

# Cluster Assignment of Global Values for Clustered VLIW Processors

Andrei Terechko

Philips Research

P. Holstlaan 4; 5656AA Eindhoven  
The Netherlands  
+31 (0)40 2742788

andrei.terechko@philips.com

Erwan Le Thénaff

Philips Research

P. Holstlaan 4; 5656AA Eindhoven  
The Netherlands  
+31 (0)40 2744384

erwan.le.thenaff@philips.com

Henk Corporaal

Technical University Eindhoven

Den Dolech 2; 5612 AZ Eindhoven  
The Netherlands  
+31 (0)40 2475462

h.corporaal@tue.nl

## ABSTRACT

In this paper high-level language (HLL) variables that are alive in a whole HLL function, across multiple scheduling units, are termed as *global values*. Due to their long live ranges and, hence, large impact on the schedule, the global values require different compiler optimizations than local values, which span across only one scheduling unit. The instruction scheduler for a clustered ILP processor, which is responsible for cluster assignment of operations and variables, faces a difficult problem of assigning global values to clusters. Our study shows that trivial assignments (e.g. mapping all global values into one cluster) may result in a severe cycle count overhead relative to the unicluster of up to 26.3% for a four cluster VLIW machine. This paper presents three advanced algorithms for assigning global values to clusters based on multi-pass scheduling and affinity of variables. Furthermore, we measure performance of these algorithms on optimized multimedia C applications and assess quality of our algorithms by comparing them to a practical higher performance bound derived from a vast random search. Our algorithms reduce the execution time overhead of the best simple algorithm *round-robin* from 10.5% to 5.9% for the two cluster VLIW machine and from 17.3% to 14.12% for the four cluster VLIW machine.

## Categories and Subject Descriptors

D.3.4 [Programming Languages]: Processors – *compilers, code generation, optimization*.

## General Terms

Algorithms, Experimentation, Performance, Languages

## Keywords

ILP, VLIW, compiler, instruction scheduler, cluster assignment, register allocation

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

CASES'03, Oct. 30 – Nov. 1, 2003, San Jose, California, USA.

Copyright 2003 ACM 1-58113-676-5/03/0010...\$5.00.

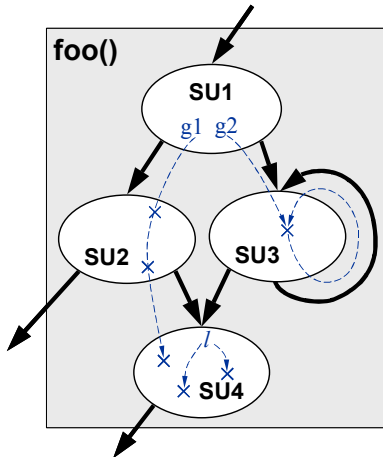
## 1. INTRODUCTION

To follow uninterruptedly increasing computing requirements of multimedia applications, embedded media processors must provide higher performance at low cost. To satisfy this demand, instances of a processor family derived along the technology scale should bring noticeable improvements (e.g. in terms of higher clock frequency). In order to improve scalability of an Instruction-Level Parallelism (ILP) architecture, which is often used in media processors, it can be split into clusters. Each cluster contains a number of function units (FU) connected to a local register file (RF). This way, the classical single RF is partitioned into smaller RFs with fewer read and write ports. Clusters with smaller RFs are faster and dissipate less energy than the corresponding unicluster [1]. Furthermore, the bypass network is partitioned to keep bypass wiring shorter. This becomes especially advantageous in future IC technologies, when wire delay dominates logic delay [2]. However, communication between clusters takes several architecturally visible cycles, incurring a cycle count overhead relative to the unicluster. Clustering is used in several VLIW and EPIC processors such as the TI TMS320C6x [3], BOPS ManArray [4], STM/HP Lx [5], Sun MAJC [7], etc.

Performance of a VLIW processor strongly depends on the quality of the compiler, which is responsible for mapping operations to parallel FUs and data variables to registers. Moreover, the compiler for clustered VLIW processors has to assign operations and data to clusters and schedule inter-cluster communication (ICC). ICC can be carried out, for example, by inter-cluster copy operations. If an operand of an operation resides in a remote cluster, the instruction scheduler adds a copy operation to the VLIW code that transfers the operand to the local RF. Note that the added copy operations may delay scheduling of regular operations, introducing, therefore, cycle count overhead. The cycle count overhead of a two-cluster machine with respect to the unicluster is reported to be between 5% and 20% [6][16][17][18][19][20][31][32]. The instruction scheduler is responsible for minimizing this overhead.

Typically, the scheduling scope is chosen smaller than a high-level language (HLL) function to decrease scheduling time and engineering complexity of the instruction scheduler. For example, some RISC and CISC schedulers operate on basic blocks; VLIW/EPIC and superscaler schedulers process traces [27], decision trees [24][25][13], regions [11], superblocks [26], etc. Let us term the data flow graph structure processed by the scheduler at a time as *scheduling unit* (SU).

Consider Figure 1 with a HLL function  $foo()$ . To simplify scheduling this function is decomposed by the compiler front-end in several scheduling units ( $SU1$ ,  $SU2$ ,  $SU3$ , and  $SU4$ ). The scheduling units connected by the bold solid edges, representing possible execution paths, depict the control flow graph of  $foo()$ . The X symbols denote accesses to variables, and the dashed lines designate live ranges of the variables.



