Instruction Assignment for Clustered VLIW DSP Compilers: A New Approach

Giuseppe Desoli
HP Laboratories Cambridge
HPL-98-13
February, 1998

E-mail: desoli@hpl.hp.com

VLIW, clustering, assignment, ILP, DSP

This report proposes a new heuristic/model driven approach to assign nodes of a computational DAG to clusters for a VLIW machine with a partitioned register file. Our approach exploits a heuristically found initial clustering to speed up the convergence of a deterministic descent algorithm. The initial configuration is determined through a longest path driven strategy that collects a number of paths or sub-dags starting from the DAG's leaves. The initial node assignment problem is then simplified to the assignment of these partial components to one of the k clusters.

We approach the component assignment problem in two different ways depending upon some heuristically detected DAG symmetries. The descent algorithm starts from the initial configuration and modifies the assignment for each partial component by minimizing a cost function being an estimate of the schedule length for all nodes in the DAG on a given machine. The estimate is carried out by a simplified list scheduler taking quantitatively into account things like register pressure, resources allocation, etc. We compared our approach with a common heuristic known as BUG (Bottom Up Greedy) on a set of scientific and multimedia-like computational kernels. Experimental results show a reduction from 5 to 50% in the static schedule length depending from the DAG's complexity, symmetry and intrinsic parallelism and from architectural parameters like number of clusters, registers banks size, etc. Best results were obtained for large DAGs (hundreds of nodes) where the assignment of nodes to clusters is determinant to reduce the inter-cluster copies and the resource conflicts; another important factor is sometimes the reduction in register spills to/from memory due to the load balancing between clusters. These results and the low computational complexity of this approach show how the proposed method can be a viable solution for node assignment in a VLIW compiler for clustered machines.

Internal Accession Date Only

Introduction

Code generation for embedded processors and DSPs has become an important research topic because of the increased presence of such devices on the market. Compiler technology is mature for general-purpose processors while many open issues remain for code generation for DSP. The complex features of DSPs' and especially ILP, have introduced new problems and expanded the classical compilers' code generation phase with some new functions. Normally some of the more important are instruction selection, register allocation and scheduling for ILP. The recent literature has proposed algorithms to tackle the complexity of some of the mentioned problems for DSPs: Leupers and Marwedel [1] recently proposed an instruction selection scheme for DSPs with complex instructions in the presence of ILP, and many other scheduling algorithms are presented in [2].

In this paper we deal with a new scheduling phase introduced by the need of partitioning a computational DAG onto k interconnected clusters for clustered DSPs. We focus our study on the partitioning of large computational DAGs like those commonly generated by optimizing compiler for scientific applications and especially multimedia and digital signal processing.

Recently, several VLIW DSPs and media processors appeared on the market or have been announced (Texas Instruments' TMS320C6xxx, Philips' Tri-Media, Chromatic's MPACT Mitsubishi's V30 [3]) requiring a solution to this problem. This new class of DSPs architectures, while simplifying hardware to a great extent, poses new problems for compiler writers in order to effectively exploit their full potential.

In this paper, we are specifically interested in finding an effective sub-optimal algorithm suitable to be implemented in a VLIW DSP compiler. The main goal is to exploit the available ILP (Instruction Level Parallelism) through the static scheduling of operations, as opposed to a superscalar where dedicated hardware exists that can carry out this task at run-time on an instruction window.

Often, to meet hardware costs and clock targets it is mandatory to partition a VLIW register file into many smaller ones, and it is also common for VLIW DSPs to have internal memory subdivided into different memory banks. Each register file or memory bank is connected directly only to a certain number of ALUs and/or computational units to reduce the number of ports, thus allowing smaller register files and smaller propagation delays ultimately leading to higher clock speeds. When the connectivity is not complete, then the association of operations to computational units becomes an important issue.

In a static scheduling paradigm, it is the compiler task to allocate operations to different ALUs in different clusters. In some cases, without special hardware, it may be the compiler's task also to directly insert copy operations when operands are needed from a register file on a different cluster.

From the compiler's point of view then we can identify two different cases:

- The hardware provides a transparent support to execute copies between register files.
- The hardware provides a non-transparent support to execute copies between register files.

In the first case the copy operation will be handled automatically by the DSP control hardware when a computational unit needs the content of a register that is not directly connected to it. It is often not possible to implement a zero delay copy, so the copy (or the copies if more than one is supported) will stall the entire machine until completed. These stalls will cause hidden cycles, that are completely 'lost' because they will not appear in the static schedule.

The second case requires the compiler to issue a *copy operation* (or more than one depending upon the connectivity) whenever a register on a different register file is needed. This copy will be scheduled just like any other operation, and will not stall the entire machine but rather delay only the operations that depend on it.

