

DEFACTO: A Design Environment for Adaptive Computing Technology

Kiran Bondalapati¹, Pedro Diniz², Phillip Duncan³, John Granacki², Mary Hall², Rajeev Jain²,
Heidi Ziegler¹

Abstract

The lack of high-level design tools hampers the widespread adoption of adaptive computing systems. Application developers have to master a wide range of functions, from the high-level architecture design, to the timing of actual control and data signals. These systems are extremely cumbersome and error-prone, making it difficult for adaptive computing to enter mainstream computing. In this paper we describe DEFACTO, an end-to-end design environment aimed at bridging the gap in tools for adaptive computing by bringing together parallelizing compiler technology and synthesis techniques.

1.0 Introduction

Adaptive computing systems consisting of configurable computing logic can offer significant performance advantages over conventional processors as they can be tailored to the particular computational needs of a given application (e.g., template-based matching, Monte Carlo simulation, and string matching algorithms). Unfortunately, developing programs that incorporate configurable computing units (CCUs) is extremely cumbersome, demanding that software developers also assume the role of hardware designers. At present, developing applications on most such systems requires low-level VHDL coding, and complex management of communication and control. While a few application developers tools are being designed, these have been narrowly focused on a single application or a specific configurable architecture [1]. The absence of general-purpose, high-level programming tools for adaptive computing applications has hampered the widespread adoption of this technology; currently, this area is only accessible to a very small collection of specially trained individuals.

This paper describes DEFACTO, an end-to-end design environment for developing applications mapped to adaptive computing architectures. A user of DEFACTO develops an application in a high-level programming language such as C, possibly augmented by pragmas that specify variable arithmetic precision and timing requirements. The system maps this application to an adaptive computing architecture that consists of multiple FPGAs as coprocessors to a conventional general-purpose processor. Other inputs to the system include a description of the architecture (e.g., how many FPGAs, communication time and bandwidth), and application-specific information such as representative program inputs.

DEFACTO leverages parallelizing compiler technology based on the Stanford SUIF compiler. While much existing compiler technology is directly applicable to this domain, adaptive computing environments present new challenges to a compiler, particularly the requirement of defining or selecting the functionality of the target architecture. Thus, a design environment for adaptive computing must also leverage CAD research to manage mapping configurable computations to actual hardware. DEFACTO combines compiler technology, CAD environments and techniques specially developed for adaptive computing in a single system.

The remainder of the paper is organized into four sections and a conclusion. In the next section, we present an overview of DEFACTO. Section 3 describes the system-level compilation based on the Stanford SUIF compiler.

1. Department of Electrical Engineering-Systems, Computer Engineering, University of Southern California. Hughes Aircraft Electrical Engineering Center, Los Angeles, CA 90089. E-mails: {kiran,hziegler}@usc.edu

2. Information Sciences Institute, University of Southern California. 4676 Admiralty Way, Suite 1001, Marina del Rey, CA 90292. E-mails: {pedro,granacki,mhall,rajeev}@isi.edu

3. Angeles Design Systems. 2 N. First Street, Suite 400, San Jose, CA 95113. E-mail: duncan@angeles.com

Section 4 presents the design manager, the tool that brings together the system-level compiler and commercial synthesis tools. Section 5 presents related work in high-level tools for adaptive computing.

2.0 System Overview

Figure 1 outlines the flow of information in the DEFACTO environment. The input for DEFACTO consists of an algorithm specification written in a high-level language such as C or MATLAB and augmented with domain and application-specific annotations. These annotations include variable precision, arithmetic operation semantics and timing constraints.

The target architecture for DEFACTO consists of a single general-purpose processor and multiple configurable computing units (CCUs) as a distributed memory machine. Each CCU contains some configurable logic and can access its own memory and communicate with the GPP and other CCUs via data and control channels. In this architecture, the general-purpose processor is responsible for orchestrating the execution of the CCUs by managing the flow of data and control in the application execution. An architecture description language is used to describe the specific features of the communication channels (e.g., number of channels per CCU and their bandwidth) and topology of the connections between the various CCUs; the amount and number of ports on each CCU and the CCU capacity in terms of logic elements. This architecture specification is used by DEFACTO to evaluate possible program partitioning strategies and to consider their relative merits.

