UC Irvine
ICS Technical Reports

Title
Architectural tradeoff analysis of partitioned VLIWs

Permalink
https://escholarship.org/uc/item/69t5n87p

Authors
Capitanio, Andrea
Dutt, Nikil
Nicolau, Alex

Publication Date
1994-03-29

Peer reviewed

eScholarship.org Powered by the California Digital Library
University of California

SLBAR
Z
699
C3
no. 94-14

Architectural Tradeoff Analysis of Partitioned VLIWs

TR #94-14

Andrea Capitanio, Nikil Dutt & Alex Nicolau

Notice: This Material
may be protected
by Copyright Law
(Title 17 U.S.C.)

Architectural Tradeoff Analysis of Partitioned VLIWs *

Andrea Capitanio

Dipartimento di Elettronica
ed Informatica
Universita' di Padova, Italia

Nikil Dutt & Alex Nicolau

Dept. of Computer Science
University of California, Irvine
CA 92717-3425, USA

March 29, 1994

Technical Report ICS-94-14

Abstract

A Very Long Instruction Word (VLIW) processor is an architectural model that has been extensively adopted as computing paradigm in the field of Instruction Level Parallelism (ILP); a common design is based on a set of functional units, each able to issue an operation per cycle, connected to a shared register file.

A VLIW has extreme requirements in terms of the number of gates, points of I/O, power dissipation and number of ports on its register file; characteristics that prevent implementation of the ideal architectural model on a single chip in current technologies, except for a limited number of functional units. A practical solution is to partition the architecture into multiple modules so as to meet technological constraints. The design of this brand of partitioned architectures requires an analysis of the overall effects of partitioning on both hardware and softwarer since the partitioning alters the performance in a non-intuitive manner.

In this paper we investigate the tradeoffs involved in the design of partitioned VLIWs with a methodology that matches data obtained through software simulation with hardware estimation models, creating a global performance model. The focus of the analysis is to study the effects of multiple register files on the overall performance and on the area requirements of the processor modules.

---

*This is an expanded version of a paper published in Proceedings of the 25th International Symposium on Microarchitecture, Portland, OR, 1992, pp. 292-300, Copyright (c) 1992, IEEE

1 Introduction

A VLIW processor is a parallel architecture in which several functional units (FU) can simultaneously execute multiple operations synchronously[9]. Operations are supplied to the processor in the form of long macroinstructions (i.e., Very Long Instruction World) that contain the opcodes and the operands for each FU.

A few commercial VLIW-like machines have already appeared on the market: early examples of this approach are the FPS series [5], the Intel 80860, and the Multiflow TRACE [6]. Other architectures, such as the IBM RS6000, the Motorola 88110, the DEC Alpha and the SUN Sparc 10, allow the issuance of few instructions per cycle but the selection mechanism is implemented in hardware. These architectures are currently referred to as Superscalars and are mostly based on the architectures of earlier machines such as the CDC 6600 [23] and the IBM System 360/91 [8].

Although all of these architectures pursue the goal of executing multiple operations concurrently, none of them has been built using the "ideal" VLIW architecture model frequently assumed by authors in the field of Instruction Level Parallelism (ILP).

The design variations from the ideal model are necessary since any practical implementation of an architecture with a large number of functional units (say more than 4) on a single chip is constrained by several technological constraints, including: the total number of gates required to implement the functional units, the number of I/Os pins for data and control signals, the power to be dissipated, and the inability to design and build a register file (RF) that provides a sufficiently large data-bandwidth to connect to all the functional units.

Although some of these restrictions become less of an impediment due to technological improvements, the limitation imposed by the single register file (RF) is unlikely to vanish. Multi-port Static RAM technology has been around for several years and yet there are no consolidated technologies for building RFs with a large (say more than 9) number of ports per RF [13] [16]. Also, to the best of our knowledge, there do not exist static memory cell designs with a large number of ports that are able to achieve access times comparable to single-port memory cells.

Even if a large multi-port RF could be built, it is likely to introduce some performance degradation – a degradation that may greatly offset the benefits of a complete interconnectivity and lessen the theoretical performances achievable with many functional units. This exact problem was faced by the designers of the Multiflow TRACE [6], 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, any practical implementation of a VLIW architecture with a reasonably large number of FU requires a partitioned scheme in which the register file is not connected to all the functional units.

A possible solution is to divide the architecture into a few interconnected modules each carrying a subset of the functional units and a register file. Several benefits accrue from this kind of architecture: it is scalable, since more clusters can be added to increase the size of the VLIW; it permits the use of standard, off-the-shelf, low-cost components and each module has simpler requirements for fabrication; finally, this approach enhances testability and maintainability, since a faulty partition can be replaced or replicated with ease.

The design of this type of architecture is a complicated effort since the partitioned structure has several intuitive, as well as non-intuitive, effects on the overall performance that are hard to quantify. A partitioned structure requires code, at best, as efficient as the code produced for an ideal model because of the additional constraints posed on the scheduling of operations; more often, it results in some performance degradation. However, it also a simpler data-path which might turn into a faster cycle time: the reduced number of ports in the RF may alleviate, if not eliminate, the bottleneck represented by the limited speed of this component.

In this paper we describe a methodological approach to the analysis of the effects on hardware and software performance of a VLIW with a partitioned RF. This approach analyzes area/performance tradeoffs to derive a broad view of the design space and its characteristics.

In Section 2 we present a sample Limited Connectivity VLIW architecture that uses multiple, port-limited RFs for architectures that are realizable in CMOS technologies. In Section 3 we propose some closed-form approximations to estimate the growth, both in terms of delay and area, for a register file with multiple I/O ports. In Section 4 we describe a code partitioning strategy that maps code generated for an ideal VLIW to the Limited Connectivity VLIW architecture under different constraints such as: fixed number of partitions, fixed number of data moves between partitions, and maximum number of ports per RF partition. In Section 5 we present the results of benchmarks for different LC_VLIW configurations. By merging these results with the hardware estimation techniques developed in Section 3, we create a parameterized performance space to allow an early investigation of the tradeoffs between different VLIW configurations.

2 Limited Connectivity VLIW Model

The ideal VLIW processor model assumes the capability of simultaneously executing different opcodes on different functional units. A functional unit (FU) is a component capable of executing one generic operation (i.e., arithmetic, logical, a memory access, etc.); each unit is completely equivalent to all the others such that no structural dependencies between the units can exist. Uniform access, by all FUs, to any register during each cycle is provided through full connectivity between FUs and the RF – we will refer to this architecture as a *Fully Connected VLIW.*

The resulting model provides a simple computing paradigm, used by several authors [9] [20] [21], which is extremely appealing for the development of parallelization techniques. However, it is impractical to realize in silicon for large numbers of FUs[2] since several technological constraints limit its realizability; among the others, one of the most severe constraints is the large number of ports required by the centralized RF.

A practical solution 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 approach was adopted by the Multiflow TRACE design team whose architecture had as many as 5 register files per board [6]. The basic concept is to trade some of the ideal performance, achieved through full complete connectivity between registers and FUs, for realizability and, possibly, a faster data-path.

2.1 A Limited-Connectivity VLIW Architecture

Several types of partitioned design schemes are possible. In this paper we consider a clustering approach in which the architecture is divided into modules, where each module (cluster) is composed of an equal number of FUs completely interconnected to a local register file (i.e., each FU has exclusive access to 2 read ports and 1 write port). Communication between different clusters is achieved through a number of buses that connect the local RFs. We refer to this architecture as *Limited Connectivity VLIW* (LC-VLIW).

The synchronization protocol adopted for interbank communications is based on a fixed cycle time (i.e. no stalls or time varying cycles) and the insertion of ad-hoc, inter-RF copy operations, is used to move data from one register file to another.

Other approaches are feasible: interlocking schemes based on a varying cycle time that stalls the execution until all the operands are available, or compiler based techniques that consider transmission delays. The approach adopted has the clear advantage of not requiring any complicated interlocking logic and also allows a high scheduling flexibility since inter-bank movement operations/delays don't have to be scheduled immediately after (or before) the use of the operand.

Figure 1 shows a simplified architecture composed of 4 clusters each with two functional units each. Each cluster is itself a small VLIW with two FUs completely connected (i.e. each FU can access any register in the module during any cycle), and all RFs are connected through 4 data-buses.

Each FU requires 2 read ports and 1 write port in a RF for full FU utilization. Thus in Figure 1 each RF requires 4 read ports and 2 write ports to provide the necessary connectivity between the registers and the FUs; furthermore each RF provides 4 additional write ports for inter-cluster data transfers for a total of 10 ports (4 read and 6 write). The read ports needed during each inter-bank copy operation to drive the buses are those used by the FU on which the operation is scheduled.

