Asynchronous Polycyclic Architecture*

Geraldo Lino de Campos

Escola Politécnica da Universidade de São Paulo
São Paulo 05508, Brazil
e-mail: RTC@BRFAPESP.BITNET

Abstract The Asynchronous Polycyclic Architecture (APA) is a new processor design for numerically intensive applications. APA resembles the VLIW architecture, in that it provides independent control and concurrent operation of low-level functional units within the processor. The main innovations of APA are the provision for multiple threads of control within each processor, the clustering of functional units into groups of functional units that show very weak coupling with each other, decoupled access/execute and eager execution. A supercomputer implementing this architecture is currently being designed, using commercially available parts.

1 – Introduction and motivation

Development of the Asynchronous Polycyclic Architecture (APA) concept was spurred by the needs of Project Omicron, an academic research effort. The project's main goal (expected to be achieved sometime in 1994) is to design and build a supercomputer for numerical applications, with a real-world performance in the same range of the then current supercomputers. APA processors are expected to provide better sustained performance in real-world problems than standard vector processors with the same peak capacity.

Since the constraints of a typical University research budget excluded expensive solutions like custom ECL circuits or exotic packaging and cooling, we were forced to develop a novel architecture that is better matched to the needs of real-world computations than the standard vector processor design, but is still realizable by using commercially available components.

We began our design effort with a critical evaluation of existing architectures. Current vector processors are machines architecturally designed in the early seventies. Since then much effort has been put in research in computer architectures, and many important results had been achieved, but had no influence on supercomputer architectures. A basic assumption was that there should be ways to use these results in high-performance machines, if there is no need of compatibility with older architectures.

* This research was partly supported by the CNPQ grant 501971/91-2.

Such an evaluation required a performance yardstick. Given our emphasis on sustained speed in real-world problems, we decided to compare the alternatives by their average performance on some representative benchmark, rather than by their theoretical peak performance under ideal conditions. As pointed out in [8], peak performance adds only to the price of a machine.

Since numerical algorithms typically spend most of their time in the innermost loops, we chose a standard collection of simple loops, the Lawrence Livermore Kernels (LLK) [9] as our benchmark. The performance of many supercomputers in these loops is well known, and the loops are simple enough to be compiled and simulated by hand for all the alternative designs that we had to consider. It is instructive to consider that the real speed observed is a very small fraction of the maximum speed.

This evidence strongly suggests that the traditional vector processor architecture is somewhat ill-adapted to the very tasks for which it was designed. The explanation for this paradox is quite simple: the vector architecture was designed with individual operations in mind, while the real problem is to process entire inner loops, not individual operations. Loops contain recurrences and conditionals, situations that are not considered in the design of vector architectures.

In vector architectures, the main source of inefficiency is the lack or inadequacy of support for operation involving recurrences or conditionals. Following up on the RISC analogy, it seems reasonable to assume that, by breaking up the special instructions of standard vector processors into simpler ones, one would obtain substantially better overall performance from the same amount of hardware. What is needed is an architecture that can offer reasonable performance on general loops, containing recurrences and conditional tests, instead of one that concentrates on optimizing only a few restricted vector forms.

Our proposed Asynchronous Polycyclic Architecture implements this principle by combining the basics of the VLIW architecture [6] with the decoupled access/execute concept [13], and with extensions for loop execution, for conditional execution of instructions, for fetching large amounts of data and for an autonomous operation of groups of functional units, and eager execution for hiding memory latency (in *eager* execution, a instruction is executed as soon as their operands are available and there is a free unit to do the operation, even when it is not sure if the control flow will warrant the need for the instruction; in *lazy* execution an instruction is executed only when reached by the control flow).

Sections 2 and 3 describe the generic APA concept in more detail, and provide arguments in support of these claims. Section 4 briefly outlines an interconnect network specially designed for connecting the above processing units to the memory subsystem. Section 5 describes one specific instance of the architecture, the preliminary design of the Omicron supercomputer. Section 6 offers some concluding remarks

2 – Description of the APA

The Asynchronous Polycyclic Architecture resulted from a critical analysis of the characteristics of the VLIW architecture. The detailed evolution leading to the APA can be found in [2]; it will be only summarized here.