**Figure 1. Globals in the control flow graph of function  $foo()$ .**

Normally, some HLL variables are alive throughout the whole function. For example, variable  $g1$  produced in scheduling unit  $SU1$  is also accessed (either read or written to) in scheduling units  $SU2$  and  $SU4$ . Note that the scheduling unit  $SU3$  constitutes a loop. Consequently, variable  $g2$  can be accessed every iteration. In the remainder of this paper, we will refer to the variables that are alive across multiple scheduling units as global values or globals. Note that in contrast to HLL global variables, our global values can not be alive across HLL functions. Variable  $l$  in scheduling unit  $SU4$  is a local variable, because it is alive in only one scheduling unit. To sum it up, we distinguish three types of variables (or DFG edges) based on their live ranges:

1. *local values*, which are alive within one scheduling unit
2. *global values*, which are alive within one HLL function
3. *HLL global variables*, which are alive in the whole program

Normally, compilers do not keep HLL global variables in registers during the whole execution of the program to decrease RF pressure when the variable is not used. On top of that, satisfying cluster assignment preferences of many SUs with a single cluster assignment for a global value is often impossible. Therefore, our cluster assignment study did not focus on the HLL global variables.

Global values often require different handling than local intra-scheduling unit values. Local values have short live ranges, which, normally, get reordered by the instruction scheduler. The globals, on the other hand, can be alive throughout the whole HLL function, and, furthermore, the scheduler is not able to reorder their live ranges. Normally, a VLIW instruction scheduler processes one scheduling unit at a time. If we do not differentiate global values from locals, the scheduler has to assign all HLL variables to physical registers (and clusters) scheduling unit by scheduling unit. However, while processing one scheduling unit (e.g.  $SU1$ ), the scheduler does not consider the following

scheduling units ( $SU2$ ,  $SU3$ , and  $SU4$ ), and, consequently, can not find the best cluster assignment (CA) for a global, reducing inter-cluster copies to/from this global. Many existing compilers do not differentiate between globals and locals, and assign globals to clusters based on one most critical SU (in Figure 1 it will most likely be  $SU3$ , which constitutes a loop body). However, scheduling the most critical SU first does not help in many cases, when the HLL function contains several SUs with comparable execution frequencies. Therefore, we believe that global values require optimizations (e.g. in register allocation) at the function level, involving multiple scheduling units, whereas locals can be efficiently handled within a single scheduling unit.

These ideas are reflected in explicit separation of HLL variables into globals and locals in several state-of-the-art ILP compilers, e.g. the CHAMELEON compiler with the CARS algorithm [21][22][23] and the TriMedia compiler [13]. The TriMedia C/C++ compiler, for example, performs register allocation for global values before instruction scheduling and register allocation for locals in the scheduler on the fly. The CARS algorithm also distinguishes global inter-region variables denoted as  $\phi$  and  $\phi^{-1}$ , and treats them differently from locals.

In this study we analyze the importance of the assignment of global values to clusters and propose three assignment heuristics – two based on the *affinity matrix* and a *feedback-directed two pass* algorithm. The paper is organized as follows. Section 2 presents related work. Then our cluster assignment algorithms are described in Section 3. We present and analyze the results of our experiments in Section 4. Finally, Section 5 concludes the paper.

## 2. RELATED WORK

J. Hiser, S. Carr, and P. Sweany described a heuristic assignment of *all* HLL variables to clusters in [27], based on the Register Component Graph (RCG). RCG is built from an “ideal” instruction schedule of operations belonging to one function. The ideal schedule uses all characteristics of the target machine except that it assumes a single infinite RF. The nodes of the RCG represent symbolic registers and the weights of the edges define affinity between registers to belong to the same cluster or RF. Once the RCG is built, the partitioning algorithm assigns each node to the cluster that has the highest benefit. The benefit is the sum of the weights of the edges between the node in question and nodes already assigned to the cluster. Our affinity matrix is more exact, because besides affinities between inputs/outputs variables of the same operations we consider affinities between inputs/outputs variables of different operations. Furthermore, local scheduling within the basic block boundaries of non-optimized C applications used in the evaluation of their algorithm can hardly deliver high ILP rates. That may explain why many functions reported in [27] yielded no cycle overhead at all and performance of the algorithm appeared so promising. Interestingly, [27] did not take execution frequencies of the operations into account. However, we believe it is crucial to give priority to and optimize scheduling of operations/data in the loop bodies with high execution frequencies.

J. Janssen and H. Corporaal presented in [10] several heuristics for assigning variables to multiple RFs, including vertical distribution, horizontal distribution, data-independence heuristic, and intra-operation heuristic. Unfortunately, direct comparison of their results with ours is impossible, because of difference

between our VLIW and their TTA architectures. In our fully clustered target machines access to remote RFs can be carried out solely by inter-cluster copy operations, which is not the case in [10]. For comparison, though, we also implemented the horizontal distribution (termed *round-robin* in our paper).

D.J. Kolson, A. Nicolau, N. Dutt et al. presented a method for register allocation in loops minimizing spilling in multiple RF architectures [29]. Their solution exhaustively tries all possible register assignments for the live variables within the loop body. By repeating this process and taking the previous assignment mapping as input for the next iteration, the algorithm seeks an optimal assignment. This algorithm implements the idea of revisiting the first assignment similar to our *two pass feedback-directed* heuristic, but, obviously, requires much higher computing time due to the exhaustive search.

K. Kailas, M Franklin, and K. Ebcioglu identify the problem of globals in their description of the CARS algorithm in [21], which assigns global values to registers in a round-robin fashion. The only concern of the CARS compiler, though, is maintaining consistency of this assignment by keeping track of the register mappings to ensure that every definition of the same global value is assigned to the same physical register. [21] did not attempt to optimize assignment of globals to clusters. Benchmarking of our algorithms against the round-robin fashion reveals superiority of our approaches in most cases.

