Multithreaded Code from Synchronous Programs: Extracting Independent Threads for OpenMP

Daniel Baudisch, Jens Brandt, and Klaus Schneider
Department of Computer Science
University of Kaiserslautern, Germany
http://es.cs.uni-kl.de

Abstract—Synchronous languages offer a deterministic model of concurrency at the level of actions. However, essentially all compilers for synchronous languages compile these actions into a single thread by sophisticated methods to guarantee dynamic schedules for the sequential execution of these actions. In this paper, we present the compilation of synchronous programs to multi-threaded OpenMP-based C programs. We thereby start at the level of synchronous guarded actions which is a comfortable intermediate language for synchronous languages. In addition to the explicit parallelism given in the source program, our method also exploits the implicit parallelism which is due to the underlying synchronous model of computation and the data dependencies of the guarded actions. We show how viable tasks can be constructed from the actions of a program and show the feasibility of our approach by a small example.

I. INTRODUCTION

System description languages (like SystemC and SpecC), hardware description languages (like VHDL and Verilog) and synchronous languages [1] (like Esterel [2] and Quartz [6]) share some common characteristics: In particular, all of them have (1) a precise notion of time and (2) the possibility to describe systems consisting of parallel processes. Hence, developers can model parallelism in these languages explicitly by using corresponding statements of these languages. However, parallelism is not only given by this explicit model, but also by the notion of time that allows one to execute different actions at the same point of time. For instance, the execution of hardware description languages and SystemC distinguishes between an elaboration and an update phase. The elaboration phase executes the program, but does not yet update the variables, and the variables’ updates are done at once in the update phase. This allows one to perform multiple assignments in the same environment before the variables are assigned the new values. Similarly, synchronous languages distinguish between micro and macro steps of the execution. Micro steps do not take time which means that all variables remain constant during a macro step and synchronously change at the next macro step. Hence, all of these languages provide inherent parallelism between subsequent synchronous variable updates, in addition to the explicit one.

However, especially in the area of synchronous languages, research over the last two decades has put its focus on creating sequential code from the synchronous languages. Various compilers have been developed (see [3] for an overview): except for a translation to hardware circuits, they all target deterministic single-threaded code in order to directly execute synchronous programs on simple micro-controllers without using complex operating systems. Hence, generation of multi-threaded code is still an open problem, which has not yet been considered in this area with only a few exceptions: The KEP processor [4] is a special processor designed to execute synchronous programs at the machine code level. For this reason, its multithreaded implementations make use of the explicit parallelism given by the programmer. However, it cannot use the implicit parallelism. A lot of research on generating multi-threaded code has already been done by the development of the compiler of the language Signal. However, Signal is a polychronous language and therefore it is not based on the purely synchronous model of computation: Its weaker assumptions about the global timing (e. g. absence of a single global clock) allows the compiler to reorder events to some degree. In particular, weakly endochronous systems [5] still preserve the input-output behavior. Unfortunately, this approach cannot be applied in general to synchronous languages.

In this paper, we therefore consider the problem to generate multi-threaded C-code from synchronous guarded actions, which is a comfortable intermediate format for the compilation of synchronous programs [3], [7], [6]. Our contribution is a translation procedure from purely synchronous guarded actions to multi-threaded OpenMP-based C-code. This is the first approach to translate imperative synchronous programs to multi-threaded code for general-purpose multi-core processors. Thereby, our implementation is still deterministic – for the same inputs we get the same outputs – independent of the scheduling of the individual OpenMP threads. Although we focus on the compilation of synchronous guarded actions, the presented approach can be also used for other system description languages, in particular for SystemC programs with clocked threads and methods.

This paper is structured as follows: In Section II, we introduce synchronous guarded actions and analyze their data dependencies. Section III shows how they can be grouped into tasks, which are subsequently translated to C-code in Section IV. After illustrating our approach by an example in Section V, we finally list some experimental results and draw some preliminary conclusions.

II. SYNCHRONOUS GUARDED ACTIONS

In general, a guarded action has the form $\gamma \Rightarrow C$, where $\gamma$ is a boolean condition called the guard and $C$ is an action called the body of the guarded action. Actions are typically assignments to variables, but could also be assertions, procedure calls etc.
In the following, we restrict our consideration to assignments. The intuitive semantics of guarded actions is that the body action \( C \) is executed if the guard \( \gamma \) holds.

Thereby, we assume the synchronous model of computation, which follows the paradigm of perfect synchrony. Generally, this divides the execution of a program into micro and macro steps. The execution of micro steps is done in zero time, while the execution of a macro step requires one logical unit of time. Variables are constant during the execution of micro steps and synchronously change at macro steps. In the programmer’s view, all macro steps take the same amount of logical time. Hence, concurrent threads run in lockstep and automatically synchronize at macro steps.

