DEFACTO: A Design Environment for Adaptive Computing Technology

Kiran Bondalapati[†], Pedro Diniz[‡], Phillip Duncan[§], John Granacki[‡], Mary Hall[‡], Rajeev Jain[‡], Heidi Ziegler[†]

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. 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 Introduction

Adaptive computing systems incorporating configurable computing units (CCUs) (*e.g.,* FPGAS) 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, mapping programs to adaptive computing systems 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 tools for application developer are being designed, these have been narrowly focused on a single application or a specific configurable architecture [1] or require new programming languages and computing paradigms [2]. 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 with application-specific information. The system maps this application to an adaptive computing architecture that consists of multiple FPGAs as coprocessors to a conventional general-purpose processor.

DEFACTO leverages parallelizing compiler technology based on the Stanford SUIF compiler. While much existing compiler technology is directly applicable

† Department of Electrical Engineering-Systems, University of Southern California, Los Angeles, CA 90089, email: {kiran,hziegler}@usc.edu

‡ Information Sciences Institute, University of Southern California, 4676 Admiralty Way, Marina del Rey, CA 90292. email: {pedro,granacki,mhall,rajeev}@isi.edu

§ Angeles Design Systems. 2 N. First Street, Suite 400, San Jose, CA 95113. email: duncan@angeles.com

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 techniques 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 three 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. Section 4 presents the design manager, the tool that brings together the system-level compiler and commercial synthesis tools.

2 System Overview

The target architecture for DEFACTO consists of a single general-purpose processor (GPP) and multiple configurable computing units (CCUs). Each CCU 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 CCUs; the amount and number of ports on each CCU and the CCU logic capacity. This architecture specification is used by DEFACTO to evaluate possible program partitioning strategies.

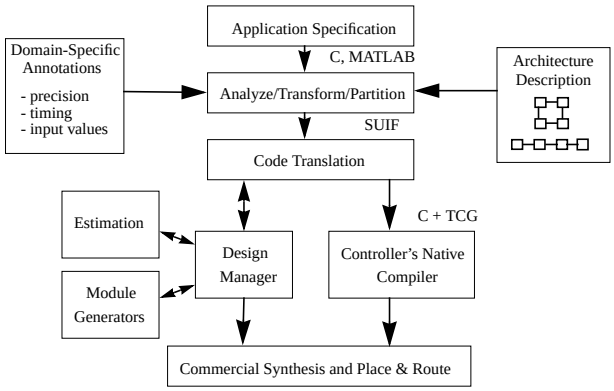

Fig. 1. DEFACTO Compilation Flow Outline

Figure 1 outlines the flow of information in the DEFACTO environment. The input for DEFACTO consists of an application 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 DEFACTO system-level compiler, represented by the Analyze/Transform/Partition box in Figure 1, first converts the application specification to SUIF's intermediate representation. In this translation, the system-level compiler preserves any application-specific pragmas the programmer has included. 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 GPP, 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 GPP 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.

The code partitioning phase is iterative, whereby 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 GPP 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 and module generator components. The estimation tool is responsible for estimating the hardware resources a given computation requires while the module generator uses a set of parameterized library modules that implement predefined functions.

An overall optimization algorithm controls the flow of information in the DEFACTO system. The optimization goal 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. 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.

3 System-Level Compiler

The DEFACTO system-level compiler 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 GPP and the CCUs.
- Manage data storage and communication within and across multiple CCUs.
- Identify opportunities for configuration reuse in time and space.

The compiler analysis produces an annotated hierarchical source program representation, which we call the task communication graph (TCG), specifying

the parallel tasks and their associated communication. Subsequently, this representation is refined to partition the computation and control among CCUs and the GPP, 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).

```
for (j = 0; j < M; j++) {
  /* Loop1 : Accumulation using Mask */
  sum = 0.0;
  for(i = 0; i < N; i++){
    if(mask[i] > 0)
      sum += a[i,j];
  }
  /* Loop 2: Energy computation */
  th = sum * coeff;
  for(i = 0; i < N; i++){
    t = a[i,j] * a[i,j];
    if(t > th)
      z[i] = 1;
    else
      z[i] = 0;
  }
  /* Loop3 : Thresholding and Bias */
  for(i=0; i < N; i++){
    if(z[i] > 0)
      b[i,j] += a[i,j];
  }
}
```

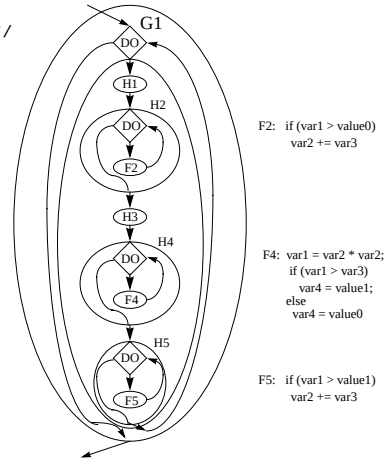

Fig. 2. Example program and its Hierarchical Task Communication Graph(TCG)

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, task G1 accounts for the outer-loop body. At the next level, tasks H1 through H5 represent each of the statements (simple and compound) in the loop body of G1, namely two statements and three nested loops. At each of the loops, it is possible to derive functions representing the loops' computations as depicted by the functions F2, F4 and F5. The compiler analysis phase also annotates the TCG at each level of the representation with data dependence and privatization information. For example, at G1, the compiler will annotate that `sum`, `z` and `th` can be safely made private to each iteration of the loop since values in the variables do not flow from one iteration of G1 to another. These task-level data dependence and privatization annotations will help later phases of the compiler to relax the scheduling constraints on the execution of the computation. For this example, the code generation phase can explore parallelism or pipelining in the execution of the iterations of task G1.

