# Use of Fine and Medium Grain Parallelism on Horizontal Architectures*

Geraldo Lino de Campos
Escola Politécnica da
Universidade de São Paulo
P. O. Box 8174
São Paulo 01051 - BRASIL
Tel (55)(11)815-9322 ext 3288
e-mail: RTC@FPSP.FAPESP.BR

**Abstract**

Horizontal architectures can control several functional units independently, and usually have a single flow of control. As such, they exploit only the finer parallelism, and a large number of functional units is useless when the code contains control or data dependencies. The performance in these cases can be increased, with the same total hardware, if the functional units are divided in groups, each with a private register file and with an independent flow of control.

The main conclusion is that this will neither impair the performance when many functional units are required, nor affect the cycle time adversely. Very important side effects are the reduction on the number of ports in the central register file, with reduction of the mean instruction size as well. Results are presented for a processor, now under design, showing that improvements in the range of 1.6 can be achieved in the geometric mean of performance when executing the Lawrence Livermore Kernels.

* This research was partly supported by CNPQ grant 105178/92-6.

## 1. Introduction

The ever increasing demand for high-speed processors forces the introduction of more sophisticated instruction pipelines and increasing degrees of parallelism on the architectures, from microprocessors to supercomputers. The clock time of the pipelines is in most part controlled by the available technology; the architect can only reduce the complexity of the control cycle, aiming at a small reduction in cycle time. On the other hand, only the nature of the programs limit the usage of parallelism to increase performance.

In this paper, fine grain parallelism is defined as the parallelism that can be achieved within the same thread of control, and medium grain parallelism is the parallelism obtained by the replicated execution of the same code block. Coarse grain is a name reserved for parallelism between different blocks of code, and it will not be considered further here. It is much simpler to detect, use and control the medium grain parallelism then the coarse grain.

The purpose of this paper is to study the limits of practical usage of parallelism on horizontal architectures. Horizontal architectures are best represented by VLIW (Very Long Instruction Word) [Fis83], and its derivatives, as APA (Asynchronous Polycyclic Architecture) [Cam92].

It is shown that it is possible to obtain higher performance, within the same hardware restrictions, if the available resources are divided for use with fine and medium grain parallelisms, as it is done in APA machines.

These results have a broader significance, since [Jou89] showed that VLIW, superscalar and superpipelined machines are roughly equivalent in terms of performance and exploitation of parallelism; we believe the results presented here are valid for the superscalar and superpipelined machines as well.

This study has been conducted with the intent of dimensioning the number and type of functional units for the implementation of an APA processor oriented for numerically intensive problems, and the examples are taken from this domain.

Limits on the available parallelism have been studied in several papers, with widely varying results, as shown by the following table, based partially on [Smi89]:

| Study | Year | Speedup |
|---|---|---|
| Weiss/Smith | 1984 | 1.58 |
| Tjaden/Flynn | 1970 | 1.8 |
| Sohi | 1987 | 1.8 |
| Acosta et al | 1986 | 2.7 |
| Kuck et al | 1972 | 8 |
| Reisman/Foster | 1972 | 51 |
| Nicolau/Fisher | 1984 | 90 |
| Smith et al | 1989 | ~2 |
| Jouppi/Wall | 1989 | ~2 to ~5 |

There are many reasons for these apparently discrepant results, because the underlying hypothesis of the various studies are very different. It is always possible to obtain any desired degree of parallelism; as an example, the following code has at least N as degree of parallelism:

```
      DO 1 I = 1, N
1     X(I) = A(I)
```

Obviously, one cannot obtain a speedup of N by replicating a ALU N times in an horizontal architecture, for large values of N, due to memory access limitations, both in the variables and in the large instruction that will result, even ignoring the question of feasibility of such a machine. The important issue is that it is easy to obtain unrealistic measures on the degree of parallelism.

What really matters is the achievable degree of parallelism in realistic applications, and under reasonable assumptions about

the underlying hardware. For shared memory architectures, it is unrealistic to expect more than a few accesses to memory in each cycle, and this factor must be taken into account to have meaningful results. The effect of latency is of paramount importance, since processors are generally superpipelined, in the sense defined in [Jou89].

