A Multipurpose Clustering Algorithm for Task Partitioning in Multicore Reconfigurable Systems

S. Arash Ostadzadeh, Roel J. Meeuws, Kamana Sigdel, Koen Bertels
Delft University of Technology
Computer Engineering Laboratory
Mekelweg 4, 2628 CD Delft, The Netherlands
{arash, roel, kamana, koen}@ce.et.tudelft.nl

Abstract

In recent years, multicore systems have become a dominant architecture, introducing new challenges that need to be addressed in order to take full advantage of their efficiency. Reconfigurable computing has also received a great deal of attention due to its ability to increase the performance of an application through hardware execution, while retaining the flexibility of a software solution. Grouping tasks within an application contributes to coarse-grained partitioning, which can eventually improve the performance of the system. In this paper, we introduce a clustering framework along with a flexible multi-purpose clustering algorithm that initiates task clustering at the functional level based on dynamic profiling information. The clustering framework can be used as the basic step to modify the granularity of tasks in the hardware/software partitioning and scheduling phases. As a result, an elaborate mapping onto the system resources and possibly a higher degree of task parallelism becomes feasible. The framework particularly targets two objectives, 1) to form workload-balanced and 2) loosely-coupled clusters. We evaluated its efficiency using MJPEG as a case study. The experimental results comply with the desired clustering metrics defined through the objectives.

1. Introduction

Over the past years processor technology has fundamentally changed. Multicore processors or Chip Multi-Processors (CMP) are gaining popularity in high-performance and mainstream computing. However, it introduces new challenges in fully exploiting increased numbers of cores for maximal performance. Meanwhile, reconfigurable computing has also received a great deal of attention due to its capacity to accelerate applications. The main advantage of reconfigurable computing is its ability to increase performance through hardware execution, while retaining the flexibility of software solutions. Typically, reconfigurable systems consist of traditional microprocessors and reconfigurable hardware. Hardware implementation of a task generally exhibits better performance than software execution. Moving selected software components to reconfigurable hardware can improve the performance of the whole system. Thus, reconfigurable systems benefit by speeding up the whole application through implementation of selected application kernels onto reconfigurable hardware. These technologies can be further combined to form a reconfigurable multicore system [4].

One of the major requirements for such systems is to identify which part of an application should be mapped onto software and which part onto reconfigurable devices. A traditional way of selecting critical application region(s) for mapping onto reconfigurable hardware is to utilize program analyses such as profiling and tracing in order to identify computation intensive kernel functions. Ending up with small kernels is a key problem adversely affecting the performance gain of the whole system, since we have to pay for extra overheads (communication, synchronization, and configuration) introduced in handling these kernels regardless of being in software and/or hardware mode. To tackle this problem, coarse-grained partitioning of an application is generally considered to improve the performance of an implementation by decreasing the costs involved, which could also result in increased parallelism [6]. Grouping related kernel functions is the first step to intensify the coarse-grained partitioning. Furthermore, after clustering the problem size is reduced, and this benefits the runtime of the synthesis process.

Most researches acknowledge the hardware/software partitioning problem as a crucial step in proper task allocation under given system constraints. Various algorithms have been proposed to solve this problem ranging from simple algorithms to sophisticated ones [2, 7, 9, 10]. Task scheduling and mapping onto underlying architecture is commonly addressed in subsequent stages [3, 5]. Here, we deviate from this routine as our general clustering algorithm is introduced early in Design Space Exploration (DSE) prior to actual HW/SW partitioning, scheduling, and mapping phases. We try to establish a comprehensive and flexible framework for creating efficient task groups with varying granularity, meanwhile taking into account predefined system/architecture resource constraints and preferences. Traditionally, these constraints and preferences are checked at late design stages, which lengthens the whole process and raises the costs involved. Somehow related to our work, [8] presents a clustering method as a preprocessing phase to a hardware/software partitioning and scheduling system. The approach is primarily focused on scheduling optimization. In contrast, our framework is more comprehensive and can be utilized for various design optimizations.