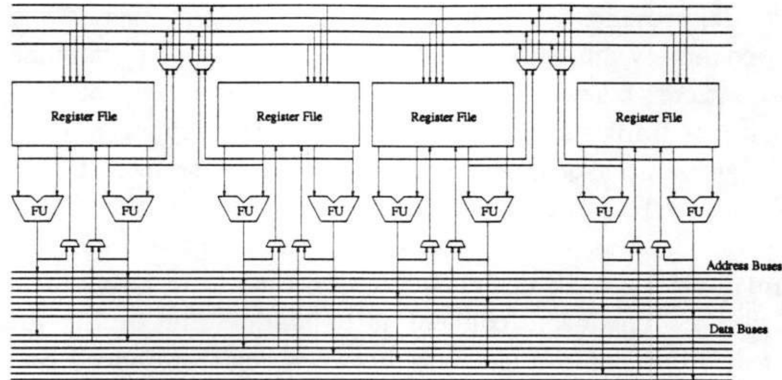

Figure 1: 4 cluster, 8 FU, 4 inter-bank bandwidth, Limited Connectivity VLIW

During an inter-cluster move operation the FU to which this is bound remains idle; one of the two read ports reserved to this unit is used to drive the bus and to transfer the data to a RF in another module. This operation in our model takes one cycle, just like any other operation performed by the machine. To support this communication scheme the copy operation has to be scheduled on a FU belonging to the module that contains the source register in order to ensure that an output port remains available.

3 Register File Complexity Analysis

The VLIW architectural model requires a RF whose number of ports grows with the number of FUs, thus it is important to analyze the complexity of a RF with a large number of ports. It is to be noted that this bottleneck is a relatively recent problem: although multi-port memory cell technology is well defined and stable for small number of ports [4, 2], only recently has a demand for memories with a very large number of ports (i.e., more than 8) arisen. This interest is related to the increasing number of multifunctional unit architectures in the microprocessor world. Despite the recent spread of interest, though, only few design have been published on SRAM memory cell design with large number of ports [12] [13] [16], and several issues need still to be investigated.

A single-port RF typically exhibits design tradeoff in terms of its aspect ratio, the number of bits stored and performance; however, multiporting adds a new dimension to the RF design space, since the number of ports affect both the area and the access timing to the registers.

In this section we describe a set of functions that model the effects of the number of ports on the area and access time of a RF for high speed, heavily pipelined processors with multiple functional units. We use these functions as estimators for our tradeoff analysis; we do not claim an exhaustive analysis of the issue, which would be beyond the scope of

this paper. The analysis has been performed assuming standard CMOS technology and a bit-slice data-path architecture, both currently popular for the design of processors [7]. Also each memory cell has to support multiple reads and exclusive writes, since concurrent writes to the same memory cell are not permitted.

3.1 Area Complexity Analysis

In a memory cell the principal factors determining its overall area are: 1) the active area, 2) the null area, and 3) the routing area. A SRAM is composed of a bi-stable circuit, usually implemented with two cross connected inverters, and some access circuitry. The number of gates required by the core of the cell is fixed (usually 4), yet its area changes with the number of ports since the two inverters must be sized to drive the increased loads represented by the added ports and the increased data (control) line length.

The total load (Ct) is the parallel of the load represented by the access circuitry and data line for each port (Cp), since in the worst case each cell must be able to drive all the read ports simultaneously: $Ct = \#outports * Cp$. Hence the size of the active area (i.e., the core memory inverters) is likely to grow linearly with the number of ports.

The growth of the area required by the access circuitry also can be assumed to be linear since the same circuit is used to govern the access per data line (i.e., per port).

A study of published designs [4, 13, 17, 11, 24, 12, 16] suggests that in a register even with a moderate number of ports, the area requirement is usually dominated by the routing area. Register files are commonly built using a bit-slice data-path architecture with cells belonging to the same register organized in rows and cells corresponding to the same bit organized in columns (bit slice) [4] [12] [15] [16]. Data-lines are routed along the bit slice, since they share all the cells in the same registers position, while the control lines are routed orthogonally to data lines across the cells belonging to the same register. Furthermore each memory cell requires a number of data-lines (select-lines) at least equal to the number of the ports (to ensure complete connectivity between each register and each port). These considerations along with the minimum distance requirements for line tracking allows us to model the routing area as proportional to the square of the number of ports.

The routing factor is predominant in multi-ported memory cells since its growing factor is more than linear, even for a limited number of ports, the cell layout is almost completely occupied by the data and control lines, thus making it impossible to accommodate other lines without increasing the dimensions of the cell. Assuming that the routing factor dominates for medium to large number of ports we derive the following cell area growth model:

$$Area(p) = CoreArea * (1 + \delta_x p) * (1 + \delta_y p) \tag{1}$$

where Area(p) is the estimated cell area for a p ported SRAM, CoreArea is the amount

of silicon required for the memory core (i.e., without access circuitries) , the two constants δ_x and δ_y are the percentage dimensional increase determined by each port.

The parameters δ_x and δ_y are used to estimate the percentage increase of dimensions x and y of memory cells per port and can be calculated by dividing the minimum allowed track-to-track distance by the layout dimensions of the core of a single memory cell. These two parameters have been evaluated to be in the range 5% to 20% for most of the designs considered.

A different general layout for the RF (e.g., a tiled architecture) doesn't affect the model since our assumptions still hold and the number of tracks to be routed per cell remains the same.

3.2 Delay Complexity Analysis

Register access time is another characteristic influenced by the number of ports and is another important factor considered in our analysis. The design of a cell and of the access circuitry is dependent on the number of ports since simultaneous access to all the RF ports require appropriate sizing of the drivers, resulting in an increased cell dimension and line length – therefore an increased propagation delay. RF performance is thus degraded by an increase in the number of ports.

Write operations have traditionally been the most sensitive and difficult to design, though in our model multiple write accesses to the same cell are forbidden, hence the number of write ports does not largely affect performance of the RF except for the effects of increased line delays. Multiple read operations to the same cell, instead, can occur (and in the worst case all read ports can access the same memory cell simultaneously); this strongly influences the design of the core of the cell, whose inverters must be sized to drive all the lines in all situations.

To proceed with our analysis we assume that the design adopts a data-line pre-charging technique, which is widely used in the design of SRAMs. Precharging brings the voltage of the data line to a logic one before a read operation. When the logic content of the cell is the same as the precharged line, there is only a minor flow of current through the access circuitry (i.e., a static read operation) thereby this operation can be quickly performed. When the content of the cell is different from the precharged value, a major flow of current is required in order to force the data-line level to the same logic value (i.e., a dynamic read operation), causing a long read operation. This technique permits a smaller cell size since only the pull-down transistor in the inverter which drives the line during a dynamic operation, must be sized for large loads (see Fig. 2).

The standard design techniques, adopted for single-ported static memory cells, can be easily extended to deal with a limited number of ports. Conflicting requirements for the read/write operations quickly bring these standard techniques to its limits.

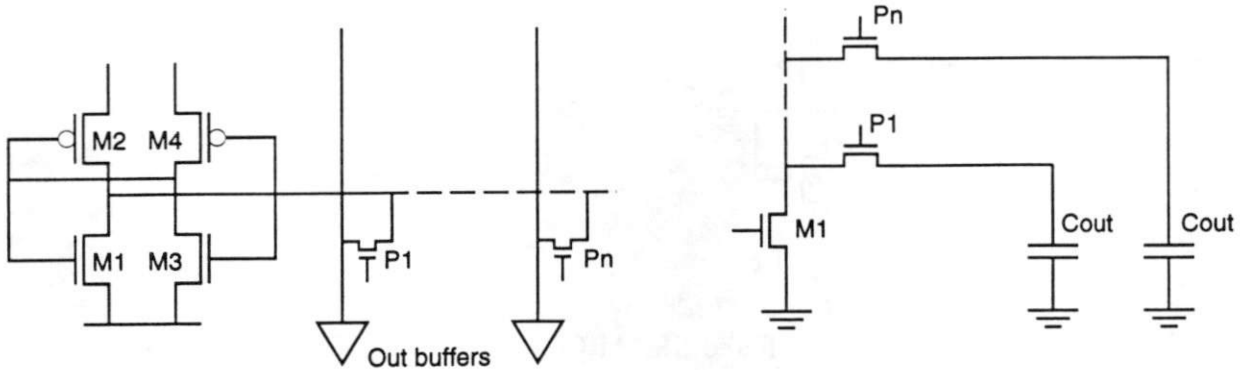

Figure 2: Basic SRAM Cell access scheme for a read operation

During a read access to a memory cell whose output is a logic 0, since the data-line is precharged to a logic 1, charge must flow through the access gate and the pull-down gate in the inverter discharging the data-line (gates P1 and M1 in Figure 2).

