Clustered Instruction-Level Parallel Processors

Paolo Faraboschi, Giuseppe Desoli, Joseph A. Fisher
HP Laboratories Cambridge
HPL-98-204
December, 1998

VLIW, registers, clustering, compilers, EPIC, scheduling

CPUs with a large amount of instruction-level parallelism must carry out many register accesses each cycle. Eventually this leads to severe hardware bottlenecks and a loss of cycle time. A solution that has been proposed and implemented a few times is "clustering". Clustered ILP CPUs have several groups of hardware each consisting of a register bank and one or more functional units. Functional units may only access registers in their associated bank. To access registers in other banks, explicit or implicit intercluster moves must be made while a program is running.

CPUs offer ILP in a great variety of ways, and thus there are many different ways clustering may be carried out. However it is done, clustering represents a tradeoff between cycle speed and cycle count: it will take more cycles to execute a program on a clustered CPU than a single clustered CPU with the same functional units and total number of registers, but the clustered CPU will have a faster clock. We measure here the cost of clustering on multiple cluster VLIW architectures, particularly using a new algorithm called Partial Component Clustering.

Remarkably, the experiments reported here suggest the same ballpark results seen in very different environments. Indeed, the results seem similar across strikingly different architectures, layout algorithms, benchmarks, and degrees of ILP. As a rule of thumb, breaking the CPU into two clusters costs somewhere around 15-20% lost cycles; four clusters costs around 25-30%. As feature sizes of microprocessors decrease in relation to communication costs, these numbers are likely to strongly favor the use of clustering, at least for CPUs which execute applications with large amounts of ILP.

Internal Accession Date Only

1 Introduction

Instruction-level parallel (ILP) architectures using VLIW and superscalar design styles achieve their high-performance by issuing multiple overlapped operations. With the increased capability of overlapping operations comes the increased need to supply register bandwidth, since a typical operation requires 3 register accesses. Thus a 4-issue superscalar CPU might require 12 register accesses per cycle; a 12-issue VLIW media processor might require 36 accesses per cycle.

Eventually the access time to and from a central register file becomes the critical path in the cycle time of the processor. The bottlenecks are due to the combined effects of:

- Register access time,
- Silicon area for address decoders and data routers,
- The additional instruction bits for register specifiers and
- The distance of a central register file to the numerous consumers and producers of data.
- Bypassing logic, which grows with the square of the number of readers and writers.

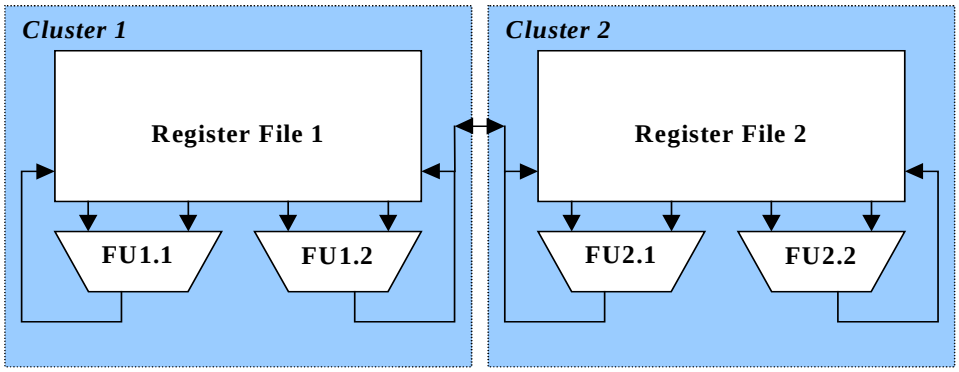
In today's technology, these bottlenecks are likely to become serious somewhere between the register requirements of the two examples above: a 4-issue CPU and a 12-issue CPU.

1.1 "Clustered" Architectures

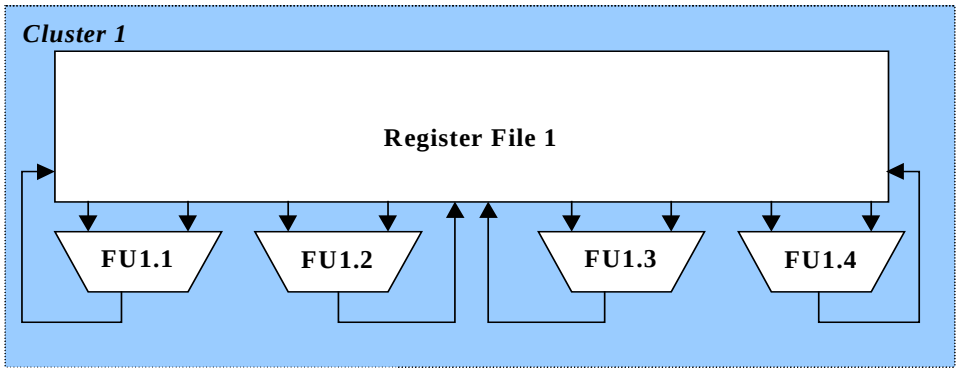
A very natural solution to these problems is "clustering". A typical clustered architecture's datapath has several groupings, identical in structure or possibly somewhat different, each consisting of a register bank and one or more functional units. Functional units most efficiently access registers in their associated bank. To access registers in other banks, explicit or implicit intercluster moves must be made while a program is running. Figure 1 (b) shows a 4-issue ILP datapath having a 8-read, 4-write central register, while Figure 1 (a) shows a similar 4-issue ILP datapath, but now having 2 clusters of 2 function units and one register bank each. There is, additionally, an "intercluster bus" or some other form of communication, used to move date between clusters.

Note that clusters neatly solve all of the problems listed above. By having smaller register banks and fewer addresses to decode, we get faster access times. The specification of which cluster is being addressed may be made once in the instruction, rather than in each reference, potentially saving instruction bits. (As

discussed below, a superscalar may or may not yield this particular savings, while a VLIW is likely to implicitly specify the cluster in the placement of the operation within an instruction, saving even more instruction bits.) Register files can be located near to their consumers and producers of data. In short, we get potentially greater parallelism in a more scalable manner. Clustered architectures also address the bypassing problem, since the number of fully connected units (i.e. the pairs of writers-readers that have to be bypassed) is reduced. This enormous advantage is often overlooked in discussion of the desirability of clustering.



(a)



(b)

Figure 1. Two possible organizations of a 4-unit machine: 4-unit 2-cluster (a) or 4-unit 1 cluster (b). Configuration (a) requires building two register files, each of them with 4 read ports and 2 write ports. Configuration (b) requires building one register file with 8 read ports and 4 write ports. For the clustered architectures, we assume to share one read port and one write port per cluster for intercluster communication.

But things rarely come for free. The need for intercluster moves brings its own potential problems. When the intercluster moves are architecturally visible, there are systems issues, which we discuss below. But in any case there will be:

- Lost performance due to cycles devoted to intercluster moves, or due to the unbalanced or poor use of the clusters.
- Hardware bus and port costs in supporting intercluster moves and
- Hardware or compile-time costs of algorithms that lay out data in order to minimize intercluster moves and/or unbalanced use of the clusters.

1.2 Past Use of Clustering and Similar Techniques

Clustered instruction-level parallel implementations were first suggested and named as such, as far as we know, in [1]. There have been implementations in the form of a VLIW (the Multiflow Trace [13]) and, to a very limited extent, in at least one superscalar processor (the DEC 21264, [2]). Several papers have investigated the effects of clustering in VLIWs and similar architectures (see, for example, [3] [4] [5] [6]), and in superscalars (the most thorough being [7]). We cover the experimental results in some of these papers below, most notably [7], and compare them with our own. Interestingly, the results seem far more consistent across experiments than results usually are in computer systems work. Indeed, the results seem invariant across strikingly different architectures, layout algorithms, benchmarks, and degrees of ILP. The results are consistent enough that we believe some rules of thumb can characterize the number of extra cycles due to clustering, and we do so below.