The RAWCC compiler [30] allocates variables and instructions of a basic block to RAW’s multiple processing units connected via a statically programmed network. First, RAWCC groups data elements and instructions depending on their affinity. An instruction has affinity with a data element, if it either consumes this element or produces the final value for this element. Then, a data-instruction placer assigns a processing unit to every data-instruction group, statically analyzing the data memory references in the code. Note that if memory references are not known statically (e.g. for unrestricted pointers in C), nothing drives the assignment in the placer. Static memory reference analysis may perform well on the RAW’s preferred array-based code, however, for irregular code rich with pointers, such as our multimedia applications, it may not suffice.

### 3. CLUSTER ASSIGNMENT ALGORITHMS FOR GLOBAL VALUES

In this section we present four simple assignment algorithms (*dense*, *round-robin*, *random*, and *shared RF*) and three complex ones (*affinity matrix 1*, *affinity matrix 2*, and *two pass*). The simple algorithms are applied to one scheduling unit at a time, in contrast to the complex assignments that simultaneously operate on all scheduling units of a HLL function. The main purpose of the simple algorithms is to estimate upper and lower performance bounds for the more complex heuristics.

#### 3.1 Dense assignment

In this case all global values are assigned to one cluster. This trivial algorithm drives cluster assignment of operations accessing the globals towards one cluster and, obviously, unbalances loads on the clusters. We use this evidently poor assignment as the lower performance bound for other heuristics. Interestingly, mapping all global values in one cluster would be beneficial for

low ILP code (e.g. our *peaking* benchmark), which fits in the issue slot(s) of one cluster.

#### 3.2 Round-robin

To overcome the imbalance of the *dense* assignment, we implemented a round-robin distribution of globals among the clusters. In this case, the globals are evenly divided among the clusters and do not disturb the load balance of the clusters. However, since the data flow and control flow graphs of the function are not taken into account, this distribution in many cases leads to unnecessary inter-cluster copies.

#### 3.3 Random search

To estimate the upper performance bound for our CA algorithms, we resorted to vast random search. We did not apply ‘smarter’ iterative algorithms (genetic algorithms, linear programming, etc.) because the random search delivered satisfactory results, and, moreover, it does not require any tuning towards the scheduling problem. The random search, including scheduling of a HLL function using 100,000 random CAs of globals, lasted on average two days on a single CPU of a HP 9000/785 workstation.

#### 3.4 Shared register file for global values

This assignment and scheduling algorithm assumes that the target architecture on top of the partitioned (local) RFs contains an added shared RF accessible without delay penalty by all clusters. The globals are allocated to these shared registers, and, consequently, incur no cycle count overhead. The local values, however, are allocated to the local RFs, and, hence, accessing them from a remote cluster causes cycle count overhead relative to the uncluster. The shared RF naturally suggests a shared resource, which contradicts to our notion of clustering. However, for example, the Sun MAJC architecture [7] avoids the shared resource by replicating the shared registers in all clusters [8]. The contents of the replicated registers are kept synchronized. The FUs always read the shared registers from the local copy, whereas the writes to the shared registers are broadcasted to all replicas, see Figure 2.

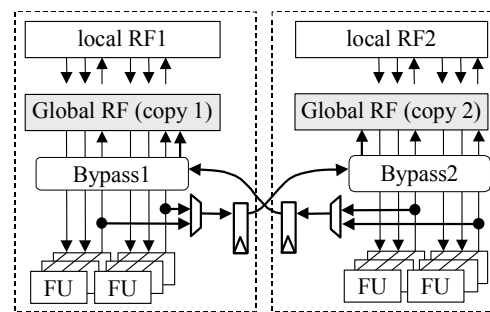
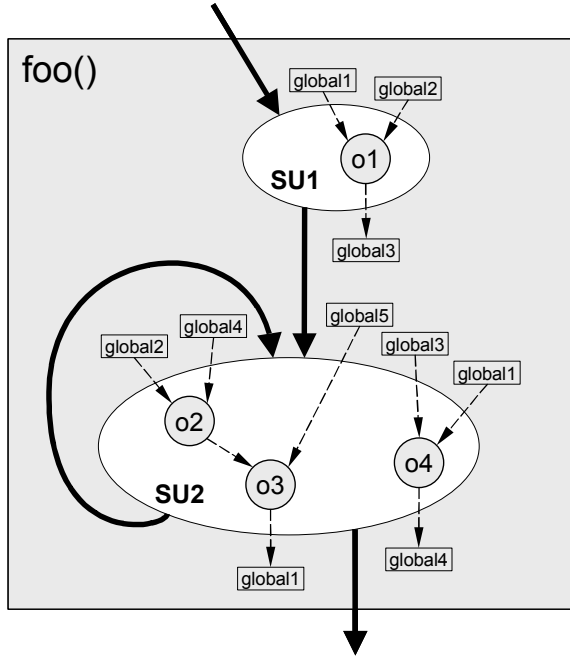


Figure 2. Globals in the control flow graph of function foo().

#### 3.5 Affinity matrix

In this algorithm prior to instruction scheduling we construct a matrix of affinities among all pairs of globals. Then, using the matrix we assign globals to the clusters and, finally, schedule the current HLL function. Note, that the affinity matrix is constructed for the whole HLL function, expanding, thus, the scheduling scope beyond the typically “small” scheduling unit. *Affinity* between two globals indicates the benefit of assigning them to the

same cluster based on the data flow graph (DFG). We further explain our affinity with a help of Figure 3.