A VLIW processor [6] is conceptually characterized by a single thread of execution, a large number of data paths and functional units, with control planned at compile time, instructions providing enough bits to control the action of every functional unit directly and independently in each cycle, operations that require a small and predictable number of cycles to execute, and each operation can be pipelined, i. e., each functional unit can initiate a new operation in each cycle.

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 of the conditions above shows that they are sufficient for the goal, but most are not necessary, at least in the length stated.

The evolution took place in 4 steps:

First, the access/execute concept [13] was introduced. The motivation was practical considerations on memory access.

The ideal memory subsystem for any supercomputer should have large capacity, very low access times, and very high bandwidth, at a low cost. Real memories must compromise some of these goals. If a large capacity is required, cost and size limitations almost force the usage of relatively slow dynamic memories. Besides, numerically intensive applications require a very high memory bandwidth.

General purpose architectures rely almost invariably on caches to speed up execution, exploiting the locality of memory references. Although this is the case with general computing loads, this assumption can be wildly wrong with numerically intensive programs, since dealing with large arrays is incompatible with any realistically sized cache. This conclusion is reported for quite different machines [1, 5, 12].

The way to go is to have extensive interleaving in the memory; the latency may be high, but as long as it is possible to maintain a large number of outstanding requests, it will be possible to obtain operands at the necessary rate.

It is therefore unfeasible to have a predictable access time for the functional units in charge of memory accesses, due to the static unpredictability of 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, 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 the *address unit*, generates and sends the required addresses to the memory subsystem; the second, called the *data reference unit*, is responsible for reordering data words coming from memory and upon request sending them to the other functional units. This is an asynchronous counterpart to the decoupled access/execution concept introduced by [13].

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

As a consequence, 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 fewer ports are required in the central register file, since the address processors can use a local copy of the required registers.

The second step was extending this concept of asynchronous operations to the other functional units, each with its own flow of control.

Experience in programing such a machine shows that it is unduly complicated. Very few situations require this full splitting of the functional units. A hierarchical system is adequate for almost all situations: the functional units can be divided into groups, composed of a certain number of arithmetic units, each capable of forking the operation of address units.

The third step resulted from the fact that experience in programming also shows that the communication between groups is infrequent. An important consequence is that there is no need to share a central register file among groups; each group can use its own, provided it is able to send values to the other groups. This communication is done by a private bus connecting the functional units together.

Three additional features are required for an efficient usage of above characteristics: eager execution, delayed interrupts and a tagged memory system. To keep the functional units busy, values should be calculated as soon as possible, regardless of 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, every word is composed of a tag and a value fields, both in the central register file and in memory. When an operation encounters an error condition, a number associated with this error condition is placed in the tag, and the address of the offending instruction is placed in the value field of the result word. All operations performed with a word containing a non-zero tag will have the original word as its result, thus preserving the error nature and the address of the offending instruction. If both operands have non-zero tags, one is arbitrarily chosen for the result.

Real interrupts occurs only when the word's value is effectively used. A value is effectively used when the computation can not proceed without the use of value. This may be a somewhat elusive concept, and a precise characterization is beyond the scope of this paper. In a simple approach, a value is effectively used when used as a final result, in output operations; in a more restrictive context, when used in an unavoidable operation, as determined by the execution flow graph.

The hardware must have special instructions that generate an interrupt when a value is effectively used. It is up for the compilers do determine when this is the case.

Efficient execution of loops is obtained by use of a variant of polycyclic support; since its basics are described elsewhere [4, 12], it will not be described in detail here. The main difference from the implementation used in the Cydra 5, described in the above references, is that there are no explicit predicates; predicates are implied in the "age" of the iteration. This allows the hardware to support automatic prolog and epilog generation. The hardware also supports predicate-controlled execution of instructions.