The main motivation of introducing the clustering framework as a preprocessing phase is to suppress undesirable task partitioning (assembling and/or splitting) cases which prove to be inappropriate, unnecessary, and non-optimal in subsequent design space exploration stages for HW/SW partitioning, scheduling and mapping. The flexible algorithm presented is capable of considering various criteria to come up with optimal or nearly optimal solutions suited for different cases. It also contributes to the portability of the partitioning scheme for different heterogeneous multicore reconfigurable systems, since system- and/or architecture-specific details can be peeled off from the core of partitioning process. In addition, the outcome of the clustering phase along with valuable dynamic profiling information extracted from an application can provide hints to application developers for individual design optimizations and refinements such as possible coarse-grained parallelism detection and extraction.

The rest of this paper is organized as follows. In Section 2, we describe our clustering framework along with the introduction of data usage analysis process. In Section 3, the clustering algorithm is proposed and discussed. A case study is presented in Section 4. Finally, Section 5 is devoted to the concluding remarks and some comments for future work.

2. Clustering framework

Currently, the proposed clustering framework focuses on two primary goals, namely, workload balancing and minimizing inter-cluster data communication. These goals address the most critical aspects of application development and execution in multicore reconfigurable systems. Needless to say, load balancing between clusters is the main property for performance gain in coarse-grained parallelism and the total communication time of a task's execution, either in hardware or software mode, is directly affected by the amount of inter-cluster data exchange. The flexible multipurpose clustering algorithm presented here is also capable of considering other objectives such as satisfying system/architecture constraints including FPGA resources (area/gate constraints, bus bandwidth, etc.).

In order to form abstractions of control and data dependencies, two graph representations are employed. Control dependencies are examined via a conventional call graph and a new graph named Quantitative Data Usage (QDU) graph is introduced to illustrate the amount of data usage dependencies between functions.

Extraction of the exact and precise data dependencies is only possible through static source code analysis involving exhaustive conditional statements analysis, input data diversity analysis and alias tracking, which have proven to be elusive when comprehensive program constructs are present in source code. Nevertheless, systems adopting this approach should impose some constraints on programming style and constructs to be able to produce desirable outcomes. An alternative approach would be to conduct simulated run(s) of the target application to track data accesses in memory. Although this technique may be prone to produce biased, inaccurate, and sometimes erroneous information about the actual application behavior, in most cases it can provide valuable information, which closely approximates the real pattern of data communication. Moreover, it's absolutely feasible with no restrictions on the structure of currently available programs.

In the proposed framework, data usage tracking is handled at function level, which implies a coarse granularity, yet delivers a sufficiently detailed overview of quantitative data dependencies within a program. Since the clustering framework generally plans to form task groups as primary building blocks for application development and execution on chip multiprocessors, we only need to go through the tracking process at coarse level in order to compensate for the resulted overheads of target application execution on the underlying architecture. Therefore, we do not consider instructions or small basic blocks within functions in this work. Tackling the problem at this level also benefits from the fact that it's more likely the clustering algorithm ends up with loosely coupled partitions regarding data communications via memory. Figure 1 depicts the clustering framework data flow along with other coupled units. It should be stressed that here we only focus on the representation of the new QDU graph, its implementation and integration within the simulation tool set and the clustering algorithm formulation.

2.1. Annotated call graph

A basic call graph reveals the relations of caller and callee functions in a program. There are several profiling tools that can report these relations based on source code

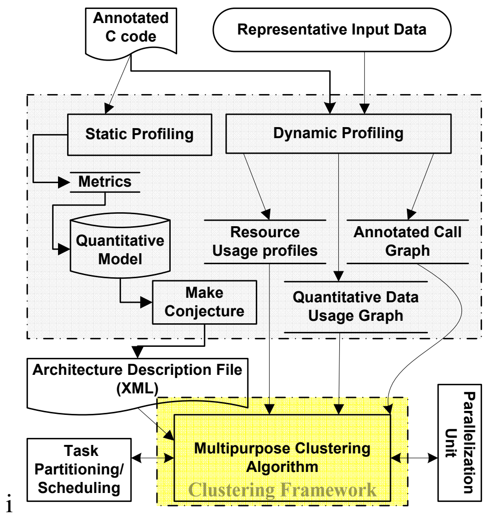

Figure 1: Clustering framework

or run-time analyses. The procedure involves recording all the function calls and returns. Initially, it's considered as a control dependency outline, since it shows the sequence of program execution in a coarse scale. However, regarding the fact that a caller function typically feeds input data to the callee, which in someway produces data as a return value, initiates also the concept of data dependency scheme among the co-operating functions.