The clustering technique is in some ways similar to several other tried-and-true CPU techniques. A trick to improve cache performance is to divide it into two caches, one for data and one for instructions. This technique, referred to as a "Harvard Architecture", allows not only the kinds of advantages listed above, but allows also for asymmetric caches, built to handle the two different types of cached objects. Similarly, several architectures divide the integer and floating point register banks, yielding what may be thought of as a floating point cluster and an integer cluster. Finally, the concept of decoupled access/execute [8] again divides the hardware, this time between memory oriented operations and non-memory operations. These techniques do not suffer the potential problem of clustering. They all deal with fundamentally different data types, thus they are taking a hardware data structure and dividing it into two structures containing distinct data. In clustering we are taking a structure with a single data type and no major difference in functionality and dividing it into several pieces.

1.3 Minimizing Performance Loss Due To Clusters

We have not yet discussed the different ways in which different systems cause data to end up in a particular cluster's register bank, or how a particular function unit is assigned to carry out an operation. It is intuitively clear, however, that if data is laid out poorly, intercluster moves will ruin the performance advantages of clustering, and will reduce instruction-level parallelism. Alternatively, if it is possible to lay data out cleverly enough that few intercluster moves are required, there might be an insignificant loss of cycles, and it might be possible to get by with a minimal bus structure to support the few moves. But life isn't as simple as one might wish: good schedules might require us to move data around far more than is minimally required for correctness. One can see this easily by recognizing that code could be laid out to occupy only a single cluster, eliminating the need for any intercluster moves whatsoever. It is intuitively obvious (and measured below) that this is a bad idea when the ILP gets even a little beyond the resources available in the single cluster. Thus, as we explain in more detail below, laying out the code for a clustered architecture is a subtle and potentially difficult process.

This paper, then, is an investigation into how one might get the advantages of clustered architectures—and those are big advantages—while laying out the data to produce a computation that runs as fast as possible. In the next section we describe briefly architectural issues in clustered architectures. Following that, we report on experiments that quantify, at least in one system, the greater need for registers as ILP increases. Finally, we present a new clustering algorithm, “Partial Component Clustering” (PCC) and compare its performance with another algorithm, that used in the Multiflow commercial VLIW compiler. From these results, and others in the literature, we deduce what we believe to be generally applicable principles.

2 Clustered Architectures and Intercluster Moves

2.1 Clustered VLIW Architectures

It is very straightforward to implement a VLIW-style architecture with clustering. Hardware structures are usually exposed to the compiler, and code is typically recompiled to match the hardware exactly. Thus it is easy to build in specific MOVE instructions, and to ask the compiler's code generator to specify the moves whenever it generates code for an operation whose operands and functional unit are not all in the same cluster.

In a CPU with four clusters, numbered 0-3, each containing a 64 register bank and two ALUs, the code generator might find it wants to add two values found in one cluster, and add that sum to a value found in another cluster. The operations the code generator might want to use to carry this out might be:

```
ADD2.1 R165 <- R160, R161
ADD0.0  R59 <- R12,  R165
```

where, for example, ADD2.1 means adder number 1 in cluster 2. Notice that the second add requires a source value in cluster 2 and another source value in cluster 0, putting its result in cluster 0. Thus the compiler cannot simply issue this sequence, since the adder in cluster 0 cannot reference the value in R165. Instead, the code generator might generate the instructions:

```
ADD2.1 R165 <- R160, R161
MOVE    R58 <- R165
ADD0.0  R59 <- R12, R58
```

If the second add is not on the critical path of the computation, and there are sufficient instruction-level parallel resources to carry out the move early enough, then the move can often be done for "free"—i.e. without increasing the schedule length.

2.2 Clustered Superscalar Architectures

Unlike VLIWs, superscalar implementations usually have significant artifacts that are not visible on the architectural level. Clustering can be one of those artifacts: superscalar hardware typically has the ability to adjust the schedule at run time. The ways in which schedules are rearranged varies significantly from implementation to implementation, thus there are a great many variants on how a superscalar-style processor could deal with clustering. We are not aware of any clustered superscalar processors that have been built except for the Alpha 21264 (which keeps its register banks synchronized), nor designed in any detail, except for the Multicluster Architecture. We believe, however, that if and when it is practical to solve the other problems involved in building a superscalar with enough ILP to warrant clusters, then clusters are probably the right choice.

The datapath of a clustered superscalar can be similar to that of a clustered VLIW. Once again, we have functional units whose operands must come from the register bank in the same cluster. And, once again, intercluster moves must support the situation in which data must be moved to the proper cluster. In this case, however, the code generator doesn't place an explicit move operation in the code. Instead the runtime hardware detects the need for the move, and moves the data as and when required. It doesn't have to do what is done in a VLIW: move the data to a register in the proper cluster. More likely, the data would be moved to a register staging area, similar to or perhaps shared with bypass queues in the proper cluster.

Thus the hardware can follow a very simple algorithm when decoding an operation:

1. For each source value found in a cluster different from the destination, issue a move to place the value into the input queue of the destination's cluster.
2. Dispatch the operation to a functional unit in the destination register's cluster .

To execute the same sequence just seen, the following would happen:

```
ADD R165 <- R160, R161       /* Addition done on an adder in cluster 2 */
NOP                          /* R165 moved to cluster 0 input queue    */
ADD R59  <- R12, 'R165'      /* Addition done on an adder in cluster 0 */
```

Compared to the explicit register target of a VLIW, this lowers register pressure. As with a VLIW, the hardware might be able to do the intercluster move in parallel with other operations.

Figure 1 shows the datapath as we have described it. In the simplest case, the superscalar's additional hardware control is minimal: it merely has to assure that the need for an intercluster move is detected and the move is issued.

We discuss below the implications of such a simple system, in terms of how many intercluster moves would be required, but there are two ways in which a superscalar system could reduce the number of intercluster moves below the level that this simplest case implies:

1. As is required with a VLIW, but optional with a superscalar, a compiler could rearrange the code with that superscalar's topology in mind before it is handed to the system. This flies somewhat in the face of superscalar philosophy, however, in that a system with different clustering would require a different set of register and functional unit assignments.
2. More compatible with superscalar philosophy, one could build a superscalar that tries to minimize intercluster moves in its assignment of architectural registers to physical registers, and in its selection of functional units. This would be done at instruction dispatch time, and is described in more detail in the next section.

In the Multicluster Architecture, both of these are proposed.

2.3 A More Complex Clustered Superscalar

Superscalar processors have already been built with extremely complex control units (see, for example, [9] and [2]). Given that complexity, there are very many variants on how one might build a control unit which tries to minimize intercluster moves. A satisfying discussion of adding clustering to a complex superscalar would be well beyond the scope of this paper—such a discussion, for one set of choices, appears in [7]. Instead, we will outline some of the considerations involved in one particular aspect of superscalar design, namely "register renaming", which could enable more effective use of clustering than within a simple superscalar. A superscalar utilizes register renaming when it maps an "architectural register",

that is, one referenced in the object code, into a physical register. The physical registers may differ in number and structure from the model given in the architecture, as long as runtime hardware consistently keeps things straight.

Register renaming is used in ambitious superscalars to reduce false dependence relationships caused when different logical data is mapped into the same register. It is particularly effective when there can be many more physical than architectural registers. Similarly, a clustered superscalar could use register renaming to change the clusters in which computations are done. In that case, runtime hardware could examine the program as it's being issued and could map architectural registers to physical registers in such a way as to minimize computation time.