As a consequence, it is proposed not to measure the degree of parallelism present in the code, but the increase in performance that can be obtained by addition of more units of each kind to a configuration under study. When the resulting performance increase is small, it is possible to conclude that the useful degree of parallelism has been reached. This allows all the restrictions imposed by the configuration to be taken into account.

This paper is organized in 4 sections. The following section presents the main features of the APA, and typical programming techniques to exploit them. In section 3, the methodology used to conduct the available parallelism study is introduced, and the results are presented. Section 4 offers a few concluding remarks.

## 2. Asynchronous polycyclic architecture

The asynchronous polycyclic architecture resulted from a critical analysis of the characteristics of the VLIW architecture.

A VLIW processor is conceptually characterized by [Fis83]:

1 - a single thread of execution;

2 - a large number of data paths and functional units, with control planned at compile time;

3 - instructions providing enough bits to control the action of every functional unit directly and independently in each cycle;

4 - operations that require a small and predictable number of cycles to execute;

5 - Each operation can be pipeline, i. e., each functional unit can initiate a new operation in each cycle.

In terms of hardware, the ideal VLIW processor should have many functional units connected to a large central register file. Each functional unit would ideally have several read ports (two for arithmetic units) and a write port to the register file. Also the register file would have enough bandwidth to allow any combination of accesses generated by the functional units.

Unfortunately, the hardware described above is unrealistic. Every implementation adopted several ad hoc solutions to make it implementable.

On examining the conceptual characterization, it becomes clear that the rationale is to have a large degree of parallelism and a simple, and therefore fast, control cycle. Closer scrutiny shows two relevant aspects:

First, condition 1 does not contribute to any of these objectives. It is possible, at least at the conceptual level, to have a situation where every functional unit has its own cache and control logic, and therefore be capable of executing its own thread of control. This would add some circuitry, but it would not add any delays to the processor cycle.

Second, condition 4 is unfeasible for the functional units in charge of memory accesses, at least for high performance processors, due to unpredictable memory bank conflicts. This can be circumvented by considering that the processor remains synchronous in virtual time, stalling if data is not available when expected to be, but this may have a substantially adverse effect on performance.

The first new APA feature solves the memory access problem by decoupling the process of memory access. Two kinds of functional units are used: the first, called address unit, generates and sends the required addresses to the memory subsystem; the second, called data reference unit, is responsible for reordering data words coming from memory and to send them to other functional units.

Address units may operate in two modes: single address and multiple addresses. In the single address mode, its role is only to receive an address calculated by an arithmetic unit and send

it to the memory subsystem; in the multiple addresses mode, its role is to autonomously generate the values of a set of arithmetic progressions, until a specified number of elements are generated; this mode is used for references to (a set of) arrays. Once started, the address unit can proceed asynchronously with the main flow of control: it generates as many addresses as the memory subsystem can accept, waits if the memory subsystem cannot accept new requests, resumes execution when this condition is withdrawn and stops when all addresses have been generated.

The consequences are the following: the memory subsystem operates at full capacity and the main flow of control is not disturbed by saturation of the memory subsystem. As for hardware, fewer bits are required in the instruction, since the units are controlled by their own instructions, and also less ports are required in the central register file, since the address processors can use a local copy of the required registers.

By extending these ideas, it is possible to conceive an architecture where subsets of functional units can be split from the main flow of control and start operating with their own flow of control, until the execution of a stop instruction. Then, the respective functional units will be returned to the main control flow.

Experience in programming such a machine shows that it is unduly complicated. Very few situations require odd divisions of the functional units. A hierarchical system of two levels is adequate for almost all situations: the functional units can be divided in *groups*, composed of a certain number of arithmetic units, each capable of forking the operation of the address units.

Experience shows also that the communication between groups is seldom done. An important consequence is that there is no need to share a central register file between groups; each group can use its own, provided it is able to send values to the other groups.

The consequences again are the reduction in the number of bits required in the instruction, since each group is controlled by their own instructions, and fewer ports are required

in the central register file, since each group uses a local set of registers.

Efficient usage of those characteristics requires two other features: eager execution and delayed interrupts. To keep the functional units busy, values should be calculated as soon as possible, regardless the possibility of the control flow rendering them unnecessary. A consequence is that abnormal conditions (like division by zero, references to invalid addresses, etc) can arise. To postpone the resulting interrupts, a result descriptor is appended to every value in the central register file. All operations performed with this value will have the original interrupt as its result descriptor. The interrupt occurs only when the value is effectively used - stored in memory or used for a branch decision.