A parameter critical to the stability of the cell is the ratio between the β of the pull-down (p.d.) transistor and the pass gate: when the pass gate is turned on the voltage between the data line and the source of the pull-down transistor is partitioned according to impedance ratio between the two transistors [4]. If the drain of the pull down is brought to a potential that turn the other inverter on the content of the cell is lost. This ratio must be kept low in order to guarantee the stability of the cell during an access; in a multiported memory cell the critical ratio is between the parallel combination of p pass gates and the pull-down gate of the inverter [16] and keeping the ratio constant require either a decrease in the β of each pass gate or an increase in the β of the pull down gate. The first solution generates a larger and slower pass transistor, whereas the second generates a larger load and hence a slower write operation.

Even assuming this scheme can be expanded to a reasonable number of ports, it still suffers from an increase in the access timings; the fall read time can be assumed to remain constant since the series of the p.d. gates and the parallel of the pass gates remain constant [25], though in this case the p.d. transistor represent a proportionally larger load which reflects in a longer write operation.

Estimating the write delay with the delay of a cascade of buffers, increasing the β of the transistor proportionally increases the delay, hence this can be estimated to grow linearly with the increase in a gate's size which is in turn proportional to the number of read ports.

The approach of decreasing the β of the pass gates leads to longer read times since the propagation delay through the pass gate is proportionally longer to the length of the channel.

A different scheme for driving read ports makes use of a buffer stage, as proposed in [12]: the memory cells are accessed through a single-ended read circuitry driven through a

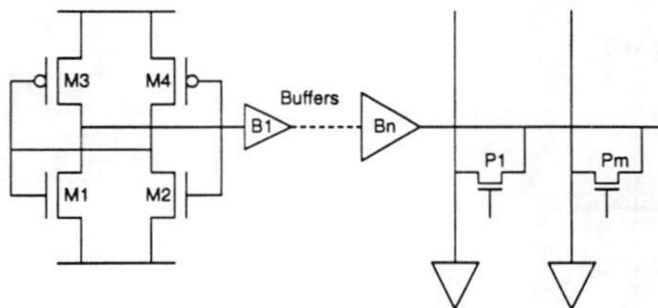

Figure 3: Buffered SRAM Cell access scheme

cascade of buffers as shown in Figure 3. A clear advantage of this scheme is that it scales well for driving an increasing number of ports by simply increasing the number of buffers and their size. The obvious drawback is the increase in the memory access time due to the latency of each inverter, the overall area used and the increase in design complexity.

For a cascade of buffers, the minimum total switching time is obtained when each buffer is e times larger than the previous one; then the delay is proportional to $e * \tau * ln(Y)$ where Y is the ratio between the parallel of the load capacitance for each port and the gate capacitance of the cell's inverter [14]. Although the logarithmic cost function looks appealing, the complexity involved in laying out this scheme has prevented the use of this technique. Instead, several projects have been based on a single buffer stage.

The single buffer stage seems a reasonable design tradeoff since it permits loads to be driven with a reasonable slowdown (i.e., the delay introduced by the buffer can be estimated to grow linearly with the output load, hence with the number of output ports concurrently accessing the cell), and a simple organization. This makes it a likely solution in the future for multiported SRAM design.

We therefore model the growth in the access time for multiported register cells as a function of only the number of output ports and estimate its increase as:

$$Delay(OP) = Delay(1)(1 + k\,OP), \tag{2}$$

where Delay(OP) is the time estimated for a memory access to a register file with OP ports and k is a parameter used to express the percentage cost increase per port.

4 The Partitioning Methodology

Compiling code for a LC-VLIW architecture requires a partitioning of code into a number of substreams, each to be executed on a different cluster; each substream has to be resource-constraint scheduled according to the architectural parameters (i.e., number of units per

module and number of inter-bank communications allowed per cycle).

Operations must be allocated such that all its operands are present in the local register file, ready for computation (i.e., in registers directly accessible from the FU that performs the operation). However, this task cannot always be accomplished statically since the operands produced in one module might be required on a different one; hence, data must be shuffled around at run time by inserting ad-hoc movement operations that provide each operation with the operands not directly available in the register file. On top of these requirements (i.e., data-availability and resource constraining) the code partitioning process has to modify the code such that it minimizes the performance degradation due to the insertion of data movement operations.

In this paper, we constrain ourselves to deal with straight line code loops; this permits simplification of the partitioning technique without affecting the meaningfulness of the results. Straight line code is typical in inner loops of most scientific applications - which frequently dominates execution time. In any case the use of guarded operations can be used to extend our technique to deal with conditionals inside loops.

4.1 Problem Definition and Approach

We begin by assuming that code for an ideal VLIW is already generated (currently generated by the PS compiler developed at U.C. Irvine [21]). This VLIW code serves as a useful reference point for the analysis of our results since it sets a baseline, in terms of software performance achievable.

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 code such that the value computed by an operation in one cluster can reach successive uses by operations in different clusters only through main memory (i.e., through a couple of store/load operations), or through specific data movement operations (also referred to as inter-bank movement operations).

The intent of the analysis is to estimate the degradation in performance introduced by the partitioning of the architecture, regardless of the scheduling technique adopted for producing VLIW code; hence the code partitioning phase was implemented as separate step in the process of compilation. An integrated approach would have not allowed such a clear distinction of the costs of partitioning from the performance of the operation scheduler.

The code partitioning approach is divided into three phases. First, a graph representation of the code's dataflow is generated; second, a partitioning algorithm is applied to the graph in order to produce substreams of code to be run on each module so that the code length increase is minimized[1]; third, intra-substream data movement operations are

[1]In straight line code the number of instructions in the loop body is directly proportional to the

inserted and the code is compacted.

Phase 1: Graph Generation

A Data Dependence Graph (DDG) graph, representing the dataflow among operations and including loop-carried dependencies, is built from the VLIW code: each operation in the code is mapped to a node of the graph and arcs connect operations between which exist a data dependence. The graph is then modified according to the following rules:

- Immediate constant assignment operations (e.g. R1 = 100) are removed from the DDG, in that they do not represent a data flow from one operation to another (i.e., the immediate is encoded in the operation and doesn't need to be transferred or stored).
- Dependencies between store and load operations through main memory are not taken into considerations since each LC_VLIW module is assumed to have complete access to main memory.
- Dependencies from outside the loop are also disregarded since they are negligible compared with internal dependencies: data can be distributed among all the register files such that each cluster has all the necessary operands (including those produced outside the loop) in its own register file (data can be duplicated if useful). Successive iterations will kill the content of the register (in which case the new value is computed within the loop and the dependency is internal) or will continue to use it (i.e., the data is stored in a local register and doesn't need to be moved again). These data communications are performed only once and are therefore negligible compared to intra-loop data movements that are executed at each iteration.

With these modifications we are able to insulate the DDG within the loop body from dependencies to and from outside of the loop. The graph contains all and only those data flow movements to be considered for code partitioning.

Phase 2: Code Partitioning

The graph partitioning algorithm is applied to the DDG. The technique uses an improved and adapted version of the Lee, Park and Kim algorithm [18] to partition the graph in a set of subgraphs each representing a substream of code. The primary goal of the algorithm is to produce a subgraph that meet each module resource constraints; as a secondary goal the algorithm seeks the minimization in the code length of the partitioned code. This is achieved by minimizing a function estimating the increase in code

---

execution time

length (i.e., the number of LC_VLIW instructions in the loop kernel) for each movement operation inserted.

In a second step the set of inter-bank movement operations are optimized to avoid unnecessary repetitions of the same move that could have been originated during the partitioning step (this might happen when an operand generated in a module is required several other times in another module).

Phase 3: Compact Code in Partitions

Once the graph has been partitioned and data movement operations have been introduced in the DDG representation, the graph is mapped back to code, through a two-step process.

First, we insert empty LC_VLIW instructions to allocate space for inter-bank movement operations that have been created during the previous phase. This is done by tracing through the code and creating a new empty instruction every time a move operation cannot be allocated in an empty slot within the existing code[2]. The move operation is later inserted in the empty instruction and assigned to the module containing the source operand.

Second, we apply a resource constrained scheduler (RCS) to compact the code and to ensure that the number of movement operations meet the constraints determined by the inter-bank communication bandwidth available.

The resulting code is now consistent with the model of execution adopted: each instruction contains a number of operations no more than to the maximum allowed, homogeneously divided into a number of subinstructions that are to be executed on different modules. Communications between operations in different clusters is accomplished through explicit data movement operations, and no more than B (where B is the *bandwidth*) operations can perform a data movement across different modules per iteration.