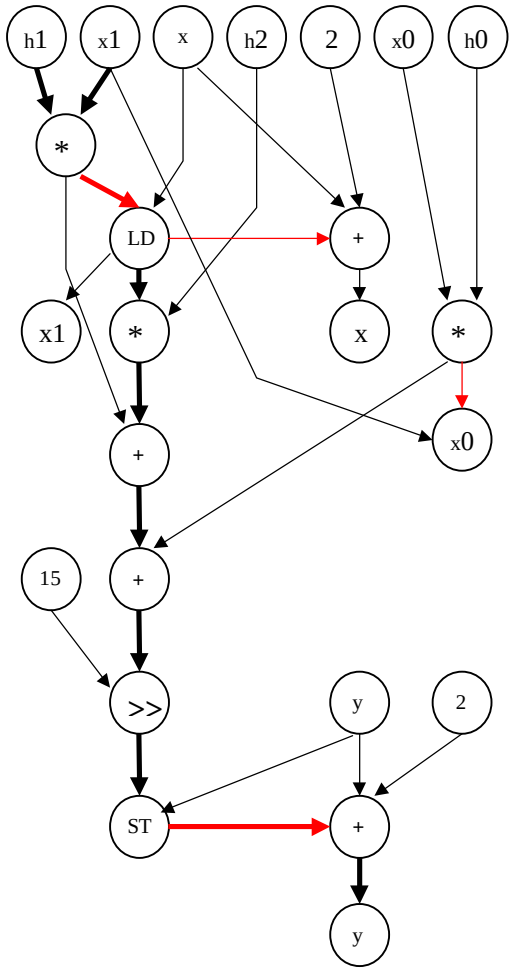

Figure 1: DAG for a 3 taps fir filter, the critical path assumes no resource limits and 3 cycles latency for a memory load, 2 cycle for a multiply and one cycle for ALU operations, red edges are not real input but constraints.

The latter case is - to some extent - harder to implement from the compiler's point of view as it changes the structure of the DAG depending upon the nodes assignment. However, it gives more control on the static schedule and potentially enables to reduce the number of stalls when compared to the first case.

Independently from the above classification, the appropriate assignment of operation to computational units may greatly reduce the number of copies required to move around values and in the end to reduce the number of cycle needed for complex DAGs.

Figure 1 shows the DAG for a simple 3 taps FIR filter (whose code is presented in the appendix). The thicker edges in the picture are used to connect the nodes on the critical path (assuming unlimited resources). Let us assume a hypothetical clustered machine where each cluster has no ILP and is only able to perform one operation per cycle. Then, for this trivial example, it is clear that to achieve the shortest schedule length (without splitting nodes) we cannot introduce any inter-cluster copy between nodes on the critical path. From the picture we can also see that only two nodes can be issued on different clusters with the necessary copies, without potentially impacting the schedule length (nodes: x0 * h0 and x = x + 2).

Ideally an optimal code generator for this kind of architectures should carry out instruction selection, partitioning (instructions assignment to computational units), register allocation and scheduling (assignment of a cycle for each operation) simultaneously. Given that clustering and scheduling are both well known NP-complete problems and considering the engineering prob-

lems involved, often they are separated and carried out independently by using heuristic and/or stochastic optimization approaches.

In this report, we focus on minimizing the impact of inter-cluster copies by pre-assigning operations to clusters. The problem is similar to the assignment problem for computational DAGs in multiprocessors system but with some important differences. It can also be seen as a *k*-way partitioning problem for a DAG where the cost function represents the final schedule length for the DAG including communication delays.

We first give a quick overview of previous similar work, then we describe our approach and we compare it experimentally against a common heuristic and to the case where only one fully connected register file exists.

We implemented our clustering algorithm within the framework of the MULTIFLOW trace compiler [4]. This compiler uses techniques like loop unrolling, function inlining, if-conversion, speculative execution, predication and a global scheduling technique known as trace scheduling [5]. All of them result in an increased size of the scheduling regions that for some applications can easily reach thousands of nodes. For such big DAGs it is extremely important to use an approach that is fast and that scale in complexity almost linearly with the number of nodes to be assigned. Finally we discuss the results and give some ideas for future directions.

Previous work

Communication costs are a well-known cause for performance degradation in multiprocessor systems [2], and being an NP-complete problem many joint scheduling/assignment algorithms have been proposed that try to solve the problem sub-optimally. Most of the existing studies approached the problem of scheduling a computational DAG of tasks on a multiprocessor system.

However almost all existing algorithms cannot be easily adopted to solve the problem of allocation of operations to clusters for a VLIW with a partitioned register file. The major limitations come from the assumption made on the structure of the DAG, or by assuming no communication delay [6][7], or by requirements on the number of processors, being either limited or infinite [8][9]. Some techniques require the nodes of the DAG to have certain properties, such as an execution time that decreases with the number of processors applied to them [10]. Interesting work has been done in [10] for the scheduling of DAGs for asynchronous multiprocessor execution. However the results cannot be applied to a clustered VLIW were all the clusters work in lockstep.

Much has been done on the multi-way partitioning problem where a graph has to be subdivided into *k* sub-sets by minimizing a given cost function. Frequently used cost functions try to balance the number of nodes in a cluster versus the number of edges (cuts) between clusters. Linear time algorithms have been proposed for a given cost function [12], and many sub-optimal ones for more complex costs. However, these approaches have only a limited applicability when the cost to be minimized is the scheduling length of a DAG given limited resources and many other complex interactions with a VLIW machine like register allocations, register spilling etc. Moreover, classical costs used do not have any relationship at all with the overall schedule length, except when the DAG has particular symmetries, like a number of separated components being a multi-

ple of the number of clusters. For these reasons we discarded the k-way partitioning approach for all cases but the highly symmetric ones, where we apply a similar strategy in the selection of the starting configuration.

