Time-Constrained Clustering for DSE of Clustered VLIW-ASP

Mario Schölzel
Department of Computer Science at Brandenburg University of Technology at Cottbus, Germany

Abstract

In this paper we describe a new time-constrained clustering algorithm. It is coupled with a time-constrained scheduling algorithm and used for Design-Space-Exploration (DSE) of clustered VLIW processors with heterogeneous clusters and heterogeneous functional units. The algorithm enables us to reduce the complexity of the DSE, because the parameters of the VLIW are derived from the clustered schedule of the considered application which is produced during a single compilation step. Several compilations of the same application with different VLIW-parameter settings are not necessary. Our proposed algorithm is integrated into a DSE-Tool in order to explore the best parameters of a clustered VLIW processor for several basic blocks of signal processing applications. The obtained results are compared to the results of Lapinskii's work and show, that, for most benchmarks, we are able to save ports in the register file of each cluster.

1. Introduction

In many signal processing applications high performance is required, while power and/or area consumption should be as low as possible. These requirements may only be satisfied by specialised hardware. Furthermore, the system should be flexible enough to handle little changes of the application in later development stages. Application Specific Processors (ASPs) provide a good trade-off between flexibility, performance and power/area consumption [13]. A widely accepted model for ASPs is the Very Long Instruction Word (VLIW) processor model. A processor of that type uses a single register file and several functional units (FU) that operate in parallel. Thus, every FU is connected to the register file by two read ports and one write port. Every FU contains a certain set of **operators**, e.g. adder, multiplier, an so on. We refer to this as the type of a FU. In heterogeneous VLIWs FUs may have different types. All FUs are controlled by an **instruction word** in every clock cycle, which encodes the operation to be performed by each FU. The scheduling of operations into instructions is done statically by the compiler. This assures high performance while the control logic of a VLIW can be kept very simple. However, the single register file is the bottleneck of a VLIW. The more parallelism is available the more ports must be provided by it. The power and area consumption of the register file grows as n^3, where n is the number of ports, and its delay grows as $n^{3/2}$ [15]. Clustered VLIW-processors (see figure 1 for an example) offer a solution to this problem.

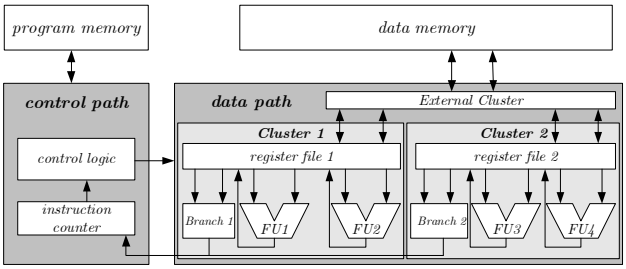

Figure 1: Two-Cluster-VLIW processor.

A clustered VLIW uses several register files. A certain set of FUs is associated with each of them. Direct access to a register file is only given to associated FUs. By this, high parallelism is preserved while the number of ports of each register file becomes small. There again, a cluster interconnection network is required to transfer data from one cluster into another one. A data transfer is done by a copy operation, which has a certain latency time and must be scheduled statically by the compiler. The copy operations are carried out by copy operators which belong to the *external cluster*. External read and write ports in each register file give access to the external cluster.

Adaptation of ASPs to a given application is typically done by a Design-Space-Exploration (DSE). Most of the existing approaches use an iterative methodology, which modifies the parameter of a (clustered) VLIW in each iteration [1, 2, 4, 6, 7]. In every iteration, compiler and simulator are adapted to the current architecture. Using them, the application is compiled, simulated and profiled. Necessary modifications of the architecture may derived from the obtained profiling data. This DSE-cycle is repeated until the desired performance is reached or a certain amount of pareto-optimal architectures was found. In [4] this cycle takes more than 2 weeks only for determining the type of five FUs in a non-clustered VLIW. If DSE is done for heterogeneous clustered architectures, beside the type of each FU also the number of clusters and the number of FUs in each cluster must be determined [17]. Thus, the design space of a clustered heterogeneous