**Figure 3. Data flow and control flow graphs of function foo().**

Figure 3 shows a fragment of the data and control flow graph of a HLL function  $foo()$  with two scheduling units  $SU1$  and  $SU2$ , four operations  $O1$ ,  $O2$ ,  $O3$ ,  $O4$ , and 5 globals. Bold arrows denote control flow graph dependencies, while thin dashed arrows – DFG dependencies. Small rectangles designate accesses to global values. Note that the same global can be accessed multiple times (e.g.  $global1$  is accessed by operations  $O1$ ,  $O3$  and  $O4$ ). The main rationale behind affinity is that globals accessed by the same or some ‘close’ operations should be allocated to the same cluster. By allocating globals with high affinities to the same cluster we can avoid long inter-cluster transfers in scheduling of some critical operations. Quantitatively, the affinity between two globals  $g1$  and  $g2$  is as follows:

$$AFF(g1, g2) = \sum_{u \in f} \left( (FREQ(u) + 1) \cdot \sum_{p(g1, g2) \in u} \frac{PRIO(g1, u) \cdot PRIO(g2, u)}{LENGTH(p(g1, g2))} \right)$$

In a nutshell, this formula assigns higher affinity to globals accessed by critical path operations with high priority from SUs with high execution frequency. Calculation of  $AFF(g1, g2)$  involves finding all accesses to  $g1$  and  $g2$  in all SUs  $u$  in function  $f$ .  $p(g1, g2)$  is the shortest sequence of DFG nodes between two accesses to globals  $g1$  and  $g2$ .  $LENGTH(p)$  is the number of nodes (operations) on the path  $p$ . For example, in Figure 3 between  $global1$  and  $global2$  there are two paths  $\{O1\}$  and  $\{O2, O3\}$  with the lengths of 1 and 2, respectively. Obviously, the longer the path, the weaker should be the affinity between the two globals, which is reflected by the  $LENGTH(p)$  term.

If a certain scheduling unit was executed multiple times (e.g. a loop body), access to globals in this scheduling unit should be optimized better than in other less frequent units. Therefore,  $FREQ(u)$  increases the score according to the execution frequency

of the scheduling unit. Execution frequencies can be obtained by profiling the code prior to the final compilation.  $+1$  after  $FREQ(u)$  ensures that we consider constraints from ‘cold’ (not executed during profiling) scheduling units too. First, this secures that our algorithm still works, if the benchmarks were not profiled at all. Second, scheduling units not executed for the current data inputs of the benchmarks may execute for other inputs. The availability of the profiling information has a big effect on the affinity. When such information is not available, static estimates about  $FREQ(u)$  could probably make more sense than setting  $FREQ(u)$  to 0 for all  $u$ . However, we did not experiment with such static estimates.

$PRIO(g, u)$  is equal to the priority of the operation accessing global  $g$  in scheduling unit  $u$ . The priority of an operation  $PRIO(o)$  reflects the urgency that it should be scheduled whenever it becomes ready for scheduling. Our instruction scheduler uses the priority function proposed by Hsu and Davidson [24][13]. For each path  $p$  from the scheduling unit entry to an exit point, the minimal completion time  $MINCOMPL(p)$  for an infinite resource machine is determined. Furthermore, we determine for each operation  $o$  on path  $p$  the latest cycle  $LATEST(o, p)$ , in which it can be placed in order to achieve  $MINCOMPL(p)$ . Operation priority  $PRIO(o)$  is computed by:

$$PRIO(o) = \sum_{p \in paths(o)} PROBAB(p) \cdot \left( 1 - \frac{LATEST(o, p)}{MINCOMPL(p)} \right)$$

where  $paths(o)$  is the set of control flow paths through  $o$ , and  $PROBAB(p)$  is the expected probability that  $p$  is executed whenever the scheduling unit is invoked. This probability is based on profiling information if available; otherwise it is estimated.

Assume that the priorities of operations  $O1$ ,  $O2$ , and  $O3$  are 0.5, 0.4, and 0.3, respectively, and execution frequency of  $SU1$  and  $SU2$  are 1 and 999, respectively. Then, for example, the affinity between  $global1$  and  $global2$  for function  $foo()$  from Figure 3 is:

$$AFF(global1, global2) = \frac{0.5 \cdot 0.5 \cdot (1+1)}{1} + \frac{0.4 \cdot 0.3 \cdot (999+1)}{2} = 600.5$$

Besides affinity, we characterize each global by *load* expressed in the number of operations directly accessing the global value. For example, in Figure 3 the load of  $global1$  is three operations  $O1$ ,  $O3$  and  $O4$ . The load is devised to promote load balancing of the operations in the issue slots of the VLIW instructions.

Based on the affinity matrix we applied two heuristics for CA of globals. In the first one termed *affinity1* we begin with selecting a group of globals with highest affinity among each other. Then we assign the whole group to the currently least loaded cluster. The load on the cluster is the sum of loads of the globals already assigned to this cluster. After evaluating performance of the algorithm with groups of 2 to 10 globals, we chose the smallest group of two globals for publication, since it yielded the best performance results. Cluster assignment of global values groups continues, until no globals are left with any affinity among each other. The rest of the global values are distributed among the clusters in a round-robin fashion.

The second heuristic termed *affinity2*, first, calculates costs of assigning each global to each cluster based on the following cost function:

$$COST(g1, c) = \sum_{g2 \in c} \frac{LOAD(c) + 1}{AFF(g1, g2)}$$

Subsequently, the global gets assigned to the cluster with the minimum cost, which is, primarily, determined by the cumulative affinities between global  $g$  and globals already assigned to cluster  $c$ . Furthermore, this cost function promotes assigning the global to the less loaded cluster.  $LOAD(c)$  is a sum of loads of all globals already assigned to cluster  $c$ . This heuristic has some similarity with [27], except for the cost function. [27] uses the number of globals as load, whereas we use the number of operations directly accessing globals as the load balancing factor in our cost function. Furthermore, we exploit execution frequencies and affinities between globals accessed by different operations. Therefore, [27] would not consider affinity between  $global4$  and  $global5$ .

### 3.6 Feedback-directed two pass assignment

The two pass algorithm also assigns globals to clusters considering the whole HLL function. This algorithm exploits the DFG, control flow graph and a probable schedule of the whole function, which gives it an advantage over the other algorithms at the cost of longer scheduling time. The two pass algorithm consists of three stages including two scheduling passes:

1. instruction scheduling, assuming a shared register file for global values. Local values are allocated to the multiple local RFs
2. feedback: calculate scores and assign global values to clusters based on these scores
3. instruction scheduling for the target clustered machine.