The MULTIFLOW compiler uses a heuristic called BUG (Bottom Up Greedy) described in [13]. BUG recursively propagates from exit nodes of the DAG to the entry nodes and estimates the best functional units to be assigned to a node. When it reaches the entries, it works its way back to the exits, while selecting final assignments for the nodes along the way. To reach the final assignment for a node, BUG estimates the cycle in which a functional unit can compute the operation and picks up the one producing the smallest delay for the output, including the theoretically minimum delay to copy operands to another cluster if that is the case.

BUG makes some simplifying assumptions: functional units are the only limiting resources in the machine, and conflicts due to scarce register-bank ports or buses are ignored. The rationale behind these assumptions is that, if the bandwidth of the register banks is not adequate for the number of computational units that can access those banks, then the machine is probably improperly designed. Such an assumption in general holds even for clustered architectures but there are some holes left in this strategy.

- First of all, BUG ignores the additional resource costs involved in explicitly scheduling the copies in machines that require them. This capability can be easily added to BUG but still the resulting strategy will have a weakness in the global cost impact of inserting copies between two nodes instead of some others ones.
- In addition, BUG ignores the problem of register pressure (and consequently of register spilling) in the presence of which the topology of the DAG can change significantly. This is especially dangerous when register allocation is done by the scheduler afterwards.
- Finally, the local cost is driven only by the delay of scheduling a node on a given computational unit and the impact on the global schedule is only taken into account by giving precedence to nodes on the critical paths.

Given these weaknesses, BUG still produces quite reasonable results with a reasonably low computational complexity. In the rest of the paper, we will show how we can achieve a significant improvement in the presence of very high parallelism and regularity, such as in scientific computational kernels.

Algorithm description

The basic data structure we use is the DAG that describes the compilation regions (basic blocks or traces) in terms of indivisible units of execution expressed by atomic nodes, and their dependencies represented by the edges between them. In this model a set of v nodes $V=\{n1,n2,....,n_v\}$ are connected by a set of e directed edges, each of which is denoted by (n_i,n_j), where node n_i produces an input for (or simply constrains) node n_j. A node without incoming edges is an entry node (or a root) of the DAG, and a node without output(s) is an exit node (or a leaf) of the DAG. The weight of a node, denoted by $w(n_i)$ represents its execution time (or pipeline latency) in cycles.

When an input for a node is needed from a node requiring a non zero delay to transfer the result, then there are two choices to model it; one assigns to the edge between them a non zero cost $d(n_i,n_j)$ being the number of cycles required to transfer the value, while the second splices a new node in between being a copy operation node. The second representation is more general and is more useful in the case where copies must be scheduled explicitly by the compiler when compiler transparent copies are not available.

Even when the hardware provides such transparent inter-cluster copies, it may be better to model them explicitly in order to prevent as much as possible machine-wide stalls in case of conflicts.

We then describe the DAG $D=(V,E)$ as the union of the set of nodes and their connecting directed edges. Given K the fixed number of architecturally visible clusters (each one having at least one register file), our problem (clustering) is to find k sets of nodes $C=\{S_1,S_2,...,S_k\}$ where each node n_i must be contained in one and only one set, in such a way as to minimize $L(D,C,A)$, L being the schedule length of the DAG D, given the node assignment C, subject to the resource constraints imposed by the architecture A. In principle, L depends on the scheduling algorithm used to assign a cycle for every node in D, in practice however we minimize L with respect to a highly simplified list instruction scheduler for A, taking into account only quantitatifiable register pressure, register spills etc. To speed up the convergence process and to reduce the problem dimensionality, we pre-clusterize together nodes according to their respective 'criticality' in the original DAG, producing a set of partial components by using the following procedures:

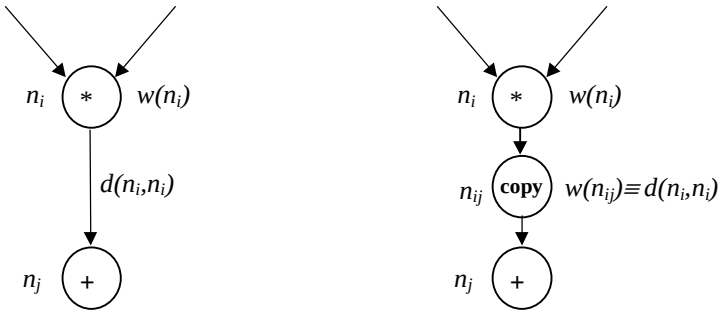

Figure 2 Assuming node ni and nj are issued in different clusters, then this two representations are equivalent but the right one allows the compiler to model copies explicitly, and if required to schedule the necessary resources.

procedure *PARTIAL_COMPONENTS*(D, ϕ_{th})

