# Design Considerations for Limited Connectivity VLIW Architectures*

Andrea Capitanio, Nikil Dutt, Alexander Nicolau
Department of Information and Computer Science
University of California, Irvine
Irvine, CA 92717

Technical Report #92-59



*This work was supported in part by the Italian "Dottorato di Ricerca Program, NSF grant CCR8704367, MIP9009239 and ONR grant N0001486K0215.

# 1 Introduction

VLIW machine approaches have received great interest recently, due to the fact that they are inherently suitable architectures for exploiting the parallelism extracted by fine-grain compilers that look beyond basic blocks [15] [24] [8]. The ideal VLIW machine has a number of concurrently executing functional units fed by a memory with a large data-bandwidth that can issue two operands per functional unit and perform one write per functional in each cycle [11] [14] [25] [24] [16]. Furthermore, smaller feature sizes for VLSI, together with better yields for large die sizes allow us to realize several million transistors and hence multiple FUs and RFs on a single chip or a multi-chip module (MCM). This trend of large-capacity chips and MCMs is expected to continue, making the realization of a complete VLIW on a chip a distinct feasibility in the near future.

A few commercial VLIW-like machines have already appeared on the market. Early examples of this approach are the Intel i860 [3] [18] that can issue two operations per cycle, the IBM System 6000 [26] that can execute 4 operations concurrently, and the Multiflow TRACE [9] which was designed to allow the concurrent execution of up to 28 operations per cycle. Although all of these architectures pursue the goal of executing multiple operations concurrently, none of them has been built using the "ideal" VLIW architecture.

The variations are necessary because the practical implementation of the ideal VLIW model is constrained by several technological considerations, the most critical being the inability to build a register file (RF) that has an excessively large number of ports to store the VLIW code.

Although multi-port memory cell technology has been around for several years [2], there are no consolidated technologies for building RFs with a large (say more than 16) number of ports RF [12] [17]. Even if such a large multi-port RF could be built, it may suffer some performance degradation due to the fact that each memory cell has to drive all output ports (i.e. all data substreams) in the worst case. Such an overhead may greatly affect the benefits of having a large number of ports on a RF for achieving large data bandwidth to the multiple functional units in the VLIW. This exact problem was faced by the designers of the Multiflow TRACE [9], who concisely noted: ".. any reasonably large number of functional units requires an impossibly large number of ports to the register file... The only reasonable implementation compromise is to partition the register files...".

Hence, a more realistic VLIW architecture needs to have its code partitioned into multiple (port-limited) register files, which we refer to as a *Limited Connectivity VLIW* architecture. Several benefits accrue from using this architecture. The first and most important, of course, is

realizability. The second is scalability: each VLIW partition can be built and replicated to fit the size of the final machine. The third is the choice of designing a partition using standard, off-the-shelf, low-cost components, or using a custom/semi-custom design approach. Finally, the partitioned VLIW scheme enhances testability and maintainability, since a faulty partition can easily be replaced or replicated.

In this paper we describe the necessity for code partitioning in actually realizing VLIW architectures that can only use port-limited RFs for the storage of VLIW code. We present a sample Limited Connectivity VLIW architecture that uses multiple, port-limited RFs for architectures that are realizable in CMOS technologies. We describe a code partitioning technique that can be used to map the ideal VLIW code into this Limited Connectivity VLIW architecture under different constraints such as a fixed number of partitions, a fixed number of data moves between partitions and a maximum number of ports per RF partition. The partitioning of code into multiple RFs (each with a limited number of ports) may result in a degradation of performance due to the extra moves required to transfer data from one partition to another. We therefore present the results of some experiments using standard benchmarks that allow a designer to perform a tradeoff analysis between limiting the number of ports in a RF, the size of each RF partition, and the effects on the performance of the specific VLIW realization.

The rest of the paper is organized as follows. Section 2 presents the limited-connectivity VLIW model. Section 3 discusses the partitioning scheme. Section 4 describes some experimental results using standard benchmarks. Section 5 concludes with a summary.

## 2 Limited Connectivity VLIW Model

The ideal VLIW model can be regarded as a horizontally microcoded engine able to simultaneously decode and execute different opcodes on different functional units. Each functional unit is completely equivalent to all the others so that there are no structural dependencies between the units, yielding a uniform functional unit model [20] [10] [25] [24]. Since the ideal VLIW assumes that any FU can access any register in a cycle through full connectivity between the FUs and the registers, we refer to this architecture as *Fully Connected VLIW*. This is a simple, yet appealing model extremely useful for code parallelization since one is only concerned with code scheduling. However, this ideal model is impractical to realize in silicon (except for a limited number of FUs) due to the large number of ports required by the centralized RF.

A practical solution to realizing the VLIWs in silicon is to limit the full connectivity between registers and FUs in an attempt to reduce the number of ports needed by each register file. This very approach was followed by the Multiflow TRACE design team who experienced this port-limitation problem, and designed a VLIW with multiple register files [9]. The basic concept is to trade some of the ideal performance achieved through full connectivity for a simpler data register structure and shorter clock cycle time. This goal is achieved by using a partitioned VLIW processor where each cluster has an equal number of FUs, and all FUs within a cluster are completely connected to a private register file. Communication between different clusters is achieved through a number of dedicated buses that connect all the register files. We refer to such a model as a *Limited Connectivity VLIW*.

When a register content is required in a cluster different from where it is stored, a specific move operation must be executed in a previous instruction to copy the data through inter-cluster data transfer buses. Hence the compiler has to to introduce some move operations to drive these buses and transfer the contents of a register from one cluster to the other.

It is clear that because of the limited connectivity and the register-to-register moves, the underlying model of execution no longer resembles a homogeneous shared memory architecture. In fact, register accesses can take different time lengths, resulting in an additional constraint that prevents us from achieving the optimal parallelization obtained using an ideal VLIW. However, there are clear advantages with the Limited-Connectivity VLIW, the most cogent being the feasibility of actually realizing the machine.

Although much work has been done on for code partitioning in the distributed computing community [22] [23] [4], very few authors to our knowledge have addressed the issue in the VLIW domain. The most closely related work is the Multiflow TRACE architecture [9].

## 2.1 A Sample Limited-Connectivity VLIW Architecture

Figure 1 shows a sample limited-connectivity VLIW consisting of 4 clusters of two functional units each. Within each cluster, the two functional units are completely interconnected to the local RF, yielding 4 output ports and 2 input ports for functional unit accesses. Furthermore, each RF has 4 additional input ports for inter-cluster data transfers.