978-3-9810801-2-4/DATE07 © 2007 EDAA

VLIW becomes much more larger than the design space of a non-clustered VLIW. Therefore, an iterative DSE methodology may leads to infeasible long runtimes, because too much architectures must be considered, even if the design space is pruned according to pareto-optimal architectures as in [2, 6]. In order to perform a DSE-cycle for a clustered VLIW, the compiler must be able to generate a clustered schedule of the application. Several approaches for clustering were proposed [3, 5, 8, 10, 11, 14, 18]. All of them are resource-constrained. This means, the available resources in each cluster must be known before compilation, and therefore, these clustering algorithms are only applicable for iterative DSE-approaches. Our proposed time-constraint clustering algorithm enables us to avoid an iterative DSE-approach and thus, helps us to reduce the complexity of the DSE. Furthermore, the optimisation goal of resource constrained clustering approaches is to minimise the length of the produced schedule. Therefore, most of them avoid to schedule copy operations in the critical path of the application. This may prohibits to find a better solution because the produced schedule may be longer than the critical path, due to other resource constraints. Then, copy operations on the critical path would not matter. Because our time constrained clustering algorithm, produces a clustering for a given schedule length L we allow to insert copy operations in the critical path from the beginning if L is larger than the critical path length.

2. Application Model

The application for which a DSE should be done is represented by a set of basic blocks. A **basic block** $b = (V, E, type)$ consists of a set of operation nodes V, a set of data dependency edges $E \subseteq V \times V$ and a labelling function $type : V \rightarrow \mathcal{O}$, where $\mathcal{O}$ is the set of all operation types supported by the processor. $lat(t)$ denotes the number of clock cycles (latency) necessary to execute an operation of type t. A **schedule** of length L consists of exactly L instructions and can be seen as a function $sc : V \rightarrow \{0, \ldots, L - 1\}$ which associates nodes with the number of an instruction in which its execution starts. Please note, that an operation v, with $lat(type(v)) = l$, and $sc(v) = i$ is executed in instructions $i, \ldots, i + l - 1$. Furthermore, a schedule sc may be partial, i.e. not every node has a start time. sc is called a **full schedule** if every node has a start time. A **clustering** of a basic block is a function $cl : V \rightarrow \{0, \ldots, mC\}$ which assigns every node the number of the cluster in which it is executed. mC is the number of regular available processor clusters. Please note, there is a special external cluster 0, which executes all *copy* operations (see figure 1). If two adjacent nodes are assigned to different clusters a *copy*-operation must be executed in order to move the result of the producer operation into the cluster of the consumer operation. In order to tread these *copy*-operations like regular operations during the scheduling they are added to the basic block. A basic block b together with a clustering cl is said to be a **clustered basic block**. It already contains all necessary *copy*-operations and is denoted by (b, cl). cl also assigns all copy operations to cluster 0. In figure 2 an example of a basic block and a clustered basic block is given.

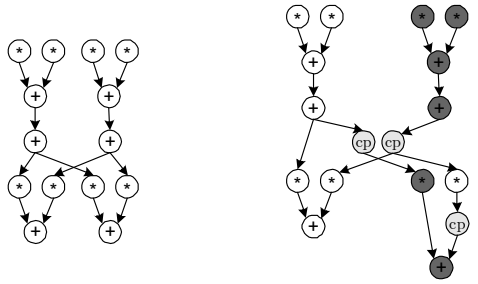

Figure 2: Basic block and clustered basic block.

A **path** $p = v_1 \ldots v_n$ in a clustered basic block is a sequence of adjacent nodes, i.e. $(v_i, v_{i+1}) \in E$. The **length** (or latency) of a path p is the sum of the latencies of its nodes. Given a certain clustering and scheduling, the set of operations, which are executed in cluster c and instruction i, is denoted by $ops(c, i)$. The number $|ops(c, i)|$ of these nodes is called the **width of instruction** i in cluster c. Every instruction i, for which $|ops(c, i)|$ is maximal in c, is called a **widest instruction** in cluster c. The **width of a cluster** is equal to the width of its widest instructions. The number of ports of the register file in cluster c immediately depends on its width w and is $3 \cdot w + ep$, where ep is the number of external ports, whose number depends on the number of simultaneously executed copy operations.