$F=\{l_1,l_2,...l_f\} \in$ D and $(l_i,n_j) \notin$ E $\forall$ i in [1,f], j in [1,v] /* the set of f exit nodes or leaves of the DAG */

$\Phi=\{\varnothing\}$ /* list of partial components initially empty */

FOR EACH($l_i \in F$)

$\phi=\{\varnothing\}$ /* start a new component */

LONGEST_PATH_GROWTH(Φ,ϕ,ϕ_{th},l_i)

ENDFOR

RETURN

procedure *LONGEST_PATH_GROWTH*(Φ,ϕ,ϕ_{th},l_i)

IF ($l_i \in \phi_i \; \forall \; \phi_i \in \Phi$) **THEN RETURN** /* l_i has been assigned already */

IF (SIZE(ϕ) > ϕ_{th}) **THEN**

$\Phi = \Phi \cup \{\phi\}$ /* add ϕ to the set of partial components Φ */

$\phi = \{\varnothing\}$ /* start a new component */

ENDIF

$\phi = \phi \cup \{ l_i \}$

$P = \{ p_1,p_2,...,p_n)$ *such that* $(pj, l_i) \in E$ /* all predecessor of node l_i */

ORDER_DEPTH(P) /* order the predecessors by their decreasing DEPTH as defined by the topological order in a depth first search over the DAG D taking $w(n_i)$ into account for every node. */

FOR EACH($p_j \in P$)

LONGEST_PATH_GROWTH(Φ,ϕ,l_i) /* follow the longest path */

ENDFOR

RETURN

Where D is the original DAG, and Φ is the set of partial components.

After this, we grow partial components starting from the DAG's leaves and following the longest path backward towards the DAG's roots until we hit a threshold ϕ_{th} for the maximum number of nodes in a partial component. When *LONGEST_PATH_GROWTH* reaches an entry point or root of the DAG or a node already visited, the recursion restarts along one of the pending paths originating from one of predecessor nodes in the stack of recursive calls. In this way, we add paths to the partial components in a "critical path first" fashion. The rationale behind this is to allow us to insert copies if needed preferably off the most critical paths in the DAG.

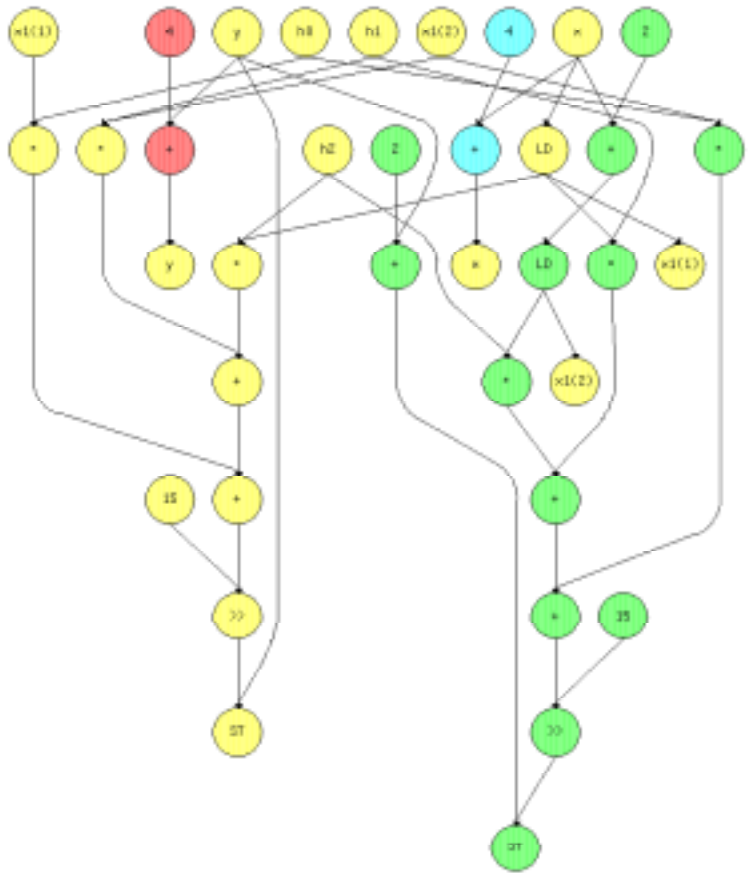

Figure 3: DAG for a 3 taps FIR filter unrolled twice. We can see four partial components (different colors), two of which correspond to the two unrolled loop iterations. This DAG is unconnected according to our loose definition of connection when following edges backward starting from the leaves.

Figure 3 shows this concept for a simple DAG being the same fir filter of Figure 1 but unrolled twice. By setting $\phi_{th}=\infty$ we get 4 partial components, two of them being the main ones representing the two unrolled iterations of the original loop. In general for infinite value of the threshold ϕ_{th}, two different situations may arise.

If the DAG is fully connected, then the process will produce only one component consuming the entire DAG. However if the DAG is made of several *separated* DAGs, then the process produces a number of components each one containing an unconnected part of the DAG.

Here, we use the concept of separation in a loose sense, as we only explore the DAG following edges backward. A typical case of this arises from loop unrolling, especially in the absence of loop carried dependencies across iterations.