3.1 Identifying Configurable Computing Computations

The compiler identifies parallel loops, including vector-style SIMD computations, more general parallel loops that follow multiple threads of control, and pipelined parallel loops using existing array data dependence analysis, privatization and reduction recognition techniques [3]. 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, the loop corresponding to task H2 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. Tasks H4 and H5 are easily parallelizable as they have no loop-carried dependences (any two iterations of these loops access mutually exclusive data locations).

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 to determine which partitions of the data can be accommodated that result in minimal communication and synchronization [5]. These data partitions are subject to the additional constraint that the corresponding tasks that access the data can be fully implemented 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 partitioning.

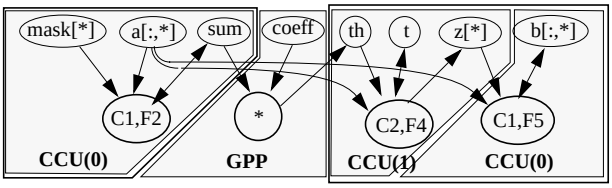

Fig. 3. Data and Computation Partition for the Example Code in Figure 2

For this example, it is possible to split the data in the `b`, `z` and `a` arrays by columns and privatize the variable `sum`, so that the iterations of the outer-most loop could be assigned to different CCUs in consecutive blocks and performed concurrently [4]. Figure 3 illustrates a possible data and computation partitioning for the code example. In this figure, we have represented each configuration C associated with a given set of hardware functions F.

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 (FSM) indicating data movement among the CCUs

and the GPP. As a computation coordinator, the control takes is represented by another FSM indicating when CCU configuration and task computation occur. In effect, the control captures the overall task scheduling. The actual implementation of these FSMs is distributed among CCUs, initially in HDL format, and the GPP, in C code.

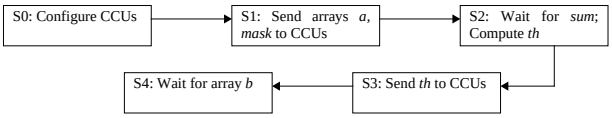

Fig. 4. GPP Control FSM Representation

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 compiler then starts the automated generation of the control by using the TCG to generate corresponding FSMs for each of the CCUs and the GPP. The GPP FSM for task F1 is shown in Figure 4.

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 communication and synchronization routines that are associated with the target architecture. 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 compiler also generates HDL for each of the CCUs' control FSMs. This HDL is used in the behavioral synthesis phase of the control generation.

3.4 Configuration Reuse Analysis

Because reconfiguration time for current reconfigurable devices is 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 F2 for task F5. Although the data inputs of these loop bodies are different, the functions are structurally the same. For the example, this information may be captured in the TCG by relabelling task F5 with task F2. 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.

4 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 partitioning phase, the design manager assists the system-level compiler by estimating the feasibility of implementing specific tasks to be executed on a CCU. 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. If a feasible partitioning 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 partitioning.

Once 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 storage to the appropriate CCU. We now address these mapping issues in a platform-independent manner.

4.1 Computation and Data Mapping

To provide feedback to the system-level compiler during the partitioning phase, the design manager collaborates with the estimation tool and (optionally) the module generator component. The design manager uses 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, and provides this HDL representation to the estimation tool. 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 compiler to help guide further efforts to find a partitioning that satisfies all of the constraints if the current one does not.

For our example, the compiler has partitioned the data into columns for the arrays `a` and `b`. The compiler has allocated array `z` to the memory of the first CCU. Figure 5 illustrates the conceptual partitioning of the computation and identifies which memories of which CCU hold which data The design manager captures this storage mapping in the RTL HDL for each CCU. If estimates are within the system-level constraints, the design manager generates the FPGA configuration bitstreams.

4.2 Estimation Tool

The design manager gives to the estimation tool a specific task in RTL HDL and expects the estimation tool to return three pieces of information: the estimated configurable logic block (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.

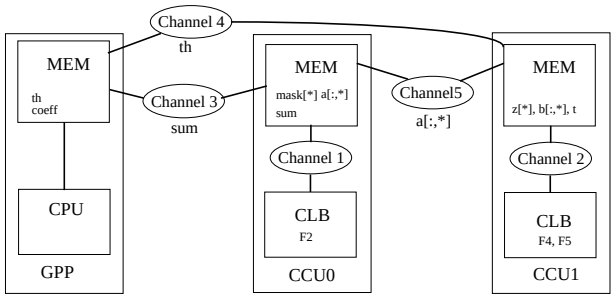

Fig. 5. Computation and Communication Mapping to 2 CCU Architecture

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, GPP and memory elements. Data communication between the tasks must be mapped onto the predefined interconnect for the 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 must 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. In this example, the columns of array `a` need to be sent to the CCUs for these later units to compute the updates to `b`. In addition the design manager can make use of the physical communication channels that exist between the adjacent units for maximum performance and rely on a GPP buffer-based scheme otherwise.

4.4 Storage Mapping

The system-level compiler specifies the data on which computation takes place as well as the communication 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 partitioning, the estimates must take into consideration 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 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 (*e.g.,* an 8×8-byte window of an image). 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.

5 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 effective mapping of applications 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.

Acknowledgements. This research has been supported by DARPA contract F30602-98-2-0113 and a Hughes Fellowship. The authors wish to thank John Villasenor and his students for their contributions to the design of the system.

References

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. Hall, M et al., "Maximizing Multiprocessor Performance with the SUIF Compiler," IEEE Computer, IEEE Computer Society Press, Los Alamitos CA, Dec. 1996.
4. 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.
5. 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.