This result descriptor also contains the condition code associated to the generation of the value, and thus can be used as a predicate to control the conditional execution of instructions.

Efficient execution of loops is obtained by use of a variant of polycyclic support; since its basics are described elsewhere [Den89], it will not be described here. The hardware support includes automatic prolog, epilog and predicate-controlled execution of instructions.

The result of these characteristics is a processor with higher performance as compared to an equivalent VLIW, implemented with register files with few ports, and with dynamic word size: each group is small, and at any moment only the active ones must fetch instructions. For instance, the machine resulting from the dimensioning studied in the next section uses register files with only two write ports, and group instructions of only 64 bits.

Furthermore, this permits the exploitation of the medium grain parallelism, by splitting loops where control or data dependencies force the use of few functional units per thread.

Medium grain parallelism is useful when executing loops. In the following, all the code contained in an inner loop will be called a *block*; it is not necessarily a basic block.

Every block can be characterized in one of the following categories:

1. Blocks without data or control dependencies

These blocks can be executed entirely by one group, or, if the relation of the number of operations to the number of variables used is high, the loop can be divided among a sufficient number of groups to allow the use of all bandwidth to memory and all functional units. This category also includes a few particular cases of easily resolved cases of data dependency, like the sum of products.

As an example, the loop

```
    DO 1 I = 1, N
1   S = S + A(I) * B(I)
```

can be split to

```
    DO 1 I = 1, N, 2            DO 1 I = 2, N, 2
1   S = S + A(I)*B(I)       1   S = S + A(I)*B(I)
```

each executed in a different group, followed by the addition of the values of S. (Since each group has its own set of registers, there is no renaming problem with S and I)

2. Blocks with control dependencies

Blocks with control dependencies have their performance limited by latency of the instructions that establish conditions and by the branch instructions. This can be improved in two ways. To a limited extent, it can be circumvented by the predicated-controlled execution. A more general solution is to divide the

loop among a sufficiently large number of groups, until the full memory bandwidth is achieved.

3. Blocks with data dependencies

It is generally difficult or impossible to parallelize blocks with data dependencies. If the data dependency is immediate and the block is short, a technique that might be used is to increase the data dependency length and execute the resulting blocks on several groups. This technique is generally limited to an expansion from a dependency of one to two dependencies of two.

As an example, the following loop

```
    DO 11 K = 2, N
11  X(K) = X(K-1) + Y(K)
```

can be split to

```
                                  X(2) = X(1) + Y(1)
    DO 11 K = 3, N, 2             DO 11 K = 4, N, 2
11  X(K)=X(K-2)+Y(K-1)+Y(K)   11  X(K)=X(K-2)+Y(K-1)+Y(K)
```

## 3. Methodology and results

This section deals with the determination of the available fine and medium grain parallelism on numerically intensive applications.

The 24 Lawrence Livermore Kernels [Mah86] were used; kernels 14 and 22 were dropped so as to avoid precision considerations with the functions involved .

To be realistic, a preliminary study of the memory subsystem characteristics was conducted, with the conclusion that the number of read accesses per cycle should be twice of the write accesses. Code was supposed to be already in cache.

Table 1 - Performance with fine grain parallelism

| | Assintotic performance in megaflops | | | | | |
|---|---|---|---|---|---|---|
| Arithmetic Units | Minimum | Harmonic Mean | Geometric Mean | Arithmetic Mean | Maximum | Variation of the geometric mean from the previous value (in %) |
| 1 | 16 | 42 | 48 | 49 | 97 | |
| 2 | 20 | 61 | 84 | 98 | 187 | 75 |
| 3 | 20 | 63 | 93 | 117 | 276 | 11 |
| 4 | 20 | 64 | 98 | 127 | 355 | 5 |
| 5 | 20 | 66 | 104 | 139 | 401 | 6 |
| 6 | 20 | 67 | 115 | 171 | 528 | 6 |
| 7 | 20 | 68 | 116 | 174 | 528 | < 1 |
| 8 | 20 | 68 | 117 | 182 | 673 | < 1 |
| ... | | | | | | |
| 16 | 20 | 68 | 118 | 182 | 673 | < 1 (relative to 8) |