It is not uncommon for the machine clusters to be homogeneous and for the main separated components of the DAG (like those generated by loop unrolling) to be isomorphic. In such cases we experimentally found that allocating each partial component to a given cluster by minimizing the number of inter-cluster copies generally leads to good solutions if some conditions are satisfied. In addition, this greatly improves the convergence process of the iterative phase.

The set Φ of n partial components is then used to perform a simple mapping n -> k to produce a k-way clustering. In the case of fully connected DAGs, the mapping simply scans the set Φ (reordered by decreasing size of its elements) for all partial components assigning them to the least utilized cluster based on the previous assignments.

procedure *ASSIGN_PARTIAL_COMPONENTS*(Φ,C)

$S_1=S_2=...=S_k=\{\varnothing\} \quad \forall\, S_j \in C$

FOR EACH($\phi_j \in \Phi$)

find S_{min} such that Size(S_{min}) = min{ $Size(S_1),Size(S_2),...,Size(S_k)$ }

$S_{min} = S_{min} \cup \{\phi_j\}$

ENDFOR

RETURN

This simple mapping effectively achieves a load balancing between clusters, but it isn't guaranteed to produce a good initial assignment for DAGs made by separated sub-DAGs, since it does not take into account inter-cluster copies in the mapping process.

To obviate this problem, we can modify the mapping for the "separated" DAGs by assigning each component to a cluster by minimizing a cost function that takes copies into account. To do this we start by computing a matrix M_Φ where each entry m_{ij} represents the number of directed edges between partial components ϕ_i and ϕ_j.

Starting from an empty assignment, allocating two partial components on the same cluster corresponds to replacing the corresponding rows and columns in the matrix M_Φ by their respective sums for all elements but with the (i_{th},j_{th}) being zeroed. In fact, it is intuitive to think of this as merging together two partial components to produce a new component requiring a number of copies with the remaining components being the sum of the original ones. While the copies needed between the components now on the same cluster are no longer necessary.

Given a closed form cost function, this problem can be formulated and solved as an integer variables linear programming problem with constraints. However, to simplify our approach, we produce a mapping by reducing the connection matrix M_Φ and by minimizing a compound cost for load balancing and number of copies between partial components iteratively.

Although sub-optimal, for the kind of DAGs we mentioned above this strategy produces a good initial assignment provided we have made an appropriate selection of the threshold ϕ_{th}. Practically, ϕ_{th} has to be chosen as a compromise between the algorithm's running time and the number of partial component to be processed by the iterative phase. We iterate the initialization phase a few times progressively decreasing ϕ_{th} (thus increasing the number of partial components) having as a feedback the estimate on the schedule length and saving the best initial state found during the process.

The entire initialization phase is described in the following pseudo-code:

function *INITIAL_ASSIGNMENT(*D)

/* determine the class of DAG we are dealing with */

Φ_0 = *PARTIAL_COMPONENTS*(D, ∞) /* $\phi_{th} = \infty$ determines the number of

un-connected components */

/* F_k is a constant dependent upon the number of clusters k */

IF (Size(Φ_0) > F_k) **THEN** *DAG_is_unconnected = TRUE*

ELSE *DAG_is_unconnected = FALSE*

ϕ_{th}= *Initial*-ϕ_{th}(Φ_0,k)

$L_{min} = \infty$

$\Phi_{min} = \{\varnothing\}$

$C_{min} = \{\varnothing\}$

DO

Φ = *PARTIAL_COMPONENTS*(D, ϕ_{th})

IF (*DAG_is_unconnected)* **THEN**

M_Φ = *BUILD_CONNECTION_MATRIX(D,*Φ)

ASSIGN_PARTIAL_COMPONENTS(Φ,C, M_Φ)

ELSE *ASSIGN_PARTIAL_COMPONENTS(*Φ,C)

IF (L_{min} > *L(*Φ*,C,A)*) **THEN**

L_{min} = *L(*Φ*,C,A)*

$\Phi_{min} = \Phi$

C_{min} = C

ENDIF

ϕ_{th}= *Next*-ϕ_{th}(Φ,ϕ_{th} ,k)

WHILE *Stop*-ϕ_{th}(Φ,ϕ_{th} ,k, L_{min})

RETURN(Φ_{min},C_{min})

Initial-ϕ_{th} ,Next-ϕ_{th} and Stop-ϕ_{th} are simple functions used to drive the initialization phase, in our experiment we set the initial value for ϕ_{th} such that the number of resulting partial component was a compromise between accuracy and speed. The assignment $C=\{S_1,S_2,...,S_k)$ produced by the initial phase is improved by an iterative descent algorithm that refines C by modifying the initial choice made for every element in the set Φ of partial components.

We investigated two different strategies:

- The first strategy orders Φ by decreasing size of its component and then tries to keep the cluster's loads balanced (assuming homogenous clusters) by swapping two element ϕ_i and ϕ_j, when the schedule length L produced by the swap is smaller.
- The second strategy simply evaluates L for any possible assignment of $\phi_i \in \Phi$ to a cluster and retains the one that leads to the shortest schedule L.