What we need for our detailed analysis is beyond just the relations between functions. An annotated call graph not only visualizes these relations, but also contains the number of times a particular function is called, a particular function's self-contribution to the whole execution time, and a particular function's entire contribution as an aggregate of its descendants. The execution time contribution is used for workload balancing in the proposed clustering algorithm.

2.2. Quantitative Data Usage (QDU) graph

Main memory data usage is tracked during simulated run(s) to identify exactly which function is reading what amount of data produced by which function. To record the required information during a program run, we need to monitor each unique memory address that is accessed inside a function. The most recent write access to a particular location is tagged by the respective caller and is then transformed into a data communication record when another function reads from that particular address. The entire records are subsequently examined and combined to form the quantitative data usage graph. In order to spot, extract and track the memory access data within the simulated run outputs, we designed and implemented a memory access recording module. The tracking process utilizes trie data structure for proper and fast storage and retrieval.

Considering hexadecimal digits in memory addresses, a trie structure with base 16 is used. Each hexadecimal digit in a 32-bit memory address corresponds to one level in the trie data structure, leaving 8 levels deep in the hierarchy for complete address tracing. In order to save space as much as possible in module implementation, we have designed the structure to grow dynamically only on demand.

The recording process is accomplished in two distinct phases. In the first phase, we trace an 8-level trie for a particular memory address provided by the simulation tool set. For each access three different arguments are specified, namely, memory address, function ID, and read/write flag. In case of a write access, the corresponding memory cell representative in the trie is labeled with the caller function ID. When a read flag is detected, the function ID responsible for the most recent write in the memory cell is retrieved and passed along with the consumer function ID to the second phase where a data communication record is created. A 16-level trie structure is employed for this phase in order to accommodate both the producer and consumer function IDs. Figure 2 depicts the dynamic trie data structure used in the recording module. There may be cases where for a consumer function there is no prior producer, i.e., reading from a location where the data has not been tagged by write flag in that address. We call these compile-time data and it occurs for example when a function tries to read a constant. The module was implemented and integrated into the open source SimpleScalar tool set [1].

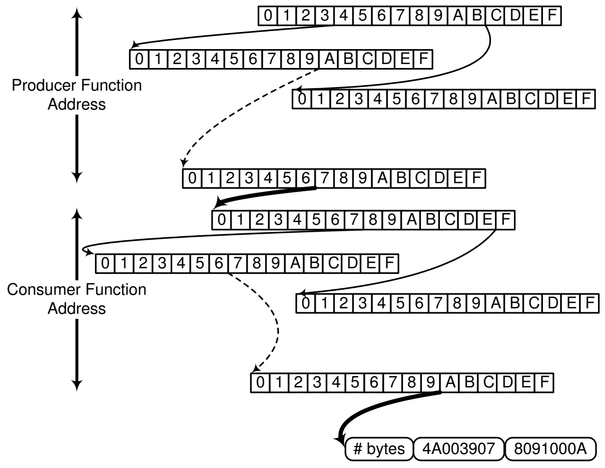
0 1 2 3 4 5 6 7 8 9 A B C D E F
Producer Function Address
Consumer Function Address
# bytes
4A003907
8091000A

Figure 2: Dynamic trie data structure

3. Clustering algorithm

The purpose of the clustering algorithm is to look for (strongly) interconnected functions in order to bind them in a cluster. As discussed subsequently, the term *interconnected* refers to more than just functions which extensively exchange data, though this characteristic is considered to have a critical role in clustering. As the main unit in the framework, the clustering algorithm role in subsequent processing is crucial. The formed clusters constitute the complex coarse kernel blocks for HW/SW partitioning/scheduling/mapping tasks (threads) on the underlying architecture. Performance estimation feedbacks by the respective units will be used for additional cluster refinements. The extracted information in the framework and the clusters are also used as hints to the parallelization unit, where application revision is done in order to exploit possible coarse-grain parallelism. Grouping the functions into clusters divides the data communication streams between functions in two classes, Intra-cluster (connecting functions within a cluster) and Inter-cluster (connecting functions in different clusters). In order to perform the clustering, some metrics and properties are taken into account. We have classified them into five different categories as follows.