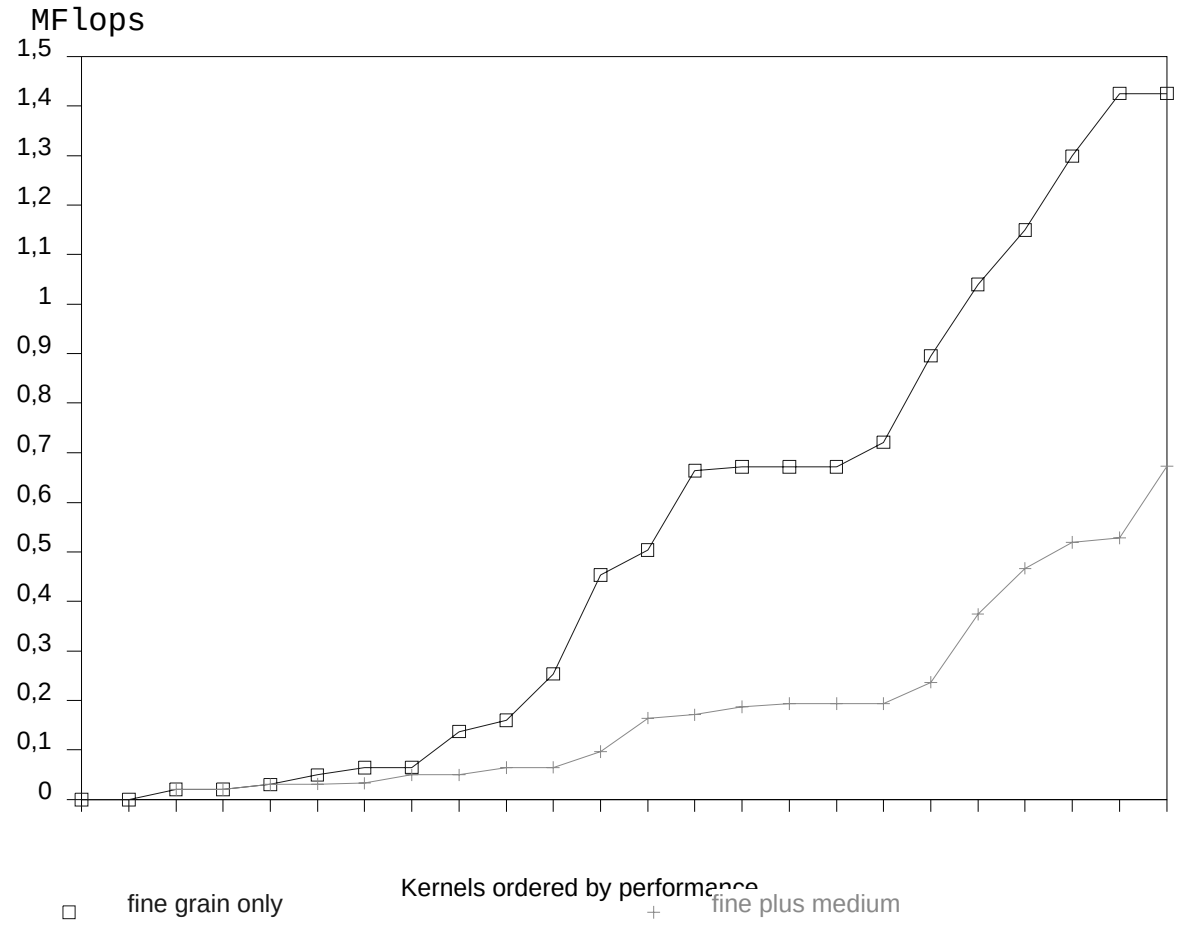
The kernels were compiled by hand, with care to use only optimizations that can be expected from a good compiler. The effect of memory conflicts was taken into account.

The hardware characteristics assumed were a cycle time of 10 ns, all operations pipelined with a latency of 5 cycles, except for divisions and square roots, with 25 and 45 cycles, unpipelined.