The interconnection among RF clusters is accomplished through a number of inter-cluster data transfer buses that connect output ports of the RF (originally targeted for a FU) to the specific inter-bank communication input ports present in each register file (the ports on top of the RFs in

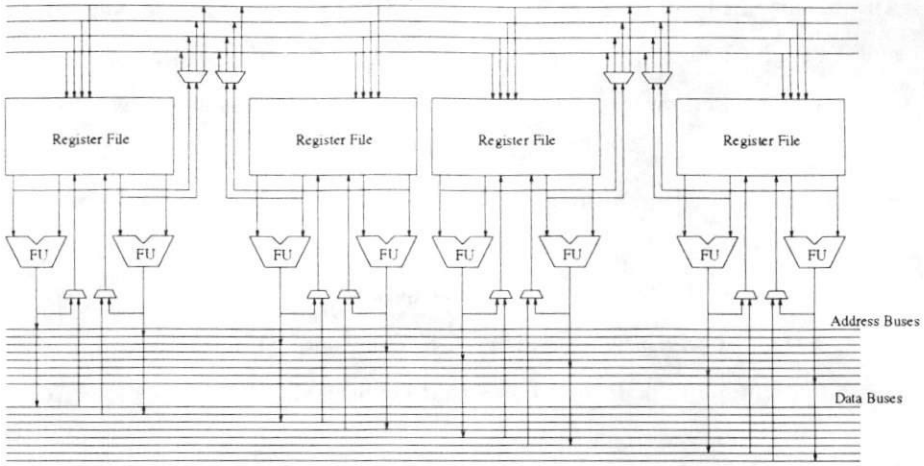


Figure 1: 4 cluster, 8 FU, 4 inter-bank bandwidth, Limited Connectivity VLIW

Figure 1). The availability of an output port of RF for a data transfer during the execution of an instruction is guaranteed since a move operation keeps the functional unit in the source cluster idle for that cycle. This can't be done with the destination cluster since the operation slot for the FU might contain normal operations. Hence an appropriate number of input ports is added to each register file to achieve the inter-cluster move operations. It is the compiler's task to automatically insert a move operation by specifying the source and destination register and bank. The source bank is automatically detected according to the instruction slot in which the move operation is inserted.

## 2.2 Architectural Characteristics

In order to analyze the performance and realizability of the Limited-Connectivity VLIW, we describe and define some architectural characteristics. We define *inter-bank communication bandwidth* of the VLIW to be the number of move instructions that can be issued per instruction, and define *relative bandwidth* to be the ratio between the communication bandwidth and the total number of operations that can be issued concurrently.

The number of reserved ports that have to be added to each register file bank is bounded by two constants: the number of inter-bank communications allowed per instruction and the total number of functional units in the other clusters. The first constant is intuitively clear, since we cannot have more than *bandwidth* communications taking place per instruction, and in the worst case all instructions are directed to move data to the same register file. The second constant

arises from the fact that there is no need to copy a register through an external bus back to the same bank. Hence there are at most $n$ functional units whose slot can be used to issue a data movement operation to a same cluster, where $n$ is the total number of functional units in other clusters (i.e., the total number of functional units less the number of functional units per cluster). The number of communications buses required in such architecture is equal to *bandwidth*. Note that when the second constant (total #FUs in other clusters) is less than the bandwidth, the buses are not connected to all input ports; this requires the adoption of a suitable scheme to ensure full connectivity. The scheme is easily obtained by connecting each bus to all clusters but one, that is, each RF is connected to $bandwidth - \frac{bandwidth}{clusters}$ different buses.

This architecture requires only a single control unit and a single conditional flags register file, since each conditional flag can be written by at most one functional unit and can be read only by the control unit. Hence there is no need to replicate the control and conditional registers.

The main advantage of the described model is in the limited number of ports required by each RF, and the limited distance that intuitively separates each functional unit from other non-local RFs. The direct consequence is an improved data-path delay time.

The gain achieved by reducing the number of ports can be attributed to the decrease in total number of ports, as well as the decrease in the number of output ports. As explained later, the number of output ports is one of the factors that most influences the cycle time of the registers, while the total number of ports accounts for the area complexity measures.

The number of *output* ports required in a limited connectivity architecture is simply the number of ports required per functional unit (2) times the number of functional units (#FU) over the number of clusters (#clusters) in which the architecture is partitioned.

The *total* number of ports instead is:

$$\#ports = 3 * \#FU + \#clusters * \min(bandwidth, (\#clusters - 1) * \frac{\#FU}{\#clusters})$$

where the first term accounts for the number of ports required for the functional units and the second term represents the number of ports added for inter-bank communication, and, as explained before, is computed as the number of clusters times the minimum between the bandwidth and the number of move operations that can be issued concurrently to move data in the same register file.

It is clear that the number of output ports is a decreasing linear function of the number of clusters and that the improvement can be measured as the ratio between the number of output ports in a limited connectivity architecture and the number in an unlimited connectivity architecture. This is obviously one over the number of clusters.

The analysis of the total number of ports per cluster is a little more complicated. The ratio of the number needed in a Limited Connectivity VLIW and the number in a Fully Connected one can be expressed as:

$$\frac{1}{\#clusters} + \frac{1}{3} * \min(\frac{bandwidth}{\#FU}, \frac{\#cluster - 1}{\#cluster})$$

The first term accounts for the percentage of the ports connected to the functional units required per RF and the second term is the number added to allow concurrent inter-bank communication. For example, the machine shown in Figure 1 shows 4 output ports per RF, i.e., 50% of the unlimited connectivity model, and a total of 10 ports, i.e., about the 42% of that in the fully connected model.

In [7], we provide a detailed analysis of the number of ports needed to build different kind of architectures by varying the number of functional units, the number of clusters and the communication bandwidth between clusters.

## 2.3 Register File Complexity Analysis

The major bottleneck in realizing an ideal VLIW comes from the inability to build register files with a very large number of ports. Accordingly, we describe some complexity measures to highlight the design consideration for multi-port RFs.

Although multi-port memory cell technology has been around for many years [2] [17] [1], only recently has there been a demand for memories with a very large number of ports, especially due to the interest in building VLIW architectures. For this reason there are no consolidated technologies for large, say more than 16-port RFs [12] [17].

The design of a multi-port memory can follow two approaches: a simple approach based on the replication of the memory cell access system (suitable for a limited number of ports), and a more complex one based on a hierarchical structure used to access the cell contents (more suitable for a large number of ports).

For a VLIW architecture, another important consideration is that a register cannot be written to concurrently. This is because the ideal VLIW guarantees full connectivity, resulting in the possibility of multiple writes to the same register location – and must be avoided by the compiler. Only concurrent reads are allowed from the same register and in the worst case all the output ports of a register file can access the same register (i.e. memory cells). This observation allows us to separate the area related complexity issues (that are dependent on the total number of input

and output ports) from the delay complexity issues (that are related to the number of output ports on a RF).

In [7] we show that for a large number of ports, the access time of a RF can be modeled as a logarithmic function of the number of output ports. This makes the number of ports per RF a fundamental issue for the design of a pipelined architecture where each constituent delay must be minimized in order to achieve the best overall throughput. We also show that the area complexity is a function that grows with the square of number of total ports; this is intuitively explained by the requirement of the number of tracks required in the the chip layout to route the input/output port signals across cells.

# 3 The Partitioning Methodology

Compiling code for a Limited Connectivity VLIW (LC VLIW) Architecture is not a trivial task. Due to the non-uniform register file access, data stored in a register is not always available to all the functional units; and code must operate within the boundaries of the clusters. If data from a *far* register, that is, in a different cluster, is needed, it must first be copied to a *local* register and then utilized.

The LC VLIW model thus requires a major modification of the ideal VLIW code. The ideal VLIW code stream must be partitioned into a number of substreams to be executed on the architecture clusters. Each of these substreams must be resource-constraint scheduled according to the architecture of the cluster itself. Also, the limited bandwidth needs to be accounted as a constraint in the scheduling process.

Our goal is to produce code that can be run on a Limited Connectivity VLIW, while minimizing the slowdown due to the introduction of data movement operations. In this paper, we constrain ourselves to deal with only straight line code loops. This is not a major limitation since in most scientific code applications, loops are the hot spots that primarily affect the performance and a large number of such scientific applications can be coded without conditional jumps.

## 3.1 Problem Definition

We begin by assuming that we already have the code available for an ideal VLIW model – this is currently generated by the PS compiler developed at U.C. Irvine [25]. This ideal VLIW code serves as a useful reference point for the analysis of our results.