Balanced clusters - forming clusters that are nearly balanced with respect to the workload (total executing time).

Loosely-coupled clusters - as few inter-cluster data streams as possible will increase the overall performance and facilitates concurrent tasks execution.

Tightly-coupled functions within clusters - functions having close interactions are likely to be related and hence should be considered as a unit. This coupling can be addressed to bidirectional data streams, huge amount of data exchange, workhorse strategy, etc.

Balanced data streams - the communication loads for inter-cluster data streams are expected to be nearly balanced. This also implicitly involves the number of required streams between clusters and the channel bandwidth specifications.

Resource constraints satisfaction - some system/architecture constraints may apply in clustering, e.g., On-chip mapping area constraints.

Based on these properties, we also present some heuristic rules which will be used as anticipation criteria in the selection process. These rules are supposed to improve the quality of formed clusters.

- If a function only has data communication with another one or is only called by another function (according to the annotated call graph), it is desirable to put them inside the same cluster.

- If current cluster is relatively small in terms of overall execution time, it is desirable to favor the node with highest contribution in the selection criterion and vice versa. Definition of a small/large cluster is totally subjective, however, it is simple to consider number of clusters and estimate a contribution percentage, e.g., for 5 clusters, we expect a contribution of about 20% by each cluster.

- It is beneficial to limit the number of bidirectional data streams between clusters as much as possible. These data streams impose severe restrictions on subsequent parallelization process.

The heuristic rules can be revised and augmented for adaptation to individual system design, architectural or user preferences and requirements. The outline of the clustering algorithm is presented in figure 3. To form clusters around compute-intensive kernels, initially we spot those kernels which pass a minimum contribution threshold. This value is defined per case. For example, if we plan to end up with approximately four clusters, minimum amount of about 10% looks appropriate. This boundary value could be increased if too many kernels are selected or decreased if insufficient kernels meet the condition. We also conduct another strategy to discard excessive picked out kernels to ensure not ending up with surplus small clusters at last. In the preprocessing step, pair of kernels that are topologically close to each other (at most within distance of *d* in the QDU graph) are compared and the kernel with lower contribution is excluded, unless the to-be-discarded kernel has a contribution percentage higher than a threshold value, making it illogical to discard. As an example, when we are heading for five clusters, two kernels with 15% and 18% contributions are both expected to form their own clusters no matter how topologically close they are. Following this step, initial clusters are created with each selected kernel as an initiator. These clusters will grow gradually during subsequent stages.

In the main body of the algorithm, at each iteration, clusters are augmented with the best interconnected neighbor node. For each cluster, the candidates are evaluated by a ranking function. The function inspects and assigns a value to each node indicating the suitability of the node for merging with the current cluster. The selection process is intensely affected by the definition of the ranking function, providing high flexibility for the clustering algorithm. In order to adapt to different policies for cluster creation, individual elements and parameters consisting the function definition could be customized. As an initial attempt to examine our algorithm, we defined the function in the following simple format, considering only two major elements in selection criterion.