We illustrate the technique with an example shown in Table 1. A small kernel in C language is converted into MIPS-like assembly language and parallelized for a 4-FU VLIW (VLIW code). The VLIW code is first partitioned for a 2 module LC_VLIW by applying phases 1 and 2 to the code; this requires the insertion of an additional operation *fmove F4' F4"* from one module to another). The VLIW code is later compacted in phase 3. In the partitioned code we adopt the convention of labeling each register with a number of apostrophes related to the module in which it is contained (e.g. fmove F4' F4" move a virtual register F4' on module 1 to virtual register F4", on module 2).

Note that although the partitioning process requires the introduction of a movement operation, this doesn't translate into longer (and slower) scheme since the move can be accommodated in the existing code without altering its length.

[2]VLIW code usually contains some empty operation slots since scheduling cannot guarantee full utilization of all the FUs.

C Code

```
for (k=1; k<=1000; k++)
q += z[k]*x[k]
```

Assembly Code

```
1: (LABEL L5)
2:          (fload F8 -16000 I2 )
3:          (fload F10 -8000 I2 )
4:          (fmul F4 F8 F10)
5:          (fadd F6 F6 F4)
6:          (iadd I2 I2 8)
7:          (iadd I3 I3 8)
8:          (iconstant I4 8000)
9:          (ile cc0 I3 I4)
10:         (if cc0 (LABEL L5))
```

VLIW Code

1: fadd F6 F6 F4	fmul F4 F8 F10	if cc0	fload F8 I2 16000
2: fload F10 I2 8000	iadd I2 I2 8	ile cc0 I3 I4	iadd I3 I3 8
3: iconstant I3 8000	-	-	-

Code after partitioning

1: fadd F6' F6' F4'	if cc0	fmul F4" F8" F10"	fload F8" I2" 16000
2: -	-	fmove F4' F4"	-
3: ile cc0 I3' I4'	iadd I3' I3' 8	fload F10" I2" 8000	iadd I2" I2" 8
4: iconstant I3' 8000	-	-	-

Code after partitioning and compaction

1: fadd F6' F6' F4'	if cc0	fmul F4" F8" F10"	fload F8" I2" 16000
2: ile cc0 I3' I4'	iadd I3' I3' 8	fmove F4' F4"	fload F10" I2" 8000
3: iconstant I3' 8000	-	iadd I2" I2" 8	-

Table 1: Example of code partitioning

4.2 Algorithm Implementation

The partitioning algorithm is based on a fast deterministic search algorithm, coupled with a stochastic process that repeats the deterministic search from distinct, randomly generated initial solutions. This process is iterated for initial solutions that lie at decreasing distances[3] from the best found solution until no change is registered. The code for the algorithm is presented in Appendix B.

The deterministic algorithm uses the partitioning mechanism described by Lee, Park and Kim (LPK) [18], and modified to improve it's flexibility in order to be applied to the specific situation. The stochastic process is used to explore a wider search space, and is implemented by randomly generating initial solutions (from which the LPK algorithm starts) differing from the best found solution in K positions (i.e., at distance K). The process is repeated for diminishing values of K until either a better solution is encountered (in which case the process is restarted) or a threshold is crossed (in which case the process is terminated).

The idea behind this implementation is to search the space around the optimal solution at decreasing distances in order to escape from a possible local minima.

This algorithm has two major advantages: it does not suffer from the long run times typically required for techniques using exhaustive search or Simulated Annealing, and it widens the search space typically covered by a deterministic approach. It can also be tuned to satisfy execution time and search space coverage, by changing the function that alters K at each cycle, as well as the termination threshold that in turns alters the number of times the deterministic algorithm is applied.

The time complexity for the LPK algorithm is known to be quadratic while the stochastic search is performed a number of times dependent on the search space and on the parameters. In our configuration the number of iterations of the outer cycle was observed be usually less than linear and frequently quite small (between 4 to 10 iterations). Hence we can argue that the total complexity of the algorithm is superquadratic (i.e., $O(N^2\alpha(N))$ with $\alpha(N) = o(N)$; N is the number of nodes in the graph).

4.3 Related Work

Although several works have been published on code partitioning by the distributed computing community (e.g., [19]), very few authors, to the best of our knowledge, have addressed the issue in the VLIW domain. In our case the number of operations that must be considered for partitioning is at least an order of magnitude higher than assumed in

[3]The distance between two partitions is defined as the number of operations allocated onto different clusters

previous work. Furthermore, the functional units execute synchronously making unlikely the reuse of any of the techniques developed for loosely connected, asynchronous systems.

The most closely related work is the code partitioner for the Multiflow TRACE architecture [6] and Ellis' BUG (Bottom Up Greedy) assignment algorithm [10]. Both approaches are based on a greedy heuristic and designed to be applied on traces. BUG is a well known algorithm that achieves reasonably good results for straight line code but that does not take into considerations loopback dependencies. The improved version used in the TRACE compiler was adapted in order to better control the greediness and to take into consideration loopback dependencies.

However both the BUG and TRACE approaches are not suitable for the assumed LC-VLIW communication model, where movement operations must be introduced to copy data across the module boundaries. Applying a greedy heuristic to this case would require the recomputation of all the information for the DAG every time an additional operation is inserted. Also BUG integrates both the phases of resource constrained scheduling and code partitioning, whereas we partition the code after scheduling.

Hence our motivations and computing paradigm are quite different justifying the need for a different approach and the inappropriateness of comparing our technique with BUG since the comparisons would be misleading and meaningless.

5 Tradeoff Analysis

The overall performance of a partitioned VLIW architecture is the results of two key factors: code performance after the partitioning process and hardware performance achievable by a clustered architecture. Increasing the level of partitioning (i.e., the number of clusters in which the architecture is divided) drives these two factors along opposite directions: an increased number of modules frequently results in longer code while it might allow a faster execution cycle-time.

The process of partitioning requires the insertion of inter-bank data movement operations when operands are not present in the local register file. These operations usually have the effect of increasing the code length since they cannot be accommodated in any of the empty slots available in ideal VLIW code.

A partitioned architecture, though, has a reduced number of functional units directly connected to each register file, therefore has smaller requirements in terms of ports. Register-file access timings are directly affected by this parameter and a reduced number of ports might results in a faster access.

Our analysis is based on the assumption that the RF is, by far, most affected by the parallel architecture of a VLIW when compared to a standard RISC processor. The RF's performance and area are, in turn, directly affected by the number of ports on the RF – involving major structural modifications; the design of other components (FUs, control

P1	State Equation Fragment
P2	Adaptive Integration Algorithm
P3	Integration Prediction Algorithm
P4	Difference Prediction Algorithm
P5	Fluidodynamics Fragment 1
P6	Fluidodynamics Fragment 2
P7	SPECmark Kernel
P8	8th order elliptic filter

Table 2:

logic, I/O units) is relatively unaffected. This makes the RF a most likely bottleneck for the entire system.

This assumption is substantiated by recently published results: cycle times as short as 5 ns (and less) are becoming possible with today's technology [22], yet multiport register files usually exhibits access times that are several times larger [16, 13, 3].

We show how the models for multiport RFs and the results from simulations of code produced for a LC-VLIW can be used to examine some of the tradeoffs, in terms of performance and silicon area requirements, between different architectural configurations. We assume fast inter-chip communication and an architecture cycle time comparable to the inter-chip communication time (MCM architectures exhibit these characteristics). This allow us to adopt a simplified model in which every operation (inter-RF movement operation included) takes one cycle, as well as any other operation.

We generated experimental results for a large design space obtained by modifying three key architectural parameters of a LC-VLIW: the number of functional units per module, the number of modules into which the architecture is partitioned, and the maximum number of movement operations among clusters allowed per instruction (inter-bank communication bandwidth).

Each configuration defines an architecture with specific requirements in terms of number of ports per RF, this allows, by using formulas in Section 3, an estimation of the area and delay of the register file. Thus we provide a mechanism for analyzing the hardware performance and the effect on area requirements for different LC-VLIW architectures.

A set of standard benchmarks (see Table 2) composed of straight-line code loops representative of scientific code were compiled for different configurations of a LC-VLIW. The complete results for all benchmarks are given in Appendix A.

Several parameters are involved in this analysis; in this first phase we do not consider

Benckmarks	Benchmark P7				Benchmark P6				Benchmark P3			
# Partitions	2 modules		4 modules		2 modules		4 modules		2 modules		4 modules	
RB = 1	43	–	48	–	31	–	34	–	47	–	54	
RB = 1/2	43	0%	48	0%	31	0%	34	0%	47	0%	54	0%
RB = 1/4	43	0%	55	+14%	31	0%	35	+3%	48	+2%	56	+3%
RB = 1/8	51	+18%	91	+89%	31	0%	38	+11%	52	+10%	63	+16%

Table 3: Partitioned code length produced for a 8 FU LC_VLIW

the relative bandwidth[4] which we assume being 1. This decision was made on the ground of experimental results that show a limited influence of this parameter, within a reasonable range (1 - 1/3), on benchmarks performance.