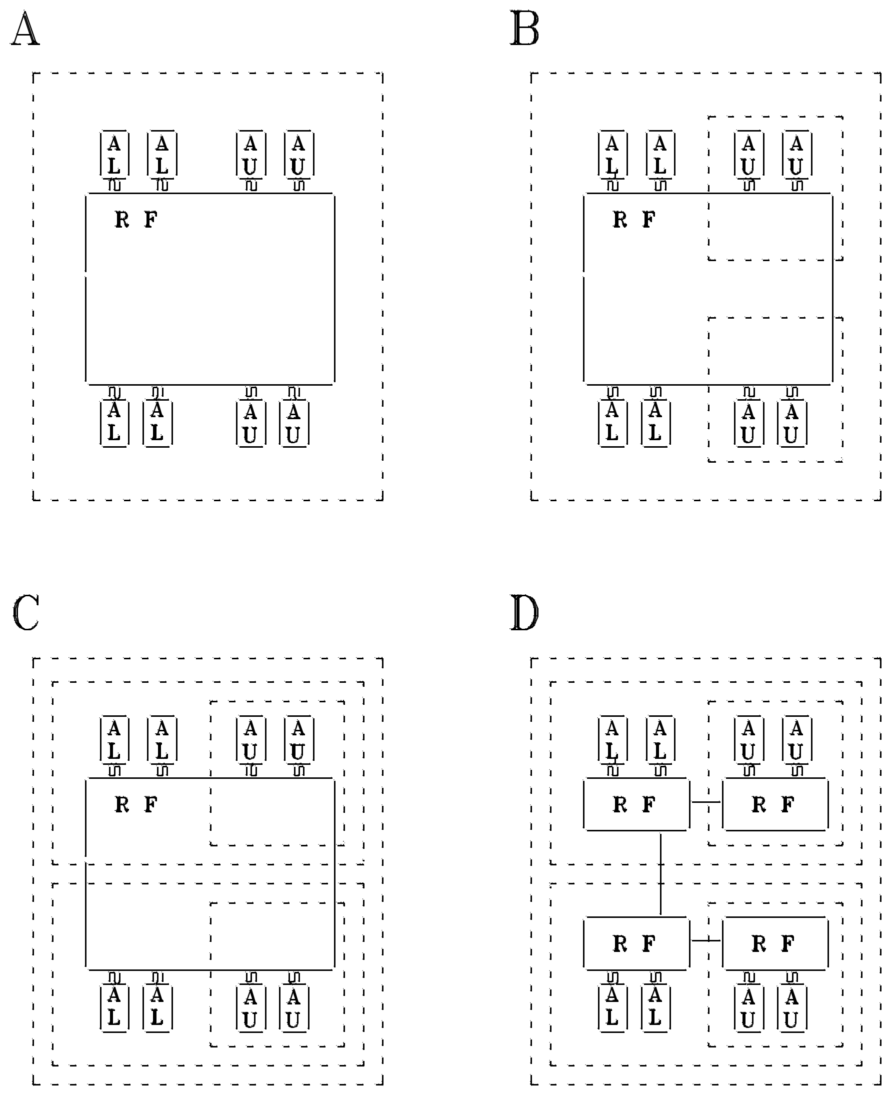

Figure 1 - Evolution from VLIW to APA architecture

In this drawing, dotted lines represent flow of control, AL stands for Arithmetic Logic Unit, and AU for Address unit.

In A, the traditional VLIW architecture. In B, evolution to the access/execute architecture. In C, the introduction of groups. Experimental analysis of programs for this architecture shows that each autonomous set may have its own copy of the register file, as shown in D. Data is exchanged via an internal bus. Other essential features for making this possible are not shown.

The result of these characteristics is a processor with higher performance as compared to an equivalent VLIW, implemented with register files with fewer ports, and with dynamic instruction 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 instructions of only 80 bits. Figure 1 depicts the evolution from the VLIW to the APA architecture.

