Proceedings of International Conference
on Computer Architecture

On the Design and Modeling of a Homogeneous VLIW Architecture

L. Wang and Ted C. Yang

Graduate Institute of Information Engineering,
Feng Chia University, Taichung, 407, Taiwan, R.O.C.
E-Mail : tcyang@fcu.edu.tw

Abstract

Following the advances in semiconductor technology and computer architecture, we can expect that the functional units on a single chip can grow to tens in the future. By considering the trend in microprocessor design, we propose a new type of parallel architecture which combines the features of both ILP and SIMD machines for instruction level parallelism and loop level parallelism, respectively.

In order to verify the efficiency of the parallel architecture, we have built an analytical model to explore architecture performance with parallelism ranging from different architecture features. The analytical results show that the inclusion of SIMD type execution can improve the execution of vector loops and gain a speedup of 1.6 over traditional ILP machine. When the execution for non-vector codes is included, the proposed parallel architecture can still gain speedups in the range of 1.023 to 1.567 depending on ratio variations of vector codes.

I. Introduction

The combined advancements of semiconductor technology and computer architecture have promoted innovations in microprocessor design from sequential execution to parallel processing. By examining more advanced commercial microprocessors, such as Alpha 21164[1], PowerPC 604[2], Pentium[3] etc., we can find that they all have made use of independent pipelined functional units(F.U.) and technologies of static/dynamic scheduling to speed up program execution. Based on microprocessor design in the last decade, we can observe some trends that already exhibited in the modern microprocessor design:

- Parallelizing a program in fine-grained granularity to promote the performance by instruction level parallelism(ILP).
- Using multiple independent F.U.s and pipelines to elaborate the potential of hardware parallelism.
- Achieving the goal of out-of-order execution by, as in superscalar, dynamic scheduling[4] and developing skills for speculative execution.

Although these trends are quite clear, there exist at least two major obstacles caused by chip layout and the method of exploiting parallelism.

1. Implementation Obstacle:

By using advanced technology of semi-conductor, we can expect that the F.U.s on a single chip can grow to tens in the future. However, the restrictions on the number of ports in uni-register file and on the number of pins on a chip are becoming more severe due to implementation difficulties.

2. Parallelism Obstacle:

Several investigations have pointed out that if the limitation of control dependency could be conquered, then the level of parallelism that can be exploited would be greatly increased. Speculative execution[5][6] is a solution for this purpose. However, the existence of challenges such as how to handle multiple control paths and how to deal with the interrupts raised by speculative instructions, still holds back rapid advancement in this area.

Adequate hardware resources have provided profound impacts on computer architecture. For example, multiple-level on-chip caches have appeared in modern microprocessors[1]. Multi-threading[7] is another successful example to use the excess hardware resources. The main goal of the multi-threading is not to speed up the execution of single program but promote the throughput of computer by using multiple register files for switching among tasks to fully utilize the hardware resources.

There have been attempts to make use of parallelism models other than the commonly known SIMD/MIMD[8] to conquer the limitations directly to achieve higher speedup of single program. The XIMD architecture[9] is such an example. XIMD is built by several F.U.s with each F.U. controlled by a dedicated sequencer and program(s) can be executed in a manner similar to MIMD. The main difference between XIMD and MIMD is that, in XIMD, the F.U.s execute in lock-step fashion and compiler can handle the executions of each F.U. in every cycle. There are many interesting ideas appeared in the

XIMD architecture. There is a global register file and a condition code distribution circuit shared by all F.U.s but the memory is distributed. The concept behind the synchronization method for lock-stepped MIMD execution from the architecture is also appealing.

Multiscalar[10] is another example for including several parallelism models into a parallel system. Different from the XIMD architecture, multiscalar is built by a line of processing elements(P.E.) but controlled by a central sequencer. Compiler partitions the workload into several tasks that can execute concurrently and the sequencer dispatches the tasks to different P.E.s. A most interesting part of multiscalar is the connection of P.E.s, researches suggest that a uni-direction ring is adequate for parallel execution. Based on the examples exhibited in [10], we can find that the connection and the attached synchronization (Forward/Mask) bits behavior well in the fashion of SIMD style.

The theme of this paper is on combining various parallelism features to promote the degree of parallelism in program execution. We do not attempt to propose a novel architecture. Instead, we focus on combining existing ideas and technologies of various parallelism models progressively to build a parallel microprocessor architecture that is suitable for the future, especially when scalability for the progress of fabrication is considered. The design philosophy is discussed in Section II. A parallel architecture named homogeneous VLIW, abbreviated as HVLIW, which can execute program in traditional ILP and SIMD fashion, is presented in Section III. Besides the architectural design, an analytical model is presented to verify the idea of HVLIW. The analysis is described in Section IV. Other related researches, such as compiler and interrupt system, and future works are discussed in Section V.

