A Model-Driven Approach for Software Parallelization

Margarete Sackmann Peter Ebraert and Dirk Janssens

Universiteit Antwerpen, Belgium
{margarete.sackmann, peter.ebraert, dirk.janssens}@ua.a.c.be

Abstract. In this paper, we describe a model-driven approach that aids the developer in parallelizing a software program. Information from a dynamic call tree, data dependencies and domain knowledge from the developer are combined in a model transformation process. This process leads to a model of the application where it becomes obvious which parts of the application can be executed in parallel.

1 Introduction

In the past, it was unnecessary to write programs that have the potential for parallel execution since most processors contained only one processing core and programs were therefore executed sequentially. Now, however, multiple processors are getting more and more popular, even for smaller devices such as routers or mobile phones, and it is crucial that the possibilities offered by the parallel processors are used well.

Despite this trend towards parallel processors, most programmers are still used to a sequential style of programming. It is difficult for them to judge where parallelization is both feasible and worthwhile. Even for programmers who are used to parallel programming, it is often less error-prone to start out from a sequential program rather than designing a parallel program from scratch, as errors that arise from the parallel execution can then be distinguished from other programming errors. In this paper, we propose a model-driven approach that aids the programmer in deciding where good opportunities for parallelization are in a program.

The starting point is the source code of a program from which a model of the behavior of the program is derived. This is done by profiling the execution of the program. In the resulting model, the developer may add information about the sequential order that may be required among program parts. Model transformations are then used to derive a parallel model of the application, taking into account both the information derived from profiling and the information added by the developer.

2 Related work

A popular way to parallelize programs is Thread Level Speculation (see for instance [1] or [2]). In this compiler-based approach, the program is parallelized

during execution. The parallelization is speculative which means that in some cases a part of the execution has to be repeated since there are dependencies that were not taken into account. This approach seems unfit for systems where resources such as energy are scarce, since parts of the application have to be executed more than once to take effect.

Other approaches, for instance [3], require specific hardware. The speedups for floating point operations are close to optimal, however porting the application to different platforms is not possible.

When trying to parallelize applications for different hardware platforms, an approach that focuses on the software is a better solution. Examples are Embla 2 [4] which links back the profiling information of the program to the source code to help the user identify possibilities for parallelizing the code. iPat [5] is an Emacs assistance tool for parallelizing with OpenMP. However, it is entirely text-based and provides no visual aid to the developer.

In [6], a model transformation process is used to generate parallel code. The difference to our approach is that information about possible parallelism in the program has to be added to the code as compiler directives. Via various model transformations, code can then be generated for different hardware platforms.

An approach that helps to extract a parallel model of the application is the Compaan/Laura approach [7]. It gives a more complete view of the program than the approach presented here since the extracted model is representative for any input data. However, it requires a massive restructuring of the code and an initial idea about the program behavior to help with this restructuring.

3 Approach

We propose a toolchain that focuses on the software so as to ensure portability of the code to different hardware platforms. The approach is based on model transformations and provides a visual interface to the developer. Several models of the application are used throughout the toolchain to represent the application at different levels of abstraction and with different levels of detail. The advantage in this context is that the approach is not bound to the profiling tools we use and that details that are not important for the parallelization can be hidden. The resulting parallel model can be used to parallelize the source code or to schedule the application on a multiprocessor platform. Parts of the application are represented as nodes in a graph, and dependencies between them – implying precedence relations – as edges in that graph. Thus, the developer gets a visual aid that helps him to understand the data and control flow in the program. The toolchain will be illustrated on the Fast Fourier Transform (FFT) application of the MiBench benchmark [8].

In Figure 1, an overview of the transformation process from source code to parallel model is given. In the next section, the different models that are used throughout the transformation, i.e. call trees, call graphs and the parallel model, are introduced. In Section 5, the transformation from source code to a parallel model of the application is explained. Section 6 shows how to implement

the model transformation in practice. The results of the transformation process for the FFT application are given in Section 7 and ideas for future work are explained in Section 8. We also explain some of the limitations of the approach, that can arise for instance from variable reuse or structural problems in the original program. Some conclusions are presented in Section 9.