In short, the architecture of one APA group is characterized by:

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

2 - functional units which are divided into groups; each group has its own register file and its own flow of control;

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 (in virtual time) of cycles to execute;

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

6 - memory accesses are decoupled;

7 - hardware support for loop execution;

8 - eager executions and delayed interrupts.

An APA *processor* is composed of a set of groups sharing a common memory.

3 – Exploitation of parallelism

As defined above, a group is a unit of execution. It can be considered as a dual of the pipe in vector machines. In the same way a vector processor can have several pipes, an APA processor may have several groups.

These several groups share a common view of memory, and must be interconnected to a common memory subsystem. Here again, the problem is not one of latency, but one of bandwidth. Section 4 describes a interconnection network, called the *Omicron Network*, specially designed for this application.

With a processor having several groups, fine-grain parallelism is exploited within groups; medium-grain parallelism, if present in loops, is exploited within a processor. Coarse-grain parallelism can be exploited by several processors; in this regard, the APA is not different from other architectures.

Exploitation of the medium-grain parallelism, which is a distinctive characteristic of the APA, is done in blocks contained in innermost loops. We call *inner block* all the code contained inside an inner loop; it is not necessarily a basic block, since it may contain conditionals; however, if it contains procedure calls, the inner loop condition allows only the call of procedures that do not contain loops.

The following considerations are based on simple compiler technology; the use of program transformations and other techniques already developed for vector machines will lead to considerable increase in performance.

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

3.1– Blocks without data or control dependencies

These blocks 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.

It should be noted that since each group has its own set of registers, there is no renaming problem with S and I; each value will be kept in a register with the same number, but on a different register file.

3.2– 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, each processing a fraction of the iterations.

3.3– Blocks with data dependencies

It is generally difficult or impossible to parallelize blocks with data dependencies in a straightforward manner. 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)
```

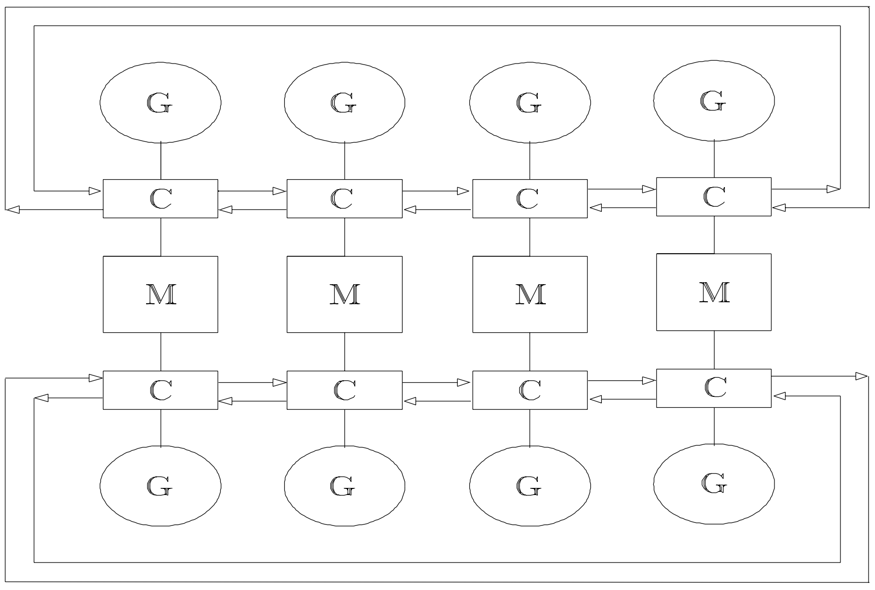

Figure 2 - An 8 x 8 omicron network

G are groups, M are memory modules and C are Omicron switches. Requests from the groups are routed to the memory modules or to the adjoining switches, depending on the address requested.

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)
```

The net result is that the APA not only allows the "vectorization" of a larger class of loops then the traditional vector architecture, but allows the paralleli-zation of a large class of loops.