The code for a VLIW comes in the form of an ordered sequence of Very Long Instructions. Each instruction is divided into a number of operations, with each operation defining the function to be performed by a distinct functional unit. The objective of our methodology is to partition the ideal code so that data flowing from one operation to another in a different cluster may happen only through main memory access, (i.e., through a couple of store/load operations) or through a specific data movement operation.

Our approach in partitioning the code is divided into three successive phases. First, a graph representation of the loop is generated. Second, a partitioning algorithm is applied to this graph in order to divide the code in substreams minimizing a given cost function. Third, inter-substream data movement operations are inserted and the code is recompacted.

A graph representation is first built from the code. This is a Directed Cyclic Graph $G = (N, E, \Lambda)$ where each operation *op* in the loop code is biunivocally mapped to a node $n \in N$ through a function $n = node(op)$; $E = E_1 \cup E_2$ is the set of directed edges. Each direct *data-dependency*, that is a dependency that flow toward the direction of execution, from an operation $op_1$ to an operation $op_2$ through an operand $x$ is mapped to an edge $e \in E_1, e = (n_1, n_2)$ and $n_1 = node(op_1)$, $n_2 = node(op_2)$. Each *loop-carried* dependency from an $op_1$ to an $op_2$ is mapped to an edge $e \in E_2, e = (n_1, n_2)$.

$\Lambda$ is an ordered list of sets such that, operations in the $i^{th}$ instruction are contained in the set $\lambda_i$. This list $\Lambda$ represents a scheduled partition of the ideal code graph since no operations can be shared between different instructions. In this way we operate on a schedule-partitioned DCG, each disjoint subset of which represents an instruction.

The $\Lambda$ partition is needed to preserve the scheduling order that otherwise would be lost if the code is mapped on a plain DCG. Operations contained in a set $\lambda_i$ are scheduled to be issued concurrently. Further, within the same loop iteration, all operations in all sets $\lambda_j$ for $j < i$ are scheduled to be executed before $\lambda_i$ and all operations in all set $\lambda_k$ for $k > i$ are scheduled to be executed after $\lambda_i$.

In each instruction a fixed number of operations $max\_op = \#FU$ can be issued. If there are less then max_op due to data dependencies, the instruction is filled with *nop* operations.

$$\| \lambda_i \| = max\_op, \ 1 \leq i \leq \#instructions$$

The edge set E of the graph G satisfies the following *causality* property $\Gamma$:

$$\forall e \in E_1, e = (n, m), \ n \in \lambda_i, m \in \lambda_j \Rightarrow i < j$$

$$\forall e \in E_2, e = (n, m),\ n \in \lambda_i, m \in \lambda_j \Rightarrow i \geq j$$

Partitioning of the code requires the creation of a new partitioned graph G' = $(N', E', \Lambda', S)$ where S = $\{S_1, .., S_s || s = \#clusters\}$ is a *balanced* partition of sets where each edge of G that goes from one node in one set of $S_i$ to a node in a different set $S_j$ must be replaced by inserting a node corresponding to the movement operation that has to be performed in order to copy data across clusters. [1]

$N' = N \cup N_m$ where $N_m = \{Inter\ Register\ Bank\ Communication\ Movements\}$, that is N' is the union of the set of nodes in graph G with the set of nodes corresponding to movement operations that have to be inserted.

$E' = (E - \{Edges(S_i, S_j)\}) \cup E_m$ where $E_m = \{e_i | the\ head\ or\ the\ tali\ of\ e_i\ is\ a\ move\ operation\}$: is the new set of edges. This represents the original set E minus the edges that go across partition's sets plus the edges that represent data flow to and from the data movement operations just inserted. Edges are to be added in order to maintain the $\Gamma$ property; i.e., edges in set $E_1$ must be replaced with a triplet (edge,node,edge) with both edges placed in $E_1$ (both must be direct dependencies), and edges in $E_2$ must be replaced with a triplet so that one edge is placed in set $E_1$ and the other in set $E_2$. In fact each loop carried dependency $e = (n_1, n_2)$ with $\lambda(n_1) > \lambda(n_2)$ can be substituted with two edges $e' = (n_1, n_{new}), e'' = (n_{new}, n_2)$ where $\lambda(n_1) > \lambda(n_{new})$ and $\lambda(n_{new}) < \lambda(n_2)$, or $\lambda(n_1) < \lambda(n_{new})$ and $\lambda(n_{new}) > \lambda(n_2)$. [2]

To satisfy this condition it might be necessary to create some new instructions so $\Lambda'$ is the new set of instructions, each of which still contains at most *max_op* operations.

The set $\Lambda'$ may be larger than the set $\Lambda$ because of the increased number of operations and the dependencies among the operations.

In this context we say that the partition S is balanced if $\| S_i \cap \lambda_j \| \leq \frac{max\_op}{\#clusters}$ $1 \leq i \leq \#clusters$, $1 \leq j \leq \#instructions$. This condition states that each $\lambda_i$ (i.e., the set of operations in the $i^{th}$ instruction) is partitioned into $\#clusters$ subinstructions, each of which holds at most $\frac{max\_op}{\#clusters}$ operations (i.e., the number of functional units per cluster).

The whole process has to be executed so as to minimize the increase in size of the partition $\Lambda'$.

[1] in this context *cluster* will be used to indicate a set of the partition S

[2] the notation $\lambda(n_1) > \lambda(n_2)$ is used to indicate that the index of the $\lambda$ set containing $n_1$ is greater then the index of the set containing $n_2$

## 3.2 Partitioning Process

The partitioning process is composed of the following successive phases.

**Phase 1: Graph Generation**

The DCG graph is built from the ideal VLIW code and is simplified according to the following rules:

- Immediate constants are removed from the DCG, since these are local to operations, and therefore do not represent a data movement.
- Loads and Stores of main-memory data are not represented in the graph because of the fully connected main memory access model. There are no edges in a load/store operation that represent the flow of data to and from the main memory.
- Communications from the outer loop are disregarded since these are negligible compared with inner loop dependencies. This is justified because before a loop is entered, data can be preorganized in such a way that each cluster has all the operands it needs from outside the loop (the data can be replicated if useful) in its own register file. Successive iterations will kill the data or continue to use it but in both cases access to data computed outside the loop is executed only once. Hence this communication is negligible as compared to the data movements that occur with the loop body for the number of loop iteration executions.

With these (justified) simplications, we are able to insulate the graph within the loop body from that outside of the loop. At this point, the DCG has been built and modified so that all and only the data flow movements that must be considered for code partitioning are present.

**Phase 2: Code Partitioning**

The graph partitioning algorithm is applied to the DCG in order to find a minimum partition cutset. The minimum number of communications is not directly related to the slowdown in the code since some edges might cause the remainder of the algorithm to lengthen the resulting code while others might not.

However, the basic idea is that by limiting the amount of communication, we should be able to limit the number of movement operations to be inserted and therefore minimize the increase in the cycle time.

The cutset is simplified according to a simple consideration: once data has been moved from a register file to another it must not be moved again unless the data in the destination register is killed. Therefore we can get rid of unnecessary edges by looking through the communications that take place from the same operation to the same cluster and performing a lifetime analysis to check for redundancies.

**Phase 3: Recompact Code in Partitions**

Once the graph has been partitioned the code must be modified to execute data movements across partition boundaries only through explicit operations. This is accomplished through a two step process.