Note that both scheduling passes comprise operation scheduling, cluster assignment, local register allocation and spill/restore code insertion.

First, the scheduler generates schedules for all scheduling units of the HLL function for the target machine with an additional shared RF. The shared registers accommodate only global values, requiring, consequently, no inter-cluster copy operations. The first produced schedule, therefore, features no cycle count overhead associated with distribution of globals among the clusters, which is equivalent to the shared RF assignment described in Section 3.4. However, this first schedule does contain cycle count overhead relative to the uncluster due to the locals, which are kept in the local RFs. The schedule from the first pass serves as a model of the final schedule, indicating how the operations are likely to be assigned to the clusters in the final schedule.

Second, the scheduler gathers information on how the global values are accessed by the scheduled operations. Based on this information we calculate a score function  $SCORE(g,c)$  of assigning global value  $g$  to cluster  $c$ :

$$SCORE(g,c) = \sum_{u \in f} (FREQ(u) + 1) \cdot \left( \sum_{o \in u} \frac{\delta(o,g,c) \cdot PRIO(o)}{DISTANCE(o,u)} \right)$$

Each global value gets assigned to the cluster with the highest score.  $SCORE(g,c)$  traverses all operations  $o$  in all scheduling units  $u$  of HLL function  $f$  and accumulates the products of four terms. The first term  $FREQ(u)$  is the execution frequency described in Section 3.5. The term  $\delta(o,g,c)$  equals to 1, if operation  $o$  was scheduled in cluster  $c$ , and it reads or writes  $g$ , and 0 otherwise.  $\delta(o,g,c)$ , hence, rules out operations that have nothing to do with global  $g$  and cluster  $c$ . Operation priority function  $PRIO(o)$  is borrowed from the core scheduling algorithm

and described in detail in Section 3.5. Since global values are used to communicate among scheduling units, accesses to globals typically concentrate at the beginning and the end of the scheduling units. The last term  $DISTANCE(o,u)$  measures the distance (in VLIW instructions) of the operation  $o$  to the boundary of scheduling unit  $u$ . It raises importance of operations close to the SU boundaries. Obviously, scheduling copies to and from operations in the beginning or the end of the SU extends the schedule length and cannot be hidden by latencies of other operations.

In the third stage, the scheduler generates the final schedule of the current function for the target machine using the distribution of the globals among the clusters from the second stage. Then, the algorithm is applied to the following HLL function.

## 4. EXPERIMENTAL ENVIRONMENT, RESULTS AND EVALUATION

Section 4 describes our experimental compiler flow in Section 4.1, benchmarks in Section 4.2, target machines in Section 4.3 and results in Section 4.4.

### 4.1 Experimental compiler flow

We measured the cycle count overhead using the state-of-the-art TriMedia VLIW C/C++ compiler TCS2.1 [13], which carries out the following major steps:

1. optional profiling (including compiling for a uncluster, simulating the application on the uncluster machine, and instrumenting the code with profile information)
2. compiler front-end (splitting the HLL function in scheduling units, separation of global and local values, various machine-independent optimizations)
3. cluster assignment of global values (using one of our presented cluster assignment heuristics for global values)
4. instruction scheduling (cluster assignment, operation scheduling, local register assignment, spill/restore code insertion)
5. assembling and linking.

Note, that the two pass algorithm has a slightly different compilation flow described in Section 3.6.

Splitting a HLL function in scheduling units is driven by the objective of increasing potential ILP for the scheduler, while evenly splitting the compilation job (e.g. register allocation) between the front-end and the scheduler. In fact, the construction of SUs is tightly tied to whether a HLL variable is to be global or local. Thus, by adjusting the heuristics of SU selection we could shift more variables to locals and, hopefully, the scheduler having detailed information on the operation scheduling could make a better cluster assignment choice than our algorithms. However, growing SUs has also negative side-effects (e.g. code duplication, increased instruction scheduling time, etc.) and is always limited by the type of the SU (superblock, decision tree, etc.). Therefore, we did not evaluate interaction of the selection of SUs with our CA algorithms.

The scheduling unit of our compiler is the guarded decision tree of basic blocks [24][25][13]. The guarded decision tree is an acyclic control flow graph without join points, which is a more general case than superblocks or traces. Our VLIW scheduler

integrates CA, instruction scheduling, local register allocation, and spill/restore code scheduling in a single phase. Thanks to integration of the phases an instruction scheduler yields significantly denser code [11]. The compiler front-end makes approximately half of the HLL variables local and the other half – global. The maximum number of globals alive in the registers though is kept under 64 to fit in the RFs of our target machines. The global register allocation, performed by the compiler front-end, does not include CA of globals, performed later in our instruction scheduler. The instruction scheduler is responsible for CA of operations as well as local and global values. Local variables are always assigned to the cluster, where the operation producing the local, executes. CA of globals is performed by the experimental algorithms as described in Section 3. A more detailed description of our scheduler can be found in [16] and [13].

## 4.2 Benchmarks

Our benchmark suite consisted of multimedia C programs optimized for the TriMedia VLIW media processor, which features a rich SIMD operation set [14][15]. To cover a significant area of the application domain, the benchmarks were chosen from different media application categories, see Table 1.