4 _ The Omicron Interconnect Network

A detailed description and evaluation results of the omicron intercon-nect network is beyond the scope of this paper; the following presentation intends to give a few general ideas to clarify the configuration of an APA processor.

To be used in the manner discussed above, independent groups must have a common view of memory. They will be able to operate efficiently only if there is an efficient way to interconnect memory modules (eventually composed of a fairly large number of memory sub-modules) to the groups.

Fortunately, this problem is alleviated by the fact that, due to memory access decoupling, the performance is sensitive to memory bandwidth and not to the access time. This allows the design of a specific interconnect network, called the Omicron network. This network is composed of switches, connected to the groups, to the memory modules and to the adjoining switches. Fig. 2 shows an omicron network, interconnecting 8 groups to 4 dual-ported memory modules.

Each omicron switch is composed of switches that can route a request originated from a group to the adjoining memory module or to neighboring switches, and, in the case of read requests, route back the datum. The innovation is to use a queue in the inputs, to hold requests until there is a free path to the next destination of the request, until the final destination is reached. Although this may add considerably to latency, the obtainable bandwidth is very high, except when the demand rate approaches unity (one request per group per cycle).

The above consideration holds for Omicron networks of up to 8 switches per ring per memory port. Systems with more elements can be constructed by serial connection of switches. An important result is that the first level reduces the request rate to the effective capacity of the switches, and hence subsequent levels do not affect the bandwidth to a significant degree.

5 – Dimensioning an APA processor

This section describes one specific instance of the architecture, the preliminary design of Project Omicron supercomputer.

5.1– General aspects

As explained above, reasonably sized data caches are usually useless for processing involving large vectors. Elimination of data caches also avoids cache coherence problems among different groups. Caches are provided for code, however, since the locality of code references is independent of the nature of data accesses. Each group has its own code cache.

To simplify design and implementation, both in hardware and in software, all IO will be implemented by a separate IO processor. A standard RISC server will be used.

The processor will be built with ECL technology, using BIT B3130 floating point processors and Motorola ECLinPS series for logic; the main memory, however, will use standard MOS memories. The target cycle time is 10 ns; this is the value assumed in the foregoing simulations.

A basic word length of 64 bits will be used. This word can also be regarded as two 32 bits words, each with an 8 bit tag, resulting in an 80 bits physical word.

The arithmetic units will be pipelined and capable of integer and floating point operations. The latency of all operations is of 5 cycles, except for divisions and square roots, with 25 and 45 cycles; these last operations are not pipelined.

The structure will be modular, allowing systems with 2, 4, 8 and 16 groups. Each pair of groups will be connected to one memory module. The processor with 16 groups corresponds to the drawing of fig 3.

Simulation studies shown that each memory module should have 32 interleaved sub-modules to allow for two simultaneous reads and two simultaneous writes, limits imposed by technical restrictions on size and density of the backplane.

5.2– Dimensioning the group

The methodology adopted for dimensioning the number of arithmetic units per group and the number of groups was to evaluate the performance when executing the Lawrence Livermore Kernels.

The 24 Lawrence Livermore Kernels [9] were used; kernels 14 and 22 were dropped so as to avoid precision considerations with the functions involved.

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. Asymptotic performance means that the inicialization code was neglected; it usually amounts to less the 10 instructions.

In a first stage the study was conducted for architectures with one group, varying from 1 to 16 arithmetic units; table 1 shows the result, with emphasis on the increment of the geometric mean from one to the next configuration.

Table 1 - Performance of one group

Number of arith-metic units	Asymptotic Performance in MFlops					Increment of the geometric mean over previous value (%)
	Minimum	Harmonic mean	Geometric mean	Arithmetic mean	Maximum	
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	117	182	673	< 1*

* relative to 8 arithmetic units

Table 2 - Performance of several groups

Number of groups	Asymptotic Performance in MFlops					Increment of the geometric mean over a single group configuration (%)	Increment of the geometric mean over a configuration with the same number of arithmetic units (%)
	Minimum	Harmonic mean	Geo-metric mean	Arithmetic mean	Maximum		
2	20	78	123	156	360	46	25
4	20	92	196	296	720	133	67
8	20	102	315	577	1414	275	169
16	20	106	426	937	2880	407	264

As can be seen, there are no significant benefits in using more then 2 arithmetic units within a group. This topic is currently receiving much attention, and, with adequate interpretation, the same conclusion has been found in different situations [7, 11, 14], and appears to be a quite general property of programs.