- The first step is to insert, for each edge between different partitions' clusters, an empty instruction. The subinstruction corresponding to the cluster to which the source operation is allocated is filled with the movement operation needed (in the case where multiple edges share the same starting instruction, only a new empty instruction is created).
- The second step is to apply a modified version of the resource constrained scheduler (RCS) to compact the code and to ensure that no more than the allowed number of movements operations are in the same instruction. Obviously the RCS is allowed to move operations only within the code substream to ensure that the current partitioning scheme is maintained.

The resulting code is now consistent with the model of execution adopted: each instruction contain $max\_op$ operations homogeneously divided into $\#clusters$ subinstructions. Each necessary communication between operations in different clusters is accomplished through an explicit data movement operation, and no more than $bandwidth$ of these operations are allowed per instruction.

We illustrate this approach with a simple example. Figure 3 shows a simple DAG on the left; this graph corresponds to the following parallelized (ideal VLIW) code:

```
set a 100 ; set b 10
set c 1   ; nop
loop:
      add y a b ; add x c b
      mul a y b ; mul c x b
      gt cc0 a b ; nop
      br cc0 loop nop
endloop;
```

Using our approach, the DAG is simplified, partitioned in two substreams, and a necessary move is added corresponding to the interbank communication. Some preloop code is inserted to distribute the variable $b$ in both the clusters (variables with suffix i are from cluster i).

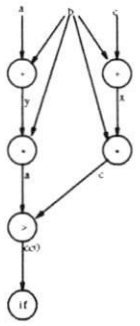

Figure 2: A simple DAG

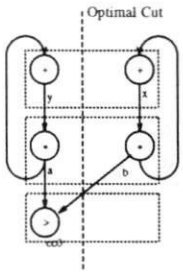


Figure 3: Simplified DCG

The resulting code corresponding to the right side of Figure 3 is:

```
set a1 100 — set c2 1
set b1 10  — nop
preloop:
          mov b1 b2 — nop
loop:
       add y1 a1 b1 — add x2 c2 b2
       mul a1 y1 b1 — mul c2 y2 b2
       nop          — mov c2 c1
       gt cc0 a1 c1 — nop
       br cc0 loop  — nop
endloop;
```

### 3.3 Algorithm Implementation

The DCG partitioning algorithm is based on the Lee, Park and Kim (LPK) [21], and the Fiduccia and Mattheyses [13] algorithms. It is modified to improve flexibility and to explore a wider search space. The detailed algorithm is explained in [5]. A brief outline is given here.

The algorithm is built around three nested loops. The two inner loops represent the deterministic part of the algorithm and essentially implements the LPK partitioning algorithm.

The innermost loop assigns, in each iteration, a node to the partition that results in the greatest

increase in the cost function. After all the nodes have been assigned, only the assignments that maximize the overall cost function benefit are accepted, while all the others are discarded. The central loop controls the repetition of these steps until no further improvement can be achieved.

Some changes were made to ensure that the structure of the code (i.e., number of operations per cluster belonging to the same instruction) is not altered. Therefore each complete execution of the two inner loops starts from a (possibly unbalanced) partition to output a new balanced partition whose cost function is minimized.

The third and outermost loop is responsible for widening the search space. The approach is to randomly select a new starting partition from the best known solution and feed it to the inner part of the code. This process is governed by a parameter that defines the average *distance*, that is the number of nodes allocated to different clusters, of the best solution from the new generated starting point. This parameter is reduced, according to a monotonic function, at each iteration of the outermost loop until a new improved solution is found, in which case the process is restarted, or a threshold is reached, in which case the algorithm terminates.

The basic idea is to increase the search space randomly picking up starting points at a given distance from the best solution. This process is repeated for decreasing distances with the goal of exiting local minima.

This technique merges a stochastic approach similar to simulated annealing [19] with a deterministic technique as the LPK algorithm with two major advantages: it does not suffer from the long run times typically required for simulated annealing runs and it widens the search space typically cover by a deterministic approach. The time complexity for the algorithm is superquadratic, that is $O(N^2\alpha(N))$ with $\alpha(N) = o(N)$, where N is the number of nodes in the graph.

# 4 Experiments

The code partitioning methodology was applied to produce code for different configurations of a Limited Connectivity VLIW architecture. A set of benchmarks consisting of straight line code loops was used: Livermore Loops #7, #8, #9, #10, #13 the Crale #10 loop, and a loop provided by Motorola Co., named motorola. These benchmarks constitute a good base for experimental tests since the number of instructions per iteration in the sequential 3-address code spans across a wide range: from 31 to 218; this provides a good testbed for experimentation since different speedup behaviors can be observed when different parallelization techniques are applied.

Our experimental results and analysis are drawn from a large design space obtained by varying

three key architectural parameters for the LC VLIW: the number of functional units, the number of clusters and the number of movement operations allowed per instruction. The ideal VLIW code was partitioned and compiled into each architectural design point to determine the resulting performance for a given set of parameters. The design space is clearly not homogeneous since certain architectural configurations are not meaningful; therefore we constrain the values of the parameters to only meaningful and consistent triplets.

## 4.1 Tradeoff Measures

Tables 1 show the number of cycles per loop in the code after it has been compiled and partitioned for different architectural models with 1, 2, 3, 4, 6 and 8 functional units. Each row present results for a VLIW model with different number of FUs, RFs and and Bandwidth. Within the row there is also an indication of the number of ports required for each register file from that specific configuration. The percentage figures on the right of each penalty measure indicates the percentage slowdown with respect the ideal VLIW model with the same number of FUs. Results were obtained without the renaming.

Some of the benchmarks, especially those with a limited number of instructions, do not scale linearly with the number of functional units, that is, do not achieve a linear speedup when increasing the number of FUs.

Some loops, especially small ones, top the achievable speedup (without renaming) for a small number of functional units. Because the partitioning algorithm tries to compact code to avoid inter-RF communication, when the clusters are larger than a certain threshold, the code can be executed in a limited number of clusters (sometimes even in just one cluster). This results in a limited amount of communication, yielding a small or even no slowdown.

## 4.2 Tradeoff Analysis

A preliminary look at the results shows very little sensitivity of the performance penalty to the relative bandwidth. Therefore the relative bandwidth can be treated as a second order parameter and can be used to fine tune the final architecture. The most important parameters appear to be the number of FUs and the number of clusters in the architecture.

Given this fairly large design space, we can use different design goals to guide the tradeoff analysis of the penalties versus the important architectural parameters. We propose three different approaches for analyzing the data obtained through our partitioning approach.

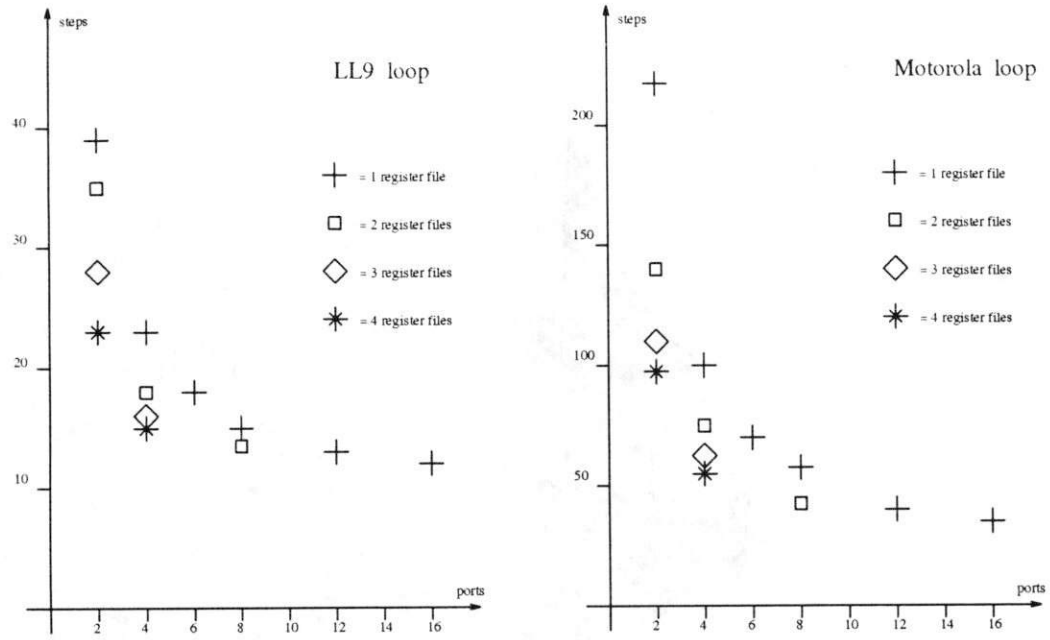


Figure 4: Instructions per iteration vs. number of output ports for LL9 and Motorola loops