In a first stage the study was conducted for architectures with one group, varying from 1 to 8 arithmetic units, and then from 2 to 4 groups with 2 arithmetic units each. The memory subsystem is supposed to have a total of 3 accesses per cycle up to 4 arithmetic units, and 6 thereafter.

Tables 1 and 2 show the results. Table 1 makes clear that there is a significant increase in performance from one to two arithmetic units, and small advantages thereafter; this was the basis for selecting groups with two arithmetic units. These

Graphic 1 - Performance with 16 arithmetic units




configurations exploit only fine grain parallelism, although they have all the other APA features described above.

Table 2 - Performance with fine and medium grain parallelism

| | Assintotic performance in MFlops | | | | | |
|---|---|---|---|---|---|---|
| Number of Groups with 2 Arithmetic Units | Minimum | Harmonic Mean | Geometric Mean | Arithmetic Mean | Maximum | Increase of the geometric mean from the fine grain equivalent configuration (in %) |
| 2 | 20 | 74 | 113 | 142 | 356 | 35 |
| 4 | 20 | 90 | 179 | 265 | 713 | 53 |
| 8 | 20 | 100 | 288 | 516 | 1426 | 144 |

Table 2 shows the same results for configurations with 2, 4 and 8 groups of 2 arithmetic units. These configurations exploit both fine and medium grain parallelism, and present an improvement of up to 144% when compared with the values in table 1 for a configuration with the same hardware resources. As should be expected, the improvement increases as hardware resources increase. This table shows also the strong influence of the number of simultaneous accesses to memory.

Graphic 1 compares the individual behavior of the kernels for the situation with 16 arithmetic units, using only fine grain and fine and medium grain parallelism. In this graph, the kernels are ordered by increasing performance.

It is important to note that the more significant increase in performance occurs at the kernels with medium performance. Since where most practical applications are in this range, the improvement occurs where it is most useful.

## 4. Conclusions

In this paper the concepts of using both fine and medium grain parallelism and of evaluating the usable parallelism by selectively adding functional units are introduced.

The asynchronous polycyclic architecture (APA), an architecture developed to exploit the medium grain parallelism, is briefly outlined.

Results of the analysis and simulation of an APA processor are presented, showing a performance increase of 144% in the geometrical mean of the full Lawrence Livermore Kernels when medium grain parallelism is added to the basic horizontal architecture.

The main conclusion is that there is no reduction in performance when the functional units of a VLIW processor are split into groups with a private register file and autonomous control. On the contrary, this splitting allows the exploitation of medium grain parallelism, resulting in an increase of performance. This splitting also allows hardware implementations with register files of only two write ports, and the use of dynamical instruction sizing: each group has its own instruction

stream, which is used only when required by the available parallelism.

The inclusion of the features needed to support medium grain parallelism not only increased the performance but resulted in a more easily implementable hardware and in shorter instructions.

## Bibliography


[Cam92] Campos, G. L. "Asynchronous Polycyclic Architecture." Parallel Processing: CONPAR 92-VAPP V (Lecture Notes in Computer Science, vol 634), Springer-Verlag, setembro de 1992

[Den89] Dehnert, J. C., Hsu, P. Y.T., Bratt, J. P., "Overlapped Loop Support in the Cydra 5", 3rd Int. Conf. on Architectural Support for Programming Languages and Operating Systems, 26-38, April 1989

[Fis83] Fisher, J. A. "Very Long Instruction Word Architectures and the ELI-512", IEEE Conf. Proc. of the 10th Annual Int. Symp. on Comput. Architecture, 140-150, June 1983.

[Jou89] Jouppi, N. P. and Wall, D. W., "Available Instruction-level Parallelism for Superscalar and Superpipelined Machines", 3rd Int. Conference on Architectural Support for Programming Languages and Operating Systems, 272-282, April 1989.

[Mah86] McMahon, F. H. "The Livermore Fortran Kernels: A Computer Test of the Numerical Performance Range," Lawrence Livermore Nat’l Laboratory Report No. UCRL-53745, Livermore, CA, Dec. 1986.

[Smi89] Smith, M. D., Johnson, M. and Horowitz, M. A. "Limits on Multiple Instruction Issue", 3rd Int. Conference on Architectural Support for Programming Languages and Operating Systems, 290-302, April 1989.