For example, suppose the lines of code shown above continued with a third operation as follows:

```
ADD R165 <- R160, R161
ADD  R59 <- R12,  R165
ADD R166 <- R161, R59
```

Unless the value put in R59 by the second operation is used extensively in cluster 0 soon after, it would make sense to do the second operation on cluster 2 rather than cluster 0, which is what the simple algorithm above implies. To accomplish this, the superscalar dispatch hardware might decide to map R12 into a physical register in cluster 2. This might require the control hardware to issue a move, if the data in R12 was created in cluster 0, but it could create a more balanced computation with a shorter critical path and fewer moves.

Such a scheme is given in detail in [7]. We report below on experiments that measure the effectiveness of more sophisticated register mapping algorithms to shorten computations on clustered architectures. However, we did not attempt to use our benchmarks, which are very different and contain far more ILP than those in [7], to characterize algorithms that might be practical to implement in runtime hardware.

3 Tools and Benchmarks

3.1 The Benchmark Suite

For our experiments we selected a set of six integer benchmarks representing typical media-intensive processing task. The reason for the choice—as opposed to more traditional workload like SPEC—is due to the fact that we are interested in applications that show significant amount of ILP and are at the same time interesting from an industrial perspective. So we decided to pick a set of important applications that handle and transform multi-media data and we hand tuned the

sources to make them more suitable for our compiler technology. Hand tuning refers simply to identifying the most important code segments (typically loops), and annotating them with proper directives to guide the compiler optimizations.

In addition to computation intensive benchmark, we also include *Dhrystone 2.1,* to measure clustering on an example of code with modest amounts of ILP.

Table 1 shows a collection of basic statistics for the individual benchmarks.

Benchmark	Lines of C code	Baseline Cycles	Description
bmark	6,700	377,680,815	Typical color printer pipeline. Composed of JPEG decompression, image scaling, color conversion and correction, under-color removal, and halftoning with error diffusion. This is the type of computation you can expect in a color laser or ink-jet printer to rasterize a photographic-quality picture.
copymark	3,500	913,100,557	Typical color copier pipeline. Composed of color-space conversion, scan filters, device correction, interpolation, edge-enhancement filters, and halftoning. This is the type of computation you can expect in a high-end color copier.
crypto	11,200	118,548,255	Cryptography benchmark. Composed of a digital file signature and authentication pair using the ECC (Elliptic Curve Cryptography) algorithm. This is the type of computation involved in a secure document transmission.
mpeg2	8,300	959,947,540	MPEG-2 motion picture decoding. This is the type of computation involved in playing digital motion pictures, such as in a DVD player.
tjpeg	2,900	44,472,146	JPEG-like compression of a full 3-color 8-bit image. This is the type of computation you can expect, for example, in a digital camera.
dhry	1,000	3,170,773	The Dhrystone 2.1 benchmark. Included as a contrasting example of code not containing interesting levels of ILP.

Table 1. Description and characteristics of the benchmarks used in the experiments.

For all of our experiments, we ran the benchmarks with representative data inputs, large enough to give significant results, but small enough to keep simulation time within practical limits. A complete compilation and execution cycle of all the benchmarks takes about 30 minutes on a Hewlett-Packard 9000/J200 workstation.

3.2 The Machine Model

In order to be able to present consistent results (and in some cases not compromise confidential benchmark data), we present all results relative to a *baseline* configuration. The baseline is a scalar architecture with a single cluster containing 32 registers and one unit that can issue one operation per cycle. All integer operations are single-cycle, except multiplies that take 3 cycles, and memory load operations that also take 3 cycles. All latencies are completely exposed to the compiler. The memory operations are pipelined, so there is some ILP already in the baseline case.

For the other machine configurations, we vary the following parameters:

- **Units:** the issue width of the CPU. Any operation consumes an issue slot. For all configurations, we assume that half of the computational units are capable of executing a memory access. We never allow more than a single control-flow operation per cycle.
- **Registers:** the number of general-purpose registers in the CPU. The compiler uses general-purpose registers for all temporary data. The calling convention is caller-saves and passes arguments and return values in registers.
- **Clusters:** the number of clusters in the CPU. A cluster contains a single register file and multiple units that are fully connected with the register file. Communication across clusters happens via *intercluster copy* operations that consume one issue slot in the source cluster and one issue slot in the destination cluster.

3.3 The Compiler and Simulation Framework

Our compiler is a descendant of the Multiflow compiler. The Multiflow compiler uses region scheduling, specifically the *trace-scheduling* algorithm, to exploit instruction level parallelism across multiple basic blocks. The compiler was originally designed for the Multiflow Trace machine, the first supercomputer class implementation of the VLIW design style.

The compiler includes C and FORTRAN front-ends and implements a set of "traditional" high-level optimization (common-subexpression elimination, if-conversion, loop unrolling, reductions, etc.). The analysis phases include loop

analysis, identification of induction variables, and “disambiguation'” of memory aliases.

The trace scheduler identifies “traces” (collections of basic blocks with multiple-entries and multiple-exits) and generates compensation code to patch possible motions of instructions below splits or above joins that may happen in the code generator.

The code generator implements an integrated schedule and register-allocation phase based on list-scheduling heuristics. The register-allocator is a region-based allocator (where the regions are the traces), starts with everything in registers and generates spills when necessary that are added to the DAG and scheduled during the code generation phase itself.

The compiler is robust, generates efficient code, and is flexible enough to be a good research platform. The remainder of our compiler toolchain includes assemblers, linkers, simulators, and analysis tools that allow us to gather precise static and dynamic information on various aspects of the benchmarks and the architecture under evaluation.

4 The Need for Registers: Experiments

We first ran a set of experiments to quantitatively evaluate the performance implications of the number of registers and the number of units on ILP.

We set up the experiments as follows.

- We compiled and simulated the benchmark suite for a set of machine configurations with varying number of registers and units. We chose a range of registers between 32 and 256 and a range of units between 2 and 16.
- To keep multiple units busy it is necessary to consider large enough compilation regions. In our compiler technology this is mainly achieved by unrolling loops. For the experiments, we established a threshold so that we unroll inner loops to have approximately 100 operations per unit. This is unfortunately sub-optimal, since the register requirements are not directly related to the number of operations, but it is one of the constraints of using a real-life compiler.

Figure 2 represents the average performance increase for all benchmarks (but *dhry*) as a function of the number of registers. From the graph we can observe that

- Machines with a larger number of units benefit more from an increase in the number of registers. For example, while a 2-unit machine is at 90% of its potential with 32 registers, a 16-unit machine is barely around 60%.

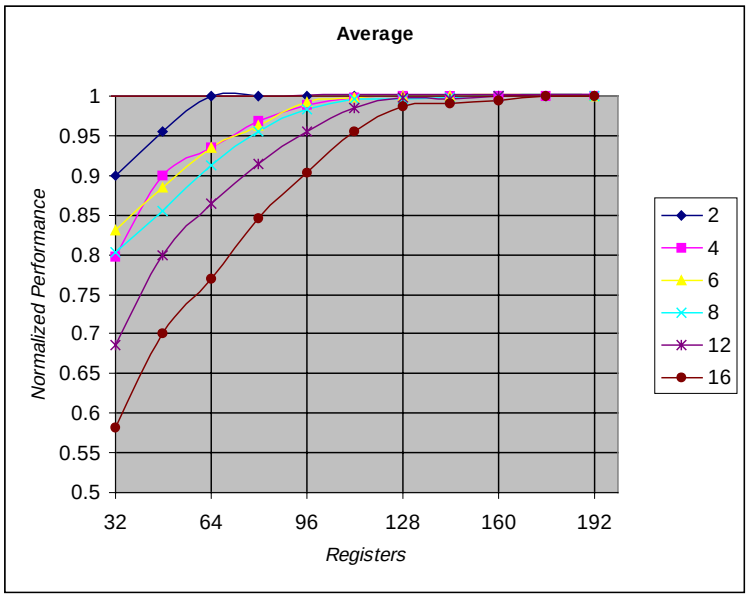


Figure 2. Average performance increase for all benchmarks (but *dhy*) as a function of the number of registers. The vertical axis is normalized to the best measured performance for the considered number of units.