3. Scheduling algorithm

Our proposed time-constrained clustering algorithm is coupled with a time-constrained scheduling algorithm. The clustering algorithm produces a clustered basic block (b, cl), which is scheduled by the scheduling algorithm, in order to value the quality of the clustering. The scheduling algorithm respects the cluster assignment cl. We shortly describe the scheduling algorithm and the used objective function. A more detailed description can be found in [9]. The objective function reflects the costs of a processor, which is required to execute a certain schedule. The power and area consumption of a VLIW strongly depends on the size of its register files, i.e. the number of ports [15]. A register file cost function $\boldsymbol{rfc} : \mathbb{N} \rightarrow \mathbb{N}$ is given, which returns for a given number of ports the register file costs, e.g. in terms of area and power consumption. For a small number of ports, this function may grow

slowly and for larger numbers it grows cubic and dominates the processor costs [15]. Furthermore, the highest possible clock rate of a VLIW depends on the size of its largest register file due to wire delay [15]. A clock rate function $\boldsymbol{cr} : \mathbb{N} \rightarrow \mathbb{N}$ defines the dependency of the clock rate on the number of ports.

The time-constrained scheduling algorithm is similar to force-directed-scheduling [12]. For a given schedule length L a schedule sc is stepwise constructed. For every operation v of (b, cl) a time-frame is determined, regarding to the given schedule length L, in which v can be scheduled. In every step an operation v is selected using a priority function and scheduled into an instructions i. The instruction i is selected by scheduling v on a trial-basis into every instruction within its time frame and updating the time-frames of all dependent operations. This leads to several (maybe partial) trial-schedules sc_i. For each of them an objective value $C(sc_i, cl)$ is determined. The objective value is based on an estimation of operator costs (*TLoad*) and register file costs (*RBLoad*) of each cluster and is defined as the sum of *RBLoad* and *TLoad*. Furthermore, *RBLoad* takes into account the limitation of the clock rate, which depends on the largest register file of the VLIW. For each trial-schedule sc_i the expected size of each register file is estimated by scheduling all unscheduled operations in sc_i, using a list-scheduling-like algorithm. By this, we obtain for each schedule sc_i an estimation schedule esc_i, which is full scheduled. The width $|ops(c, i)|$ of a widest instruction i in esc_i can be determined for each cluster c and gives the number of internal read and write ports of the register file in cluster c. In order to take the size of the interconnection network into account, the number of required external read and write ports in each register file is considered. It depends on the width of the widest instruction in cluster 0. Therefore,

$$tP(\mathrm{c}) = 3 \cdot |ops(c, i)| + 2 \cdot |ops(0, i)|$$

is the number of ports of the register file of cluster c and

$$mP = \max\{tP(c) \mid 0 < c < mC\}$$

is the number of ports of the largest register file of the VLIW. Having these values for an estimation schedule esc_i, the objective value *RBLoad* of the corresponding trial-schedule sc_i is obtained by

$$RBLoad(sc_i, cl) = \alpha_1 \cdot \sum_{c=1}^{mC} rfc(tP(c)) - \alpha_2 \cdot cr(mP)\,.$$

The first term represents the overall costs of all register files and the second term reflects the highest possible clock rate, which becomes smaller with growing mP. Highest priority is given to the performance degradation by weights α_i.

The operator costs of each cluster in sc_i are estimated by a cost function similar to the cost function used in force-directed-scheduling. It is based on an estimation of the number of parallel executed operations of the same type. I.e., for every operation type t, every cluster c and every instruction i a load-value $ILoad(t, i, c)$ is computed. $ILoad(t, i, c)$ reflects the expected number of operations of type t that may be executed in instruction i and cluster c. The cluster-load-value of a certain type t

$$CLoad(t, c) = \max\{ILoad(t, i, c) \mid 0 \leq i < L\}$$

is the maximal *ILoad*-value for a given cluster c and type t. The operator costs $TLoad(sc_i, cl)$ of a trial-schedule sc_i and clustering cl are the sum of its cluster-load-values for each type t.

The selected operation v is schedule into instruction i for which $C(sc_i, cl)$ is minimal. If the objective value is computed for a full schedule it will reflect the hardware costs of the VLIW processor required for its execution. However, the objective value may also be computed for partial schedules in which no operation is scheduled. This is done at several stages of the clustering algorithm in order to estimate roughly the quality of the clustering.

4. Clustering algorithm

The optimisation goal of our clustering algorithm is to produce a clustering cl, so that a full schedule sc of a given length L can be generated and the VLIW, necessary to execute sc, has minimal hardware costs regarding to the objective function C.

4.1 Clustering scheme

The clustering algorithm is coupled with scheduling, in order to consider dependencies between both tasks which arise from inserted copy operations. The iterative proceeding is shown in figure 3.