Both algorithms are deterministic and always follow the direction of maximum local descent. Thus they tend to get trapped into local minima.

As mentioned, the descent is driven by the estimate on the schedule length obtained through a simplified model of a real scheduler. The model is based on a simple list instruction scheduler, which uses a ready node queue ordered by the nodes' respective priorities. The model keeps track quantitatively of allocated and deallocated registers, of number of copies needed, and of register spills. The resources allocation for every cycle is modeled based on the architecture A.

The accuracy of the estimate for our implementation is on the average about ±10% of the schedule length generated by the real scheduler in our compiler, while the execution time is more than 10 times faster.

The schedule length $L(D,C,A) \equiv L(\Phi,C,A)$ as a function of the clusterization C, presents many flat areas (plateaus). Hence, we need an alternative criterion to drive the descent when the function gets stuck in a plateau. After some experiments, we decided to use the number of copies as a secondary criterion to choose a direction in flat areas of the target cost function. It turned out that such a number is more sensitive to small changes of the configuration in C and it is relatively easy to compute. Figure 4 shows a convergence process for a convolution computational kernel.

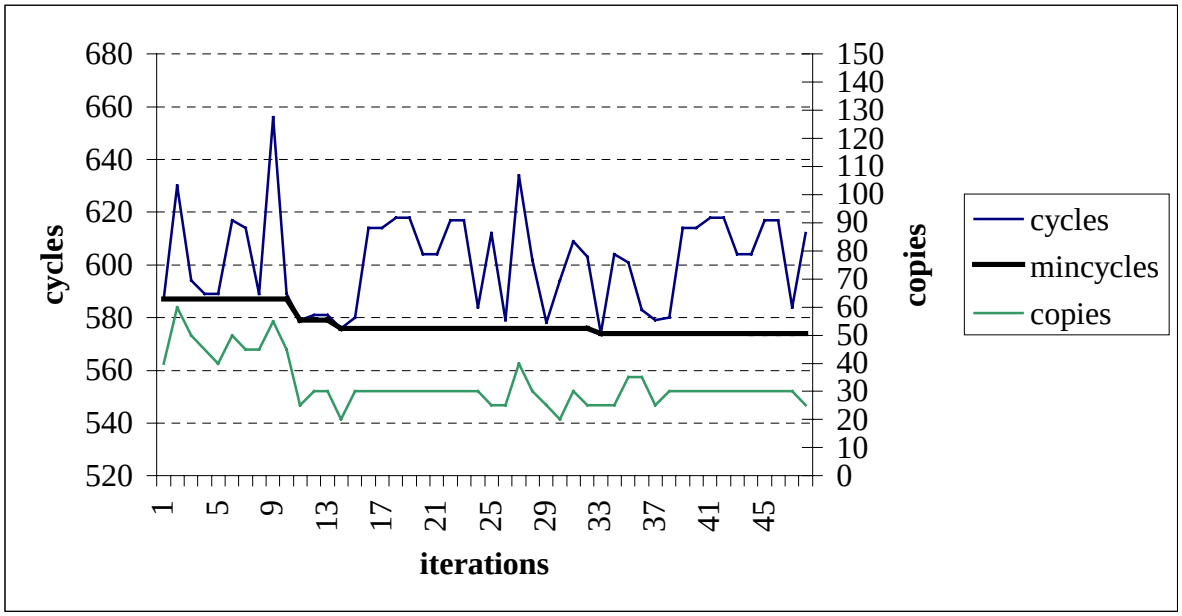

Figure 4 Evolution of the descent for one of the tested multimedia kernels (convolution). The bold line is the lowest number of cycles found so far. It is evident from the graph that the number of copies is locally related to the minimum but it can't be used as a global criterion for the descent.

The descent phase is described in the following pseudo-code for the two cases:

- The general case (in which the DAG is connected):

procedure *OPTIMIZE_ASSIGNMENT (*Φ,C,iterations)

(L_{min} , $Copies_{min}$)= *L(*Φ,*C,A)*

DO

progress = **FALSE**;

FOR EACH($\phi_j \in \Phi$) /* *for all partial components* */

find *x* such that $\phi_j \in S_x$ /* *find cluster containing* ϕ_j */

$S_x = S_x - \{\phi_j\}$

$k_{min} = x$

FOR EACH($S_k \in C, k \neq x$) /* *for all possible assignment of* ϕ_j */

$S_x = S_x \cup \{\phi_j\}$

(L,Copies) = $L(\Phi,C,A)$

IF($L < L_{min}$ || ($L == L_{min}$ && $Copies < Copies_{min}$) **THEN**

$k_{min} = k$

$L_{min} = L$ /* *main descent criterion* */

$Copies_{min}$ = Copies /* *secondary criterion* */

progress = **TRUE**;

ENDIF

$S_x = S_x - \{\phi_j\}$

ENDFOR

$S_{kmin} = S_{kmin} \cup \{\phi_j\}$

ENDFOR

WHILE(iterations-- && progress == **TRUE**) /* *while max number of iterations done* */

RETURN /* *or no more progress are made* */

- **The case in which the DAG is unconnected:**

procedure *OPTIMIZE_ASSIGNMENT_LOAD_BALANCE (*Φ,C,iterations)