- There seems to be a clear threshold around 128 registers for the machine width and benchmarks considered in the experiments. Even for a 16-unit machine, the advantage of having more than 128 registers rapidly decreases below the 5% range. We are not sure this result holds in general, since this depends on many factors. As it is more evident from Figure 3, the amount of ILP in the benchmark suite rarely goes above 10 (times the small amount already in our baseline). This means that there is plenty of room for a 16-issue machine to hide spill and restore operations in the otherwise idle units. Bear in mind that our 16-unit machine is capable of operating 8 memory accesses in the same cycle, so the cost of a spill/restore pair is definitely low. For machines with a more limited memory access rate or a heavier spill/restore penalty, the register pressure is going to grow and the need for more registers may arise.

Figure 3 shows the individual benchmark results in terms of performance relative to the baseline configuration. We can see how the register requirements vary as a function of the available ILP. If we look at the two extremes, at one extreme *dhry* yields a very flat profile, meaning that additional registers would not increase performance and probably even less than 32 would suffice. At the other extreme, *crypto* yields levels of ILP in excess of 10, and does require a considerable amount of registers to be able to exploit this potential. For example, doubling the number the registers (from 64 to 128) improves performance of about 40% (from 9.8 to 14.0).

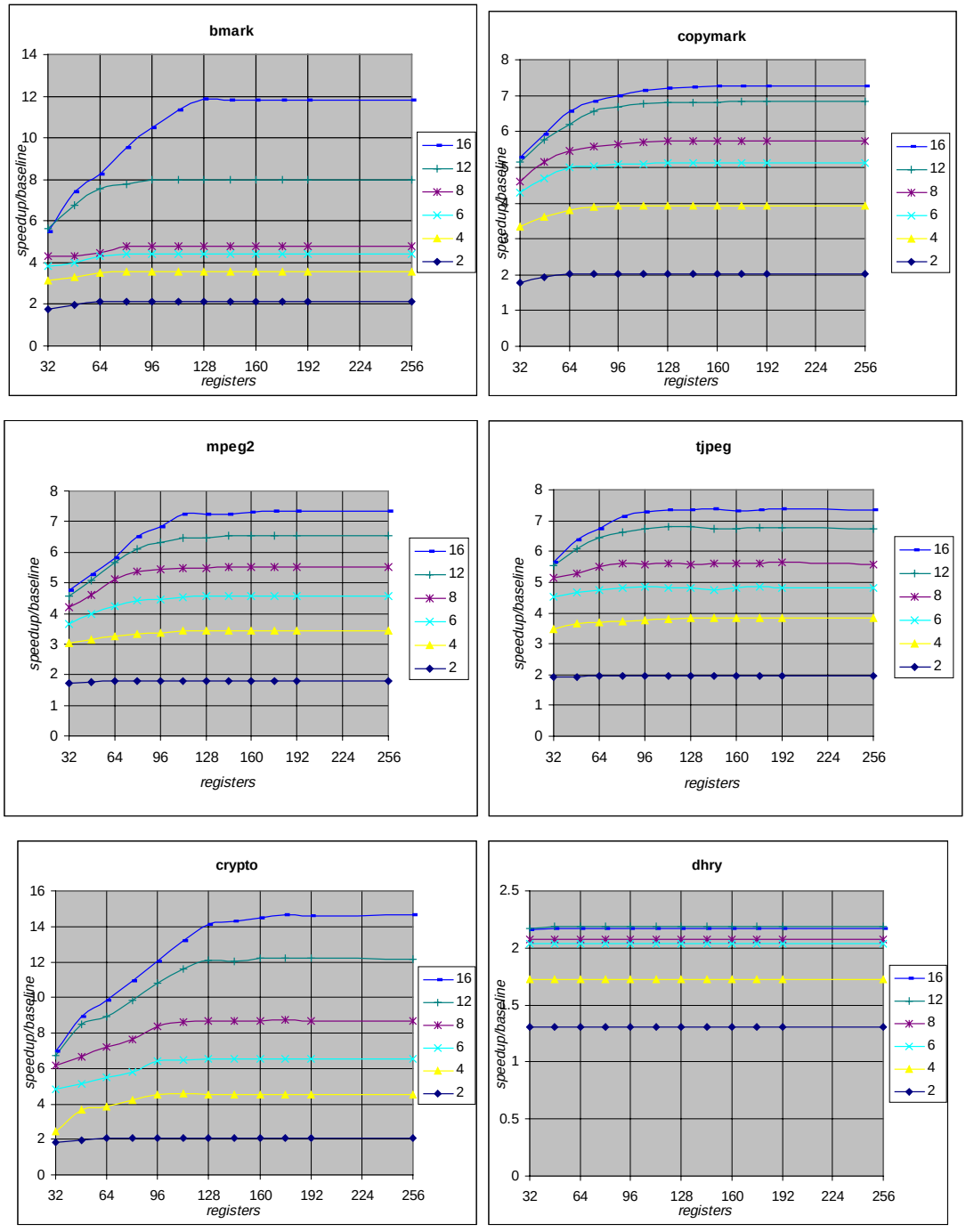


Figure 3. Detailed individual results for the six benchmarks.

5 Generating Code for Clustered Machines

As we have seen in the previous sections, many considerations suggest the opportunity of building "clustered" architectures, where the register space is physically separated in distinct register files with limited connectivity. This has important consequences on the choice of the compiler algorithms for the *code generation* phase[1].

Traditionally, a compiler does not need to make a decision concerning *where* an operation has to be executed, either because there are no choices (e.g., for a scalar machine) or because the functional unit assignment is done transparently by the hardware (e.g., for a completely connected superscalar machine).

When the connectivity is not complete and is exposed at the architectural level, the association of operations to computational units becomes a compiler responsibility. In other words, the compiler is presented with the non-trivial task of *choosing* one particular functional unit for each operation.

The nature of the problem to be solved depends on the sophistication of the hardware support for the interconnection among register files and functional units:

- When the hardware provides transparent support to fetch the operands from a "remote" register-file when needed (possibly with a dynamic penalty), the compiler's task is to apply heuristics to minimize the number of dynamic stalls. In this case compiler choices have impact on performance degradation, but still yield correct code.
- When functional units can only read and write certain locations and the connectivity is completely exposed at the architectural level, the compiler's task becomes significantly harder, since it is now the compiler's responsibility to issue the "copy" operations to move data to appropriate locations. In this case, compiler choices affect correctness on top of performance. We believe that this fact has important consequences on the algorithms to be chosen for the scheduling and the register allocation phases.

5.1 Previous Algorithmic Work

The problem we are trying to solve is the following:

---

[1]The terms: "instruction selection" and "code generation" are often used to describe different concepts in literature. To avoid terminology confusion, we call "*instruction selection*" the phase that lowers the intermediate representation to sequences of machine-specific operations. Conversely, we call "*code generation*" the phase that produces the bindings between machine elements (such as register and units) and machine-specific operations. Code generation includes *assignment of units and registers* and *scheduling* of operations into machine-level cycles.