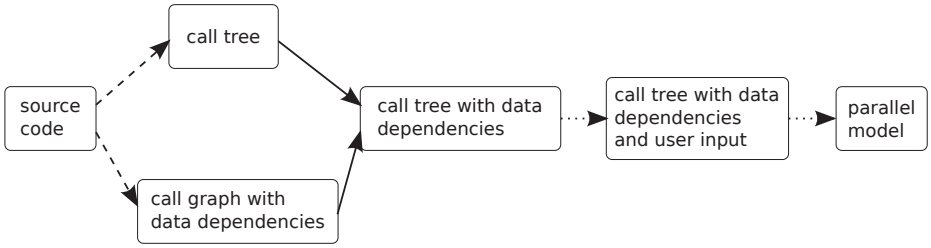


Fig. 1. The transformation process. Dashed lines stand for the use of a profiler. Solid lines represent a parsing and transformation step. Dotted lines represent model transformations.

4 Models used during the Transformation

In this section we present the models that can be seen in Figure 1: the parallel model that is the goal of the model transformation, the call tree model and the enriched call tree model with data dependencies and dependencies added by the developer that is used during the model transformation.

To abstract from implementation details, we chose to represent the program at the level of functions. This means that a single function execution is the smallest part of the application that can be discerned in the different models. In case that is too fine-grained, several functions can be grouped into coarser-grained modules. If a finer grain is required, functions have to be broken up: If the user for instance suspects that a loop could be parallelized, then the calculations that are done within the loop body should be put into a separate function.

4.1 Parallel Model

The goal in the transformation process is to derive a model of the program that shows which parts of the application can be executed in parallel. The Parallel Model (PM) that is the outcome of the transformation process is a graph-based model. The nodes represent function instances that occur during an execution of a program. The edges in the parallel model represent a precedence relation between functions. An edge from function instance f_1 to f_2 means that f_1 has to start execution before f_2. The more important information in this model

is however the lack of edges or a path between two functions. If there is no path between two function instances, they can be executed in parallel. This information can then be used for scheduling or to insert code to parallelize the program, for instance with OpenMP. A second type of edge, which we will call a subgraph edge, represents a stronger precedence relation. A subgraph edge from function f_1 to function f_2 means that f_1 and all its successors, i.e. its children, their children and so on, have to finish execution before f_2 can start execution.

The information provided by the subgraph edges can be used to get a greater flexibility of the model when the software has to be executed on different platforms with varying numbers of processors. In case the number of processors is small, a function f that is the source or target of a subgraph edge can be combined with all its successors into a subgraph that is scheduled as a whole. For a larger number of processors, the subgraph edges can be replaced by regular edges and more nodes can be distributed over the parallel processors.

4.2 Call Trees

To derive the parallel model, we start by analyzing the program code. One way to analyze the relation between the different functions is to track the calls from one function to another. This can either be done statically on the whole source code, or dynamically at run time. We chose for dynamic analysis, since it represents the actual execution of the program. During program execution, a profiler tracks which functions are called in which order. The resulting model, the call tree, is specific to a single execution of the application. Therefore, this approach works best for applications that always have a similar behavior (i.e. the same function calls are executed even if the actual data on which the calculations are performed is different), for instance Digital Signal Processing (DSP) applications such as digital filters or matrix multiplications. An advantage of using dynamic call trees is that functions that are not used during execution (and thus do not have to be distributed on the parallel platform) are not considered. Recent research has shown that in many circumstances dynamic analysis is better suited for parallelization than static analysis (see for instance [9]). A disadvantage of using dynamic analysis is that programs may be executed differently based for instance on the input. In that case, a system scenario [10] approach, where different types of execution behaviors are identified can be used.

Call trees can be represented as graphs, where the nodes are function instances and the edges are calls between the function instances. An excerpt of the call tree that is derived from the FFT application can be seen in Figure 2.