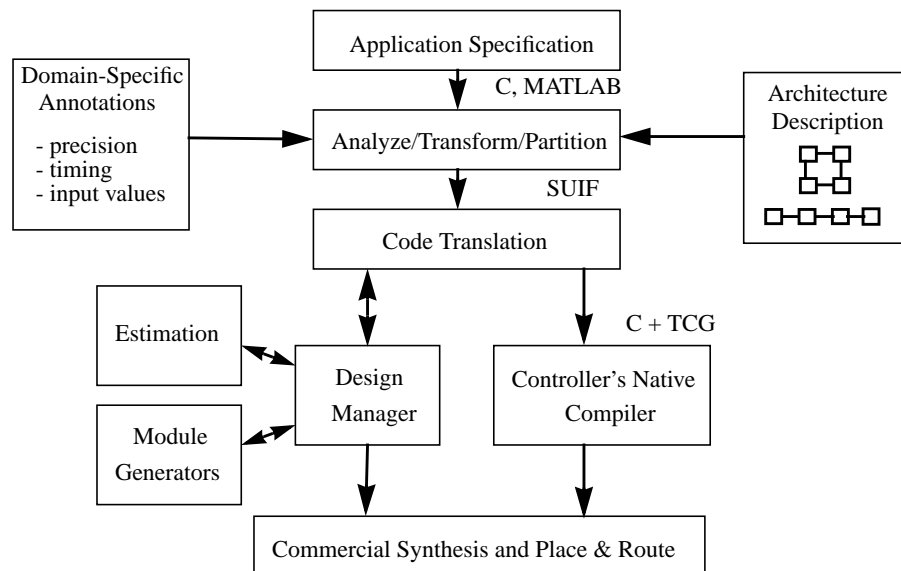


FIGURE 1. DEFACTO Compilation Flow Outline.

The DEFACTO compiler uses a standard front-end to convert the input program specification to SUIF's intermediate representation. In this translation, the system-level compiler preserves any pragmas the programmer has included regarding variable precision and timing constraints. The system-level compiler then analyzes the input program for opportunities to exploit parallelism. Next the compiler generates an abstract representation of the parallelism and communication. This representation consists of the partitioning of tasks from the input program, specifying which execute on CCUs and which execute on the general-purpose processor, and capturing communication and control among tasks. When the compiler has derived a feasible program partitioning, it generates C source code that executes on the general-purpose processor and an HDL representation (i.e., a vendor-neutral hardware description language) of the portions of the computation that execute on the configurable computing units. While the C portion is translated to machine code by the GPP's native compiler, the portions of the computation that execute on the CCUs provides input to the design manager, and subsequently, commercial synthesis tools, to generate the appropriate bitstreams representing the configurations.

Because the compiler needs to determine a feasible allocation of the configurable resources, the code partitioning phase is iterative. During this phase, the system-level compiler interfaces with the design manager component. The design manager is responsible for determining that a given partition of the computation between the general-purpose processor and the CCUs is feasible, i.e., the resources allocated to the CCU actually fit into the hardware resources. For this purpose, the design manager makes use of two additional tools, the *estimation tool* and the *module generator* component. The estimation tool is responsible for estimating the hardware resources a given computation requires while the module generator uses a set of parametrized library modules that implement predefined functions.

An overall optimization algorithm controls the flow of information in the DEFACTO system. The compiler iterates on its selection of partitions and program transformations until it has satisfied the hardware resource constraints and the application timing constraints subject to the semantics of the original program. The optimization goal of the system is to derive a complete design that correctly implements the application within the constraints imposed by the architecture platform, minimizes overall execution time, avoids reconfiguration and meets any timing constraints specified by the programmer.

3.0 System-Level Compiler

The DEFACTO system-level compiler uses traditional and parallelizing compiler technology to perform transformations and partition the original input program specification. The system-level compilation has the following goals:

- Identify computations that have the potential to yield performance benefits by executing on a CCU;
- Partition computation and control among the general-purpose processor and the CCUs.
- Manage data locality, communication and storage within and across multiple CCUs;
- Identify opportunities for configuration reuse in time and space.

In the first phase, the system-level compiler performs analysis to identify the parallelization opportunities in the program and the corresponding data dependences between parallel tasks [8]. The output of this first phase is a hierarchical source program representation specifying the parallel tasks and their associated communication. Subsequently, this representation is refined to partition the computation and control among CCUs and the general-purpose processor, and identify the communication and storage requirements. This refined representation is the input to the design manager. We now describe how the compiler identifies and exploits transformation opportunities using a simple code example depicted in Figure 2 (left).

After the input code, shown in Figure 2 (left) has been transformed into the SUIF intermediate representation, the system level compiler derives an abstraction of the input code in the form of a hierarchical task communication graph (TCG) similar to the one described in [9] and labelled in an hierarchical fashion by loop nest identifier and nesting level. As part of this TCG derivation, the compiler identifies individual tasks, and define the data flow and data dependence between tasks at each level of the representation. These analyses are drawn from parallelizing compiler techniques [8]. In particular, the compiler identifies the loops whose bodies can be executed independently and whose computational functions can be implemented by a single CCU. These computation representations are described by the parameterized functions F1 through F3.