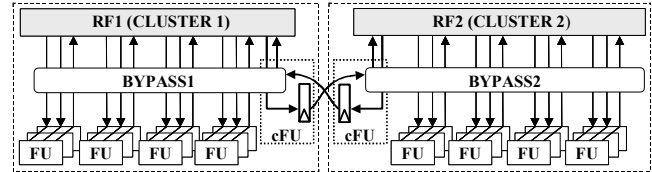
**Table 1. Optimized benchmarks**

Benchmark	Category	ILP	# DFGs	# C functions	lines of C code
Viterbi decoder	Data communication	4.1	17	3	140
Peaking	Video processing	2.3	183	15	2544
Median filter	Video processing	3.3	61	4	588
MPEG II encoder	Video coding	4.5	427	65	12649
Layered Natural Motion	Video processing	3.2	1036	109	16025
DVC decoder	Video coding	4.3	140	19	4491
Renderer (MESA 3D)	3D graphics	2.8	542	66	7159
Transform (MESA 3D)	3D graphics	3.3	34	7	1399

The benchmarks underwent optimizing source-to-source transformations to increase ILP. Furthermore, the code was enhanced with 64-bit SIMD intrinsics, cache prefetch operations, loop unrolling, software pipelining, function inlining, restricted pointers, etc. The presented ILP rates were measured dynamically in the simulator for the uncluster 8 issue slot machine. Note, that SIMD operations are equivalent to 2-24 RISC operations. This makes the actual ILP for the optimized code much higher than presented in Table 1. In total, the optimization of these applications brought 10x-40x speedups with respect to the initial source code. We believe that it is heavily optimized benchmarks, reflecting high utilization of the target machine, that should be used to evaluate compiler/architecture performance as opposed to non-optimized mostly sequential code. In the low ILP code (as used in evaluations in many other publications) the compiler does not get confronted with complex and dense DFGs, and cluster assignment of globals is far less critical.

## 4.3 Target machines

Performance of a clustered VLIW processor strongly depends on the implementation of inter-cluster communication (ICC) in the instruction set architecture [16]. In this paper we evaluate the ICC implemented in *dedicated issue slots*, where ICC is executed in additional dedicated slots, not interfering, therefore, with regular operations, see Figure 4. Issue slots with inter-cluster copy FUs (*cFUs*) are dedicated solely for inter-cluster communication, whereas other slots with FUs execute regular operations. Note that in our machines each issue slot contains multiple FUs.



**Figure 4. Two-cluster 8 issue slot VLIW machine.**

According to our previous studies [16], this ICC implementation strikes a good balance between the cycle count performance and hardware complexity influencing the cycle time. The dedicated issue slots model alleviates ICC relative to the other ICC models (e.g. with explicit copy operations or send/receive operations), and, consequently, poses a bigger challenge for our cluster assignment algorithms than many others. We believe that the other ICC models, where inter-cluster transfers are highly penalized, may benefit from our presented algorithms to a greater extent.

The baseline uncluster architecture is an 8-issue slot VLIW with the TriMedia operation set. All operations are pipelined and can contain a guard (predicate), two operands and one result. The distribution of the function units among issue slots was made such, that the derived clustered architectures contained equal functionality in all their clusters. The latencies of the TriMedia function units are given in Table 2. The two-cluster architectures used in our experiments have slots 1, 2, 3, and 4 in cluster one and slots 5, 6, 7, and 8 in cluster two. Each of the two clusters possesses 64 registers. The four-cluster architectures have slots 1 and 2 in cluster one, slots 3 and 4 in cluster two, slots 5 and 6 in cluster three, and slots 7 and 8 in cluster four. Each of the four clusters contains 64 registers. The inter-cluster bandwidth is set to one inter-cluster transfer per cluster per VLIW instruction for all clustered machines. An inter-cluster transfer takes one cycle. All our target machines have ideal memory causing no extra cycle penalty on loads and stores.

**Table 2. Baseline uncluster architecture**

Function Units	Issue slots	Latency
alu, shifter	1,2,3,4,5,6,7,8	1
imul, fmul	1,3,5,7	3
load/store	2,4,6,8	3
branch	1,3,5,7	4

## 4.4 Results and evaluation

Table 3 and Table 4 present the execution cycle count overhead of clustered VLIW machines with respect to the baseline uncluster machine. We used the uncluster as our baseline to put our performance results in perspective with other publications on clustering in VLIWs.

**Table 3. Performance results for the 2 cluster VLIW machine**

benchmarks	unicluster	dense	round-robin	sharedRF	affinity1	affinity2	2pass	random
viterbi	100.00%	111.85%	111.80%	105.89%	111.78%	100.09%	105.91%	-
peaking	100.00%	110.58%	105.41%	100.10%	107.03%	103.22%	103.45%	-
median	100.00%	131.60%	118.28%	117.14%	114.67%	114.11%	117.55%	-
mpeg2enc	100.00%	116.64%	112.49%	107.80%	106.49%	105.70%	113.19%	-
natmot	100.00%	111.12%	110.89%	104.81%	107.02%	108.64%	106.76%	-
dvc_dec	100.00%	109.73%	107.42%	104.41%	106.69%	107.23%	106.01%	-
renderer	100.00%	106.93%	106.44%	101.51%	104.35%	102.56%	102.65%	-
transform	100.00%	111.11%	111.12%	102.69%	106.52%	105.99%	106.01%	-
<i>arithmetic mean</i>	<i>100.00%</i>	<i>113.69%</i>	<i>110.48%</i>	<i>105.54%</i>	<i>108.07%</i>	<i>105.94%</i>	<i>107.69%</i>	-