Given a direct acyclic graph (DAG) of operations, a set of heterogeneous resources (computation and storage) with different latencies and constraint, and a limited interconnection network between computational and storage units.

Find a mapping between nodes and resources and a temporal ordering of the operations that minimizes the time to complete the execution of the DAG without violating latency, resources and communication constraints

The problem of scheduling tasks on a set of resources with limited connectivity and communication costs is well studied in many scientific areas, such as task scheduling on multiprocessor systems, and so on [11]. Being an NP-complete problem, many combined scheduling/assignment algorithms have been proposed that try to sub-optimally solve the problem. Several of the existing studies on the subject approached the problem of scheduling a computational DAG (Direct Acyclic Graph) of tasks on a multiprocessor system [11][15][16][17][18][19][20].

However, it is hard to directly adopt these algorithms to solve the problem of generating code for clustered architectures. The major limitations come from the assumptions on the structure of the DAG [15], the assumptions on zero-delay communication costs [16]; or from the requirements on the number and type of processors [17][18]. Some techniques require the nodes of the DAG to have certain properties (such as an execution time that decreases with the number of processors applied to them [19]). Interesting work has been done in [19] for the scheduling of DAGs for asynchronous execution on a multiprocessor system. However, their results cannot be directly applied when communication and execution happen in lock step, for VLIW architectures.

An alternative approach to solve the problem could be to view it as a multi-way-patitioning problem. Much has been done on the multi-way-partitioning problem where a graph has to be subdivided into k subsets by minimizing a given cost function. Frequently used cost functions attempt to balance the number of nodes in a cluster versus the number of edges (cuts) between clusters. Linear time algorithms have been proposed for a given cost function [21], and many sub-optimal ones for other types of cost functions.

Unfortunately, there is only a limited applicability of these approaches when the cost to be minimized is the scheduling length of a DAG given limited resources and many other complex interactions with issues like register allocation, register spill/restore and so on. In addition, traditionally used cost functions have a very small correlation (if any at all) with the overall schedule length, except when the DAG has particular symmetries. This can be true when the DAG is composed of separated components whose number is an integer multiple of the number of clusters, but is not in general. For these reasons we discarded the k-way

partitioning approach for all cases but the highly symmetric ones, where we apply a similar strategy in the selection of the starting configuration.

5.2 The Bottom-Up-Greedy Algorithm (BUG)

Probably the most exhaustively described algorithm for the cluster code generation problem, BUG (for the Bottom-Up-Greedy), was originally designed within the Bulldog compiler at Yale by John Ellis in the mid 80's [22]. It then became part of the Multiflow compiler [13], which is currently used as a research platform in various industrial and academic institutions around the world.

BUG works in two phases:

1. In the first phase BUG traverses the DAG from the exit nodes (leaves) to the entry nodes (roots), estimating the likely set of functional units to be assigned to a node based on the location of previously assigned operands and destinations. When it reaches the roots, it works its way back to the leaves, and selects the final assignment for the nodes along the way. To reach a final assignment, BUG estimates the cycle in which a functional unit can compute the operation based on resource constraints, the location of the operands and the machine connectivity. The available cycle of a node is computed differently, depending on whether the operand nodes are already assigned or not:
   - If any operand node is already assigned, then the available cycle is computed as the sum of operand cycle and distance between the operand unit and the current unit
   - If no operand node is assigned, then the available cycle is guessed on the basis of the operand depth (the distance of the longest path from the roots) and possibly the feasible locations of the operand destinations.

   Once the cycle estimates for all the feasible units for a node are available, BUG selects the unit producing the smallest output delay for each node.
2. In the second phase, BUG assigns initial and final locations to the variables that are live in and out of the DAG. This phase is quite delicate, since it affects the adjoining regions of code, and particular care has to be taken to avoid redundant duplication of locations for critical values (such as induction variables in loops) without sacrificing the parallelism opportunities. Various heuristics that can be applied are well documented in [22].

In its operating mode, BUG makes a few simplifying assumptions:

1. Functional units are the only limiting resources in the machine, and conflicts due to scarce register-bank ports or buses are ignored. The rationale behind this assumption is the following: if the bandwidth of the register banks is not adequate for the number of computational units that can access those banks,

then the machine is probably not balanced and improperly designed. We can say that such an assumption in general holds true for well-designed architectures.

2. The additional resource costs and delays involved in explicitly scheduling the copies in machines that require them are ignored. This may degrade the precision of BUG cycle estimates when copy resources are critical. However, this is not a fundamental limitation of the algorithm and the capability of tracking copy resources could be easily added to BUG.
3. Register pressure is ignored. Under register pressure the topology of the DAG can change significantly due to the presence of spill/restore operations. This is on e of the major limitation of the algorithm, and is mainly due to the fact that the scheduling model of BUG is very simple and not accurate in the presence of serious register pressure.
4. If we view BUG as an optimization algorithm, we can say that the cost that it tries to minimize is only driven by the delay of scheduling a node on a given computational unit. In other words, the impact of the choice of a unit on the global schedule is only taken into account by giving precedence to nodes on the critical paths. This is a fundamental limitation of the algorithm, since it means that choices are based on local criteria only and overlook "global" optimization possibilities based on the overall DAG structure.

In some sense, we can say that the major weakness of BUG is the fact that it is "short-sighted". BUG assigns nodes in a greedy way without knowing what the impact of a node assignment is on the global schedule length. On top of that, node assignments are never changed and the effectiveness of previous choices is never evaluated.

Given all these limitations, BUG still produces reasonable results (sometimes excellent ones) with a quite low computational complexity.

6 Clustering through Iterative Improvement

If we step back and look at the problem from a system point of view, we can think of clustering as a combinatorial optimization problem whose goal is to produce a set of parameters that minimize the output of a strongly non-linear system. In our case the parameters are the assignment of units to nodes, the output is the global schedule length and the non-linear system is the block composed of the scheduler and the register allocator.

From an optimization perspective, the target system (i.e., the scheduler and register allocator) presents a contrasting set of features:

- The system is strongly non-linear and impossible to characterize in an analytic form. Small changes in the parameters, such as an operation assigned to another

unit, may cause dramatic changes in the final schedule length. This is mainly due to the register allocation task, whose behavior is dominated by hard thresholds (the number of physical registers) and causes the schedule length to degrade ungracefully, mostly depending on the burden of spill/restore code.

- The computational requirements of obtaining a single value of the cost function for a given assignment is quite high, since it requires to complete scheduling and register allocation on *all* the nodes of the DAG. On top of that, we should not underestimate the engineering challenges of writing a real-life scheduler and register allocator that could be invoked iteratively on the same region.
- The dimensionality of the problem gets easily out of control for real-life cases. For example, to be able to exploit ILP in the range of ten, we need to consider compilation regions with at least thousands of nodes (typically achieved by inlining procedures, predicating control flow and unrolling loops). For a 1,000 node DAG and a four-cluster machine, there are 4^{1000} legal cluster assignments. Even if we consider all the possible problem symmetries, it is easy to see that an exhaustive search is probably not practical.

These consideration lead to the following dilemma.

- The fact that it is very hard to characterize the cost function rules out any traditional *gradient-descent*-like technique. Ideally, this seems to push in the direction of explicit enumeration algorithm, such as *branch-and-bound*, or even a non-deterministic system like *simulated annealing, TABU* or *genetic* search. These algorithms have been proved to be very effective for "black-box" systems in many other areas.
- However, the cost of computing a data point for the cost function is quite high, and this adversely affects the same iterative algorithms, that usually demand a very large number of trials to converge to good minima.

To reach a reasonable compromise, we need to operate on two fronts.