Table 3 shows the code lengths obtained for various configurations of 8-FU LC-VLIW. Three benchmarks (P7, P6 and P3) are compiled for architectures with 2 or 4 modules and different values of RB; each line contains the set of results (i.e., number of instruction per iteration in the compiled loop) for a fixed value of relative bandwidth, with RB ranging from 1 (i.e. up to 8 move operations per cycle) to 0.125 (i.e. 1 move operation per cycle at most). Each value has on the side the percentage increase over the ideal code that that configuration requires.

Among all the benchmarks P7 is the one that has proved to be the most sensitive to RB variations, probably because of the complex communication pattern among the operations in it; P6 and P3, instead show an average behavior. In all the cases, it can be seen how the performance degradation due to a reduced inter-cluster communication bandwidth is negligible. In all the experiments a RB of 1/2 doesn't affect the best results at all, and a RB of 1/4 determines a drop in performance at worst equal to 14% and from 2% to 3% in all the other cases.

This is the first important result since not only do these observations allow us to proceed in the rest of the analysis without considering RB as a parameter of primary importance, but they also identify a RB between 0.25 and 0.5 as the amount of communication resources required so as not to affect the performance in a LC-VLIW. Should the need for a more precise analysis arise, this parameter can be taken into consideration after a first analysis step has identified a set of architectures as candidates most suited to the needs of the designer.

[4]We define the Relative inter bank communication Bandwidth (RB) as the ratio between the maximum number of movement operations allowed per cycle and the number of operations per instruction.

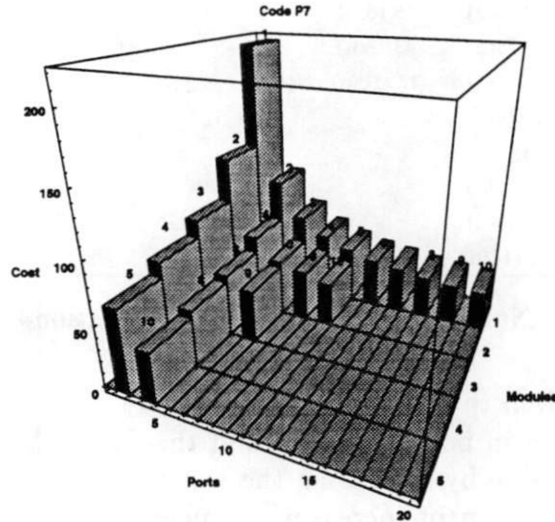

Figure 4: Instructions per iteration of P7 for several configurations

5.1 Performance Analysis

Table 4 presents a set of results obtained by compiling P7 for several different architectural configurations of a LC-VLIW, where each configuration is defined in terms of the number of functional units and the number of modules in which the architecture is partitioned.

The results are in the form of number of instructions contained in the loop body of the benchmark, hence higher values indicate lower performance. The results are also organized such that values of code length obtained by compiling code for configurations with the same number of read ports per RF are placed along the same row and values for configurations with the same number of modules appear in the columns. In our architectural model the number of read ports per RF is equal to twice the number of functional units connected to it (i.e., each FU connected to a RF requires 2 read ports), hence each row contains results for architectures with the same cluster size (i.e., number of FU per cluster).

The same set of results of Table 4 are also plotted in form of 3D-bars in Figure 4. The number of ports is plotted on the X-axis, the number of modules per VLIW module on the Y-axis and the cost (#instructions) on the Z-axis. Taller bars means worse values (larger number of instructions per iteration). On top of each bar is printed the number of FUs in

Out Ports	1 Module	2 Modules	3 Modules	4 Modules	5 Modules
2 ports	218 (1fu)	132 (2fu)	96 (3fu)	82 (4fu)	70 (5fu)
4 ports	100 (2fu)	66 (4fu)	53 (6fu)	48 (8fu)	42 (10fu)
6 ports	70 (3fu)	48 (6fu)	42 (9fu)	-	-
8 ports	56 (4fu)	43 (8fu)	-	-	-
10 ports	47 (5fu)	37 (10fu)	-	-	-
12 ports	41 (6fu)	-	-	-	-
14 ports	38 (7fu)	-	-	-	-
16 ports	35 (8fu)	-	-	-	-
18 ports	33 (9fu)	-	-	-	-
20 ports	32 (10fu)	-	-	-	-

Table 4: Number of instructions per iteration of P7

the architectural configuration the code was compiled for.

The design space defined in both the table and the figure shows the software performance improvement achievable by increasing the number of functional units per module (i.e., moving along the direction of increasing number of ports) and by increasing the number of modules in the architecture.

Performance increases monotonically in both directions, as is intuitive, since in both cases we increase the number of functional units and the scheduling technique is able not to make use of additional FUs in another module if the cost of communication affects performance.

Architectures with the same number of FUs but different number of modules frequently provide different results since the number of instructions is altered by the partitioning phase. We define the effect of an increased number of modules as partitioning *slowdown* and measure it as the percentage code length increase.

The data in Table 4 presents a level of slowdown ranging from 17% for a (6-FU and 2-cluster) architecture, up to 48% for a (5-FU and 5-cluster) architecture. A superficial analysis of this data could lead to the conclusion that a partitioned architecture performs worse than the ideal architecture, since any LC-VLIW requires code which can be, at best, as efficient as the code produced for an ideal architecture. However the software performance space does not take into consideration changes in propagation delay through the critical data-path, hence the processor execution cycle time.

It is to be remembered, though, that in the adopted architectural model, communications are performed in parallel to other operations (in the form of inter-bank data movement operations) and do not represent an additional delay to be accounted for in the execution cycle time. This observation, together with the assumption of an inter-module communication time smaller than the execution time, allows us to compare configurations with the same number of ports as architectures with the same cycle time.

Consider the row relative to 6 read ports (out-ports) in Table 4. We considered 3

architectural configurations requiring that number of ports per RF[5]: a (3-FU 1-module) , a (6-FU 2-module) and a (9-FU 3 module).

The last configuration is the one whose code contains the smallest number of instructions (42 vs 70 for the ideal 3 FU architecture). This result is obviously worse that the ideal 9 FU VLIW (33 steps), which doesn't have to deal with transfer of register values from RF to RF, but it provides a reasonable way to improve performance under the 6-port per RF constraint. 6 read ports allow the realization of an 3-FU ideal VLIW and the only way to increase performance is through a LC-VLIW created by coupling 3-FU modules together.

A more comprehensive analysis is required to account for different data-path timings achievable by each configuration, leading to different estimates of the execution cycle-time. The approach adopted was to weight the number of instructions per iteration with an estimation of the register access. Our focus is to analyze the effect on performance behavior induced by a partitioned RF since we assume the cycle time to be dominated by the register access time. Hence we approximate the overall execution time of a benchmark with the product of the code length times the register access time.

In case of different architectural models this approximation can be changed to better suit the needs of each model. A possible alternative is to count the influence of the register access time only partially as in:

$$Texec = \#instr * (1 + k)Tacc$$

. ,

where $Texec$ is the execution time for an iteration of a benchmark and $Tacc$ is the estimated register access time. Note that $k = 0$ is the case adopted in our analysis.

Results obtained by weighting values in Table 4 with the estimated RF access time obtained by using formula (2), are presented in Figure 5. Each vertical bar is proportional to the estimated execution time.

The introduction of the estimated cycle time into the model greatly affects the performance space. In this space (See Fig. 5) the architectures with the best performance are those that have a large number of FUs distributed in many modules (so as to keep the number of read-ports low) whereas the ideal architectures perform poorly; this behavior differs from the previous results suggesting that the best architectures are ideal VLIWs with a large number of FUs. The most overall efficient architecture for running P7 among those analyzed is a LC-VLIW with 10 FUs distributed among 5 modules (2 FU per module).

In this case the performance of the ideal VLIW peaks at about 5 functional units and further increments in the number of FUs decrease performance instead of improving it. This can be explained since the estimated RF access time grows faster than the software

[5]The analysis was constrained to architectures with no more that 10 FU since in most of the cases that is the maximum amount of parallelism that could be found in the benchmarks

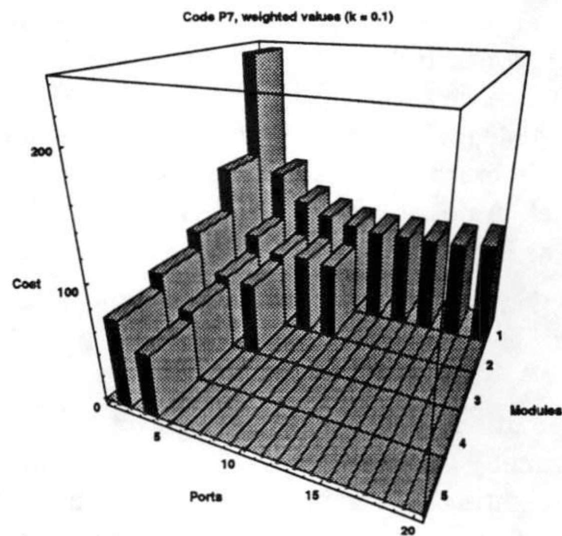

Figure 5: Estimated overall performance (hardware + software performance)

speedup achievable by scheduling the code for a number of functional units larger than 4 or 5, making the overall performance level off at first, and then decrease.