**Table 4. Performance results for the 4 cluster VLIW machine**

benchmarks	unicluster	dense	round-robin	sharedRF	affinity1	affinity2	2pass	random
viterbi	100.00%	146.87%	129.36%	117.58%	117.74%	117.65%	117.65%	111.80%
peaking	100.00%	106.41%	113.55%	105.92%	111.10%	110.96%	107.28%	105.66%
median	100.00%	152.22%	124.48%	114.21%	124.84%	124.91%	128.14%	114.42%
mpeg2enc	100.00%	113.93%	116.19%	108.11%	114.78%	114.27%	111.95%	-
natmot	100.00%	129.92%	117.79%	107.94%	116.80%	116.21%	114.87%	-
dvc_dec	100.00%	116.60%	112.16%	107.80%	112.30%	113.28%	112.74%	-
renderer	100.00%	114.52%	111.18%	103.70%	109.94%	110.01%	107.97%	-
transform	100.00%	129.83%	113.29%	105.61%	113.61%	114.48%	112.39%	107.14%
<i>arithmetic mean</i>	<i>100.00%</i>	<i>126.29%</i>	<i>117.25%</i>	<i>108.86%</i>	<i>115.14%</i>	<i>115.22%</i>	<i>114.12%</i>	<i>111.12%</i>

Each benchmark was compiled and simulated using six different methods of distributing global values among the clusters: *dense* assignment see Section 3.1; *round-robin*, see Section 3.2; *sharedRF*, see Section 3.4; *affinity1* and *affinity2* from Section 3.5; *2pass*, see Section 3.6. Before executing complex algorithms (*affinity1*, *affinity2*, and *2pass*) all benchmarks were profiled on the 8 issue slot unicluster VLIW machine.

According to Table 3 and Table 4, the trivial *dense* assignment shows a high performance loss of up to 13.7% for our two-cluster machine and 26.3% for our four cluster VLIW machine. Analysis of the generated code reveals sever penalty due to numerous inter-cluster copies from the dedicated cluster with globals. Remarkably, the cycle count overhead of the *sharedRF*, indicating the performance level of a hardware solution, is only a half of the overheads of the trivial heuristics *dense* and *round-robin*. In fact, this high performance potential triggered our research on efficient assignment of globals to clusters. Despite its simplicity *round-robin* performs rather well on all benchmarks and target machines. The good results of *round-robin* also mean that our cluster assignment algorithm for operations can efficiently compensate for drawbacks of the even distribution of globals.

Our complex heuristics achieve higher performance than the simple algorithms due to the analysis of the cluster assignment of globals at the entire HLL function level. There is no obvious winner among the complex heuristics. This roots in diversity of our SUs, which respond differently to our CA heuristics on different machines. An interesting example of this diversity is benchmark *peaking*, which, obviously, benefits from the *dense*

assignment on the four cluster machine. Therefore, most of our CA heuristics trying to maintain load balancing show rather poor performance on this benchmark. However, for the two-cluster VLIW machine *affinity2* nearly reaches the performance of the hardware solution *sharedRF*. Remarkably, a number of benchmarks (*viterbi*, *median*, and *mpeg2enc*) scheduled with our algorithms *affinity1* and *affinity2* outperformed the *sharedRF* on a two cluster machine, see Table 3. We attribute that to the shortcomings of our instruction scheduler that failed to find better schedules for the *sharedRF* architecture on these benchmarks. The *feedback-directed two pass* heuristic outperforms simple heuristics for the two cluster machines. Note that an important premise of effectiveness of the *feedback-directed two pass* algorithm is that the final schedule resembles the schedule from the first pass. That is why the feedback-directed technique performs worse than a simple round-robin distribution on the *mpeg2enc* in the two-cluster machine and *median* in the four cluster machine. In these benchmarks the final schedule of some critical functions differed significantly from the schedule of the first pass.

Our algorithms perform rather well on the two cluster VLIW machine. To assess quality of our algorithms and find out whether higher performance is achievable for the difficult four cluster VLIW architecture, we used the random search described in Section 3.3. Unfortunately, we could not random search all our benchmarks due to time complexity. Therefore, Table 4 presents random search results of only four benchmarks *viterbi*, *peaking*, *median*, and *transform* for the four cluster machine. These results

show there is still some performance potential in CA of globals, which we consider challenging for our future explorations.

The two passes of the feedback-directed algorithm are executed internally in the scheduler, causing no time overhead in writing and reading the assembly file. We observed that the scheduling time of the two passes approximately doubles compared to other cluster assignment heuristics (*dense*, *round-robin*, *sharedRF*, *affinity1*, and *affinity2*). For our fast list scheduler, the doubled scheduling time equals to merely fractions of a second per C file. Consequently, using an estimation model of the final schedule instead of actual scheduling (e.g. see [6]) would save very little scheduling time, while resulting in less accurate predictions of accesses to globals and, hence, less efficient final schedules. Iterative algorithms, on the other hand, would extend the scheduling time to several seconds or even minutes, becoming disruptive for interactive compilation of large software. Scheduling time of the affinity-based heuristics was even shorter than that of the two pass algorithm.

## 5. CONCLUSIONS

In this study we emphasize the importance of optimizing the assignment of global values to clusters for clustered VLIW processors. We found that accessing globals in remote clusters causes approximately half of the cycle count overhead of the clustered VLIW architecture. This large portion of the overhead can be almost entirely eliminated by using a technique of broadcasting to replicated RFs at the cost of high die area and power dissipation. However, if the power (or area) is to be reduced, the cycle count can be improved in the compiler by the proposed *feedback-directed two pass* or *affinity-based* cluster assignments of globals at the cost of longer compilation time. These algorithms reduce the cycle count overhead of the best trivial algorithm *round-robin* from 10.5% to 5.9% for the two cluster VLIW machine and from 17.3% to 14.12% for the four cluster VLIW machine.