As can be seen in Figure 1, besides function calls we also add data dependencies and dependencies added by the developer to the call tree in our toolchain. A data dependency edge from a function f_1 to a function f_2 means that f_1 reads or writes data that is also read by f_2. This implies that at least some part of f_1 has to occur before f_2. Otherwise, a correct outcome of the program cannot be guaranteed. f_1 and f_2 can consequently not be executed in parallel. At the moment, we use the Paralax [11] compiler to automatically detect the data dependencies. It logs for each data element that is used in the program, such as

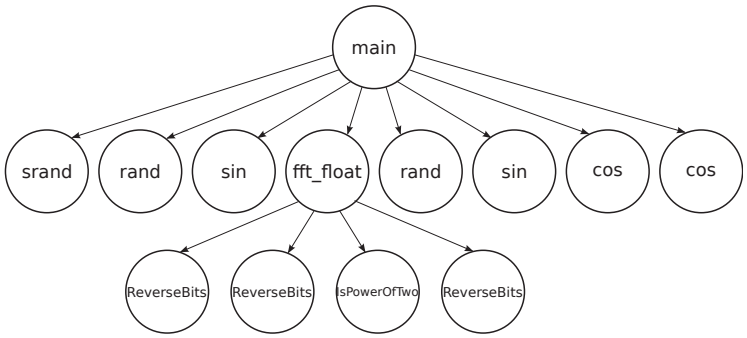


Fig. 2. Part of the call tree of the FFT application.

arrays or integer variables, which functions access this data in which order. This is done by tracking accesses to the address space of the data. Paralax also logs the access type, i.e. whether it is a read or write access. The read accesses are the critical ones and therefore the ones that are included into the model. The functions accessing a specific piece of data are identified by their call path, so that if a function is called from several other functions, a data dependency is only added if the whole call path matches. The data dependencies can be automatically included in the call tree model via a model transformation. To avoid loops, data dependencies are only added from a function f_1 to f_2 if f_1 occurred earlier in the call stack than f_2. The position in the call stack is recorded while obtaining the call tree and is a function attribute in the model transformation.

The third type of edge can be added by the user, in contrast to the function calls and data dependencies that can be detected and combined into a model automatically. The user can add edges between two function instances that are called by the same function instance. In Figure 2, the user can for instance add edges from the `srand` function instance to `rand`. An edge from f_1 to f_2 should be added if the execution of f_1 has to be finished before the execution of f_2 can start. Again, this implies that f_1 and f_2 cannot be executed in parallel. These extra edges can for instance be useful when the profiling tools could not detect all data dependencies due to the use of pointer arithmetics.

It could be argued that using call graphs rather than call trees would reduce the number of nodes. This approach is taken in [11], which we use to profile data dependencies. In a call graph, all instances of a function are combined into a single node. The problem with this approach is that the information contained in a call graph is not detailed enough. Consider for instance the call graph in Figure 3. The information that is included in this call graph is too limited to be able to parallelize the application properly. Many different executions can correspond to a call graph. In Figure 4 two call trees are shown that would both result in the call graph of Figure 3. For the parallelization step that is described

in the following chapter it is important however, that the exact instance of a function that calls another function is known.

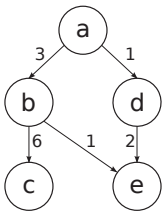
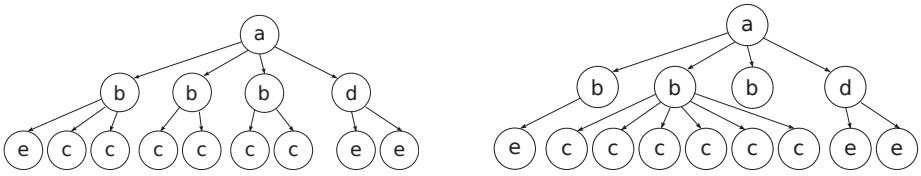


Fig. 3. A call graph



(a) Evenly distributed call tree

(b) One instance of b calls all instances of c

Fig. 4. Two possible call trees for the call graph in Figure 3

5 Transforming Source Code into a Parallel Model

In this section, the transformation process from the source code of an application to a parallel model of this application is described. The starting point is a dynamic call tree of the application (see 5.1) that is then enhanced with additional edges that represent data dependencies (5.2) and domain-specific knowledge of the user (5.3). The automatic transformation into the parallel model is described in Subsection 5.4. In Subsection 5.5, the usage of special edges that represent a dependency between subgraphs in the parallel model is explained.