The first approach is based on the assumption of a *limited number of ports* for each RF. Note that this is a practical constraint, particularly important for realizing RFs using current CMOS design techniques, and is one of the major reasons for going to a LC VLIW architecture.

The achievable gain obtained through the use of partitioned architecture can be visualized by plotting the penalty data versus the number of output ports for each implementation. Figure 4 shows the plots of the number of steps versus the number of ports using a different number of RFs for the Motorola and LL9 loops. Note that that a reasonable increase in performance can be obtained by constraining the number of ports and replicating the largest ideal VLIW model that can be built with the maximal number of ports. For instance, if we constrain ourselves to 2-output-ports per RF, we can achieve a 55% speedup using a 2-cluster architecture (where each cluster contains a single FU connected to a RF), and a 98% speedup with 3 clusters.

Although this straightforward approach is useful as a guideline for tradeoff analysis when we are primarily limited by the number of ports on a RF, we can use other approaches to better search the entire design space based upon other design goals.

We illustrate the other tradeoff analysis schemes by generating a fairly comprehensive design space that covers a different number of ports, different number of register files and different

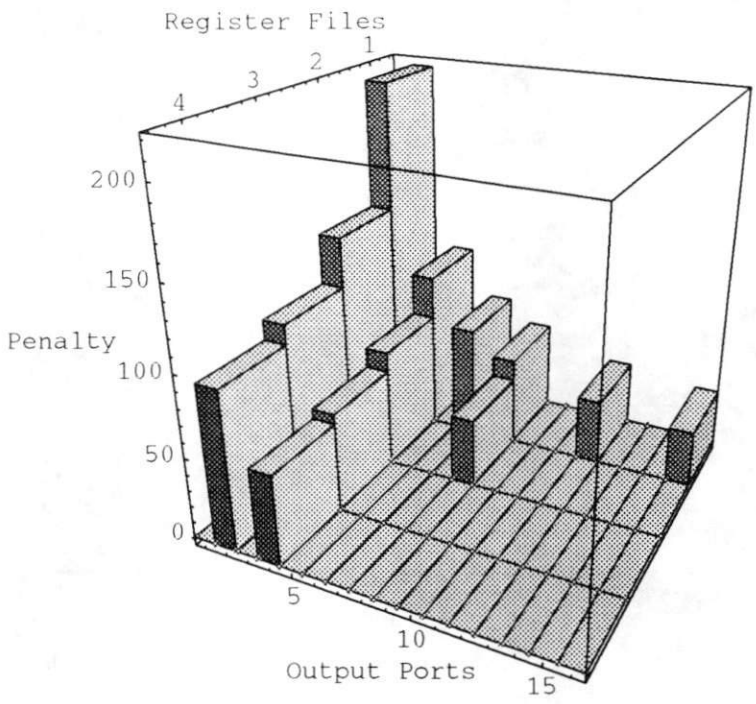


Figure 5: Number of instructions vs output ports vs clusters in Motorola loop

communication bandwidths. To get a better understanding of the tradeoff space, we recall the area/delay complexity figures previously outlined: the area of a RF is roughly proportional to the square number of ports, while the access time is roughly proportional to the logarithm of the number of output ports.

Tables 2, 3, 4 and 5 present the data for the Motorola and LL9 loops. The penalties are displayed on a two dimensional matrix: the horizontal dimension defines the number of clusters and the vertical dimension measures the number of ports per register file for that configuration. The remaining parameters (number of functional units and bandwidth) are printed on the side of each penalty number. Tables 2 and 3 refer to the global number of ports while Tables 4 and 5 refer to output ports only. Figure 5 presents the results for the Motorola loop in a three-dimensional form.

We now propose two other general tradeoff schemes that scan the entire design space in an attempt to perform tradeoffs. The *top-down* approach begins with a fully-connected, ideal VLIW

model and investigates the effects of successively partitioning this ideal model into a number of VLIW clusters to meet a constraint such as the desired number of clusters. The *bottom-up* approach, on the other hand, starts with a small, ideal VLIW, and attempts to replicate these ideal VLIW "slices" to build the target machine; the major goal is then to study the performance penalty relative to the size of the replicable VLIW "slice". For the rest of this section, we illustrate the tradeoff analysis using the *top-down* approach approach; a similar analysis can be performed for the *bottom-up* approach.

Since the *top-down* approach compares the advantages and disadvantages of splitting an ideal VLIW architecture, thanks to the complexity figures defined, we can use it to compare more feasible LC VLIW design models in terms of the area and access time required by the RFs.

Consider Table 4 which shows the variation of number of instructions per loop to output ports for the Motorola loop. If we start with a large, e.g., 8-FU ideal VLIW (i.e., with 1 RF), splitting the architecture into two results in an increased penalty from 35 to 43 steps, versus 56 steps for a 4-FU ideal architecture (i.e., with 1 RF). The two cluster 8-FU architecture and the ideal 4-FU architecture have the same number of output ports, so it is reasonable (due to our previous assumptions) that the register access time is about the same. We can therefore conclude that 50% (43/35) of the achievable speedup in the ideal model (56/35) can be achieved in the partitioned architecture with the same number of output ports.

We must also consider that for an 8-port RF, the memory cycle time can be squeezed to $\frac{log(8)}{log(16)} \approx 0.75$ [3] times the 16-port cycle time; although this percentage is not directly proportional to the execution cycle time, this reduction can be used to meet a specification. This weighted penalty figure for the Motorola loop is displayed in figure 6, where the results of Table 4 are weighted with the logarithm of the number of output ports.

Partitioning the 8-FU architecture into two allows us to reduce the area of each RF by a factor of $\frac{24^2}{16^2} \approx 2.25$ [4]. If we simply duplicate the register file, that is, assign each cluster a register file with as many registers as in the fully connected model, the total area is almost the same (about the 90% of the original area). This assumption is probably too restrictive since each cluster has half the FUs and thus probably requires a smaller number of registers. However, it can be considered as an upper limit for the total area that must be reserved for the registers.