The observed behavior can be generalized to most of the programs whose amount of ILP is limited (i.e., all the programs that are not perfectly parallelizable): the code produced for a VLIW with an increasingly large number of FUs makes the performance benefits level off as the number of available FUs get closer to the inherent ILP limit of the program; this up to a point where it might not be sufficient to counterbalance the cost increase, in terms of hardware delays, that the newly added resources determine.

Hence we can state that for every program not perfectly parallelizable exists a limit after which an increase in the number of FUs is not profitable since it decreases the overall performance. This "wall" can be broken by making use of a partitioned architecture.

5.2 Area Analysis

RFs with large number of ports also exhibit a noticeable increase in area. As mentioned in Section 3, the area required by a multiported SRAM cell is roughly proportional to the square number of ports. This value grows rapidly to become unacceptable, particularly in a VLIW architecture whose area availability is already tight.

We use an estimation model for approximating the area of a multiported memory cell, with p ports, based on function (1). Assuming $\delta_1 = \delta_2 = \delta$ (i.e., both cell's dimensions are increased of the same amount per each port) the area can be modeled as:

$$Area(p) = CoreArea * (1 + \delta p)^2 \qquad (3)$$

Function 3 is plotted[6] in Figure 6 for two possible values of δ (0.1 and 0.2), based on data from published designs [2] [4] [16], and a number of ports in the range (1 - 30). The RF area growth is extremely fast: if $\delta = 0.1$ in formula 3, then a fivefold increase is obtained for 12 ports, (i.e., the number of RF ports required in a 4 FU VLIW) and a tenfold increase for 22 ports.

This area requirement is certainly not acceptable for a large VLIW where the need for registers[7] is extremely high. It is, therefore, imperative to explore the possibilities offered, by a partitioned approach in order to reduce the amount of silicon required for each cell.

The number of ports required by a VLIW can be reduced with a partitioned design scheme as outlined in Section 2: the total number of ports is the number of ports required to connect each FU to the RF (i.e., 3 * FU since each FU uses a write port and two

[6]The area is measured in units (1 unit being the area required by the core cell).

[7]A high level of fine grain parallelism can be exposed only through aggressive renaming techniques that frequently turn into a large number of register requirement [21].

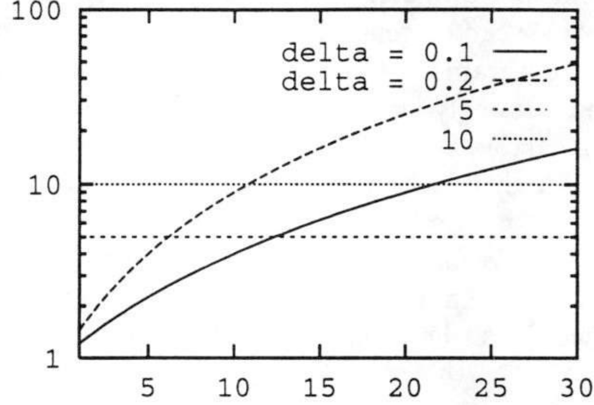

Figure 6: Estimated SRAM area increase vs number of ports

read ports) plus the additional ports introduced in the design by the RF interconnection scheme.

The total number ports per register file P can be calculated as:

$$P = 3\,(\frac{FU}{MO}) + (MO - 1)\,(\frac{FU}{MO}) \tag{4}$$

where FU is the total number of functional units in the LC_VLIW and MO is the number of modules, or clusters, in which it is partitioned. The first term accounts for ports used to connect the set of FUs in a module to the register file, and the second term accounts for additional write ports connected to the inter-bank communication buses[8].

The plots of the memory cell's area requirement for a specific configuration in terms of FUs and modules is presented in Figure 7. Each curve estimates the area growth (expressed as area units) of a register file in a LC-VLIWs for a fixed level of partitioning and a given number of FUs. The equation is:

$$A_{RF}(p) = A_{RF}(0) * (1 + K * p)^2 \tag{5}$$

where $A_{RF}(p)$ is the area of a register file with p ports ($A_{RF}(0)$ is the core cell area, which is used as a unit of measure). Only few points along each curve in Fig. 7 correspond

[8]The number of additional ports per RF is calculated based on the maximum number of operations that can be simultaneously issued in a LC-VLIW to move data to the same module (i.e., the number of the FUs in all but one module). Every module has $\frac{FU}{MO}$ functional units and, under the assumption of BR = 1, up to (MO - 1) modules can move data to a same module.

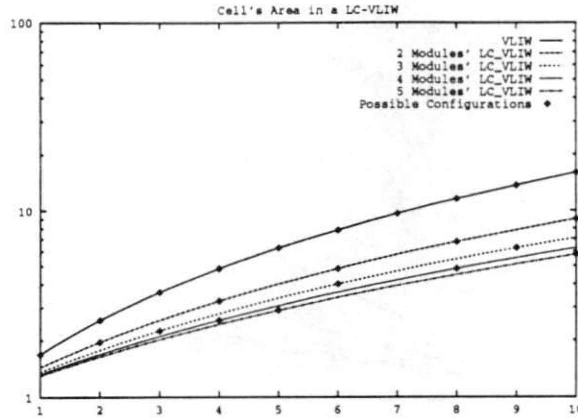

Figure 7: Area requirement for a multiported register file ($A_{RF}(0) = 1$)

to a physically realizable architecture and these are marked with black dots.

Figure 7 suggest that the area requirements for the registers of a LC-VLIW can be kept reasonably low by selecting the appropriate level of partitioning.

A further reduction in the RF area requirement can be achieved by relaxing the assumption of a full relative bandwidth (RB=1): limiting the bandwidth to 1/2, or 1/4, affects the performance only slightly and permits a reduction in the number of additional write ports which, in turn, results in a smaller cell area.

The cost in terms of total area requirements per module can be evaluated by adding the estimations for the RF area with the FU area. Splitting an architecture in 2 or more modules, though, opens up the problem of how many registers every bank should contain; we adopt the conservative approach of each RF in the partitioned architecture containing an identical number of registers as the RF in the ideal architecture. This assumption ensure the ability of partitioning code without a chance of running out of registers, since in the process we do not introduce new registers.

The estimation of the area required by a module (considering only FUs and RFs) can be obtained as:

$$A_{MO} = \frac{FU}{MO} A_{FU} + A_{RF} \tag{6}$$

where A_{MO} is the estimated area per module required for the set of FUs and the local RF, A_{FU} is the area occupied by a functional unit and A_{RF} is the estimated area of the register file, evaluated by using Equation (5). Assuming $A_{FU} = A_{RF}(0\,port) = 1\,unit$, for the sake of simplicity, we can plot the area per module as in Figure 8. Each curve, in Figure 8, plots an estimation of the the silicon required by a single module in an architecture with a fixed number of clusters for a growing number of FUs. Again since the curves are derived

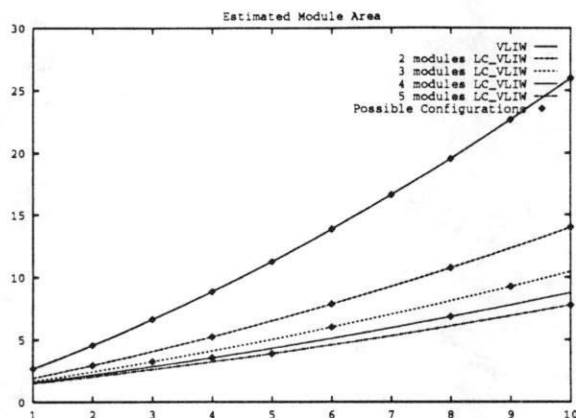

Figure 8: Area requirement for a module ($A_{RF}(0) = A_{UF} = 1$)

from equation 6 not all the points are meaningful and the realizable configurations are identified by diamonds.

Figure 8 can be used to determine a correct partitioning strategy to satisfy the area constraints determined by available technology.

The non linear behavior of each curve is due to the quadratic growth in the size of the RF. A higher ratio $\frac{A_{FU}}{A_{RF(0)}}$ tends to straighten the nonlinearity since the area required by the FUs is split evenly among each cluster and a FU's size is not affected by a partitioned design scheme – we assume in our analysis a ratio of 1 (i.e., $A_{FU} + A_{RF(0)}$).