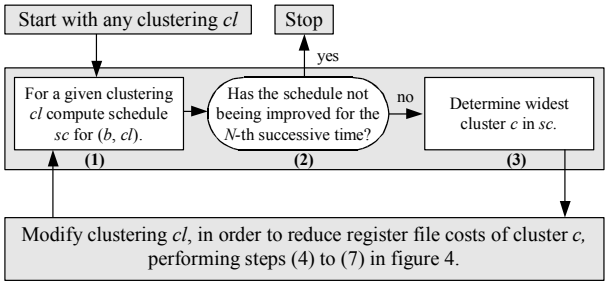

Figure 3: Iterative proceeding during clustering.

Clustering starts with any initial clustering cl. In step (1) a schedule sc for (b, cl) is generated according to the algorithm proposed in section 3. It is secured, that in step (1) all operations can be scheduled within the given length L. I.e., the length of every path of (b, cl) is at most

L, including all copy operations. In step (3) a widest cluster c is selected regarding to the schedule sc, which was produced in step (1). Let w be its width. The register file size of c is reduced by moving some operations from c into another cluster z, because the register file size has the largest impact on cost function C. This is done in steps (4)-(7). Therefore, the next iteration, which starts again in step (1), uses a modified clustering cl. If the modified cl is better than the best clustering bcl which has been produced so far (i.e. $C(cl, sc) < C(bcl, bsc)$), bcl is set to be cl and bsc is set to be sc. Otherwise, if bcl could not be improved for the N-th time, clustering is finished. The modified clustering cl is used for the next iteration anyway, even if it is worst than the clustering of the previous iteration. By this, local minima could be avoided. A more detailed description of the steps 4 to 7 is given in figure 4.

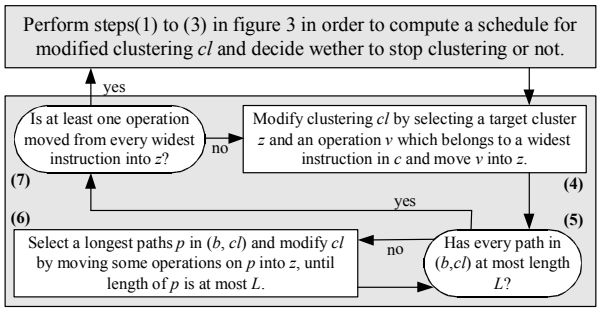

Figure 4: Reducing cluster costs.

The operation v (**target node**) to be moved from a widest instruction in c, as well as the **target cluster** z in which v should be moved, are selected in step (4). The problem that occurs, due to moving v into cluster z, is, that this may leads to copy operations, which prolong some paths of the clustered basic block. If one of these paths becomes longer than L, a schedule of length L cannot be produced in step (1). Therefore these paths must be shortened by moving other operations on these paths into the target cluster z too (step (6)). These operations "follow" operation v into cluster z. Therefore, some copy operations become needless and can be removed, which shortens the path. This proceeding is repeated, until all paths have a length of at most L (step (5)). In order to reduce the register file size of considered cluster c, at least one operation of every widest instruction in c is moved into the target cluster z (see step (7)). Therefore, steps (4) to (7) are repeated, if there is at least one instruction in c, that still has the width w. Furthermore, there is no limit how often a node can be moved between different clusters. Thus, previously made assignments of nodes to clusters can be cancelled.

The proposed clustering scheme is repeated twice and both **stages** differs only in step (6), i.e., how a path is shortened:

- During the first stage whole paths of a basic block are moved between different clusters in order to obtain a feasible clustering within a short time.
- During the second stage only sub-paths are moved, in order to obtain a better balance of cluster utilisation.

The first stage starts with an initial clustering, where all nodes are assigned to the same cluster. The best clustering bcl, which was generated by the first stage, is used as the initial clustering of the second stage.

4.2 Shortening paths

The most important step during clustering is shortening of all paths that have a length greater than L (steps (5) and (6) in figure 4). The length of a path only becomes larger than L, if copy operations were inserted. During shortening a path, the operation v, which was selected in step 4 and moved into the target cluster z, should stay in cluster z. Therefore, other operations on the considered path must be moved into cluster z in order to remove copy operations. In the following it is explained how these operations are selected.