1. We need to approximate the behavior of the system. To do this we can use a *simplified* form of the scheduler to get an estimate of the quality of a tentative cluster assignment. The simplified scheduler needs to take into account resource models and register pressure, but can only do a half-hearted job as long as its behavior is more or less consistent (monotonic) with respect to the real scheduler.
2. We need to reduce the dimensionality of the problem. Even with a simplified scheduler, we cannot even think of exploring 4^{1000} combinations within the time frame of a compile job. A possible way of reducing the dimensionality is to

construct *"macro-nodes"* for partially connected components of the original DAG that are treated as indivisible units assigned to a single cluster.

The proposed algorithm (called *PCC – Partial Component Clustering*) works in three phases:

1. Partial Component Growth.
2. Initial Assignment.
3. Iterative Improvement.

6.1 Partial Component Growth

In the first phase of the algorithm, we grow partial components starting from the DAG's leaves and following the longest path backward towards the DAG's roots until we hit a threshold (ϕ_{th}) of the maximum number of nodes in a partial component. When the component growth reaches an entry point of the DAG or a node already visited, recursion restarts along one of the pending paths originating from one of predecessor nodes in the stack of recursive calls. Larger values of ϕ_{th} get a smaller number of components and vice-versa.

In this way, we add paths to the partial components in a "critical path first" fashion. Since one component is assigned to a single cluster, the rationale behind this is to avoid inserting copies along the critical paths in the DAG. **Figure 4** shows this concept for a simple DAG of a code fragment representing a four-tap FIR filter unrolled twice. By setting $\phi_{th}= \infty$ we get four partial components, two of which represent the unrolled iterations of the loop.

In general, for $\phi_{th}, = \infty$, two different situations may arise:

1. When the DAG is *fully connected*, the growth process will produce one single component containing the entire DAG.
2. When the DAG is made of several *separated* sub-graphs, the process produces a larger number of components, each one containing an unconnected portion of the DAG. Note that here we use the concept of separation in a loose sense, as we only traverse the DAG following backward edges. A typical case of a separated DAG arises from loop unrolling, especially in the absence of loop carried dependencies across iterations.

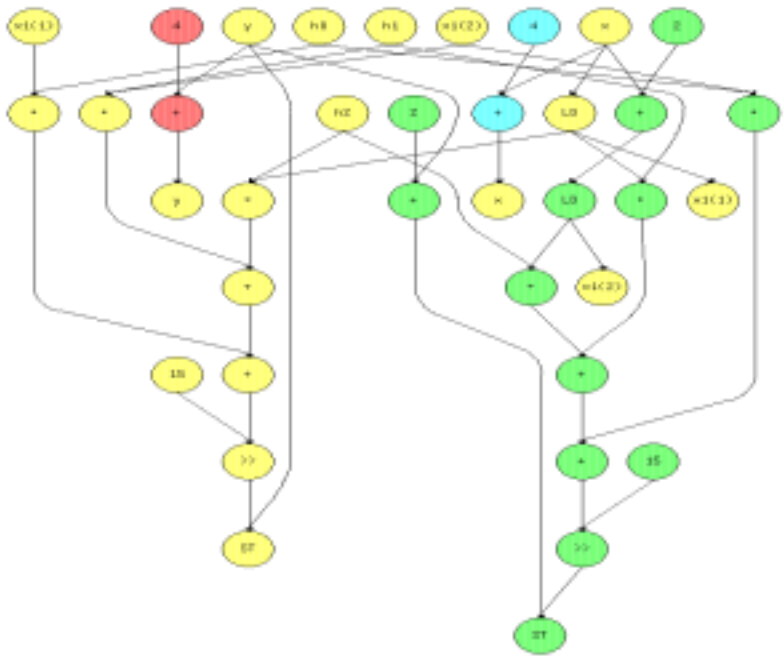


Figure 4: DAG for a three-tap FIR filter unrolled twice. We can see four partial components (in different colors), two of which correspond to the two unrolled loop iterations. According to our definition (see text), this DAG is separated.

6.2 Initial Assignment

In this phase, the set Φ of partial components is used to perform a simple mapping to produce a *k*-way cluster assignment. We apply different heuristics for the different classes of DAGs.

- For *fully connected DAGs*, the mapping simply scans Φ, reordered by decreasing size of its elements, for all partial components and assigns them to the least utilized cluster based. This simple mapping effectively achieves a good load balancing between clusters, but it does not guarantee to produce a good initial assignment for DAGs made by separated sub-graphs, since it does not take into account inter-cluster copies in the mapping process.
- For *separated DAGs*, we empirically found that minimizing the number of inter-cluster copies generally leads to good solutions. In addition, this greatly improves the convergence process of the iterative phase. In this case, we assign each component to a cluster by minimizing a cost function that takes copies into account. This problem can be formulated in closed form and solved as a

constrained integer linear programming problem. To simplify our approach, we choose to produce a mapping by reducing the connection matrix and by iteratively minimizing a compound cost that considers both load balancing and intercluster copies.

Although sub-optimal, we found that this strategy produces a good initial assignment provided an appropriate selection of the threshold ϕ_{th}. A smaller ϕ_{th} produces a larger number of partial components, but it also increases the computational requirements of the iterative phase. To select a reasonable value of ϕ_{th}, we repeat the initialization phase for a few decreasing values of ϕ_{th} and we choose the one that produces the shortest length according to our simplified scheduler.

6.3 Iterative Improvement

In the final stage, the assignment produced by the initialization phase is improved by an iterative descent algorithm that refines it by modifying the choice made for every element in the set Φ of partial components.

We investigated two different strategies:

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

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

As mentioned earlier, the descent is driven by the estimate on the schedule length obtained through a simplified model of a real scheduler. The model is based on a simple list instruction scheduler, which uses a queue of data-ready nodes ordered by their respective priority. The model quantitatively keeps track of target resources, allocated and de-allocated registers, copies, and register spill/restore pairs. The accuracy of the estimate is on average within 10% of the "real" scheduler, while the execution time is more than 10 times faster.

The schedule length L as a function of the clustering C presents many flat areas (also known as "plateaux"). Since we are using a local cost function, when the function gets stuck in a plateau we need an alternative criterion to drive the descent. After some experiments, we decided to use the number of copies as secondary heuristics to choose the descent direction in a plateau. It turns out that

such a number is more sensitive to small changes of the cluster assignment C and it is relatively easy to compute.

6.4 Effectiveness of the PCC Clustering Algorithm

To measure how PCC compares versus BUG (and in general more traditional "greedy"-style techniques), we can look at static performance measurements. Table 2 shows a set of static measurements that demonstrate how PCC effectively reduces the complexity of the problem. Section 7 reports dynamic measurements on the effect of PCC on the overall performance.

Benchmark	Number of loops	DAG Nodes in largest loop	PCC Components in largest loop	Average number of DAG nodes per loop	Average number of PCC components per loop
bmark	85	1836	32	158.3	11.0
copymark	29	1438	32	565.1	10.5
crypto	177	1515	64	99.9	9.4
dhry	5	88	24	51.0	11.2
mpeg2	72	1656	95	338.5	15.7
tjpeg	43	1673	64	176.1	10.5
Average	58.71	1172.29	44.43	198.41	9.76

Table 2. Characteristics of the benchmark loops for a 16-unit configuration where the unrolling limit was set to 1,600 nodes per loop. From the table we can see the effectiveness of the PCC algorithm in reducing the complexity of the problem through the identification of the *partial components*. In average we get a reduction of about a factor of 20 in complexity, that goes up to a factor of 30-50 for the largest loops.