Architectures partitioned into a large number of clusters allow an important reduction in terms of silicon estate per module. Consider the 10-FU 5-RF configuration, which allows the best overall performance among the considered architectures. Each module of that LC-VLIW requires an amount of silicon comparable to a 3-FU ideal VLIW (see Fig. 8). Partitioned architectures therefore not only perform better than ideal models but also allow a feasible implementation based on area concerns.

6 Area/delay tradeoff

The separate analysis performed for time and area can be merged into a comprehensive model that can be used to evaluate the tradeoffs involved in a partitioned architecture.

The design of a LC-VLIW requires the evaluation of several parameters at once; our approach allows the simultaneous consideration of three important parameters: silicon area occupation for the FUs and RF per module, number of ports per register file and overall performance. A fourth parameter, the inter bank communication bandwidth, can also be

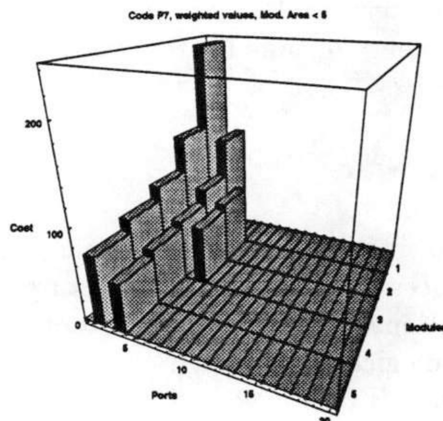

Figure 9: Cost for configurations limited by area constraints

considered, but since its relative influence on both performance and number of ports is limited we do not consider it as a first order parameter in our analysis.

The primary criteria of evaluation used is feasibility. In our analysis, the area and ports per RF are the parameters that might prevent physical realizability. Large VLIW configurations both occupy excessive areas and require RFs with large number of ports that can degrade performance. We can limit both parameters to values that are proven to be achievable with available technology and apply them to equations (4) and (6) in order to define a set of feasible configurations. By mapping these two constraints on the performance space of Fig. 5, it is possible to delete all the bars relative to architectures that do not match the requirements and hence perform a search only on the restricted (feasible) design space.

This process is illustrated in Figure 9 where all the configurations requiring an estimated area per module of more than 8 units and register files with more than 6 read ports have been deleted. It is interesting to note that the best solution of the unrestricted space (i.e, before area/ports constraining) still remains in the pruned design space; hence the 10 FU architecture composed by 5 modules is the most effective LC-VLIW configuration in terms of performance but also require a small die size and a limited number of ports.

These kinds of architectures are well suited for a realization using Multi Chip Module (MCM) technology: each die can contain one (or more) modules and buses can be routed on the substrate so as to achieve fast inter-chip communications. The fast communication allowed by MCM permits to design fast and scalable architectures where more modules

can be added to fit the needs of the user.

The scalability is limited from the propagation time of a signal from module to module; yet the decoupling between the intermodule communication path and the register-FU that the LC-VLIWs provide allows plenty of time for this task.

7 Summary

There is great potential in exploiting spatial parallelism (i.e., replication of FUs) to scalably increase the performance of processors. A VLIW architecture model is suitable for exploiting ILP through multiple functional units, though the ideal model is seriously limited in its realization by several technological constraints.

For this reason only few real VLIWs have been designed and only two, the Multiflow Trace and the Intel i860/i960, have reached the market. The first adopted a highly partitioned design and a slow cycle time, the second aimed at a limited parallelism with two functional units on a single chip.

Recently some technological limitations have been pushed a step forward by technologies like MCM and TAB wiring; also CMOS has further reduced the feature size allowing to use new schemes of partitioning. The performance of these architectures are deeply affected from the interaction of software and hardware phenomena, hence the need has arisen for tools to explore the resultant design.

In this paper we presented a design space exploration model for the limited-connectivity VLIW architecture, and we investigated some of the tradeoffs, in terms of area and delay, for a partitioned VLIW model focusing on the effects of a partitioned RF. Data obtained through the compilation of code for several architectural configurations and some estimation figures for the hardware performance were developed.

By merging these models together we were able to demonstrate that the overall performance space is unintuitive and that configurations with large number of FUs are likely to perform significantly better than an implementation of the ideal VLIW.

The area requirement for this kind of architecture was also analyzed and the results of the estimations suggest that a partitioned architecture allows to greatly decrease the requirement of area per chip up to a point where the area constraints do not pose a problem anymore.

Several benchmarks were compiled and all showed similar behavior, although in different measures; the code for benchmark P7 was used as an example during the course of the analysis presentation and an optimal architecture was found in the 10 FUs LC-VLIW partitioned in 5 modules.

Large VLIWs divided on a large number of modules, each composed by few FU and a RF, seems to be the way to go to create feasible, fast processors. Several benefits accrue from this approach: high performance, feasibility, scalability, small die size, testability are

only some. The number and the size (i.e., number of FUs) of the modules are functions of the design, the technology used and the set of programs taken into consideration. Future work needs to address the feasibility of applying this approach to newer technologies such as MCMs.

References

[1] N. Dutt A. Capitanio and A. Nicolau. Partitioned register files for vliws: A preliminary analysis of tradeoffs. In *MICRO-25, The International Symposium on Microarchitecture*, 1992.

[2] 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.

[3] Arthur Abnous, 1993.

[4] 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.

[5] Alan E. Charlesworth. An approach to scientific array processing: The architectural design of ap-120b/fps-164 family. *IEEE Computer*, pages 18–27, Sept. 1981.

[6] 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.

[7] D.D. Gajski, N.D. Dutt, A. Wu and S. Lin. *High-Level Synthesis: Introduction to Chip and System Design.* Kluwer Academic Publishers, 1992.

[8] F.J. Sparacio D.W. Anderson and R.M. Tommasulo. The IBM System/360 Model 91: Machine Philosophy and Instruction Handling. *IBM Journal*, page pp. 8, January 1967.

[9] 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.

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

[11] T. Wada et al. A 34-ns 1-Mbit CMOS SRAM Using Triple Polysilicon. *IEEE Journal of Solid State Circuits*, Vol. 22(No. 5):pp. 727–732, October 1987.

[12] H. Shinohara et alii. A Flexible Multiport RAM Compiler for Data Path. *IEEE Journal of Solid State Circuits*, Vol. 26(No. 3):pp. 343–349, March 1991.

[13] W. Maly et alii. Memory Chip for 24-Port Global Register File. In *IEEE 1991 Custom Integrated Circuits Conference*, 1991.

[14] L. A. Glasser and D. W. Dobberpuhl. *The Design and Analysis of VLSI Circuits.* Addison Wesley, Publishing Company, Inc., 1988.

[15] J. Gray, A. Naylor, A. Abnous, and N. Bagherzadeh. VIPER: a 25-MHz, 100-MIPS Peak VLIW Microprocessor, TR 92-78. Technical Report TR 92-78, Univ. of California, Irvine, Dept. ECE, 1992.

[16] R. D. Jolly. A 9-ns, 1.4-Gigabyte/s, 17-Ported CMOS Register File. *IEEE Journal of Solid State Circuits*, Vol. 26(No. 10):pp. 1407–1412, October 1991.

[17] K. Yamaguchi et alii. A 1.5 ns Access Time, 78sq-microns Memory Cell Size, 64 kb ECL-CMOS RAM. *IEEE Journal of Solid State Circuits*, Vol. 27(No. 2), February 1992.

[18] 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.

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

[20] Alexander Nicolau. Percolation Scheduling: a Parallel Compilation Technique. Technical report, TR 85-678, Cornell University, 1984.

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

[22] R. L. Sites. Alpha AXP Architecture. *Communications of the ACM*, Vol. 36(No. 2):pp. 3, February 1993.

[23] J.E. Thornton. *The Design of a Computer, the Control Data 6600.* Scott, Foresman and Co., Glenview, Ill., 1970.

[24] J. C. Tou, P. Gee, J. Duh, and R. Easley. A Submicrometer CMOS Embedded SRAM Compiler. *IEEE Journal of Solid State Circuits*, Vol. 27(No. 3):pp. 417–424, March 1992.

[25] Neil Weste and Kamran Eshraghian. *Principles of CMOS VLSI Design, A System Perspective.* Addison Wesley, Publishing Company, Inc., 1988.

A Appendix

Results of the compilation of 8 benchmarks for several configuration of LC_VLIW are presented in Tables 5,6,7,8,9,10,11,12,13. Each table contains the results for a set of architectures with the same number of functional units ranging from 1 (RISC code) to 10. Values on the same line share the same configuration parameters: number of FUs, number of modules and number of inter-bank move operation allowed per macroinstruction.