A copy operation cp is called a z-cluster-copy operation, if there exists an operation u with $cl(u) = z$ and $((u, cp) \in E$ or $(cp, u) \in E)$. I.e., cp writes a value into cluster z or reads a value from cluster z. A sub-path $sp = v_k v_{k+1} \ldots v_m$ of a path $p = v_1 \ldots v_n$ is called a z-cluster-sub-path, if sp is entirely executed in cluster z and (either $k = 1$ or $type(v_{k-1}) = copy$) and (either $m = n$ or $type(v_{m+1}) = copy$). In order to shorten a path in step (6), we select the longest one. If there are several paths of the same length, we select the one which contains the most z-cluster-copy operations. These copy operations are significant positions in the selected path. Either its predecessor belongs to cluster z and its successor not or vice versa. In every case the selected path contains at least one z-cluster-copy operation and is of the form:

$$v_1 \ldots v_{n-1}\, v_n\, c\, v_{n+1}\, v_{n+2} \ldots v_m,$$

where c is a z-cluster-copy operation and v_i are nodes of arbitrary type. Considering c, the operation, which must be moved into cluster z can be determined as follows:

I. If v_n belongs to cluster z, node v_{n+1} is moved to cluster z. We obtain the path

$$v_1 \ldots v_{n-1}\, v_n\, v_{n+1} c\, v_{n+2} \ldots v_m.$$

II. If v_{n+1} belongs to cluster z, node v_n is moved to cluster z. We obtained the path

$$v_1 \ldots v_{n-1}\, c\, v_n v_{n+1} v_{n+2} \ldots v_m.$$

By moving the predecessor or successor of a z-cluster-copy operation into cluster z the copy operation is "shif-

ted" along the selected path. This will not always reduce the length of the path. But in the following cases, it is:

(a) c is moved according to (II) and $type(v_{n-1}) = copy$.

(b) c is moved according to (I) and $type(v_{n+2}) = copy$.

(c) c moved according to (II) and v_n has no predecessor, i.e. $n = 1$.

(d) c is moved according to (I) and v_{n+1} has no successor, i.e. $n + 1 = m$.

If in case (a) or (b) v_{n-1} respectively v_{n+2} is a z-cluster-copy operation the path length is reduced by $2 \cdot lat(copy)$. Otherwise the path length is reduced by $lat(copy)$. A selected z-cluster-copy operation is shifted along the selected path according to the rules (I) and (II) until one of the cases (a) to (d) applies. Thus, the length of the considered path is shortened. However, the length of other paths may increased. Figure 5 gives an example.

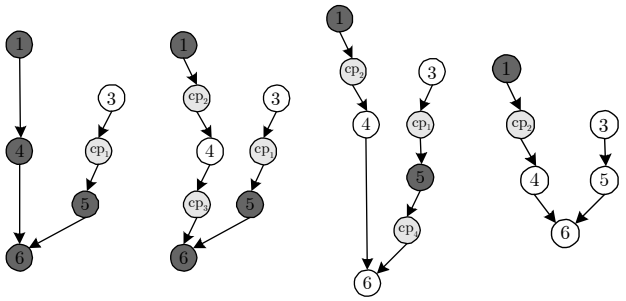

Figure 5: Example of shifting copy operations.

Let us assume the basic block in figure 5 should be scheduled within four instructions, i.e. $L = 4$. Operations 1, 4, 5 and 6 (grey) belongs to the cluster 1. Operation 3 (white) belongs to cluster 2. Thus, a copy operation has been inserted between operation 3 and 5. Furthermore, let us assume operation 4 is selected in step (4) of figure 4 and moved to target cluster 2, as shown in figure 5 (b). Therefore, the copy operations cp_2 and cp_3 has been inserted. Now the path (1 cp_3 4 cp_2 6) has a length of 5. Now, either operation 1 or 6 must be moved to cluster 2, because operation 4 must remain in cluster 2. As in figure 5 (c) shown, operation 6 has been moved. Now the path (1 cp_3 4 6) has a length of 4, but the length of path 3 cp_1 5 cp_4 6 was increased. Therefore, this path must be shortened. Because the assignment to cluster 2 is fix, operation 5 is also moved to cluster 2. This leads to the clustering in figure 5 (d). There, every path has a length of at most 4.