Hence, the semantics of synchronous guarded actions is simply defined as follows: In every macro step, all guards are simultaneously checked. If a guard is true, its action is immediately executed. Thereby, actions are assignments of the form \( \gamma \Rightarrow A \). Both kinds of assignments evaluate the right-hand side expression \( \tau \) in the current environment/macro view, all macro steps take the same amount of logical time. Hence, concurrent threads run in lockstep and automatically synchronize at macro steps.

If guarded actions should be executed in a single sequential thread (or with a bounded number of threads), they have to be executed in data flow order to avoid read-after-write conflicts. Therefore, synthesis tools may create Action Dependency Graphs (ADGs), which are bipartite graphs consisting of vertices representing variables, vertices representing the guarded actions, and edges representing the dependencies between the actions and the variables. The edge \( (\gamma \Rightarrow A) \to x \) expresses that \( x \) is written by action \( A \), and the edge \( x \to (\gamma \Rightarrow A) \) expresses that \( x \) occurs in the guard \( \gamma \) or in the right-hand side of action \( A \). Thus, the graph can be derived by a syntactic analysis of the read and write sets of all actions.

From the graph, we can immediately derive the restrictions for the execution of the guarded actions of a synchronous system. An action can be only executed if all variables occurring in its guard and the right-hand side of the assignment (i.e., its read variables) are known. A variable is thereby known if all actions writing to it have already been executed in previous micro steps of the same macro step.

An extreme case is obtained if the graph consists of several unconnected components. Since they do not have any dependencies among each other, the corresponding parts of the graph do not need to be synchronized. Since the graph does not reflect the original threads of the synchronous program, the parallelism found in the program is, in general, independent of the use of threads.

III. GENERATING TASKS

The synchronization of threads usually requires a significant amount of time, which can easily exceed the time for actual computation. Hence, in order to get a significant speed-up from multi-threaded code, one should try to keep the synchronization effort as low as possible. To this end, our code generation algorithm uses the ADG as a starting point for a task structure, which represents a possible thread structure of the final OpenMP-based C-code.

Before we present the transformation, we make the following definitions: A task action is either a guarded action \( \gamma \Rightarrow A \) or a task team. A task team is a list of tasks. A task is list of task actions. A task structure is a task. The root element of a task structure is usually called the master task. It represents a program that starts with one thread containing an arbitrary sequence of actions. These correspond to selection statements for the guarded actions and forks for the task teams. Each member of the team may then look like any task again. They will join after termination, and the execution of the parent task resumes. An example is shown in Figure 1 (c) and (d).

In order to create a task structure from a given ADG, we translate its nodes as follows: Each guarded action represents a conditional statement in C, i.e., an if-statement containing a single assignment. Each variable represents a barrier for all tasks that may write to this variable. A special case appears if there is only one action that may write this variable immediately: then, no barrier is necessary.

For the actual creation of the task structure, i.e., which task actions are merged to a particular task, our algorithm maintains a set of open tasks, which are not yet closed by a trailing action in its parent thread. They are potential places, where actions can be appended. At the beginning, this set only contains the master task, which initially contains no task actions. Then, the following steps are repeated as long as there are still unscheduled actions in the graphs.

First, the actions that can fire are determined: these are the ones which only read variables that cannot be written by another action (i.e., input variables). Then, they are scheduled to the open tasks, which may require to synchronize some of the open tasks. This is needed in the following two cases: (1) An action reads variable \( x \), and two or more tasks may write to this variable. (2) An action reads two or more variables and each variable may be written by one or more actions.

In both cases, the concerning tasks have to be synchronized. In particular, the common parent tasks have to be determined. These tasks contain all affected tasks, i.e., after an arbitrary number of forks, all affected tasks are executed. From the set of the common parent tasks, the latest common parent task is selected, i.e., the task that does not contain other tasks from this set. This task has to be added to the set of open tasks and all of its subtasks have to be removed. After the indispensable synchronization has been introduced, the actions can be scheduled to tasks. An action has to be assigned to that task which writes variables that are read by the action. Due to the previous synchronization, each action is assigned to exactly one uniquely determined task.