We assume that, if a function f calls a function g, then there are no further computations (only possibly calls to other functions) in f after g returns. This form can be easily established in the source code: in case there are computations in f after the call to g, then these can be put into a new function h, and a call to h is added in f after the call to g.

5.1 Pruning the Call Tree

As explained above, the size of a call tree can be quite large. Even for simple programs, thousands of functions are called. However, many of these functions are library calls or other functions that the user does not have control over. Therefore, we concentrate on functions that either are part of the source code that the developer programmed, or that are called from within that source code.

Example 1. In Figure 5 a loop from the `main` routine of the FFT application is shown. The functions `sin` and `cos` are called inside a loop. Even though these functions were not implemented by the user, they can be responsible for a large amount of execution time. Additionally, the user can add code to parallelize the different calls to these functions. Therefore, these functions are included in the call tree, but the functions that are not visible in the source code are not included (for instance functions that are called by `sin`).

```
for(i=0;i<MAXSIZE;i++) {
    RealIn[i]=0;
    for(j=0;j<MAXWAVES;j++) {
        if (rand()%2) {
            RealIn[i]+=coeff[j]*cos(amp[j]*i);
        }
        else {
            RealIn[i]+=coeff[j]*sin(amp[j]*i);
        }
    }
}
```

Fig. 5. Standard library functions called within a program

Even when leaving out calls to library functions, the number of function instances can be overwhelming. Therefore, only the functions with the largest instruction count are picked. Although this instruction count can vary on some processors with specialized computation units and may also be different for different compilers, this number is in general a good indicator on the expected execution time. This way, the user can later concentrate on parallelizing those parts of the software that make up the largest part of the execution time.

5.2 Data Dependencies

Besides the calling relation between functions, there exist other relationships between functions that can require them to be executed sequentially. If two functions for instance change the value of the same variable, then they have to be executed in a given order. However, these data dependencies are not visible in

the call tree. Adding data dependencies that are later taken into account when producing the parallel model prevents race conditions that might occur when the program is parallelized.

As we work with models of the application, adding precedence relations between functions simply means adding directed edges in the graph representation of the model. Therefore, different profilers can be used to derive the data dependencies. Even profilers that can extract different dependencies between function instances could be used to include more edges in the call tree.

5.3 User Input

Since it is possible that some dependencies that will prohibit a parallelization are not included in the dependencies from the call tree and the data dependencies, the developer gets to add precedence relations that were not discovered by the profilers. Sometimes, as in the case of the FFT application, not all dependencies can be recognized by the profiler for instance due to the use of pointer arithmetics. In that case, the user has to add these dependencies by hand. Simply allowing the user to add edges between any two nodes of the graph would, however, not result in a very good parallel model. A call tree can be rather large and the user is bound to make mistakes if he is allowed to add edges unrestrictedly. In addition, parallelizing function instances that are called from the same function can be realized rather easily and without having to rewrite parts of the code. Therefore we chose a step-by-step approach, where only a part of the call tree is presented to the user in each step.

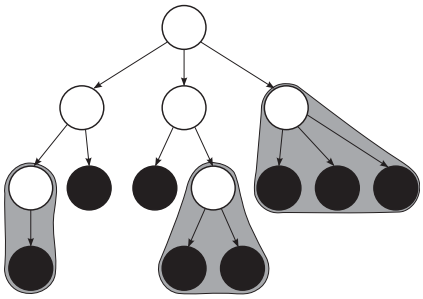


Fig. 6. A call tree where all leaves are shown in black. The subtrees marked in gray consist of one parent node and its children nodes that all have to be leaves.

A subgraph that consists of a parent node and its children nodes is presented to the user in each step. The children nodes all have to be leaves of the call tree, as illustrated in Figure 6. If there is a dependency between two children nodes of the subgraph, the user adds an edge between these nodes. In case that the user does not add any precedence relations for some children, it is assumed that these children nodes can be executed in parallel.

5.4 Transformation to a Parallel Model

In the automatic step from the enhanced call tree to the parallel model of the application, all the information from the profilers and the developer is combined. A model transformation is used to build up the parallel model of the application, while at the same time deleting nodes from the call tree in a bottom-up way. Before presenting the first subgraph to the user, all nodes of the call tree are copied to a new graph, the parallel model PM, without however copying the edges. The edges are inserted step-by-step, as the subgraphs are modified by the user. Redundant edges are not copied to the parallel model. An edge between nodes f_0 and f_1 is redundant if there is a directed path from f_0 to f_1 that consists of edges of the same or a higher importance. The call tree edges are the least important edges and the data dependencies are the most important edges.