The overall result is that using a 2-cluster architecture, we can design a machine with a 30% smaller register cycle time and whose motorola benchmark code is 22% longer; a smaller area is

[3] Refer to section 2.3
[4] Idem

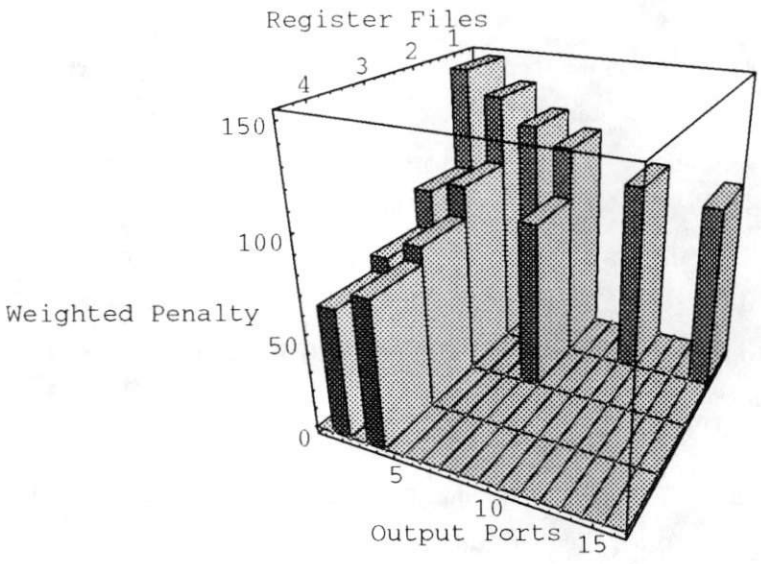


Figure 6: Weighted number of steps (Instr*log(Out Prt.)) vs. number of output ports for Motorola loop

needed for the memory and a much smaller number of ports per register file are required. Table 2 shows 14 total ports for a 8-FU VLIW with 2 clusters versus vs 24 total ports for an ideal 8-FU architecture. The same configurations require respectively 8 vs 16 output ports, as can be seen on Table 4. This seems to suggest that we can achieve about the same performance as a fully connected architecture, but with a smaller amount of area reserved for registers and with the number of ports significantly reduced.

These results can be improved when a 4-cluster 8-FU model is considered. Table 2 shows it takes 56 step to execute the Motorola benchmark but because it needs only 4 output ports per RF, the memory cycle time can be cut in half, maybe more, since the 4-output port RF can be built using a different that is faster as compared to a 16-output port RF. Following the thread of reasoning outlined above we can conclude that the area can improved by a factor of four, and although we have to pay a larger cost in terms of number of registers because now there are four register files, it seems reasonable for the substantial improvement achieved. This model can be compared with the 4-FU fully connected (ideal) VLIW (i.e 1 RF) that takes the same number of steps to execute, but has a 50% longer register cycle time due to the larger number of output ports (8 vs. 4, see Table 4). Hence the partitioned approach still appears to be better in terms of performance.

Using the *bottom-up* approach, we can perform a similar tradeoff analysis. If we duplicate a 2-FU ideal VLIW using the interconnection scheme proposed earlier, we obtain a 35% speedup for a 2-cluster, a 63% for a 3-cluster and a 78% for a 4-cluster architecture; all with the same number of ports on the RF and hence with the same register access time.

Table 4 shows that the 4-cluster 8-FU VLIW requires the same number of steps to execute the Motorola loop as the ideal 4-FU VLIW, but with half the number of output ports (that is 4 vs 8). This means a 50% faster register cycle time for the 4-cluster approach. In table 4 it can be seen that the LC VLIW with 8 moves allowed per instruction uses the same number of ports as a ideal 4-FU VLIW requires, but this measure can be improved if we allow only 2 moves per instruction; the total number of ports per RF then goes down to 8 (from 12) and the number of steps only grows by 7%.

A final tradeoff consideration can be made by looking at the Figure 6 that gives a three-dimensional view of the weighted penalty variation. The figure shows the delay contribution, calculated as the number of instructions per loop times the logarithm of the number of output ports in the configuration, of the register access to the data-path cycle time.

From this figure, it is clear that the penalty is no longer a monotonic function of the number of functional units. Note that this penalty now refers to an estimated RF access delay times the number of steps taken to execute a loop iteration, and hence does not directly relate to a real execution time. However, this figure suggests that if the RF access time is the major bottleneck for the system clock cycle, then the fastest VLIW architectures are built by replicating small clusters, rather than by increasing the number of FUs in the ideal VLIW.

# 5 Summary

This paper described the need to examine architectural tradeoffs for Limited Connectivity VLIWs, since the ideal VLIW requires a RF with an unrealizable number of ports. We claim that a partitioned architecture is a suitable way to design VLIWs since it allowed us to sensibly reduce the complexity of each cluster with a reasonable amount of execution overhead. The reduced complexity can be used to exploit better hardware performances and to design components that can be feasibly implemented.

We presented a sample Limited Connectivity VLIW architecture that uses multiple, port-limited RFs for architectures that are realizable in CMOS technologies. We described a code partitioning technique that can be used to map the ideal VLIW code into this Limited Connectivity VLIW architecture under different constraints such as a fixed number of partitions, a fixed number of data moves between partitions and a maximum number of ports per RF partition. The partitioning of code into multiple RFs (each with a limited number of ports) may result in a degradation of performance due to the extra moves required to transfer data from one partition to another. We therefore presented the results of some experiments using standard benchmarks that allow a designer to perform a tradeoff analysis between limiting the number of ports in a RF, the size of each RF partition, and the effects on the performance of the specific VLIW realization.

We presented these results, along with an analysis that provides some insights for architectural tradeoffs in the actual implementation of these Limited Connectivity VLIWs.