At the end of the assignment of actions to tasks, a task may have no new actions, exactly one new action or more than one new action. ‘No action’ means that the task has not been modified. An assignment of one action just appends this action to the task. If two or more actions have been assigned to a task, this task has to be forked into a new team. In particular, a new team is added to the task. This team contains one new subtask for each action that has been assigned to the task.
start => l1 = true
l1 => x1 = true
l1 => x2 = true
l1 => next(l1) = true
l1 & x1 => y1 = true
l1 & x2 => y2 = true
l1 & !x2 => z1 = true
l1 & y1 & x2 => z1 = true
l1 & y1 & y2 => z2 = true

(bool start, l1, _next_l1, x1, x2, y1, y2, z1, z2;
if ( start ) l1=true;
#pragma omp parallel sections reduction (|: z1) {
#pragma omp section { if ( l1 ) _next_l1 = C(); }
#pragma omp section {
#pragma omp parallel sections {
#pragma omp section { if ( l1 ) x1 = A(); if (l1 && x1) y1 = D(); }
#pragma omp parallel sections {
#pragma omp section { if (l1) x2 = B();
#pragma omp parallel sections {
#pragma omp section { if (l1 && x2) y2 = E(); }
#pragma omp parallel sections {
#pragma omp section { if (l1 && !x2) z1 = F(); }
}
}
#pragma omp parallel sections {
#pragma omp parallel sections {
#pragma omp section { if (l1 && y1 && x2) z1 = true; }
#pragma omp parallel sections {
#pragma omp parallel sections {
#pragma omp section { if (l1 && y1 && y2) z2 = true; }
}
}
}
}
}
}
}

Fig. 1. Example: Guarded Actions, Dependency Graph, Task Graphs, OpenMP-Based C-Code
Each new subtask has to execute one of these actions, hence, each action is added to an empty subtask.

IV. OpenMP-Based C-Code

OpenMP is an API based on compiler directives for shared-memory parallel programming in C/C++ and Fortran on multiple platforms. In our translation, we make use of some of its directives. First, to create parallel code, we use the parallel directive, which marks a fork in the current thread and thereby creates a new team. To describe each of its members, we use the section directive. Second, to control race conditions between a team of threads, we use the OpenMP directives shared and reduction. The shared directive declares variables to be shared among all threads. In contrast, the reduction directive takes two parameters, a variable and an operation, and attributes each thread its own local copy of the variable. Thus, accesses do not require synchronization during the execution of the thread. Only when a thread terminates, it requests exclusive write access to the global variable. Then, it performs the reduction operation on the current value of the global variable and its local copy.

With these directives, generating OpenMP-based code from the task graph can be achieved fully structurally: Task actions are translated into blocks in the C-code. If the task action is a simple guarded action, a conditional statement is generated. If the task action is a team, we first generate code for it and insert it at this point. For task teams, we first generate code for all its members and then put each of them into a section directive of OpenMP, marking it as a separate thread. These parts are concatenated and finally enclosed by a sections directive. For all variables, which are shared among two or more members of the team, we need to add a reduction clause. Tasks, which consist of a list a task actions, are simply translated to a sequence of C statements. The final task structure just wraps the master task.

A particular challenge for the OpenMP programming model are interleaved dependencies as shown in Figure 1 (b). They can be handled in the following two ways: First, the barrier of a team is moved up to the barrier of its parent task, i.e. if a task has to be synchronized, all of its subtask will be closed. The second solution uses the duplication of actions to move a fork of a task out of its parent task. Hence, these subtasks do not have to be closed, if the origin parent task has to be synchronized. In general, the efficiency of these solutions depends on the ADG and the target platform.

V. Example

Figure 1 illustrates the process of translating synchronous programs to multithreaded C-programs. We start with the set of guarded actions generated by the compiler of our synchronous language (a). The read and write dependencies between the guarded actions build up the action dependency graph (b). The graph has interleaved dependencies, which can be either resolved by moving the barrier of a task before the barrier of its parent task (c) or by moving the fork of a team before the team’s parent task’s team (d). The generated C-code of task graph (d) is shown in (e).

Figure 2 shows some experimental results for this example. The benchmark was run on a Dual Quad Core Xeon using a Linux operating system. The programs were compiled using GCC 4.4.1, which comes with OpenMP 3.0. Furthermore, the number of OpenMP threads was statically set to 1, 2, 4, and 8, respectively. The graph shows the speedup of the runs with 2, 4 and 8 threads compared to the single-threaded run. The results show that speedups near to the number of cores can be achieved.

VI. Conclusions

In this paper, we presented a method to generate multithreaded C-code from synchronous programs. Using this approach, we are able to exploit both the explicit and the implicit parallelism of synchronous languages to utilize the computing capacity of shared-memory multi-core processors. Our method identifies concurrently running tasks from the action dependency graph constructed of guarded actions, and tries to keep the synchronization costs low. Experimental results have shown that our approach is feasible.

REFERENCES