We chose to keep all data dependency edges in the transformation from the call tree to the parallel model. This is not strictly necessary to get a parallel model of the application. However, it is better to keep the data dependencies intact if this parallel model is used for scheduling where the data communication is important, such as embedded systems. If the data on a data dependency edge is for instance very large, it is better to schedule the functions on both ends of this data dependency on the same processor to avoid a large overhead for communication.

After copying the edges of the call tree subgraph in the manner just described to the parallel model, all children nodes of the subgraph are now deleted in the call tree. That way, it is ensured that for each function f, the subgraph with f as the parent node is presented to the user at some point.

5.5 Subgraph Edges

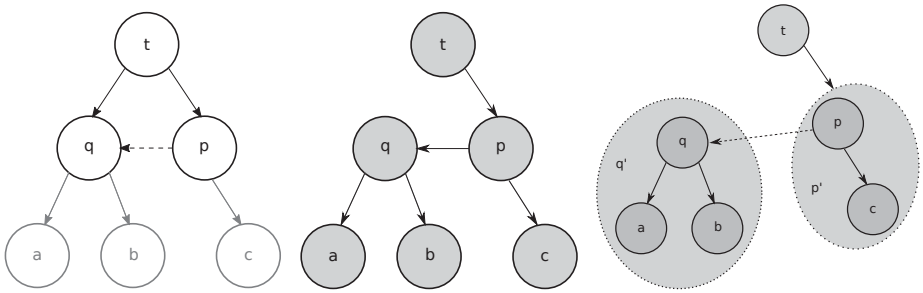
In the parallel model PM one needs to be able to see which subgraphs were already considered, i.e. for each node it needs to be obvious which function instance it was called from. This is best explained by way of an example.

Example 2. Imagine a subgraph with parent node p that has a child c. According to the transformation algorithm, the edge from p to c is copied to PM. In a later step, p and another node q are the children in a subgraph with parent t as shown in Figure 7(a). The user now indicates that q has to be executed after p as indicated by the dotted line in the figure. If no information is kept that p and c formed a subgraph earlier, then edges will now go from p to both c and q in the parallel model, indicating that c and q can be executed in parallel. This is shown in Figure 7(b). However, this is impossible to implement in practice without rewriting a part of the code.

This is why subgraph edges are allowed in the parallel model. These edges can be removed in a later step and replaced with the regular edges in the parallel model, but are useful during the transformation phase. For the example that was just discussed, the parallel model in Figure 7(c) is the result of using a subgraph edge. All nodes inside the dotted circle on the right p' have to be executed before

all nodes inside the dotted circle q' on the left. The subgraph edge is indicated by a dashed arrow.

We could of course also allow users to specify that c and q in the example in Figure 7 can indeed be executed in parallel. However, in practice this will most likely be difficult to realize. It would force the programmer to ensure that q can be executed in parallel with c, but only after some of the computations of p are already done. While our approach is more conservative, it allows for an easier adaptation of the source code and will lead to less errors.



(a) A call tree with a precedence relation from p to q. The parts that are in gray are already deleted from the call tree, but are shown here to indicate the precedence relations in the subtrees with parents p and q.

(b) Transforming the call tree into a parallel model without subgraph edges. Here, it seems as if q and c could be executed in parallel.

(c) Transforming the call tree into a parallel model using a subgraph edge. In this case, all parts of p' are executed before any nodes in q' can start execution.

Fig. 7. Using subgraph edges to prevent undesired parallelism in the parallel model after the transformation from a call tree

Subgraph edges can e.g. be useful if the parallel model is used for scheduling, but the number of nodes in the call tree is much too large compared to the number of processors. In that case, subgraphs can be seen as a single node.

An optional model transformation step allows to replace subgraphs and subgraph edges with nodes and regular edges to adapt the granularity of the parallel model.