$$R_{overall}(node_i) = C_1 R_{exe}(node_i) + C_2 R_{com}(node_i)$$

where $R_{overall}$ returns the final rank for $node_i$. R_{exe} refers to the execution-time rank of $node_i$ among the candidates to merge with the questioned cluster. A proper rank is expected to be determined according to heuristic rules. For example, if we are currently eager to merge a compute-intensive kernel with our small cluster, the node with the most contribution to the execution time of the application

```
I  Initialization    select initial kernels
II Preprocessing     revise kernel set. suppose $k$ kernels are left in the set
III ClusterInit      for each selected kernel create an initial cluster, $C=\{C_1,C_2,..,C_k\}$
IV ClusterCreate     do {
                         for ( all unfinished clusters )
                         {
                             if ( no neighbor node left ) mark current cluster as finished and continue;
                             for ( all neighbor nodes of current cluster )
                             {
                                 if ( candidate node does not satisfy any constraint ) continue;
                                 evaluate with Ranking Function;
                             }
                             if ( all candidates failed to pass constraints) mark current cluster as finished and continue;
                             sort candidates in decreasing order based on ranks;
                             mark the candidate with the highest rank for merging;
                         }
                         while ( there exists conflict between clusters on marked candidates)
                             if ( clusters $C_1$ , $C_2$ , … , $C_m$ have marked the same candidate $N$ )
                             {   unmark $N$ in all clusters except the cluster in which $N$ appears with the highest rank;
                                 for ( all clusters in which $N$ has been unmarked )
                                     if ( any candidate left )  mark the next candidate in list;
                             }
                         while ( there exists unmerged marked candidates )
                         {
                             merge marked candidate with its respective cluster;
                             update cluster properties;
                         }
                     } while (all clusters are marked as finished);
V  Completion        resolve the problem of unassigned nodes
```

Figure 3: Clustering algorithm

gets the highest rank among candidates. Higher rank implies better chance for selection. R_{com} is estimated to reveal data communication intensity of $node_i$ with respect to the nodes currently residing in the questioned cluster. Tighter data usage interactions (quantitatively) infers greater likelihood for inclusion in the current cluster. C_1 and C_2 are weighing coefficients to adjust individual ranks in the final multi-objective function output. In our experiment, we have assumed balanced stressing factors for these two parameters, setting both to 0.5. Should we have some constraints that by definition disqualify candidate(s), they must be applied prior to ranking function evaluation. Some heuristic rules could also act as rank revisors in favor or against certain candidate(s). As an example, regarding the first rule mentioned here previously, a candidate which has a dedicated caller in the cluster is expected to get the highest rank possible.

Conflict resolution should also be addressed for clusters competing to draw inward an identical candidate. As clusters set off to grow, this could happen when a particular candidate appears on top of the lists of two or more clusters due to high scores assigned by the ranking function. In this case, we favor the cluster in which the examined candidate has achieved the highest score. For all other clusters, new top candidates (if any) appearing just next in lines are selected. The same process is repeated until all conflicts (some new conflicts may appear during substitution) are resolved. When we merge the candidates with relevant clusters, some cluster properties and parameters should be updated in order to present the current status within the growing cluster. These parameters are particularly used in checking the conditions to finalize clusters as well as in ranking function and constraints checking. For example, if a cluster is growing big with respect to the total execution time contribution and it is exceeding the predefined threshold, we may opt to mark the cluster as finished. Generally, the point where we decide not to continue with a particular cluster is subjective and should be declared in constraints. The last phase in the clustering algorithm is concerned with handling the nodes which had no chance to end up inside clusters in the main loop. This rarely happens provided that we define proper and reasonable constraints complying with the underlying system and corresponding application. However, we propose two solutions to handle this situation. In the first, we can set about to put all the remaining nodes in an extra cluster or more if desired. The same procedure as described in step IV of the algorithm could also be employed. Another solution would be to relax constraint(s) which prevent the nodes to be merged with the currently chosen number of clusters. It should also be noted that if we are already applying some hardware mapping constraints or some uncompromising conditions for the clusters, it may not be feasible to go for the second solution. In that case, we certainly have difficulties running the application totally in hardware.

4. Case study

For an initial evaluation of the proposed clustering framework and algorithm, we have used an implementa-

Table 1: Clusters in the MJPEG application

Cluster	Main Kernel	Number of Functions	Kernel's contribution	Cluster's contribution
1	DoubleReferenceDct1D	1	34.34%	34.34%
2	UseHuffman	19	21.08%	24.69%
3	Quantize	6	5.42%	18.07%
4	ReferenceDct	7	19.88%	22.89%

tion of the MJPEG algorithm as a case study. The application contains 33 functions with an absolute computation-intensive DCT kernel (*DoubleReferenceDct1D*), identified by the profiling tool. To have a clear estimation of the memory access recording module performance, we measured CPU utilization and memory usage delivered by executing the application. The values were obtained on a 2.0GHz Intel Core2 Duo machine with 2.0GB of physical memory. The instrumented simulator augmented with the memory access recording module is about $8\times$ slower than the uninstrumented version, which itself is $128\times$ slower than a native implementation. The integration of the module within the simulator seems promising as it takes about 25 minutes for the whole simulation to conclude, which could be the case for similar computation-intensive applications. Furthermore, it is apparent that the instrumented version exhibits a large memory overhead compared to a normal simulation since we had to keep track of all memory accesses, however, it is still acceptable. In this application, approximately 199828 KB is recorded as the memory overhead.