## 6. REFERENCES

- [1] S. Rixner, W.J. Dally, et al. "Register organization for media processing", In Proceedings of 26th International Symposium on High-Performance Computer Architecture, Orlando, January 1999.
- [2] R. Ho, K. Mai, and M. Horowitz, "The Future of Wires", In Proceedings of the IEEE, pp. 490-504, April 2001.
- [3] Texas Instrument *TMS320C64xx* DSP Generation. <http://www.ti.com>.
- [4] M. Levy, "ManArray devours DSP code", Microprocessor report, October 2001.
- [5] P. Faraboschi, G. Desoli, et al. "Lx: A technology platform for customizable VLIW embedded processing", In Proceedings of 27th Annual International Symposium on Computer Architecture, pp. 203-213, Vancouver Canada, June 2000.
- [6] P. Faraboschi, G. Desoli, J.A. Fisher, "Clustered instruction-level parallel processors", HPL-98-204, HP Laboratory, Cambridge, December 1998.
- [7] S. Sudharsanan, P. Sriram, et al., "Image And Video Processing Using MAJC 5200", In Proceedings of International Conference on Image Processing, Canada, September 2000.
- [8] Sun MAJC architecture tutorial, <http://www.sun.com/>.
- [9] S. Sudharsanan, "MAJC-5200: a high performance microprocessor for multimedia computing", White paper, <http://www.sun.com>.
- [10] J. Janssen, H. Corporaal, "Partitioned Register Files for TTAs", In Proceedings of 28th Annual International Symposium on Microarchitectures, Michigan, November 1995.
- [11] J. Janssen, "Compiler Strategies for Transport Triggered Architecture", PhD thesis, TU Delft, The Netherlands, 2001.
- [12] S. Roos, H. Corporaal, et al., "Clustering on the Move", In Proceedings of 4th International Conference on Massively Parallel Computing Systems, Ischia Italy, April 2002.
- [13] J. Hoogerbrugge, L. Augusteijn, "Instruction scheduling for TriMedia", The Journal of Instruction-Level Parallelism, February 1999.
- [14] S. Rathnam, G. Slavenburg, "An architectural overview of the programmable multimedia processor, TM-1", In Proceedings of 41st IEEE International Computer Conference, pp. 319-326, Santa Clara CA, 1996.
- [15] A.K. Riemens, et al., "TriMedia CPU64 Application Domain and Benchmark Suite", In Proceedings of International Conference on Computer Design, pp. 580-585, Austin Texas, October 1999.
- [16] A.S. Terechko, E. Le Thénaff, J.T.J. van Eijndhoven, H. Corporaal, "Inter-cluster communication models for clustered VLIW processors", In Proceedings of Symposium High Performance Computer Architectures, Anaheim, CA, February 8-12, 2003.
- [17] E. Ozer, S. Banerjia ad T. Conte, "Unified assign and schedule: a new approach to scheduling for clustered register file microarchitectures", In Proceedings of 31st Annual International Symposium on Microarchitectures, pp. 308-315, November 1998.
- [18] V. S. Lapinskii, M.F. Jacome, and G.A. de Veciana, "High-Quality Operation Binding for Clustered VLIW Datapaths", In Proceedings of Design and Automation Conference, Las Vegas USA, June 2001.
- [19] P. Mattson, W.J. Dally, et al. "Communication Scheduling", In Proceedings of International Conference on Architectural Support for Programming Languages and Operating Systems, 2000.
- [20] K. Kalias, M. Franklin, K. Ebcioğlu, "A Register File Architecture and Compilation Scheme for Clustered ILP Processors", In Proceedings of International Conference Euro-Par, Paderborn Germany, August 2002.
- [21] K. Kailas, K. Ebcioğlu, et al., "CARS: A New Code Generation Framework for Clustered ILP processors", In Proceedings of 7th International Symposium on High Performance Computer Architecture, pp. 133-134, January 2001.

- [22] J.H. Moreno, M. Moudgill, K. Ebcioğlu, et al., "Simulation/evaluation environment for a VLIW processor architecture", IBM Journal of Research & Development, issue 41(3), pp. 287-302, 1997.
- [23] M. Moudgill, "Implementing an Experimental VLIW compiler", IEEE Technical Committee on Computer Architecture Newsletter, pp. 39-40, June 1997.
- [24] P. Y. T. Hsu, E. S. Davidson, "Highly Concurrent Scalar Processing", In Proceedings of 13th Annual International Symposium on Computer Architecture, pp. 386-395, June 1986.
- [25] W.A. Havanki, S. Banerjia, T. M. Conte, "Treegion scheduling for wide-issue processors", In Proceedings of 4th International Symposium on High Performance Computer Architecture, February 1998.
- [26] W.W. Hwu, S.A. Mahlke, et al., "The Superblock: an Effective Technique for VLIW and Superscaler Compilation", The Journal of Supercomputing, vol. 7, pp. 229-249, May 1993.
- [27] J.A. Fisher, "Trace Scheduling: a Technique for Global Microcode Compaction", IEEE Transactions on Computers, vol. C-30, pp. 478-490, July 1981.
- [28] J. Hiser, S. Carr, P. Sweany, "Global register partitioning", In Proceedings of International Conference on Parallel Architectures and Compilation Techniques, Philadelphia, PA, October 2000.
- [29] D.J. Kolson, A. Nicolau, N. Dutt, K. Kennedy, "A method for register allocation to loops in multiple register file architectures", In Proceedings of 10th IEEE International Parallel Processing Symposium, April 1996.
- [30] W. Lee, R. Barua, M. Frank, D. Srikrishna, "Space-time scheduling of instruction-level parallelism on a Raw machine", In Proceedings of 8th International Conference on Architectural Support for Programming Languages and Operating Systems, San Jose, CA, October 1998.
- [31] J. Zalamea, J. Llosa, et al., "Modulo Scheduling with Integrated Register Spilling for Clustered VLIW Architectures", In Proceedings of 34th Annual International Symposium on Microarchitecture, Austin, Texas, December 2001.
- [32] J. Sánchez, A. González, "Clustered Modulo Scheduling in a VLIW Architecture with Distributed Cache", Journal on Instruction Level Parallelism (JILP), Volume 3, October 2001.