Figure 2 (right) shows the tasks in the TCG generated for the example code. At the highest level of the hierarchy we have a single task that encompasses the entire computation. Next we have a single task G1 that accounts for the whole outer-loop body. Because this body has a conditional statement, it is not possible to derive a simple closed-form functional representation of the computation. In the next level we have several tasks H1 through H5 for each of the statements (simple and compound) in the loop body of G1. These tasks represent the execution of the two statements and three nested loops. At each of these loops, it is possible to derive functions representing the loops' computations as depicted by the functions F1, F3 and F5. The compiler analysis phase also annotates the TCG (at each level of the representation) with data dependence information. For example, at the G1 task level, the compiler

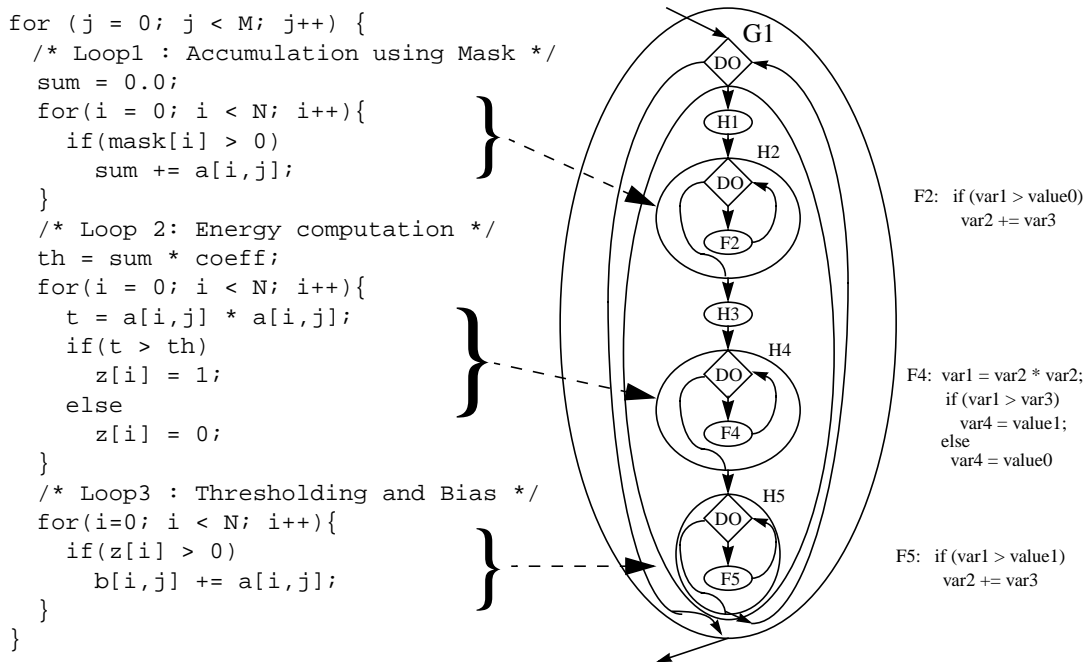


FIGURE 2. Example Program and its Hierarchical Task Graph (TCG) Representation.

will determine by data dependence analysis that the *sum*, *z* and *th* variables are privatizable, that is no data stored in any of these variables flows from one iteration of the outer-most loop to another iteration of the same loop. These task-level data dependence annotations will help later phases of the compiler to relax the scheduling constraints on the execution of the computation. For this particular example the code generation phase can explore parallelism or pipelining in the execution of the various iteration of the task G1 loop body.

3.1 Identifying Configurable Computing Computations

Fine-grain parallel loops are well-suited for execution on a CCU because their bodies can be replicated for parallel execution without incurring a large hardware cost. Parallelization analysis in the SUIF system locates loops whose iterations can be executed in parallel through the use of data dependence analysis and privatization and reduction recognition [8]. The compiler identifies parallel loops, including vector-style SIMD computations, more general parallel loops that follow multiple threads of control, and pipelined parallel loops. In addition to these parallelization analyses, the DEFACTO compiler can also exploit partial evaluation, constant folding and special arithmetic formats to generate specialized versions of a given loop body.

In the example above loop 1 performs a sum reduction (the successive application of an associative operator over a set of values to produce a single value), producing the value in the *sum* variable. Because of the commutativity and associativity of the addition operator, the compiler can execute in any order the accumulations in the *sum* variable or even compute multiple partial summations in parallel. Loops 2 and 3 are easily parallelizable as they have no loop-carried dependences (any two iterations of these loops access mutually exclusive data locations. At the outermost loop level, all iterations of the loop can execute in parallel as well provided that variables *sum*, *th* and the array *z* are privatized. That is, values stored in these variables do not flow across iterations. As a result, each iteration can use a private copy.