(L_{min} , $Copies_{min}$)= *L(*Φ,*C,A)*

DO

progress = **FALSE**;

FOR EACH($\phi_j \in \Phi$) /* *for all partial components* */

find x such that $\phi_j \in S_x$ /* *find cluster containing* ϕ_j */

find y and i such that $\phi_i \in S_y$, $x \neq y$, $i > j$ /* *find next* ϕ_i *on a different cluster* */

$S_x = S_x - \{\phi_j\}$ /* *try to swap* ϕ_i *and* ϕ_j */

$S_y = S_y - \{\phi_i\}$

$S_x = S_x \cup \{\phi_i\}$

$S_y = S_y \cup \{\phi_j\}$

(L,Copies) = $L(\Phi,C,A)$

IF($L < L_{min}$ || ($L == L_{min}$ && $Copies < Copies_{min}$) **THEN**

$L_{min} = L$ /* *main descent criterium* */

$Copies_{min}$ = Copies /* *secondary criterium* */

progress = **TRUE**;

ELSE

$S_x = S_x - \{\phi_j\}$ /* *swap* ϕ_i *and* ϕ_j *back to the original clusters* */

$S_y = S_y - \{\phi_i\}$

$S_x = S_x \cup \{\phi_i\}$

$S_y = S_y \cup \{\phi_j\}$

ENDIF

ENDFOR

WHILE (iterations-- && progress == **TRUE**) /* *while max number of iterations done* */

RETURN /* *or no more progress are made* */

Experimental results

Our compiler is retargetable through an architecture machine description, so we generated code for three equivalent machines in terms of total number of register (128) and computational units (8 ALUs and 4 multipliers) but with different number of clusters (1,2,4). This allows us to compare the results versus a reference machine composed of only one cluster and one register file fully connected to the computational unit.

The maximum issue width is the same for all configurations and for all the experiments we set it to a large enough number so that issue limitations do not affect scheduling. The clustering procedure is switched off for the single cluster machine and in this case the schedule length should be theoretically the shortest one (in very few cases the non-linearity of the scheduling may negate this).

For all the machines we run a set of experiment on a number of computational DAGs coming from the innermost loops of digital signal processing and multimedia-like kernels: color-space conversion, DCT/IDCT, convolution, FIR filtering, halftoning, etc. with and without the clustering optimization.

Figure 5 summarizes the results for all the benchmarks.

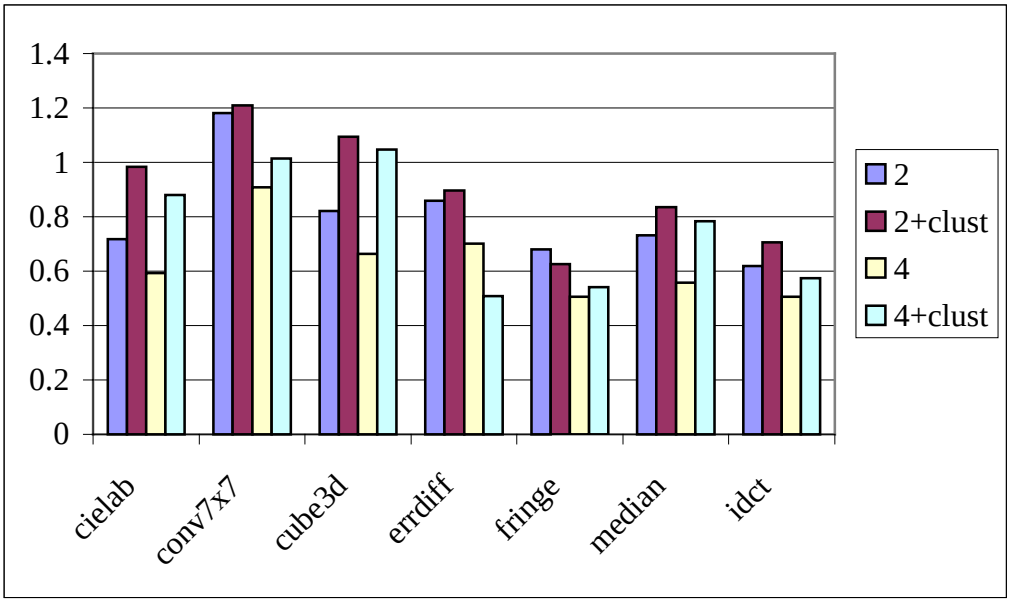

Figure 5 The graph shows the speedups relative to the 1 cluster reference machine for the two different machines with and without clustering optimization.

In all the experiments (but one), the clustering optimization gives better results than the BUG heuristics, and in some cases provides big improvements (up to 50% reduction in the cycle count).

In a few cases, the speedups are bigger than 1.0, meaning that the clustered machines are faster than the reference. This can be explained by the non-linear behavior of the list scheduler and specifically by its greediness while allocating registers. For a clustered machine, the insertion of intercluster copies modifies the structure of the DAG, and in some cases may reduce register spilling.