II. Design Considerations

Computer architectures that are suitable to combine various parallelism features exhibit a set of common characteristics, such as on-chip multi-cluster topology, on-chip multi-level high speed buffer, and software-dominated process scheduling. Is a 2-level cache with the first level distributed to each cluster and the second level built as a global backup an adequate solution and suitable for implementation?

Although the scheduling of modern high-end microprocessors is mainly implemented by the dynamic techniques, such as Tomasulo's algorithm[11], the inability and complexity of dynamic sheduling for extracting parallelism often lead the method to become less attractive, especially for larger instruction windows. Furthermore, the techniques for static speculative execution and memory disambiguation can alleviate the inability of static scheduling caused by the insufficient dynamic information. Based on researches about this prospect, we tend to believe that the scheduling for parallel processing can be most efficiently achieved by software, mainly statically, and by assisted hardware support.

To push our design considerations further, the target architecture is provided with the following additional features.

1. Clusters are distributed and homogeneous.

   Although a centralized control of parallel execution can make the design simpler, it will certainly limit the scale of parallelism. On the other hand, distributed resources allocation can help avoid the bottleneck. Homogeneousity means the clusters and the connections among clusters are identical and makes scheduling and workload partitioning easier than asymmetric topology.
2. Clusters are controlled by dedicated program counters.

   A centralized sequencer violates the distribution principle and is unfavorable to system scalability. In order to fully facilitate the control of various parallelism features, each cluster is provided with its own execution stream.
3. Parallel execution is in lock-step fashion.

   Identification of parallelism in a program is completely done by the compiler. A program is transformed into a parallel executable format before execution. A global system clock is designed to synchronize the execution in clusters in each cycle. Since the target architecture is designed for fabrication in a single chip, the building of a global clock is much easier than traditional multiprocessors. The complexity of clock generator would not be increased dramatically by an increasing on the number of clusters, either.

Fig.1 is the block diagram of the target architecture. Since each cluster is built by a line of independent F.U.s that can execute instructions simultaneously for a program stream, techniques for the ILP machines can be included without major modifications. Moreover, the clusters are synchronized by a global clock, thus compiler can schedule operations for all F.U.s in all clusters. Static scheduling can reduce the hardware complexity for scheduling and provide larger parallelism by adequate hardware support. So we decide to build the cluster as a VLIW processor[12][13] to inherit the features of ILP machine. The issue on VLIW drawbacks needs to

be considered. For example, the drawback on bigger code size can be reduced by code compaction. For another example, in order to reduce the hardware complexity for scheduling, the process of re-compiling is an inevitable and costless way to exploit the variant features of parallelism. The main criticism of re-compiling is not to avoid but to use it efficiently.

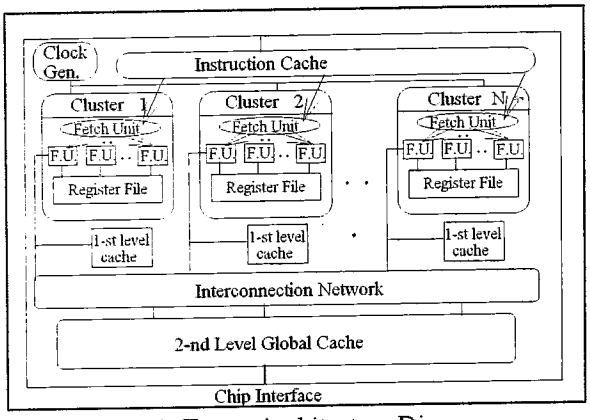

Fig. 1. Target Architecture Diagram

By checking the dependence relationship in a loop and arranging independent iterations so that they can be executed simultaneously, data parallelism exploits the loop-level parallelism existing in iterations. Commercial SIMD machines are all so designed to speedup the program execution.

Since for either SIMD or VLIW processors, programs are scheduled by the compiler in a form suitable for parallel processing, so we started out to include the merits of SIMD into the target architecture. Although some well-known scheduling techniques such as loop unrolling and software pipeline[14] can extract massively parallelism in a loop, they would not be suitable for a multi-cluster environment. This criticism can be explained clearly by the next few paragraphs and an example.