3.2 Locality and Communication Requirements

Based on the representation and the data dependence information gathered for the program for a particular loop level in a nest, the compiler evaluates the cost associated with the movement of data and the possible improvement through the execution of the tasks on CCUs. For a multi-CCU architecture, we can use the data partitioning analysis described in [11] augmented with reconfigurable-specific constraints to determine which partitions of the data can

be accommodated that result in minimal communication and synchronization. These data partitions are subject to the fact that the corresponding loops allow their functional representation to be fully incorporated on a single CCU.

The compiler has an abstract or logical view of communication channels that it can use to determine the impact of I/O bandwidth on partitioning and therefore on performance and size. For example, if the configurable machine provides for a bandwidth of 20 bits between CCUs with a transfer rate of 10 Mbps, the compiler can use these constraints to define the partition. However, the compiler is not concerned with the hardware details such as the physical names of each of the 20-bit lines, and the hardware control signals needed to initiate a specific communication event – those details are provided by the design manager.

For this particular example, assuming a multi-CCU architecture, it is possible to split the data in the b , z and a arrays by columns and privatize the variable sum , so that all iterations of the outer-most loop could proceed concurrently and are assigned to different CCUs in blocks as defined in loop-scheduling techniques [10]. Figure 3 below illustrates a possible data and computation partition for our code example. In this figure we have represented each configuration C associated with a given set of hardware functions F.

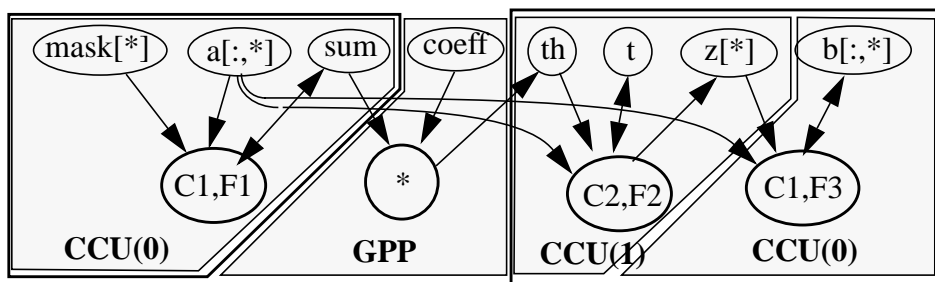


FIGURE 3. Data and Computation Partition for the Example Code in Figure 2.

3.3 Generating Control for the Partition

The final program implementation cannot operate correctly without a control mechanism. The control keeps track of which data is needed by a certain component and at what time that data must be available to a certain computation. Therefore, the system control has a dual role; it is both a communication and a computation coordinator. As a communication coordinator, the control takes the form of a finite state machine indicating data movement among the CCUs and the GPP. As a computation coordinator, the control takes the form of a finite state machine indicating when CCU configuration and task computation occur. In effect, the control captures the overall task scheduling. The actual implementation of these finite state machines is distributed among CCUs, initially in HDL format, and on the GPP, in C code.

As part of the partitioning process, the system-level compiler must annotate the TCG to indicate which tasks run on which CCU or GPP. The system-level compiler then starts the automated generation of the control by using the TCG to generate corresponding finite state machines (FSMs) for each of the CCUs and the GPP. The GPP FSM for task F1 is shown in Figure 4. Each node in the FSM represents an action or set of actions, such as sending or receiving data, or performing a computation. The edges represent transitions between the nodes. The transition from one node to another occurs based on one or more of the following: an action completion signal, a clock cycle or a counter incrementing. The GPP FSM contains nodes representing an initial CCU configuration and the transmission of program input data to the CCUs' local storage.

The GPP control is implemented using a set of C library routines that are architecture specific. The library consists of low-level interrupt handlers and calling routines that are associated with the target architecture. The C library also contains synchronization utilities. To complete the system-level compiler's role in the control generation, it translates the control information for the GPP into calls such as send, receive and wait, using the C library, and inserts these into the GPP C code. The system-level compiler also generates HDL for each of the CCUs' control FSMs. This HDL is used in the behavioral synthesis phase of the control generation.