Assembly Code. 3 ports per register file							
P1	P2	P3	P4	P5	P6	P7	P8
31	130	39	43	73	18	218	381

Table 5: Number of instructions (sequential assembly code)

ideal VLIW, 2 op. per istr., 6 ports/BR							
P1	P2	P3	P4	P5	P6	P7	P8
19	69	23	22	41	9	100	204
2 Modules, 2 move-op per istr., 4 ports/BR							
P1	P2	P3	P4	P5	P6	P7	P8
23	90	30	32	50	11	131	277

Table 6: Number of macroinstructions. 2 FUs LC_VLIW

ideal VLIW, 3 op. per istr, 6 ports/RF							
P1	P2	P3	P4	P5	P6	P7	P8
18	53	18	17	34	6	70	171
3 Modules, 3 move-op per istr., 5 ports/RF							
P1	P2	P3	P4	P5	P6	P7	P8
22	82	23	31	40	9	96	236

Table 7: Number of macroinstructions. 3 FUs LC_VLIW

ideal VLIW, 4 op. per istr, 12 ports/RF							
P1	P2	P3	P4	P5	P6	P7	P8
18	48	15	15	31	5	56	165
2 Modules, 4 move-op per istr., 8 ports/RF							
P1	P2	P3	P4	P5	P6	P7	P8
18	54	18	18	33	6	66	170
2 Modules, 2 move-op per istr., 8 ports/RF							
P1	P2	P3	P4	P5	P6	P7	P8
18	54	18	19	33	6	66	170
4 Modules, 4 move-op per istr., 6 ports/RF							
P1	P2	P3	P4	P5	P6	P7	P8
22	72	20	27	36	7	82	233
4 Modules, 2 move-op per istr., 5 ports/RF							
P1	P2	P3	P4	P5	P6	P7	P8
22	76	21	28	36	7	90	233

Table 8: Number of macroinstructions. 4 FUs LC_VLIW

ideal VLIW, 5 op. per istr, 15 ports/RF							
P1	P2	P3	P4	P5	P6	P7	P8
18	47	14	14	31	4	47	169
5 Modules, 5 move-op per istr., 7 ports/RF							
P1	P2	P3	P4	P5	P6	P7	P8
22	71	18	25	36	6	70	210

Table 9: Number of macroinstructions. 5 FUs LC_VLIW

ideal VLIW, 6 op. per istr, 18 ports/RF							
P1	P2	P3	P4	P5	P6	P7	P8
18	47	13	13	31	3	41	164
2 Modules, 6 move-op per istr, 12 ports/RF							
P1	P2	P3	P4	P5	P6	P7	P8
18	47	15	16	31	4	48	165
2 Modules, 3 move-op per istr., 12 ports/RF							
P1	P2	P3	P4	P5	P6	P7	P8
18	47	15	16	31	4	48	165
3 Modules, 6 move-op per istr., 10 ports/RF							
P1	P2	P3	P4	P5	P6	P7	P8
18	55	16	16	34	4	53	182
3 Modules, 3 move-op per istr., 9 ports/RF							
P1	P2	P3	P4	P5	P6	P7	P8
18	55	16	16	34	4	53	182
6 Modules, 6 move-op per istr., 8 ports/RF							
P1	P2	P3	P4	P5	P6	P7	P8
22	68	17	24	36	6	66	238
6 Modules, 3 move-op per istr., 6 ports/RF							
P1	P2	P3	P4	P5	P6	P7	P8
22	69	17	25	36	6	69	238

Table 10: Number of macroinstructions. 6 FUs LC_VLIW

ideal VLIW, 8 op. per istr., 24 ports/RF								
P1	P2		P3	P4	P5	P6	P7	P8
18	47		12	12	31	3	35	164
2 Modules, 8 move-op per istr., 16 ports/RF								
P1	P2		P3	P4	P5	P6	P7	P8
18	47		14	13	31	4	43	164
2 Modules, 4 move-op per istr., 16 ports/RF								
P1	P2		P3	P4	P5	P6	P7	P8
18	47		14	13	31	4	43	164
2 Modules, 2 move-op per istr., 16 ports/RF								
P1	P2		P3	P4	P5	P6	P7	P8
18	48		14	13	31	4	43	164
2 Modules, 1 move-op per istr., 16 ports/RF								
P1	P2		P3	P4	P5	P6	P7	P8
18	52		14	15	31	5	51	164
4 Modules, 8 move-op per istr., 12 ports/RF								
P1	P2		P3	P4	P5	P6	P7	P8
18	54	14	14	15	34	5	48	180
4 Modules, 4 move-op per istr., 10 ports/RF								
P1	P2		P3	P4	P5	P6	P7	P8
18	54		14	15	34	5	48	180
4 Modules, 2 move-op per istr., 10 ports/RF								
P1	P2		P3	P4	P5	P6	P7	P8
18	56		14	15	35	5	55	180
4 Modules, 1 move-op per istr., 10 ports/RF								
P1	P2		P3	P4	P5	P6	P7	P8
18	63		16	22	38	6	91	181

Table 11: Number of macroinstructions. 8 FUs LC_VLIW

ideal VLIW, 9 op. per istr, 27 ports/RF							
P1	P2	P3	P4	P5	P6	P7	P8
18	46	12	12	31	2	33	161
2 Modules, 6 move-op per istr, 12 ports/RF							
P1	P2	P3	P4	P5	P6	P7	P8
18	47	15	16	31	4	48	165
2 Modules, 3 move-op per istr, 12 ports/RF							
P1	P2	P3	P4	P5	P6	P7	P8
18	47	15	16	31	4	48	165
3 Modules, 9 move-op per istr, 15 ports/RF							
P1	P2	P3	P4	P5	P6	P7	P8
18	48	15	14	34	4	42	164
3 Modules, 3 move-op per istr, 12 ports/RF							
P1	P2	P3	P4	P5	P6	P7	P8
18	48	15	14	34	4	43	164
6 Modules, 6 move-op per istr, 8 ports/RF							
P1	P2	P3	P4	P5	P6	P7	P8
22	68	17	24	36	6	66	238
6 Modules, 3 move-op per istr, 6 ports/RF							
P1	P2	P3	P4	P5	P6	P7	P8
22	69	17	25	36	6	69	238

Table 12: Number of macroinstructions. 9 FUs LC_VLIW

ideal VLIW, 10 op. per istr., 30 ports/RF							
P1	P2	P3	P4	P5	P6	P7	P8
18	46	11	12	31	2	32	164
2 Modules, 10 move-op per istr., 20 ports/RF							
P1	P2	P3	P4	P5	P6	P7	P8
18	46	13	14	31	3	37	164
2 Modules, 5 move-op per istr., 20 ports/RF							
P1	P2	P3	P4	P5	P6	P7	P8
18	46	13	14	31	3	37	164
5 Modules, 10 move-op per istr., 14 ports/RF							
P1	P2	P3	P4	P5	P6	P7	P8
18	52	16	15	33	4	42	182
5 Modules, 5 move-op per istr., 11 ports/RF							
P1	P2	P3	P4	P5	P6	P7	P8
18	53	16	16	33	4	42	182

Table 13: Number of macroinstructions. 10 FUs LC_VLIW

B Appendix

```
/* ALGORITHM */

float GAIN[MAX_NO_OF_NODES][MAX_NO_OF_CLUSTER]; /* GAIN[i][c] = gain to move op i to cluster c */
int   STATE[MAX_NO_OF_NODES];                    /* STATE[i] = op i has been moved (true/false) */
int   HISTORY[MAX_NO_OF_NODES][2];               /* HISTORY = infos on previous move */
int   PART[MAX_NO_OF_NODES];                     /* PART[i] = cluster of op i */
int   BEST_PART[MAX_NO_OF_NODES];                /* BEST_PART temporary best solution */


/* INPUT VARIABLES */
int nodes_number, cluster_number;

do { /* loop 0 */

   calculate_initial_gain();
   while (TRUE) { /* loop 1 */

         for (i=1; i<= node_number; i++) { /* loop 2 */

            best_gain = find_best_gain(best_op,best_cluster);
            STATE[best_op] = TRUE;
            HISTORY[i][FROM] = PART[best_op];
            HISTORY[i][OP] = best_op;
            PART[best_op] = best_cluster;
            TEMP[i] = best_gain;
            update_gain();
            G = max_partial_sum_of_TEMP(index_of_max);
            if (G > ε)
              backtrack_up_to(index_of_max);
            else {
              backtrack_all_movement();
              break;
            }
         } /* end of loop 2 */
   } /* end of loop 1 */

   communication_value = compute_comm_val();

   if (communication_value < minimum) {
     K = K_INIT;
     minimum = communication_value;
     BEST_PART = PART;
   } else {
     PART = BEST_PART;
   }

   if (K < ε)
     break;
   else
     K = f(K);

   randomize(PART,K);

} while (TRUE) /* end of loop 0 */
```