During the first stage of the clustering algorithm all z-cluster-copy operation nodes of the selected path are removed, which in turns mean, that long operation chains are formed in the target cluster and the considered path gets a length of at most L. During the second stage only z-cluster-sub-paths of the selected path are moved to the target cluster, until the selected path has a length of at most L. By this, it is possible to refine the clustering and to obtain a well balanced clustering. In order to select the next copy operation, which should be removed from the selected path p during the second stage, we select the shortest z-cluster-sub-path in p. This sub-path is responsible for a very short computation chain in the target cluster. Therefore, by moving a z-cluster-copy operation at the beginning or end of this sub-path, longer operation chains are formed in cluster z.

4.3 Determining target node and cluster

So far it was not explained how operation v in step (4) is selected. Let cl be the current clustering, BI be the set of all widest instructions of cluster c, $CL = \{1, \ldots, mC\}$ the set of available processor clusters and w the width of a widest instruction in BI. In step (4) the first instruction $i \in BI$ is selected. We construct the set of all possible assignments $A = \{(v, z) \mid v \in ops(c, i) \text{ and } z \in CL\}$. For every $(v, z) \in A$ cl is modified by moving v into target cluster z and performing steps (5) and (6). The obtained clustering is denoted by $\langle v, z \rangle$ and every path has a length of at most L. Let NC be the set of all clusterings $\langle v, z \rangle$, where $(v, z) \in A$. Then, the clustering $\langle v, z \rangle$, for which $C(s, \langle v, z \rangle)$ is minimal is selected. Here s is a schedule where no operation is scheduled. Please note, that all available clusters mC will only be used by the clustering algorithm, if this minimises the objective function C. If not, some clusters remain empty. Thus, by continuously increasing mC and performing the clustering algorithm the largest number of used clusters can be determined.

5. Results

In order to evaluate the quality of our clustering algorithm it was integrated in the DSE-Tool DESCOMP [9]. We used several basic blocks of typical signal processing applications from the thesis of Lapinskii [16] as benchmarks. The largest basic blocks have up to 49 nodes. A DSE was done for these benchmarks with DESCOMP and varying schedule lengths, in order do determine the parameters of well adapted clustered VLIWs for each benchmark and each schedule length. The generated DESCOMP-architectures were compared to the architectures from Lapinskii's thesis. In figure 6 the comparison of number of ports of the largest register file is shown for several number of clusters. The number in parentheses behind the benchmark names on top of each table is the number of used clusters. In most cases the DESCOMP

approach leads to better utilisation of the FUs and could save register file ports. The number of ports of the largest register file could be reduced in average by 20%. The total number of ports, which is not shown in figure 6 due to a lack of space, could be reduced in average by 26%. This means, not only the cost of the largest register file could be reduced but also the costs of register files in all clusters. Every result was obtained within a few seconds up to a few minutes which is comparable to the runtimes of Lapinskii's resource constrained approach. However, in contrast to the work of Lapinskii the DESCOMP-approach allows to combine different operators in a single FU, which makes the DSE more complex, because the type of each FU must be determined. Thus, we were able to consider a larger design space at comparable runtime and to save register file ports by this.

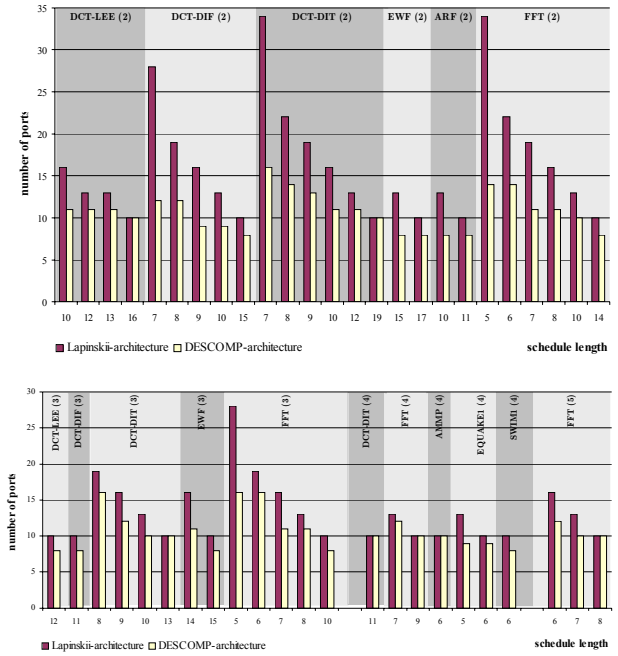

Figure 6: Benchmark results.