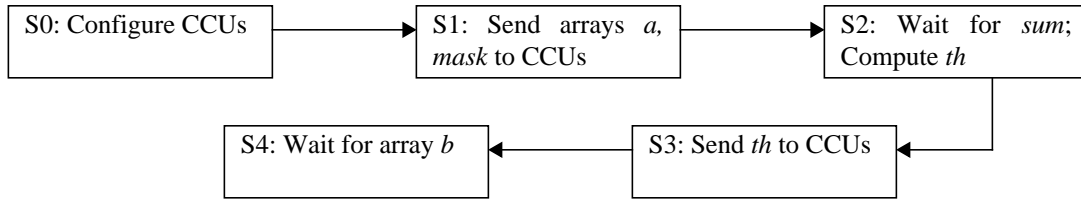


FIGURE 4. GPP Control FSM Representation

3.4 Configuration Reuse Analysis

Because reconfiguration time for current reconfigurable commodity parts is still very slow (on the order of tens of milliseconds), minimizing the number of reconfigurations a given computation requires is the most important optimization. For the example code in Figure 2, it is possible to reuse a portion of the hardware that executes task F1 in the loop 1 body for task H3. Although the data inputs of both these loops are different, the loop body functions are structurally the same.¹ For the example, this information may be captured in the TCG by relabelling task F3 with task F1. Identifying common configuration reuse is only part of the solution. The compiler also uses the dependence analysis information to bring together (by statement reordering) portions of code that use common configurations. This strategy aims at minimizing the total number of configurations, and thus, minimizing reconfiguration needed for a given computation.

4.0 Design Manager

The design manager provides an abstraction of the hardware details to the system-level compiler, and guides derivation of the final hardware design for computation, communication, storage, and control. In most cases, there will be a pre-defined configurable computing platform, where the available CCUs, storage elements (e.g., SRAM) and communication paths are all known a priori. In this case, the main responsibilities of the design manager are to map the computation, storage and communication from the high-level task abstractions to these pre-defined characteristics.

The design manager operates in two modes, designer and implementer. In the initial compilation or partitioning phase, the design manager estimates the feasibility of implementing specific tasks to be executed on a CCU as generated by the system-level compiler. Working in tandem with the estimation tool, the design manager reduces the number of time-consuming iterations of logic synthesis and CCU place-and-route to find a feasible design. If a feasible partition is not presented, the design manager provides feedback to the system-level compiler to guide it in further optimization of the design, possibly resulting in a new partition.

When a feasible design is identified, the design manager enters into its second mode of operation, that of implementer. During this phase, the design manager interfaces with behavioral synthesis tools, providing them with both HDL and other target-specific inputs, culminating in the generation of the FPGA configuration bitstreams.

For example, for the TCG in Figure 2 we need to define the physical location, at each point in time, of the arrays a and z , how they are accessed, and how the data is communicated from the storage to the appropriate CCU. We now address these mapping issues in a platform-independent manner.

1. Notice that although the output variables in both F1 and F3 are, in the source code, respectively a scalar and an array this does not constitute a problem. Because these functions are implemented in hardware the compiler generated addresses bind the sequence of values produced by the hardware to memory location. While in the scalar case the address generated is always the same, in the second case the sequence of address is an affine memory access function. The hardware that implements the function is however the same.

4.1 Computation and Data Mapping

During the partitioning phase, the design manager provides feedback to the system-level compiler about the feasibility of a particular partitioning. To do this, the design manager presents to the estimation tool an RTL HDL representation of the partition. The design manager uses the system-level compiler-generated HDL associated with each task or group of tasks as input to a behavioral synthesis tool such as Monet to generate an RTL HDL description. Or if the system-level compiler specifies an HDL module binding, the design manager can invoke the corresponding module representation in order to generate the RTL HDL. This representation will have captured both the details of the computation and data movement as specified in the TCG. In response to the design manager's request, the estimation tool returns both a timing and a sizing estimate. The design manager then returns these estimates to the system-level compiler to help guide further efforts to find a partition that satisfies all of the constraints if the current one does not do so.

We now examine the example code in Figure 2 assuming we have an architecture with 2 CCUs, each of which can only hold, at most, two of the three functions F1 through F3. With the goal to minimize the number of configurations for the whole computation, while meeting all other constraints, the system-level compiler might generate two (non-disjoint) configurations. Configuration C1 holds the functions F1 and F2, and Configuration C2 holds functions F2 and F3. Notice that the single multiplication statement can in fact be accomplished by the function F2 if the output is extracted from the register *Reg3*. This type of partial reuse of configuration is also part of the system-level compiler's effort to minimize the total number of configurations. Figure 5 below illustrates a schematic representation of the implementation of each of the functions. For these functions the behavioral synthesis tool has inserted temporary registers into the design in order to allow the pipelined execution of the functions as specified by the system-level compiler (of importance when the data has to be fetched from the local CCU memory and the number of ports on the SRAM is limited).