| 3 address code, 3 ports/register file | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| LL7 | | LL8 | | LL9 | | LL10 | | LL13 | | crale10 | | motorola | total |
| 31 | | 130 | | 39 | | 43 | | 73 | | 18 | | 218 | 422 |
| fully connected, 2 ops per instr, 6 ports/RF required | | | | | | | | | | | | | |
| LL7 | | LL8 | | LL9 | | LL10 | | LL13 | | crale10 | | motorola | total |
| 19 | | 69 | | 23 | | 22 | | 41 | | 9 | | 100 | 283 |
| 2 clusters, 2 ops per instr, 4 ports/RF required | | | | | | | | | | | | | |
| LL7 | | LL8 | | LL9 | | LL10 | | LL13 | | crale10 | | motorola | total |
| 24 | %21 | 90 | 30% | 35 | 47 % | 34 | 54 % | 50 | 21 % | 11 | 22 % | 140 | 40% 294 |
| fully connected, 3 ops per instr, 9 ports/RF required | | | | | | | | | | | | | |
| LL7 | | LL8 | | LL9 | | LL10 | | LL13 | | crale10 | | motorola | total |
| 18 | | 53 | | 18 | | 17 | | 34 | | 6 | | 70 | |
| 3 clusters, 3 ops per instr, 5 ports/RF required | | | | | | | | | | | | | |
| LL7 | | LL8 | | LL9 | | LL10 | | LL13 | | crale10 | | motorola | total |
| 23 | 27% | 86 | | 28 | 55% | 32 | 88% | 44 | 29% | 9 | 33% | 110 | 57% 332 |
| fully connected, 4 ops per instr, 4 moves per instr, 12 ports/RF required | | | | | | | | | | | | | |
| LL7 | | LL8 | | LL9 | | LL10 | | LL13 | | crale10 | | motorola | total |
| 18 | | 48 | | 15 | | 15 | | 31 | | 5 | | 56 | 140 |
| 2 clusters, 4 ops per instr, 4 moves per instr, 8 ports/RF required | | | | | | | | | | | | | |
| LL7 | | LL8 | | LL9 | | LL10 | | LL13 | | crale10 | | motorola | total |
| 18 | 0% | 55 | 14% | 18 | 20% | 19 | 26% | 33 | 6% | 6 | 20% | 74 | 32% 223 |
| 2 clusters, 4 ops per instr, 2 moves per instr, 8 ports/RF required | | | | | | | | | | | | | |
| LL7 | | LL8 | | LL9 | | LL10 | | LL13 | | crale10 | | motorola | total |
| 18 | 0% | 55 | 14% | 18 | 20% | 19 | 26% | 33 | 6% | 6 | 20% | 74 | 32% 223 |
| 4 clusters, 4 ops per instr, 4 moves per instr, 6 ports/RF required | | | | | | | | | | | | | |
| LL7 | | LL8 | | LL9 | | LL10 | | LL13 | | crale10 | | motorola | total |
| 23 | 35% | 79 | 64% | 23 | 53% | 29 | 93 % | 43 | 38% | 8 | 60% | 98 | 75% 303 |
| 4 clusters, 4 ops per instr, 2 moves per instr, 5 ports/RF required | | | | | | | | | | | | | |
| LL7 | | LL8 | | LL9 | | LL10 | | LL13 | | crale10 | | motorola | total |
| 23 | 35% | 80 | % | 23 | 53% | 30 | 100% | 45 | 45% | 9 | 80% | 98 | 75% 308 |
| fully connected, 6 ops per instr, 18 ports/RF required | | | | | | | | | | | | | |
| LL7 | | LL8 | | LL9 | | LL10 | | LL13 | | crale10 | | motorola | total |
| 18 | | 47 | | 13 | | 13 | | 31 | | 3 | | 41 | 166 |
| 3 clusters, 6 ops per instr, 3 moves per instr, 8 ports/RF required | | | | | | | | | | | | | |
| LL7 | | LL8 | | LL9 | | LL10 | | LL13 | | crale10 | | motorola | total |
| 18 | 0% | 54 | 13% | 16 | 23% | 17 | 30% | 34 | 10% | 4 | 33% | 62 | 51% 205 |
| 3 clusters, 6 ops per instr, 6 moves per instr, 10 ports/RF required | | | | | | | | | | | | | |
| LL7 | | LL8 | | LL9 | | LL10 | | LL13 | | crale10 | | motorola | total |
| 18 | 0% | 53 | 14% | 16 | 23% | 17 | 30% | 34 | 10% | 4 | 33% | 61 | 48% 203 |
| 1 cluster (limited connectivity), 8 ops per instr, 24 ports/RF | | | | | | | | | | | | | |
| LL7 | | LL8 | | LL9 | | LL10 | | LL13 | | crale10 | | motorola | total |
| 18 | | 46 | | 12 | | 12 | | 31 | | 3 | | 35 | 157 |
| 2 cluster (limited connectivity), 8 ops per instr, 8 moves per instr, 16 ports/RF | | | | | | | | | | | | | |
| LL7 | | LL8 | | LL9 | | LL10 | | LL13 | | crale10 | | motorola | total |
| 18 | 0% | 48 | 5% | 14 | 18% | 15 | 25% | 31 | 0% | 3 | 0% | 43 | 22% 171 |
| 2 cluster (limited connectivity), 8 ops per instr, 4 moves per instr, 16 ports/RF | | | | | | | | | | | | | |
| LL7 | | LL8 | | LL9 | | LL10 | | LL13 | | crale10 | | motorola | total |
| 18 | 0% | 48 | 5% | 14 | 18% | 15 | 25% | 31 | 0% | 3 | 0% | 43 | 22% 171 |
| 2 cluster (limited connectivity), 8 ops per instr, 2 moves per instr, 14 ports/RF | | | | | | | | | | | | | |
| LL7 | | LL8 | | LL9 | | LL10 | | LL13 | | crale10 | | motorola | total |
| 18 | 0% | 48 | 5% | 14 | 18% | 15 | 25% | 31 | 0% | 3 | 0% | 43 | 22% 171 |
| 4 cluster (limited connectivity), 8 ops per instr, 8 moves per instr, 12 ports/RF | | | | | | | | | | | | | |
| LL7 | | LL8 | | LL9 | | LL10 | | LL13 | | crale10 | | motorola | total |
| 18 | 0% | 53 | 15% | 15 | 25% | 16 | 33% | 34 | 10% | 4 | 33% | 56 | 60% 196 |
| 4 cluster (limited connectivity), 8 ops per instr, 4 moves per instr, 10 ports/RF | | | | | | | | | | | | | |
| LL7 | | LL8 | | LL9 | | LL10 | | LL13 | | crale10 | | motorola | total |
| 18 | 0% | 53 | 15% | 15 | 25% | 16 | 33% | 34 | 10% | 4 | 33% | 60 | 71% 200 |
| 4 cluster (limited connectivity), 8 ops per instr, 2 moves per instr, 8 ports/RF | | | | | | | | | | | | | |
| LL7 | | LL8 | | LL9 | | LL10 | | LL13 | | crale10 | | motorola | total |
| 18 | 0% | 55 | 19% | 15 | 25% | 17 | 41% | 34 | 10% | 4 | 33% | 62 | 77% 205 |

Table 1: Number of instructions per loop, different configurations

| tot. # ports | 1 Register File | 2 Register Files | 3 Register Files | 4 Register Files |
|---|---|---|---|---|
| 4 ports | - | 140 (2fu/2mv) | - | - |
| 5 ports | - | - | 110 (3fu/3mv) | - |
| 6 ports | 100 (2fu) | - | - | 98 (4fu/4mv) |
| 7 ports | - | - | - | - |
| 8 ports | - | 74 (4fu/(2/4)mv) | 62 (6fu/3mv) | 60 (8fu/2mv) |
| 9 ports | 70 (3fu) | - | - | - |
| 10 ports | - | - | 61 (6fu/6mv) | 60 (8fu/4mv) |
| 11 ports | - | - | - | - |
| 12 ports | 56 (4fu) | - | - | 56 (8fu/8mv) |
| 13 ports | - | - | - | - |
| 14 ports | - | 43 (8fu/(2mv) | - | - |
| 15 ports | - | - | - | - |
| 16 ports | - | 43 (8fu/(8/4)mv) | - | - |
| 17 ports | - | - | - | - |
| 18 ports | 41 (6fu) | - | - | - |
| 19 ports | - | - | - | - |
| 20 ports | - | - | - | - |
| 21 ports | - | - | - | - |
| 22 ports | - | - | - | - |
| 24 ports | 35 (8fu) | - | - | - |

Table 2: Number of instructions vs ports vs clusters Motorola loop

| tot. # ports | 1 Register File | 2 Register Files | 3 Register Files | 4 Register Files |
|---|---|---|---|---|
| 4 ports | - | 35 (2fu/2mv) | - | - |
| 5 ports | - | - | 28 (3fu/3mv) | - |
| 6 ports | 23 (2fu) | - | - | 23 (4fu/4mv) |
| 7 ports | - | - | - | - |
| 8 ports | - | 18 (4fu/(2/4)mv) | 16 (6fu/3mv) | 15 (8fu/2mv) |
| 9 ports | 18 (3fu) | - | - | - |
| 10 ports | - | - | 16 (6fu/6mv) | 15 (8fu/4mv) |
| 11 ports | - | - | - | - |
| 12 ports | 15 (4fu) | - | - | 15 (8fu/8mv) |
| 13 ports | - | - | - | - |
| 14 ports | - | 14 (8fu/(2mv) | - | - |
| 15 ports | - | - | - | - |
| 16 ports | - | 14 (8fu/(8/4)mv) | - | - |
| 17 ports | - | - | - | - |
| 18 ports | 13 (6fu) | - | - | - |
| 19 ports | - | - | - | - |
| 20 ports | - | - | - | - |
| 21 ports | - | - | - | - |
| 22 ports | - | - | - | - |
| 24 ports | 12 (8fu) | - | - | - |

