonaffine iterator boundary.

```
tldm_iteration_domain id_ij;
id_ij.insert_iterator(iterator_i); // "i"
id_ij.insert_iterator(iterator_j); // "j"
id_ij.insert_affine_constraint("i", 1, ">=", 0); // i*1 >= 0
id_ij.insert_affine_constraint("i", -1, "N", 1, ">", 0); // -i+N > 0
id_ij.insert_affine_constraint("j", 1, ">=", 0); // i*1 >= 0
id_ij.insert_affine_constraint("j", -1, "nonAffine(i)", 1, ">", 0); // -j+(i*i)>0

[N] -> <id_ij: i, j> {0<=i<N; 0<=j<i*i;};
```

Listing 17: TLDM specification of nonaffine iterator boundary.

```
while (!convergence)
    for (i=0; i<N; i++)
        task1(i, convergence,...); // convergence is updated in task0
```

Listing 18: Example C-code of convergence algorithm.

```
tldm_data convergence_data("convergence");

tldm _iteration_domain id_ti;
id_ti.insert_iterator(iterator_t); // "t" for the while loop
id_ti.insert_iterator(iterator_i); // "i"
id_ti.insert_affine_constraint("i", 1, ">=", 0); // i*1 >= 0
id_ti.insert_affine_constraint("i", -1, "N", 1, ">", 0); // -i+N > 0
id_ti.insert_affine_constraint("t", 1, ">=", 0); // t >= 0
tldm_expression exe_condition(&iterator_t, "!", &convergence_data);
id_ti.insert_exe_condition(&exe_condition);

// non-DSA access: different iterations access the same scalar data unit
tldm_access convergence_acc (&convergence_data, WRITE);

tldm_task task1("task1");
seq_cholesky.attach_id(&id_ti);
seq_cholesky.attach_access(&convergence_acc);

// dependence is needed to specify to avoid access conflicts
tldm_dependence dept(&task1, &task1);
// dependence : task1<t> → task1<t+1>
// dept.insert_affine_constraint ("t", 1, 0, "->", "t", 1, 1); // (t+0) → (t+1)
// dependence are added to make the while loop iterations to execute in sequence, 0, 1, 2, ...
<> :: [convergence];
[N] -> < id_ti : t, j> {cond(!convergence); 0<=i<N;};
<id_ti> ::(task1) {body_id("task1")};
(task1: t, i) -> [convergence]
(task1 : t-1, 0..N) -> (task1 : t, 0..N);
```

Listing 19: TLDM specification of convergence algorithm.

```
for (i=0; i<N; i++)
    for (j=0; j<N; j++)
        task2(... ← A [idx[j]][i],...); // implication : x<= idx[x] <= x+M
```

Listing 20: Example C-code of indirect access.

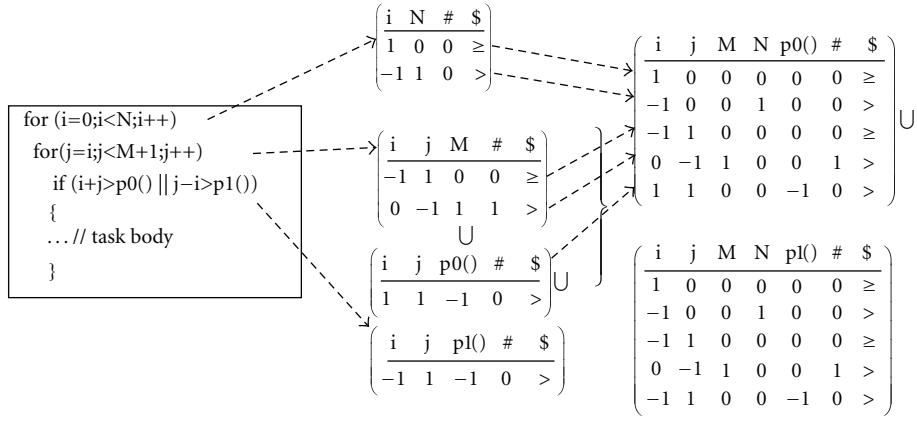

Figure 13: Polyhedral representation for iterator domain.