Example 3. An example is given in Figure 8, where h represents the subgraph that consists of h_0 and its successors. The edges that lead away from h_0 have to be redirected. For that, all leaves within h have to be found. In Figure 8 this would be nodes h_1, h_3 and h_4 since they have no outgoing edges. The subgraph edges going out from h_0 will each lead to three outgoing edges from h_1, h_3 and h_4 respectively. Finally, the subgraph edges can be deleted. The result of replacing the subgraph edges of h_0 in Figure 8 is shown in Figure 9.

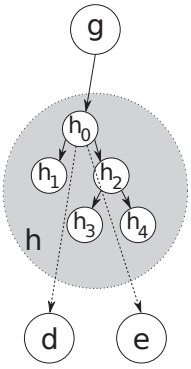


Fig. 8. Subgraph edges in the parallel model

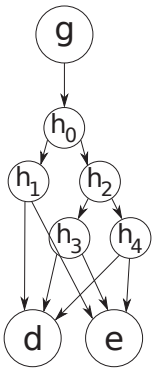


Fig. 9. Replacing the subgraph edges in Figure 8

6 Implementation

We implemented the above approach in a toolchain that goes from the software source code via call trees to a parallel model of a software application.

The call tree is represented graphically within the AToM^3 [12] modeling tool. Each edge type – function call, data dependency and user-defined dependency – has a different color, so that the user can keep track of what kind of dependency an edge represents.

The first type of edge is a function call. To derive these edges, the toolchain uses a modified version of the Callgrind profiler [13] that is part of the Valgrind tool suite [14]. The adaptations were necessary to filter out the functions that do not appear in the source code of the application. In addition, Callgrind had to be modified to produce a call tree as an output rather than a call graph. The call tree that is produced as the output of the modified Callgrind tool is then parsed into a Python file that can be opened in AToM^3. In the parsing step, we allow to adapt the granularity of the call tree by transferring only the function instances with the largest instruction count to the Python file.

The second type of edges are data dependencies. They are derived from the output of the Paralax [11] compiler. Data dependencies in Paralax are dependencies between different call paths. For a data dependency between call path $f_1, \ldots, f_n$ and call path $g_1, \ldots, g_m$ that is found by the Paralax compiler, an edge has to be inserted in the call tree between each instance of f_n and each instance of g_m in the call tree, provided that the predecessors of f_n resp. g_m are instances of $f_1, \ldots, f_{n-1}$ resp. $g_1, \ldots, g_{m-1}$.

Example 4. Consider the call tree in Figure 10. Assume that there is a data dependency from call path $a \rightarrow b$ to $a \rightarrow c \rightarrow d$. Then, two data dependencies would be added, from b_1 to d_1 and d_2. There will be no data dependency from b_1 to d_3, since the call path of d_3 does not match with the one described by the

call path $a \to c \to d$. The resulting call tree with the added data dependencies is shown in Figure 11.

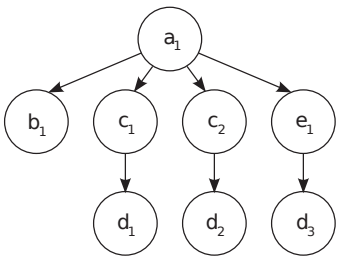


Fig. 10. A call tree where function d can be called from functions c and e.

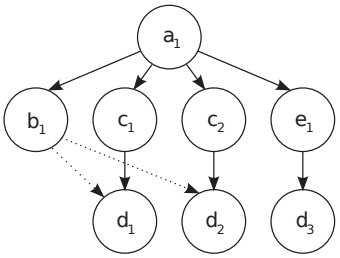


Fig. 11. Adding the data dependency from call path $a \to b$ to $a \to c \to d$ in Figure 10.

The last type of edge is a user-defined edge. As explained above, these user-added edges can be added one subgraph at a time. In each step, a subgraph of the call tree is highlighted. This subtree consists of a function instance p and all its children that all have to be leaves in the call tree. The user can add missing dependencies by drawing edges in the model. The position of each function instance in the call stack is given, so that the developer can see in which order the different function instances were actually executed during the profiling.