6. Conclusions

We have introduced a new time-constrained clustering algorithm which is coupled with a time-constrained scheduling algorithm. By this, a time consuming DSE-cycle can be avoided, even for DSE of clustered VLIW processors. The results show, that the generated clustered schedule saves register file ports, compared to a state-of-the-art approach. Therefore, area and power consumption of the register files is reduced and a higher clock rate can be used.

7. REFERENCES

[1] B.Middha, V.Raj et. al.: *A Trimaran Based Framework for Exploring the Design Space of VLIW ASIPs with Coarse Grain Functional Units*. Proc. of the 15th International Symposium on System Sythesis (ISSS'02), pp. 2-7, 2002.

[2] D.Fischer, J.Teich et. al.: *Efficient architecture/compiler co-exploration for ASIPs*. Proc. of the International Conference on Compilers, Architecture and Synthesis for Embedded Systems (CASES'02), pp. 27-34, 2002.

[3] E.Özer, S.Banerjia et. al.: *Unified Assign and Schedule: A New Approach to Scheduling for Clustered Register File Microarchitectures*. Proc. of 31th Int. Symposium on Microarchitecture, pp. 308-315, 1998.

[4] G.J.Hekstra, G.D.La Hei et. al.: *TriMedia CPU64 design space exploration*. Proc. of the IEEE International Conference on Computer Design, pp. 599-606, 1999.

[5] G. Desoli: *Instruction assignment for clustered VLIW DSP compilers: A new approach*. Technical Report, HP Laboratories Cambridge, HPL-98-13, 1998.

[6] G.Snider: *Spacewalker: Automated Design Space Exploration for Embedded Computer Systems*. Technical Report, HP Lab. Palo Alto, HPL-2001-220, 2001.

[7] H.Corporaal and H.J.M.Mulder: *MOVE: a framework for high-performance processor design*. Proc. of the Conference on Supercomputing, pp. 692-701, 1991.

[8] J.R.Ellis: *Bulldog: A Compiler for VLIW Architectures*. MIT Press, Cambridge, Massachusettes, 1985.

[9] M.Schölzel and P.Bachmann: *DESCOMP: A New Design Space Exploration Approach*. Proc. of the 18th Int. Conference on Architecture of Computing Systems, pp. 178-192, 2005.

[10] M.L.Chu, K.C.Fan et. al.: *Cost-Sensitive Operation Partitioning for Synthesizing Custom Multicluster Datapath Architectures*. Proc. 2nd Workshop on Application Specific Processors, pp. 40-47, 2003.

[11] M.L.Chu, K.C.Fan et. al.: *Region-based Hierarchical Operation Partitioning for Multicluster Processors*. Proc. of the Conference Programming Languages Design and Implementation, pp. 300-311, 2003.

[12] P.G.Paulin and J.P.Knight: *Force-directed scheduling in automatic data path synthesis*. Proc. of the 24th Design Automation Conference, pp. 195-202, 1987.

[13] P.Faraboschi, G.Brown et. al.: *Lx: a technology platform for customizable VLIW embedded processing*. The 27th Annual International Symposium on Computer Architecture 2000, pp. 203-213, 2000.

[14] R.Leupers: *Instruction Scheduling for Clustered VLIW DSPs*. Proc. of the Conference on Parallel Architectures and Compilation Techniques, pp. 291-300, 2000.

[15] S.Rixner, W.J.Dally et. al.: *Register Organization for Media Processing*. Proc. of the 6th High-Performance Computer Architecture, pp. 375-386, 2000.

[16] V.Lapinskii: *Algorithms for Compiler-Assisted Design-Space-Exploration of Clustered VLIW ASIP Datapaths*. Dissertation, University of Texas at Austin, 2001.

[17] V.Lapinskii, M.F.Jacome et. al.: *Application-Specific Clustered VLIW-Datapaths: Early Exploration on a Parameterized Design Space*. IEEE Transactions on Computer Aided Design of Integrated Circuits and Systems, 21(8), pp. 889-903, 2002.

[18] V.Lapinskii, M.F.Jacome et. al.: *Cluster Assignment for High-Performance Embedded VLIW Processors*. ACM Transactions on Design Automation of Electronic Systems, 7(3), pp. 430-454, 2002.