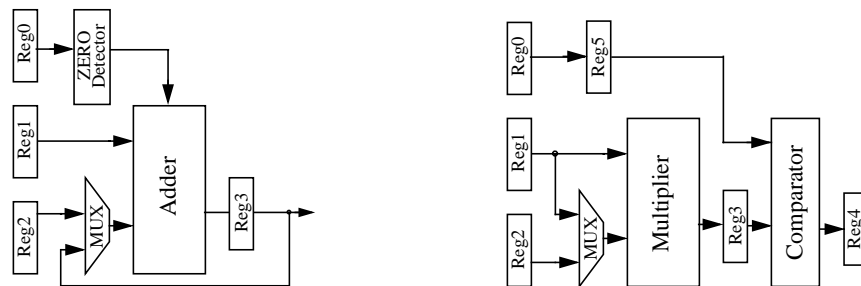


FIGURE 5. Schematic Implementation of Functions F1 (left) and F2 (right)

Given configurations C1 and C2, the design manager next evaluates the feasibility of the partition the system-level compiler has generated. For our example, the system-level compiler has partitioned the data into columns for the arrays *a* and *b*. For array *z*, the system-level compiler has allocated it to the memory of the first CCU. Figure 6 below illustrates the conceptual partitioning of the computation and identifies which memories of which CCU hold which data. An RTL HDL representation is passed to the estimation tool which returns timing and sizing estimates. If these are within the system-level constraints, the design manager then switches to its second mode of operation in order to generate the FPGA configuration bitstreams.

4.2 Estimation

One of the most important aspects of the interaction between the system-level compiler and the back-end implementation tools involves determining if the partition to perform a particular set of tasks will fit on a CCU. The *estimation engine* and *off-line estimation manager* serve a critical role in providing this information to the system-level compiler via a design manager interface. The design manager gives to the estimation engine a specific task and expects the estimation engine to return three pieces of information: the estimated CLB count, timing, and threshold information specific to the FPGA technology. The estimates should also consider routing for a specified task placement. The off-line estimation manager maintains a table of useful estimates to allow for more expedient estimation.

4.3 Communication Mapping

While the system-level compiler uses virtual communication channels, the design manager implements the physical communication channels required for data communication among the CCUs, general-purpose processor and memory elements. Data communication between the tasks must be mapped onto the predefined interconnect for target architecture. While some of the channels can be implemented as part of the pipelined execution of the functions assigned to different CCUs, other channels require buffering. In the former case, the design manager must add hardware handshaking circuits to the configuration specification. In the latter, the design manager generates circuits that temporarily store data in local memory and later ship it to another CCU's memory.

For example, if there is a 20 bit channel between CCUs and 4 bit words need to be communicated between the CCUs according to the TCG specification, the design manager can implement a serial/parallel converter at the CCU I/O port to communicate 5 values in parallel. On the other hand, if 40 bit signals need to be communicated, the design manager implements a parallel/serial converter or multiplexer in the CCU I/O port.

Figure 5 illustrates a possible mapping of the computation and communication in the code example in Figure 2. In Figure 6, we have assumed 2 CCUs connected in a ring topology and a GPP that can have direct access to every CCU's memory via a bus. Overlaid in this architecture figure we have the communication channels the application uses and the assignment program variables to storage. Not shown is the set of channels that the design manager has to introduce to communicate data to and from the GPP's memory and between the CCU's memory. For our example, the columns of the array a need to be sent to the CCUs for these later units to compute the updates to the b array. In this particular example the design manager can make use of the physical communication channels that exist between the adjacent units for maximum performance. If no such channel exists in the target architecture the design manager always has as a fall back option to use the GPP's memory for implementing many-to-many communication. Partition into larger number of CCUs can be accomplished by partitioning the variables assigned to a single CCU into multiple parts and replicating the communication channels accordingly..

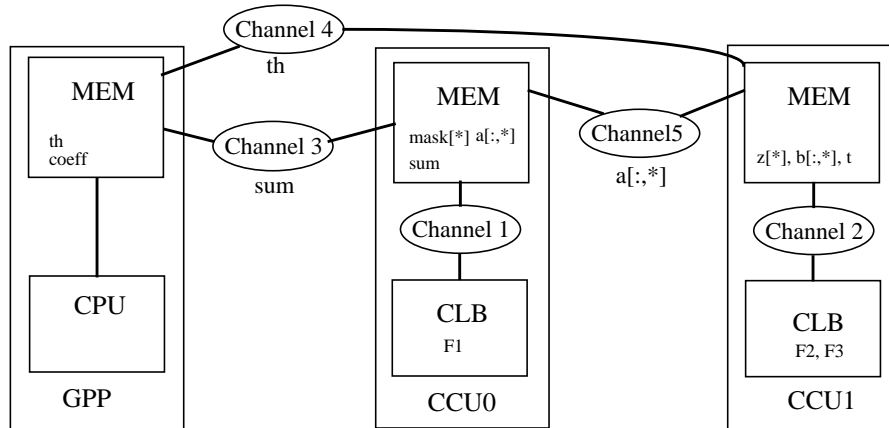


FIGURE 6. Computation and Communication Mapping to a 2 CCU Reconfigurable Architecture.