Once the user inserted all edges in a subgraph, he can indicate this with the click of a button and wait until the next subgraph is highlighted. The edges of the current subgraph are then copied to the parallel model in the manner described above. All children of the subgraph are then deleted along with the edges leading to and from them. A rule that determines for each node if it is a leaf of the call tree is invoked, so that the parent of the call tree will now be marked as a leaf. Then, the next subgraph is highlighted and so on.

After all nodes of the call tree have been deleted there is an optional step for replacing the subgraph edges. This is done in a top-down way, meaning that the subgraph edges closest to the root node (usually the `main` function) are replaced first so that the granularity of the model can be adapted.

7 Results

We tested our approach on the FFT implementation of the MiBench benchmark. In order to get usable results, we had to divide the loop that computes the FFT in the program into function calls. Although this was done within a few minutes, it shows a limitation of our approach that is due to it using functions as the smallest entities.

During the transformation phase, it was shown that it is a good idea to present the model to the user instead of just relying on the information given by the profiling tools. When we profiled the FFT application, the dependency that arises from filling the input signals with random numbers as seen in Figure 5 to using these signals in the actual Fast Fourier Transform was not detected by the Paralax profiler.

From the resulting parallel model, it became apparent that the best opportunities for parallelization are in two loops. The first loop is the inner loop of the actual Fast Fourier transformation (the outer loop has data dependencies) and the second loop is the one shown in Figure 5. These two loops make up more than 50 percent of the execution time. This shows that our approach can indeed help the developer in identifying the program parts where parallelization will bring the largest benefit.

The code was parallelized by inserting OpenMP [15] directives according to the parallel model. To parallelize the loop that computes the FFT, it was enough to add a simple line of code that allows the concurrent execution of the loop. Parallelizing the loop that initiates the input signals for the FFT is more difficult. As seen in Figure 5, there is a conditional execution within the loop. Therefore, the code had to be changed into the form seen in Figure 12. This already gives a slight improvement in the sequential execution of the code. This new for-loop can then be parallelized. Note that in order to get the same results as with the sequential version of the code, it has to be ensured that the random numbers are used in the same order. Therefore, they are stored beforehand in a separate array. Creating this array takes about one quarter of the total execution time of the parallelized program. When executing the program on an Intel Core i3 processor (dual core model with hyper-threading technology), the overall speedup when parallelizing both loops in the manner just described was 2.5.

```
for(i=0;i<MAXSIZE;i++) {
  RealIn[i]=0;
  for(j=0;j<MAXWAVES;j++) {
    x = rand()%2;
    RealIn[i] += x*coeff[j]*cos(amp[j]*i)+
                 (1-x)*coeff[j]*sin(amp[j]*i);
  }
}
```

Fig. 12. Modifying the code in Figure 5 so that there is no conditional statement in the for-loop.

8 Future Work

A line of future work would be to automatically trace back the parallelism that becomes apparent in the parallel model to the source code (as for instance seen in [16]). This could then be used to include the necessary code that parallelizes the application automatically. However, since the correctness of the parallel model depends on the user input, the model is not guaranteed to be correct. It might be more difficult to remove automatically added code that is incorrect than to add the code. Additionally, as can be seen from the FFT example, parallelizing the code can sometimes mean that some part of the code has to be rewritten before parallelizing into different threads can have a benefit. Another problem with automatically generating code is variable reuse. In case that during sequential execution a variable is used in a part of the code and then again in a different part of the code in a different context, these two parts could be executed in parallel (there will only be a write-after-read dependency, which will not be included in the model transformation). However, to actually implement this parallelism two distinct variables are needed. In order to be able to do that automatically, the write-after-read dependency would have to be tracked.

When testing our approach on a medical imaging software, it became apparent that pipeline parallelism is difficult to detect with our approach. In the case of the medical imaging software, a 4-stage pipeline is used. Each pipeline stage writes on an array that is then read by the next stage. Since the data dependency profiler does not distinguish between function instances, dependencies are added from each stage 1 function instance to each stage 2 function instance that occurred after it in the call stack. This makes the model very cluttered and the developer might not be able to see the possible parallelism.

The approach can be easily extended to include other profiling information about dependencies between different functions. Our approach already considers dependencies that can be left out in the parallel model if alternative paths exist in the model (the function calls), and dependencies that are always copied from the original call tree to the parallel model. Therefore, it would be straightforward to include other dependencies without having to adapt the graph transformation algorithm.