From an analysis of Table 4 (previous section), we can say that in average, the PCC algorithm outperforms BUG by about 5% in the 2-cluster case and about 7% in the 4-cluster case. At the same time, the compile-time penalty due to computational increase grows but remains within an acceptable range, considering that we limit PCC to run only on the inner loop of the code. Finally, we have to say that PCC is still in its infancy and we believe that there is still some significant headroom to

improve the algorithm and further reduce the performance degradation due to clustering.

7 Experiments and Results

To gather interesting statistics, we decided to pick two of the most significant configurations (8 units and 16 units) with enough register to be able to achieve the best performance (128 registers for 8 units and 256 registers for 16 units), and run them for several combinations of clustering. The compiler heuristics were tuned so that we limited unrolling to about 800 nodes per loop for an 8-unit machine and about 1,600 nodes per loop for a 16-unit machine. Table 3 shows a summary of the characteristics of the tested machine configurations.

Units	Clustering	Registers per cluster	Register file ports per cluster
8 units	single cluster	128 registers	16 read + 8 write
	2 clusters, 4 units/cluster	64 registers / cluster	8 read + 4 write
	4 clusters, 2 units/cluster	32 registers / cluster	4 read + 2 write
16 units	single cluster	256 registers	32 read + 16 write
	2 clusters, 8 units/cluster	128 registers / cluster	16 read + 8 write
	4 clusters, 4 units/cluster	64 registers / cluster	8 read + 4 write

Table 3. Characteristics of the clustered configurations used for the experiments.

The goals of our experiments are dual.

1. Demonstrate that clustered architectures are an effective solution to build wide VLIW machines without putting too many burdens on register file ports and bypass logic.

2. Show that better clustering algorithms have a significant impact on performance and that there is still quite a bit of headroom to exploit in software to make clustering even more effective.

7.1 Effectiveness of Clustered Architectures

Figure 5 and Figure 6 show the results for the experiments. Table 4 summarizes the statistics averaging all the benchmarks. In Figure 6 we present the detailed performance measurements for the five individual benchmarks (except *dhry*, that was not interesting for our purposes due to lack of ILP). The bars in the figure represent performance relative to the baseline, where the single bars are for non-clustered configurations and the overlapped bars are for clustered machines with the two different scheduling algorithms (BUG and PCC). Figure 5 reports an average of the same values, together with the performance of single cluster machines at various widths.

From the graphs we can see that in average clustering is convenient but there are cases where it can hurt performance. For example for tjpeg, a 16-unit / 4-cluster machine performs definitely worse than an 8-unit / 1-cluster and slightly worse than an 8-unit / 2-cluster configuration. While cost considerations may advise against the single cluster solution, in this particular case the narrow 2-cluster machine is a clear win versus the wider 4-cluster one.

The first observation is that, relative to the non-clustered machine, we lose 10-20% performance when we go to a 2-cluster machine and 15-20% performance when we go to a 4-cluster machine. It is beyond the purpose of this paper to decide what is the best configuration, since this depends on the particular point in the technology space, and on the impact of the number of register file ports and bypassing logic on the overall cycle time.

The graphs show some clear winner and loser. According to the level of ILP and the type of computation, indiscriminate clustering is not always beneficial. For example, for benchmarks like *bmark* and *crypto* it is a clear win to build a 2-cluster, or even a 4-cluster, machine. On the other side, for *copymark, mpeg2* and *tjpeg* a 4-cluster 16-unit machine performs worst than a single-cluster 8-unit machine, and even worse than a 2-cluster 8-unit configuration for *tjpeg*. Obviously, a single-cluster 8-unit machine will have a significant impact on the cycle time, so it is unclear which is the best choice in this case.

In summary, we can conclude that a moderate amount of clustering is – in general – a good compromise:

- Performance degrades gracefully (about 15% for two clusters)
- We get benefit from the increased ILP

- The cycle penalty to build a single register file would very likely offset the clustering overhead.

However, when we increase the number of clusters, the advantage starts to be less clear and strongly depends on the amount of ILP available and on the nature of computation. Our numbers show that the performance degradation of four clusters is in the 25-30% range.

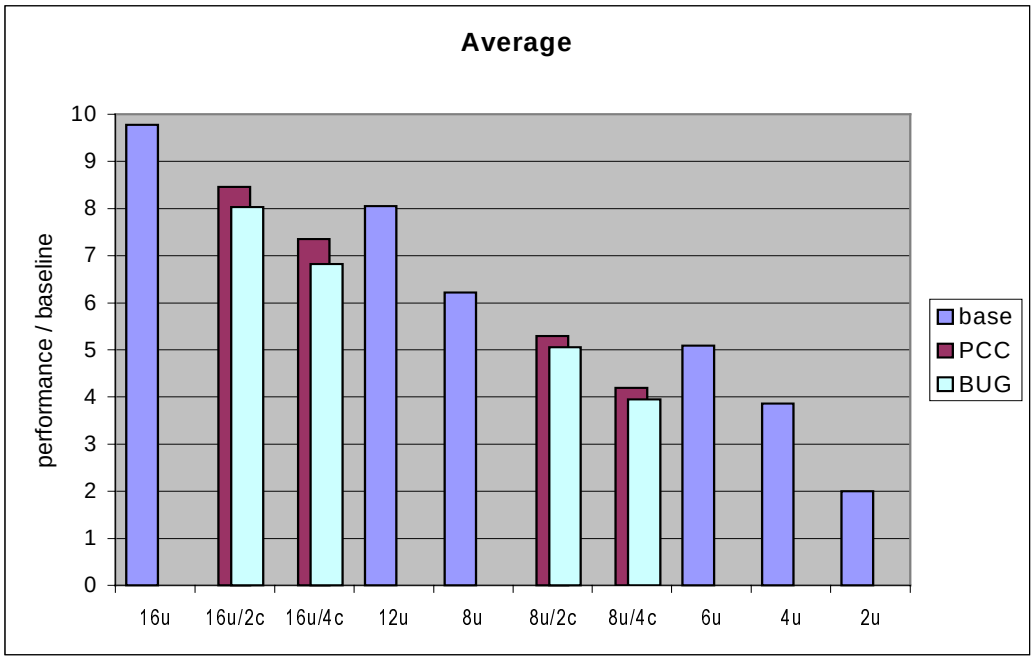


Figure 5. Average results for the five significant benchmarks. From the graph we can see that clustering costs you about one fourth of the machine functional units in terms of performance. For example a 16-unit / 2-cluster machine performs roughly like a 12-unit / 1-cluster and an 8-unit / 2-cluster like a 6-unit / 1-cluster.