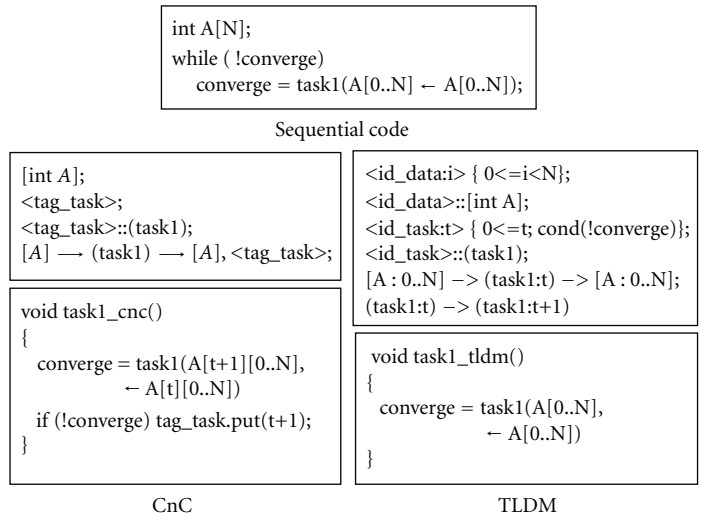

Figure 14: DSA (CnC) versus user-specified dependence (TLDM).

```
tldm_data array_A("A");

// accesses A[idx[j]][i] // implication: x<= idx[x] < x+M
tldm_access indirect_acc(&array_A, READ);
indirect_acc.insert_affine_constraint("A(0)", 1, "j", -1, ">=", 0); // A0 >= j
indirect_acc.insert_affine_constraint("A(0)", 1, "j", -1, "M", -1, "<", 0); // A0 < j+M
indirect_acc.insert_affine_constraint ("A(1)", 1, " i", -1, "=", 0); // A1 = i

[A : j..(j + M), i] -> (task2 : i, j)
```

Listing 21: TLDM specification of indirect access.

```
1 for (i=1; i<=N; i++)
2     for (j=0; j<=N; j++)
3         t0: A[i][j] = A[i-1][j] + . . .;
4 for (i=0; i<=N; i++)
5     for (j=N-1; j>=0; j- -)
6         t1: A[j][i] = A[j + 1][i] +. . .;
```

Listing 22: Example application for buffer reduction.

polyhedral model, we support the extensions in the following three aspects. First, if the boundaries of the iterator are in the nonaffine form of outer iterators and task-independent parameters, parallelism and execution order transformation for this non-affine-bound iterator is still possible. We introduce pseudoparameters to handle these inner-loop-independent nonaffine boundaries. Second, if the boundaries of the iterator are dependent on data items generated by the task itself—for example, in the case of a convergent algorithm—nonaffine data-dependent execution conditions are specified explicitly by *tldm_expression* objects, which are easy to analyze in the optimization and hardware generation flows. Third, for nonaffine array accesses or block data access in an array, we support a polyhedral-based form to specify the affine (rectangle) hull of the possible accessed array elements. With this approach, we can conservatively and efficiently handle the possible optimizations for many of the nonaffine array accesses—such as indirect accesses (shown in Listing 21).

(4) CnC restricts dynamic single assignment (DSA) in the data access specification to achieve the intrinsic deterministic characteristics in a concurrent specification. But practical experience shows that the DSA restriction is not convenient for designers when specifying their application for efficient hardware implementation. Memory space reuse in CnC is forbidden, which makes the designer lose the capability of specifying memory space reuse schemes. In addition, loop-carried variables need to maintain one duplication for each iteration; this leads to an unbounded data domain when the iteration domain is unbounded. For example, in Figure 14 array A needs to have different copies for each iteration in the CnC specification; this leads to an additional dimension for A to index the specific copy. In TLDM we do not enforce the DSA restriction in the specification, but DSA is still recommended. For those cases in which designers intend to break DSA restriction, write access conflicts need to be handled by the designer using our user-specified dependence specifications to constrain the possible

```
1  col_buf[N];
2  for (i=0; i<=N; i++)
3  { // pragma hls loop pipeline
4       t0(1..N, i, col_buf); // write one column of array A[N][N]
5       t1(i, N-1..1, col_buf); // read one column of array A[N][N]
6  }
```

Listing 23: Generated HW module after task-level scheduling.

conflict accesses into different time spans. In Figure 14 dependence (task1 : t)->task(1 : t+1) ensures the sequential execution of the convergence iterations, which can guarantee the correctness without DSA constraints.