The number of functional units has therefore been fixed as two. There are two couples of address units, one for reads and one for writes; each is composed of one single reference and one multiple reference unit. A branch unit completes the set.

5.3– Dimensioning the number of groups

Table 2 shows the same results for configurations with 2, 4, 8 and 16 groups of 2 arithmetic units. Here, the column increment shows the performance increase over a single group with the same number of arithmetic units. As can be seen, the exploitation of fine and medium grain parallelism, allowed by the APA offers an improvement of up to 3.5 times in performance, for configurations with the same hardware resources. What is more important, the improvement increases as the number of arithmetic units increases. Table 2 takes in consideration the reduction in memory bandwidth introduced by the omicron network as the number of groups increase.

As a result, it is worthwhile to use up to 16 arithmetic units in the APA architecture, while only 2 makes sense with a conventional polycyclic architecture. What is more, this can be achieved with register files having the same number of ports and instructions that have the same basic size in both cases.

6 – Conclusions

The first 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.

Results of the analysis and simulation of an APA processor are presented, showing a performance increase of 3.5 times in the geometrical mean of the full Lawrence Livermore Kernels over a conventional VLIW machine with polycyclic extensions.

The inclusion of the features needed to asynchronous operation not only increased the performance but resulted in a more easily implementable hardware and in shorter instructions.

A final note is about software. It is generally regarded that the horizontal architectures are inferior to other architectural alternatives on the ground that code is not portable between implementations with different numbers of functional units. This may be true if one takes a strict view about the concept of object code. Nevertheless, if we call object code the result of compilation just prior to final code generation, the horizontal architectures can be code-compatible.

The CONVEX series of machines proves that it is possible to break the compilation process in a series of front-ends for different languages, followed by a generic optimizer and a generic code generator [3]. For a horizontal architecture, we can call object code the output of the optimizer, distribute it under this form, and customize it for a specific machine.

7 – Bibliography

1. Abu-Sufah, W and Mahoney, A. D., "Vector Processing on the Alliant FX/8 Processor", Proc. Int'l Conf. Parallel Processing, 559-563, 1986.
2. Campos, G. L., Asynchronous Polycyclic Architecture, Proc. of the 12th Word Computer Congress, vol I- Algorithms, Software, Architecture, Sept 1992.
3. Chastain, M et al., "The Architecture of the CONVEX C240", Proc. of the Fifth Conf. on Multiprocessors and Array Processors, 28-31, March 1989.
4. 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
5. Diede, T., et al. "The Titan Graphics Supercomputer Architecture", IEEE Computer 21 (9):13-30, September 1988.
6. 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.
7. 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.
8. Jouppi, N. P., "Architectural and Organizational Tradeoff in the Design of the MultiTitan CPU", IEEE Conf. Proc. of the 16th Annual Int. Symp. on Comput. Architecture, 281-289, May 1989.
9. 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.

10. McMahon, F. H. "The Livermore Fortran Kernels Test of the Numerical Performance Range", in "Performance Evaluation of Supercomputers", 143-186, edited by Martin, J. L.., North Holland, 1988.

11. Oyang, Y. et al, "A Cost-Effective Approach to Implement a Long Instruction Word Microprocessor", Computer Architecture News 18(1):59-72, March 1990.

12. Rau, B. R., Yen, D. W. L. and Towle, R. A. "The Cydra 5 Departmental Supercomputer", IEEE Computer 22 (1):12-35, February 1989.

13. Smith, J. E., "Decoupled Access/Execute Architecture Computer Architectures", ACM Trans. Computer Systems, 2(4):298-308, Nov 1984.

14. 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.