Table 3: Number of instructions vs ports vs clusters LL9 loop

| output ports | 1 Register File | 2 Register Files | 3 Register Files | 4 Register Files |
|---|---|---|---|---|
| 1 ports | - | - | - | - |
| 2 ports | 218 (1fu) | 140 (2fu/2mv) | 110 (3fu/3mv) | 98 (4fu/4mv) |
| 3 ports | | | | |
| 4 ports | 100 (2fu) | 74 (4fu/(2/4))mv | 62 (6fu/3mv)<br>61 (6fu/6mv) | 60 (8fu/(2/4)mv)<br>56 (8fu/8mv) |
| 5 ports | | | | |
| 6 ports | 70 (3fu) | - | - | - |
| 7 ports | | | | |
| 8 ports | 56 (4fu) | 43 (8fu/(2/4/8)mv) | - | - |
| 9 ports | | | | |
| 10 ports | | | | |
| 11 ports | | | | |
| 12 ports | 41 (6fu) | - | - | - |
| 13 ports | | | | |
| 14 ports | | | | |
| 15 ports | | | | |
| 16 ports | 35 (8fu) | - | - | - |

Table 4: Number of instructions vs. output ports vs clusters in Motorola loop

| output ports | 1 Register File | 2 Register Files | 3 Register Files | 4 Register Files |
|---|---|---|---|---|
| 1 ports | - | - | - | - |
| 2 ports | 39 (1fu) | 35 (2fu/2mv) | 28 (3fu/3mv) | 23 (4fu/4mv) |
| 3 ports | - | - | - | - |
| 4 ports | 23 (2fu) | 18 (4fu/(2/4)mv) | 16 (6fu/(3/6)mv) | 60 (8fu/(2/4/8)mv) |
| 5 ports | - | - | - | - |
| 6 ports | 18 (3fu) | - | - | - |
| 7 ports | - | - | - | - |
| 8 ports | 15 (4fu) | 14 (8fu/(2/4/8)mv) | - | - |
| 9 ports | - | - | - | - |
| 10 ports | - | - | - | - |
| 11 ports | - | - | - | - |
| 12 ports | 13 (6fu) | - | - | - |
| 13 ports | - | - | - | - |
| 14 ports | - | - | - | - |
| 15 ports | - | - | - | - |
| 16 ports | 12 (8fu) | - | - | - |

Table 5: Number of instructions vs output ports vs clusters in LL9 loop

## References


[1] A. Abnous, C. Christensen, J. Gray, J. Lenell, A. Naylor, and N. Bagherzadeh. VLSI Design of the Tiny RISC Microprocessor. Technical report, University of California, Irvine, 1991.

[2] M.L. Anido, D.J. Allerton, and E.J. Zaluska. A three-port/ three-access register file for concurrent processing and I/O communication in a RISC like graphics engine. In *The 16th Annual International Symposium on COMPUTER ARCHITECTURE*, page 354, 1989.

[3] Mark Atkins. Performance and the i860 microprocessor. *IEEE Micro*, 11(5), October 1991.

[4] S. H. Bokhari. A Shortest Tree Algorithm for Optimal Assignments Across Space and Time in a Distributed Processor System. *IEEE Trans. on Software Engineering*, SE-7(6), November 1981.

[5] Andrea Capitanio, Nikil Dutt, and Alexander Nicolau. An Improved Partitioning Algorithm. Technical Report 92-57, UC Irvine, ICS Dept., 1992.

[6] Andrea Capitanio, Nikil Dutt, and Alexander Nicolau. Design Considerations for Limited Connectivity VLIW Architectures. Technical Report 92-59, UC Irvine, ICS Dept., 1992.

[7] Andrea Capitanio, Nikil Dutt, and Alexander Nicolau. Multi ported register file complexity analysis. Technical Report 92-58, UC Irvine, ICS Dept., 1992.

[8] P. P. Chang, S. A. Mahlke, W. Y. Chen, N. J. Warter, and W. W. Hwu. IMPACT: An Architectural Framework for Multiple Instruction Issue Processors. In *The 18th Annual International Symposium on Computer Architecture*, page 266, May 1991.

[9] Robert P. Colwell, Robert P. Nix, John J. O'Donnell, David B. Papwoth, and Paul K. Rodman. A VLIW Architecture for a Trace Sceduling Compiler. *IEEE Trans. on Computers*, 37(8):967, August 1988.

[10] K. Ebcioglu. Some Design Ideas for a VLIW Architecture for Sequential Natured Software. In *Parallel Processing, Proc. IFIP WG 10.3 Working Conference on Parallel Processing*, 1988.

[11] Jonh R. Ellis. *Bulldog: A compler for VLIW Architectures.* PhD thesis, Yale University, Dept. of Computer Science, 1985.

[12] W. Maly et alii. Memory chip for 24-port global register file. Technical report, Technical Report, Carnegie Mellon University, 1990.

[13] C.M. Fidduccia and R.M. Mattheyses. A Linear Time Heuristic for Improving Network Partitioning. In *19th Design Automation Conference*, page 175, 1982.

[14] J. Fisher, J. Ellis, J. Ruttenberg, and A. Nicolau. Parallel Processing: A Smart Compiler and a Dumb Machine. In *Proc. ACM SIGPLAN 84, Symp. on Compiler Constr.*, pages 34 - 37, June 1984.

[15] J. A. Fisher. Trace Scheduling: A technique for Global Microcode Compaction. *IEEE Trans. on Computers*, 30:478, July 1981.

[16] J. A. Fisher and B. R. Rau. Instruction Parallel Processing. *Science*, (253):1233 - 1241, September 1991.

[17] Dan Gajski. Dual port register file. Personal Communication, 1989.

[18] Intel. *i860 - 64 Bit Microprocessor, Assembler and Linker Reference Manual*, 1989.

[19] S. Kirkpartrick, C.D. Gelatt, and M.P. Vecchi. Optimization by Simulated Annealing. *Science*, vol. 220(no. 4598):671 - 680, 1983.

[20] M. Kuga, K. Murakami, and S. Tomita. DSNS: Yet Another Superscalar Processor Architecture. *Computer Architecture News (SIGARCH)*, 19(4):14, June 1991.

[21] C.H. Lee, C.I. Park, and M. Kim. Efficient Algorithm for graph partitioning problem using a problem transformation method. *Computer Aided Design*, 21(10):611, December 1989.

[22] S. Lee and J.K. Aggarwal. A Mapping Strategy for Parallel Processing. *IEEE Trans. on Computers*, C-36(4), April 1987.

[23] David M. Nicol. Optimal Partitioning of Random Programs Across Two Processors. *IEEE Trans. on Software Engineering*, SE-15(2), February 1989.

[24] Alexander Nicolau. Perculation Sceduling: a Parallel Compilation Technique. Technical report, TR 85-678, Cornell University, 1984.

[25] Roni Potasman. *Perculation Based Compiling for Evaluation of Parallelism and Hardware Design Trade-Offs.* PhD thesis, University of California, Irvine. Dept. of Information and Computer Science, 1992.

[26] C. Stephens, B. Cogswell, J. Heinlein, G. Palmer, and J. P. Shen. Instruction Level Profiling and evaluation of the IBM RS/6000. In *The 18th Annual International Symposium on Computer Architecture, ACM SIGARCH*, page 180, May 1991.