(5) The traditional CnC specification is not scalable because it does not support the hierarchy of steps. In our TLDM specification, task graphs are built in a hierarchical way. Designers can specify a set of highly coupled subtasks into a compound task. Hardware synthesis flow can select various methodologies to perform coarse-grained or fine-grained optimizations based on the task hierarchy. In addition, specific synthesis constraints or optimization targets can be applied to different compound tasks according to the computation characteristics of the compound tasks. Data elements can also be defined in a smaller scope for better locality in memory optimizations.

5.4. Discussion and Comparison. Table 2 compares some widely used concurrency specifications in different aspects. CSP is the general underlying MoC of the event-driven Metropolis framework. The PSM model is specified in a structural language, SpecC, and used in the SCE framework. SDF and KPN are the most popular models for data-intensive applications and have been adopted by StreamIt and Daedalus projects. CnC and TLDM were recently proposed to separate implementation optimizations from concurrency specifications.

In Table 2 the first column lists the features for the concurrency specifications. In the table, solid circles, hollow circles, and blanks mean full support, partial support, and no support, respectively. All of these models are designed to support data streaming applications because they are the most popular application types in electronic system design. For control-intensive applications, CSP and CnC can support general irregular (conditional) behavior like dynamic instance generation and conditional execution; PSM has an intrinsic support for concurrent FSMs; KPN and TLDM support data-dependent execution, but their process instance is statically defined; SDF has regular data accesses and actor firing rates, which make it inefficient to model conditional execution. CSP is nondeterministic because of its general trigger rules; PSM has parallel processes that can write the same data; other models are all deterministic. Hierarchy is fully supported by CSP, PSM, and TLDM, and hierarchical CnC is under development now. In CSP, PSM, and KPN, data parallelism has to be specified explicitly because they do not have the concept of task instance as CnC and TLDM. SDF has partially scalable data parallelism because the data parallelism can be relatively easily exploited by automated transformations on the specification like StreamIt. Reordering the task instance can help to improve the task parallelism and memory usage, which can only be supported by CnC and TLDM. CSP and CnC support dynamic generation of task instances, and the dynamic behavior is hard to analyze statically by a hardware synthesizer. On the contrary, SDF, KPN, and TLDM are limited in multicore processor-based platforms because the user needs to transform the dynamic specification into equivalent static specification. CSP (Metropolis), PSM and TLDM have explicit specification for the interfaces and accesses between tasks, which can fully encapsulate the implementation details of the task body for different platforms. In SDF, KPN, and CnC, the access information is embedded in the task body specification, and additional processes are needed to extract the information from different specifications of heterogeneous platforms. From the comparison, we can see that the proposed TLDM is the most suitable specification for hardware synthesis, especially for data-intensive applications.

6. Conclusion and Ongoing Work

This paper proposes a practical high-level concurrent specification for hardware synthesis and optimization for data processing applications. In our TLDM specification, parallelism of the task instances is intrinsically modeled, and the dependence constraints for scheduling are explicit. Compared to the previous concurrent specification, TLDM aims to specify the applications in a static and bounded way with minimal overconstraints for concurrency. A polyhedral model is embedded in the specification of TLDM as a standard and unified representation for iteration domains, data domains, access patterns, and dependence. Extensions for nonaffine terms in the program are comprehensively considered in the specification as well to support the analysis and synthesis of irregular behavior. Concrete examples show the benefits of our TLDM specification in modeling a task-level concurrency for hardware synthesis in heterogeneous platforms. We are currently in the process of developing the TLDM-based hardware synthesis flow.