We opted to partition the application in four clusters. With this respect, the cluster corresponding to the top computation-intensive kernel *DoubleReferenceDct1D* was immediately concluded with only one function, while other clusters grew gradually. *UseHuffman* is the largest cluster regarding the number of functions inside, since it contains all the I/O-related routines in the application with nearly no contributions to the whole execution time. *Quantize* and *ReferenceDct* include 6 and 7 functions respectively with contributions near the optimum values. The results are summarized in Table 1.

5. Conclusion and future work

In this paper, we introduced a robust clustering framework coupled with a flexible multipurpose clustering algorithm that initiates task clustering at the functional level. The clustering framework uses dynamic profiling information as a base for partitioning and produces clusters of balanced interconnected functions. The case study showed that the clustering algorithm can produce proper clusters which can be further utilized in application design optimizations, HW/SW partitioning and mapping, and task scheduling on the target architecture. To come up with optimal objectives, a thorough inspection of various parameters is needed in order to utilize them in the ranking function, which is the key component in forming clusters. Furthermore, we plan to extract additional dynamic information from simulated run(s) to reveal the pattern of data accesses within an application. The data is crucial for task parallelism detection and hence needed for the parallelization unit development.

References

[1] SimpleScalar LLC. `http://www.simplescalar.com/`.

[2] K. B. Chehida and M. Auguin. HW/SW Partitioning Approach for Reconfigurable System Design. In *Proceedings of the 2002 international conference on Compilers, architecture, and synthesis for embedded systems*, pages 247–251, 2002.

[3] Z. Gu, M. Yuan, and X. He. Optimal Static Task Scheduling on Reconfigurable Hardware Devices Using Model-Checking. In *13th IEEE Real Time and Embedded Technology and Applications Symposium*, pages 32–44, 2007.

[4] T.-O. Kwok and Y.-K. Kwok. On the Design, Control, and Use of a Reconfigurable Heterogeneous Multi-core System-on-a-chip. *IEEE International Symposium on Parallel and Distributed Processing*, pages 1–11, April 2008.

[5] Y. Qu, J. Soininen, and J. Nurmi. Static Scheduling Techniques for Dependent Tasks on Dynamically Reconfigurable Devices. *Journal of Systems Architecture*, 53(11):861–876, 2007.

[6] V. Srinivasan, S. Govindarajan, and R. Vemuri. Fine-grained and Coarse-grained Behavioral Partitioning With Effective Utilization of Memory and Design Space Exploration for Multi-FPGA Architectures. *IEEE Trans. Very Large Scale Integr. Syst.*, 9(1):140–159, 2001.

[7] G. Wang, W. Gong, and R. Kastner. Application Partitioning on Programmable Platforms Using the Ant Colony Optimization. *J. Embedded Comput.*, 2(1):119–136, 2006.

[8] T. Wiangtong, P. Cheung, and W. Luk. Cluster-Driven Hardware/Software Partitioning and Scheduling Approach for a Reconfigurable Computer System. *Lecture Notes in Computer Science*, 2778:1071–1074, 2003.

[9] T. Wiangtong, P. Y. K. Cheung, and W. Luk. Comparing Three Heuristic Search Methods for Functional Partitioning in HardwareSoftware Codesign. *Design Automation for Embedded Systems*, 6(4):425–449, 2002.

[10] T. Wiangtong, P. Y. K. Cheung, and W. Luk. Tabu Search with Intensification Strategy for Functional Partitioning in Hardware-Software Codesign. In *Proceedings of the 10th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, pages 297–298, 2002.