4.4 Storage Mapping

The system-level compiler specifies the data on which computation takes place as well as the communication buffers of data. However, it does not specify how or where the data is stored. Mapping data onto existing memories and corresponding addresses on each of the CCUs, as well as creating storage within the CCUs is the responsibility of the design manager. Note that when the compiler generates a partition, it needs to leave room in its estimate for the I/O circuits, control logic and storage circuits that the design manager adds. Thus in an initial iteration, the compiler produces a best guess of a partitioning that should fit on the available CCUs and the design manager estimates the cost of the I/O, control and storage within the CCU to provide feedback on the feasibility.

The compiler specifies initial storage of data (GPP memory, result of a previous configuration on a CCU, etc.) as part of the data dependence annotations associated with its partitions. When the system control initiates the execution of the TCG, data is needed on chip within the CCU. Without some a priori defined models of how the data may be stored on chip, it is difficult to generalize the synthesis of the storage. The storage design is linked to the control design and the communication channel design. The communication channels are feed the data to the CCU and transfer results from the CCU back to the GPP.

As an example of a design decision involved here, suppose that the tasks in a CCU need to operate on a matrix of data and tasks are executed sequentially on sub-matrices of the data. Further suppose that the TCG specifies parallel execution of the computation on all the data in each sub-matrix. In this case, it would be essential to access the sub-matrices sequentially from the external storage and store them on the CCU. Furthermore, the on-chip storage element on the CCU must be designed so that all the data stored can be accessed simultaneously (thus a sequentially accessed RAM is not feasible).

4.5 Control Mapping

While the compiler has derived the abstract control FSMs, capturing data flow among system components and task ordering in behavioral HDL, the specifics associated with the target architecture must still be integrated into the HDL description before synthesis can occur. This is somewhat analogous to the control generation phase performed by the compiler when it translated the control information into C library calls for the GPP. The design manager is responsible for refining the HDL into register transfer level (RTL) HDL. To accomplish this, the design manager uses behavioral HDL library routines that have been written for the target architecture as well as task timing information obtained from the synthesized RTL HDL for each task.

For example, the bus architecture shown in Figure 6 suggests that each CCU configuration must include logic that captures a bus protocol necessary for sending and receiving data in this system. These HDL library routines plus any other control constructs needed to support the gluing together of overall program flow, such as reconfiguration logic, are added, by the design manager, to the RTL HDL.

5.0 Related Work

The SPLASH project [1] built a SIMD compiler for their FPGA board based architecture. The compiler translates code written in dbC, a bit parallel version of C, into a specialized SIMD engine in structural and behavioral VHDL. The VHDL code is then compiled into hardware configurations and optimized using commercial synthesis tools. But the compiler could only translate SIMD style code and not generic input. The partitioning of the design between the various available FPGAs also had to be performed manually most of the time to achieve good performance.

The National Semiconductor NAPA project [2] targets exclusively the NAPA 1000 architecture. The architecture consists of a RISC core and configurable logic packaged as a system on a chip. The NAPA C language provides pragma directives for the user to specify the computation and data partitioning between the processor, configurable logic and various memory modules in the architecture. Compiler optimizations include synthesis of hardware pipelines from pipelineable loops. NAPA C targets only the NAPA architecture and relies heavily on user supplied hints to partition the computation and the data.

The RaPid compiler project [3] is geared for coarse grained, reconfigurable pipeline architecture. The programmer specifies the pipelined computation in a high-level language (RaPiD-C) and uses the compiler to generate the control required to execute the computation on the RaPiD pipeline. RaPiD compiler is very restricted and compiles only for the pipeline datapath with coarse grained functional units.

Hartenstein et. al. developed the Xputer paradigm and an operating system for machines based on the paradigm [4]. CoDe-X takes as input a C like application program and compiles and executes on Xputer hardware. The framework does the partitioning, compiling and library mapping of the application tasks. Data scheduling to

improve performance is also addressed. CoDe-X is an end-to-end software system but the compilation environment is restricted to the Xputer architecture.

The Reconfigurable Architecture Workstation (RAW) [5] is studying a systolic-array-like tiled architecture whose components are hybrid RISC/FPGA processors. They are studying complex compiler optimizations to exploit instruction-level parallelism, and to partition and map computation to the array of tiles. The target for compilation in the RAW project is very different from a general co-processor architecture as being envisioned in the DEFACTO project.

The BRASS project [6] is designing a hybrid MIPS architecture that includes a reconfigurable coprocessor on the same chip. The coprocessor is an array of small functional units which can be reconfigured at a high rate. Their compiler techniques focus on generating configurations by mapping dataflow graph nodes to library modules. The BRASS compiler exploits instruction level parallelism and data flow graph mapping by using library of components geared for their architecture.