Acknowledgments

This work is partially funded by the Center for Domain-Specific Computing (NSF Expeditions in Computing Award

Table 2: Summary of the concurrency specifications.

	CSP	PSM	SDF	KPN	CnC	TLDM
Data streaming application	•	•	•	•	•	•
Control-intensive application	•	•		◦	•	◦
Nondeterministic behavior	•	◦				
Hierarchical structure	•	•			◦	•
Scalable data parallelism			◦		•	•
Task instance reordering					•	•
HW Synthesis		•	•	•		•
Multicore platform	•	•	◦	◦	•	◦
Heterogeneous platform	•	•	◦	◦	◦	•

CCF-0926127) and the Semiconductor Research Corporation under Contract 2009-TJ-1984. The authors would like to thank Vivek Sarkar for helpful discussions and Janice Wheeler for editing this paper.

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 high-level 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.

[22] W. Thies, M. Karczmarek, and S. Amarasinghe, "Streamit: a language for streaming applications," in *Proceedings of the 11th International Conference on Compiler Construction (CC '02)*, R. Horspool, Ed., vol. 2304 of *Lecture Notes in Computer Science*, pp. 49–84, Springer, Berlin, Germany, 2002.

[23] M. Chiodo, P. Giusto, H. Hsieh, A. Jurecska, L. Lavagno, and A. Sangiovanni-Vincentelli, "A formal specification model for hardware/software codesign," in *Proceedings of the International Workshop on Hardware/Software Co-Design*, 1993.

[24] T. Kangas, P. Kukkala, H. Orsila et al., "UML-based multiprocessor SOC design framework," *ACM Transactions on Embedded Computing Systems*, vol. 5, no. 2, pp. 281–320, 2006.

[25] F. Balarin, M. Chiodo, and P. Giusto, *Hardware-Software Co-Design of Embedded Systems: The POLIS Approach*, Kluwer Academic, Norwell, Mass, USA, 1997.

[26] C. Park, J. Jung, and S. Ha, "Extended synchronous dataflow for efficient DSP system prototyping," *Design Automation for Embedded Systems*, vol. 6, no. 3, pp. 295–322, 2002.

[27] S. Ha, S. Kim, C. Lee, Y. Yi, S. Kwon, and Y. P. Joo, "PeaCE: a hardware-software codesign environment for multimedia embedded systems," *ACM Transactions on Design Automation of Electronic Systems*, vol. 12, no. 3, 2007.

[28] K. Strehl, L. Thiele, M. Gries, D. Ziegenbein, R. Ernst, and J. Teich, "FunState—an internal design representation for codesign," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 9, no. 4, pp. 524–544, 2001.

[29] K. Grüttner and W. Nebel, "Modelling program-state machines in SystemC^{TM}," in *Proceedings of the Forum on Specification, Verification and Design Languages (FDL '08)*, pp. 7–12, Stuttgart, Germany, September 2008.

[30] R. Dömer, A. Gerstlauer, J. Peng et al., "System-on-chip environment: a SpecC-based framework for heterogeneous MPSoC design," *EURASIP Journal on Embedded Systems*, vol. 2008, no. 1, Article ID 647953, 13 pages, 2008.

[31] M. Fujita and H. Nakamura, "The standard SpecC language," in *Proceedings of the 14th International Symposium on System Synthesis (ISSS '01)*, pp. 81–86, ACM, New York, NY, USA, 2001.

[32] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou, "Cilk: an efficient multithreaded runtime system," in *Proceedings of the 5th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pp. 207–216, August 1995.

[33] Habanero-C project, 2011, https://wiki.rice.edu/confluence/display/HABANERO/Habanero-C.

[34] V. Cave, J. Zhao, J. Shirako, and V. Sarkar, "Habanero-Java: the new adventures of old x10," in *Proceedings of the 9th International Conference on the Principles and Practice of Programming in Java (PPPJ '11)*, 2011.

[35] Intel CnC distribution, 2011, http://software.intel.com/en-us/articles/intel-concurrent-collections-for-cc/.

[36] Center for Customizable Domain-Specific Computing (CDSC), http://cdsc.ucla.edu/.