9 Conclusion

In this paper, we proposed a model-driven approach to help developers parallelize sequential software programs. Profiling results are combined with domain-knowledge of the user to transform source code into a parallel model of the application. We believe that by abstracting away the implementation details we can assist the developer significantly in the parallelization process. By providing the developer with an abstract model of the software that already includes function calls and data dependencies, the developer gets a graphical representation of his program that hides irrelevant details. As the developer can add precedence relations, a parallel model can be derived. It can for instance be used

for scheduling the application on a multi-processor platform or adding compiler directives for parallel execution into the source code.

References

1. Quiñones, C.G., Madriles, C., Sánchez, J., Marcuello, P., González, A., Tullsen, D.M.: Mitosis compiler: An infrastructure for speculative threading based on pre-computation slices. (In: 2005 Conference on Programming Language Design and Implementation (PLDI))
2. Steffan, J.G., Colohan, C., Zhai, A., Mowry, T.C.: The STAMPede Approach to Thread-Level Speculation. Transactions on Computer Science (2005)
3. Chen, M.K., Olukotun, K.: The Jrpm System for Dynamically Parallelizing Java Programs. (In: 30th Annual International Symposium on Computer Architecture (ISCA '03))
4. Mak, J., Faxèn, K.F., Janson, S., Mycroft, A.: Estimating and exploiting potential parallelism by source-level dependence profiling. (In: Euro-Par 2010)
5. Ishihara, M., Honda, K., Sato, M.: Development and Implementation of an Interactive Parallelization Assistance Tool for OpenMP:iPat/OMP. IEICE Transactions on Information and Systems (2006)
6. Ismail Assayad, Valérie Bertin, F.X.D.P.G.O.Q.S.Y.: Jahuel: A formal framework for software synthesis. In: Proceedings of the Seventh International Conference on Formal Engineering Methods ICFEM. (2005)
7. Stefanov, T., Zissulescu, C., Turjan, A., Kienhuis, B., Deprettere, E.: System design using kahn process networks: The compaan/laura approach. In: Proceedings of the conference on Design, Automation and Test in Europe (DATE). (204)
8. Guthaus, M.R., Ringenberg, J.S., Ernst, D., Austin, T.M., Mudge, T., Brown, R.B.: Mibench: A free, commercially representative embedded benchmark suite. (In: Proceedings of the Workload Characterization 2001 Workshop)
9. Tournavitis, G., Wang, Z., Franke, B., O'Boyle, M.F.: Towards a holistic approach to auto-parallelization. (In: 2009 Conference on Programming Language Design and Implementation (PLDI))
10. Miniskar, N.R., Hammari, E., Munaga, S., Mamagkakis, S., Kjeldsberg, P.G., Catthoor, F.: Scenario based mapping of dynamic applications on mpsoc: A 3d graphics case study. In: SAMOS Proceedings of the 9th International Workshop on Embedded Computer Systems: Architectures, Modeling and Simulation. (2009)
11. Vandierendonck, H., Rul, S., de Bosschere, K.: The paralax infrastructure: Automatic parallelization with a helping hand. In: Parallel Architectures and Compilation Techniques (PACT). (2010)
12. de Lara, J., Vangheluwe, H.: Using AToM3 as a Meta-CASE Tool. (In: 4th International Conference on Enterprise Information Systems (ICEIS 2002))
13. Weisendorfer, J., Kowarschik, M., Trinitis, C.: A tool suite for simulation based analysis of memory access behavior. (In: Proc. of the 4th Int. Conference on Computational Science (ICCS 2004))
14. Nethercote, N., Seward, J.: Valgrind: A framework for heavyweight dynamic binary instructions. (In: Proc. of ACM SIGPLAN 2007 Conference on Programming Language and Design (PLDI 2007))
15. Chapman, B., Jost, G., van der Pas, R.: Using OpenMP. The MIT Press (2007)
16. Johnson, S., Evans, E., Ierotheou, C.: The parawise expert assistant – widening accessibility to efficient and scalable tool generated openmp code. (In: Workshop on OpenMP Applications and Tools (WOMPAT 2004))