Pande et. al. have developed heuristic techniques for scheduling loops onto reconfigurable architectures. They focus on exploiting operator parallelism within loop nests. The Program Dependence Graph (PDG) [12] is analyzed to determine cut-sets (and corresponding configurations) which reduce the reconfiguration cost. But their scope is limited in terms of identifying the opportunities for mapping computations onto the configurable logic. Also, the management of configurations to actually map the computation onto the architecture is not addressed.

Though the projects mentioned above address some of the issues in compiling for adaptive architectures, they do not encompass all the required components of a full-fledged compiler framework for generic adaptive architectures. DEFACTO is an end-to-end design environment where the application is mapped to the architecture automatically. The input is an application in a high-level language such as C, with additional pragmas to guide the optimizations. The partitioning and mapping of the application is performed automatically by the compiler without user intervention. The compiler automatically identifies the computations which can be mapped onto the configurable logic and also performs optimizations such as identification reusable configurations etc. The target architecture is a general-purpose processor with FPGA logic as co-processor. The DEFACTO compiler is not architecture specific and can take the description of the architecture as an input.

6.0 Conclusions

This paper has presented an overview of DEFACTO, a design environment for implementing applications for adaptive computing systems. The DEFACTO system uniquely combines parallelizing compiler technology with synthesis to automate the mapping of application to reconfigurable computing platforms. As the project is still in its early stages, this paper has focused on the necessary components of such a system and their required functionality. The system-level compiler uses parallelization and locality analysis to partition the application between a general-purpose processor and CCUs. The design manager maps from the compiler's partitioning to the actual configurations on the FPGA, including the required communication, storage, and control for the CCUs and the GPP. A key distinguishing feature of DEFACTO is that it is designed to be architecture independent and retargetable.

7.0 Bibliography

- [1] D. Buell, J. Arnold and W. Kleinfelder, "Splash 2: FPGAs in a Custom Computing Machine", *IEEE Symposium on FPGAs for Custom Computing Machines*, Computer Society Press, Los Alamitos CA, 1996.
- [2] M. Gokhale and J. Stone, "NAPA C: Compiling for a Hybrid RISC/FPGA Architecture", *IEEE Symposium on FPGAs for Custom Computing Machines*, Computer Society Press, Los Alamitos CA, April, 1997.
- [3] C. Ebeling and D. Cronquist and P. Franklin, "RaPiD - Reconfigurable Pipelined Datapath", In *Proceedings of the 6th International Workshop on Field-Programmable Logic and Applications*, 1996.
- [4] R. Kress and R. Hartenstein and U. Nageldinger, "An Operating System for Custom Computing Machines based on the Xputer Paradigm", In *Proceedings of the 7th International Workshop on Field-Programmable Logic and Applications*, 1997.
- [5] E. Waingold, M. Taylor, D. Srikrishna, V. Sarkar, W. Lee, V. Lee, J. Kim, M. Frank, P. Finch, R. Barua, J. Babb, S. Amarasinghe, and A. Agarwal. "Baring it all to Software: Raw Machines", *IEEE Computer*, pp. 86-93, Sept. 1997.

- [6] J. Hauser and J. Wawrzynek, "Garp: A MIPS Processor with a Reconfigurable Coprocessor", *IEEE Symposium on FPGAs for Custom Computing Machines*, Computer Society Press, Los Alamitos CA, April, 1997.
- [7] N. Ramasubramanian and R. Subramanian and S. Pande, "Automatic Analysis of Loops to Exploit Operator Parallelism on Reconfigurable Systems", In *Proceedings of the 11th International Workshop on Languages and Compilers for Parallel Computing*, Springer-Verlag, New York, August, 1998.
- [8] Hall, M et al., "Maximizing Multiprocessor Performance with the SUIF Compiler", *IEEE Computer*, IEEE Computer Society Press, Los Alamitos CA, Dec. 1996.
- [9] Girkar M and Polychronopoulos, "Automatic Detection of Task Parallelism in Sequential Programs", *IEEE Transactions of Parallel and Distributed Systems*, Vol. 3., No. 2., March 1992.
- [10] Polychronopoulos, C. and Kuck D., "Guided-Self-Scheduling A Practical Scheduling Scheme for Parallel Computers", *ACM Transactions on Computers*, Vol. 12, No. 36., pp. 1425-1439, Dec. 1987.
- [11] Anderson J. and Lam, M., "Global Optimizations for Parallelism and Locality on Scalable Parallel Machines", In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'93)*, pp. 112-125, ACM Press, NY, July 1993.
- [12] J. Ferrante and K. Ottenstein and J. Warren, "The Program Dependence Graph and Its Use in Optimization", *ACM Transaction on Programming Languages and Systems*, Vol. 9, No. 3, pp. 319-349, Jul, 1987.