Finally, it is worth reporting that the increase in compilation time was not excessive in the average going from 1.5 to 2 times and only in few cases bigger than 2 times the original.

Conclusion and future directions

From our result it is evident that clustered architectures are efficient with respect to single register file machine only if ad hoc clustering heuristics are used.

The approach proposed, however, still suffers from some drawbacks. The need for low computational complexity forced us to develop an initialization phase that essentially exploits the DAG's symmetries. This can lead to wrong clustering in those cases where such symmetries are not pre-

sent. The proposed technique however, behaves quite well in presence of loop unrolling optimization, but it is still to be validated in the presence of more complex high-level optimization such as software pipelining. Also the pre-clustering phase does not take into account the balance between classes of machine operations for a given component, thus possibly leading to wrong decision in the presence of highly unbalanced DAGs. For example, in a DAG with a high density of multiplies (relative to ALU operations), the initialization phase tends to select initial partially connected components that are probably unbalanced.

The novelty of the proposed approach is the use of a simplified list scheduler to drive a global descent optimization phase that, for the computational kernels used, outperforms local blind heuristics without feedback.

Constraints posed by compilation time pose limitations upon the use of fully exhaustive searches of the solution space. As a consequence, we believe that future directions should investigate:

- Different heuristics for pre-selecting initial components according to the specific structure of a DAG
- Backtracking procedures in the descent phase to break up partial components can be another promising direction.
- The integration of instruction selection and register allocation with the assignment in order to be able to optimize not only performance, but also other criteria such as code compaction in presence of complex instructions.

References

[1] R. Leupers, P. Marwedel, “Instruction selection for embedded DSPs with complex instructions”, Proceedings EURO-DAC ’96. European Design Automation Conference with EURO-VHDL ’96 Geneva, Switzerland 16-20 Sept. 1996, pp. 200-205

[2] Ahmad, I.,Yu-Kwong Kwok,Min-You Wu “Analysis, evaluation, and comparison of algorithms for scheduling task graphs on parallel processors”, Proceedings. Second International Symposium on Parallel Architectures, Algorithms, and Networks (I-SPAN '96), IEEE Comput. Soc. Press, 12-14 June 1996 pp. 207-213

[3] M. Kagan, “The P55C Microarchitecture: The First Implementation of MMX Technology”, Hot-Chips 8, Stanford, CA, Aug. 1996, pp. 5.2

[4] P.G. Lowney, S.M. Freudenberger, T. J. Karzes, W. E. Lichtenstein, R. P. Nix, J. S. O’Donnell, J. C. Ruttenberg. The Multiflow Trace Scheduling Compiler. The Journal of Supercomputing 7, 1&2 , May 1993, 51-142.

[5] B.R. Rau, J.A. Fisher. Instruction-Level Parallelism. The Journal of Supercomputing 7, 1&2 , May 1993, 9-50.

[6] J.J. Hwang, Y.C. Chow, F.D. Angers and C.Y. Lee. “Scheduling Precedence Graphs in systems with Interprocessors Communication Times”, SIAM J. Computing, Vol. 18, pp 244-269, 1989

[7] K.B. Irani and K.W. Chen, Minimization of Interprocessor Communication for Computation. IEEE Trans. Comput., Vol. c-31, pp 1067-1075, 1982

[8] H. Jung, L. Kirousis and P. Spirakis. Lower bounds and Efficient Algorithms for Multiprocessor Scheduling of DAGs with communications delays. ACM Proc. Symposium on Theory of Computing (STOC), pp 254-264, 1989

[9] P. Markenscoff and Y.Y. Li. An Optimal Algorithm for Scheduling the Nodes of a Computational Tree to the Processors of a Parallel System. Proc. Of the 1991 ACM Computer Science Conference, pp 256-297, 1991

[10] G.N. Srinivasa Prasanna, , B.R. Musicus, Generalized multiprocessor scheduling for directed acyclic graphs. Proceedings Supercomputing '94 IEEE Comput. Soc. Press 14-18 Nov. 1994 pp. 237-246

[11] B.A. Malloy, E.L. Lloyd, M.L.Soffa, Scheduling DAG's for asynchronous multiprocessor execution. IEEE Trans. Parallel Distrib. Syst. Vol 5 no 5 May 1994 pp. 498-508

[12] S.T. Barnard, H.D. Simon, "A fast ,ultilevel implementation of recursive spectral bisection for partitioning unstructured problems". NASA AMES Research Center, Tech. Rep. RNR-92-033, Nov. 1992.

[13] J.R. Ellis, "Bulldog: A Compiler for VLIW Architectures", Doctoral Dissertation, MIT Press Cambridge MA 1985.

Appendix

```
void fir(const short x[], const short h[], short y[], int len)
{
   int j, sum;
   short x0, x1, x2, h0, h1, h2;

   h0 = h[0];
   h1 = h[1];
   h2 = h[2];
   x0 = x[0];
   x1 = x[1];

for (j = 1; j < len-1; j++)
   {
      x2 = x[j+1];
      sum = x0 * h0 + x1 * h1 + x2 * h2;
      y[j] = sum >> 15;
      x0 = x1;
      x1 = x2;
   }
}
```