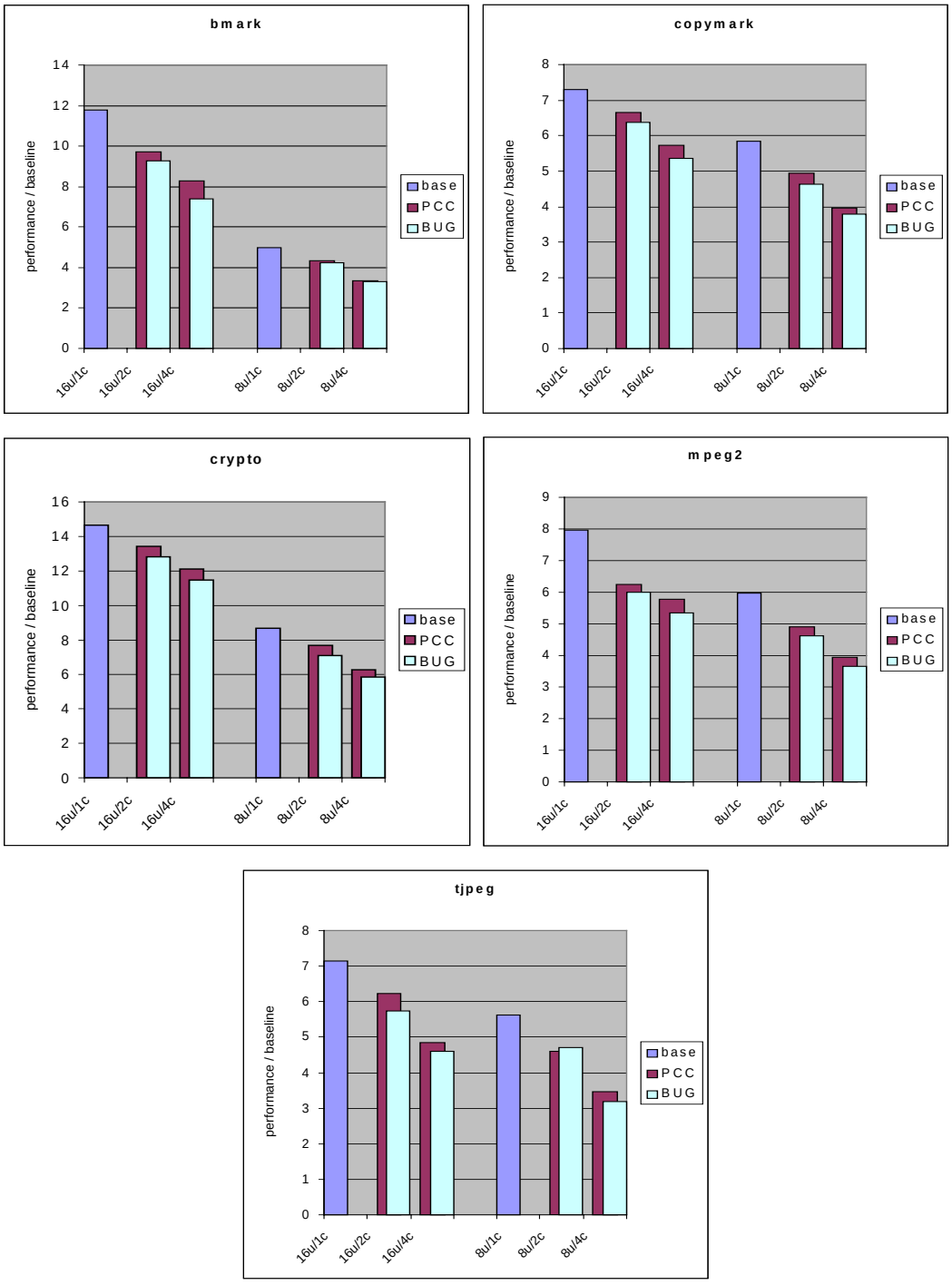


Figure 6. Detailed individual results for the five significant benchmarks

	2 clusters 16 units	2 clusters 8 units	2 clusters average
PCC	86.6%	85.2%	85.9%
BUG	82.2%	81.3%	81.8%

	4 clusters 16 units	4 clusters 8 units	4 clusters average
PCC	75.2%	67.5%	71.4%
BUG	69.9%	63.6%	66.7%

Table 4. Performance summary of the 2-cluster and 4-cluster configurations. The numbers are expressed as a percentage of the maximum performance achievable with a 1-cluster architecture. With the PCC algorithm, a 2-cluster machine loses about 14% and a 4-cluster machine about 29% against a 1-cluster version of the same resources. Our experiments show that this relation is valid regardless of the absolute number of units, in the considered range.

8 Conclusions

Table 4 suggests that as a rule of thumb, breaking the CPU into two clusters costs somewhere around 15-20% lost cycles; four clusters costs around 25-30%. Similar results have been seen before: [7] found slow- downs in virtually the same range, even though they had a superscalar design, and were using a compiler, processor and application suite with far less ILP. The other works we cited for VLIWs also had results in a similar state, though they offered a far less sophisticated compiling system then our, and didn't present industrial-strength codes. In all cases these numbers must be increased due to the longer latencies of a memory system built for the one cluster CPU, but we feel this is a small effect. An interesting open question is whether these numbers can be improved upon, or are generally at the theoretical limit for the code and clustering involved. Obviously, we cannot compute the optimal solution to answer this.

We found that the PCC algorithm performed quite well, getting noticeably better results than BUG, almost always.

Finally we note that as feature sizes of microprocessors decrease in relation to communication costs, clustering becomes more attractive. [7] points out that at hardware feature sizes of .35µ, the cycle time loss due to too many register ports might not be so bad, but at .18µ and beyond--the point VLSI technology is now reaching--the loss will be severe. The numbers presented here are likely to strongly favor the use of clustering, at least for CPUs which execute applications with large amounts of ILP.

References

[1] J. A. Fisher, "Very Long Instruction Word Architectures and the ELI-52," Proceedings of the 10th Annual International Symposium on Computer Architecture, 1983, pp. 140-150.

[2] Details from a presentation given by Jim Keller of Digital Equipment Corporation at the Microprocessor Forum on October 22-23, 1996, entitled "The 21264: A superscalar Alpha processor with out-of-order execution". A summary was written as: Linley Gwennap, "Digital 21264 Sets New Standard", Microprocessor Report, 10 (14) 11-16. Original available in March, 98 at: http://www.asia-pacific.digital.com/info/semiconductor/a264up1/index.html..

[3] A. Nicolau, N. Dutt, and A. Capitanio. "Partitioned Register Files for VLIWs: A Preliminary Analysis of Tradeoffs." Proceedings of the 25th Annual Workshop on Microprogramming (MICRO-25), pp. 292-300, November 1992.

[4] Sanjeev Banerjia, "Instruction scheduling and fetch mechanisms for clustered VLIW processors",PhD Thesis in the Department of Electrical and Computer Engineering, North Carolina State University, to be published, 1998.

[5] Saurabh Jang. "Generating Efficient Code of VLIW Architectures with Partitioned Register Files." MS thesis, Michigan Technological University, 1996.

[6] Johan Janssen and Henk Corporaal. "Partitioned Register Files for TTAs." Proceedings of the 28th Annual Workshop on Microprogramming (MICRO-28), pp. 303-312, January 1996.

[7] Keith I. Farkas, Paul Chow, Norman P. Jouppi, and Zvonko Vranesic, "The Multicluster Architecture: Reducing Cycle Time Through Partitioning." In the proceedings of The 30th Annual IEEE/ACM Symposium on Microarchitecture, December 1997.

[8] James E. Smith, “Decoupled access/execute computer architecture.” Proceedings of the 9th Annual International Symposium on Computer Architecture, 1982, pp. 112-119.

[9] Doug Hunt, “Advanced Performance Features of the 64-bit PA-8000,” COMPCON 1995 Digest of Papers, March 1995.

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

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

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

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

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

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

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

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

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

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

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

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

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

[23] B. Kernighan and S. Lin An efficient heuristic procedure for partitioning graphs. The Bell Syst. Tech. Jl 49, pp 291-307 1970

[24] H. Pirkul and E. Rolland, "New heuristic solution procedures for the uniform graph partitioning problem: Extensions and Evaluations." Computers and Operations Research, oct 1994

[25] E. Rolland, "Abstract heuristic search methods for graph partitioning," PhD dissertations, The Ohio State Univ., Columbus, 1991

[26] J. Oommen and E. de St. Croix, "Graph Paritioning Using Learning Automata," IEEE Trans. on Comp. Vol. 45, No. 2 Feb 1996 pp. 195-208

[27] A. Lim and Y. Chee, "Graph Partitioning Using Tabu Search", 1991 IEEE International Symposium on Circuits and Systems, pp. 1164-1167