Assume there exists a vector loop with a data dependence graph as described in Fig.2. Loop unrolling is to unroll the loop codes several, even tens of, times for scheduling. Since complexity in obtaining the optimal solution for the scheduling is NP-complete, heuristic list scheduling is usually adopted to assign codes to clusters. The algorithm of list scheduling is based on a greedy method that schedules the instructions according the precedence order and arranges instructions into empty operation slots as earlier as possible. By assuming that the target architecture is built by 4 clusters with 1 issue capability per cluster and the instruction latencies are 1, the scheduled object codes by data parallelism can fully utilize the hardware resources as shown in Fig.3.a The object codes scheduled by unrolling the loop 4 times is shown in Fig.3.b.

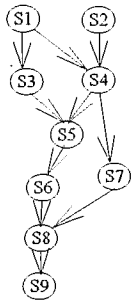

Fig.2. Data Dependence Graph of a Sample Loop

If the loop is unrolled even more times, the possibility to fully occupy all the instruction slots will be increased to what the SIMD can achieve. However, this does not necessarily mean that the parallelism will be promoted to the exact achievable level in SIMD. Since the instructions of an iteration may be distributed to distinct clusters, additional instructions for communication among clusters are added and may lead to a less condensed density of effective operations. Furthermore, the larger code size produced by unrolling can lead the hit ratio of instructions to be degraded.

Clock Cycle				
1	$S1^1$	$S1^2$	S^3	$S1^4$
2	S^1	S^2	S^3	S^4
3	$S3^1$	$S3^2$	$S3^3$	$S3^4$
4	$S4^1$	$S4^2$	$S4^3$	$S4^4$
5	$S5^1$	$S5^2$	$S5^3$	$S5^4$
6	$S6^1$	$S6^2$	$S6^3$	$S6^4$
7	$S7^1$	$S7^2$	$S7^3$	$S7^4$
8	$S8^1$	$S8^2$	$S8^3$	$S8^4$
9	$S9^1$	$S9^2$	$S9^3$	$S9^4$

(a) SIMD Scheduling

Clock Cycle				
1	$S1^1$	S^1	$S1^2$	S^2
2	$S3^1$	$S4^1$	$S3^2$	$S4^2$
3	$S5^1$	$S7^1$	$S5^2$	$S7^2$
4	$S6^1$	S^3	$S6^2$	S^3
5	$S8^1$	$S3^3$	$S8^2$	$S4^3$
6	$S9^1$	$S5^3$	$S9^2$	$S7^3$
7		$S6^3$	$S1^4$	S^4
8		$S8^3$	$S3^4$	$S4^4$
9		$S9^3$	$S5^4$	$S7^4$
10			$S6^4$	
11			$S8^4$	
12			$S9^4$	

(b) Loop Unrolling

Fig.3. Scheduling for the Example in Fig.2

The above example is not meant to negate the value of loop unrolling. Quite contrary, loop unrolling is an effective method to schedule codes for non-vectorized loops. What we try to point out is that traditional scheduling algorithm designed by greedy method may not be suitable for the environment of multi-clusters. Software pipeline[14] is another well-known scheduling method for vector loops. Again, the kernel of software pipelining is still a greedy method as discussed above. If the algorithm needs to be modified to make it more suitable for the environment, then the complexity of the algorithm will increase dramatically and still may not be proven to be an optimal solution. The inclusion of SIMD type execution is a choice we believe that is suitable for the circumstances.

III. Homogeneous VLIW Architecture

The block diagram of a homogeneous very long instruction word(HVLIW) processor architecture is proposed in Fig.4. The organization of the basic execution engine, a cluster, is classified as a VLIW processor. The major components of the cluster include two integer functional units(I0 and I1), a floating point functional unit for addition/subtraction (FA), and a floating point functional unit for multiplication/division(FM).

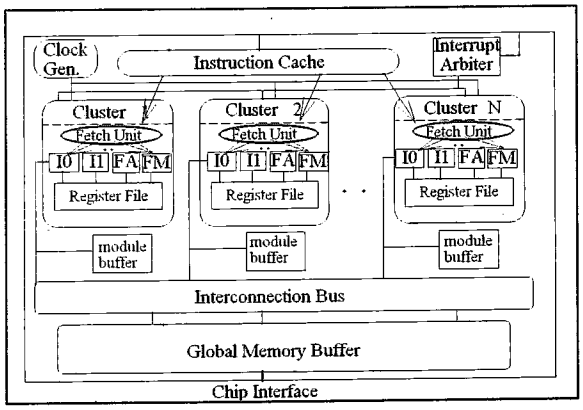

Fig. 4. The HVLIW Block Diagram

Two Execution Modes

With N clusters, the two execution modes of HVLIW are shown in Fig. 5. The first execution mode is called the Long-instruction mode, abbreviated as L-mode. In this mode, the system operates like a conventional large-scale VLIW machine driven by very long instructions with 4*N operation fields.

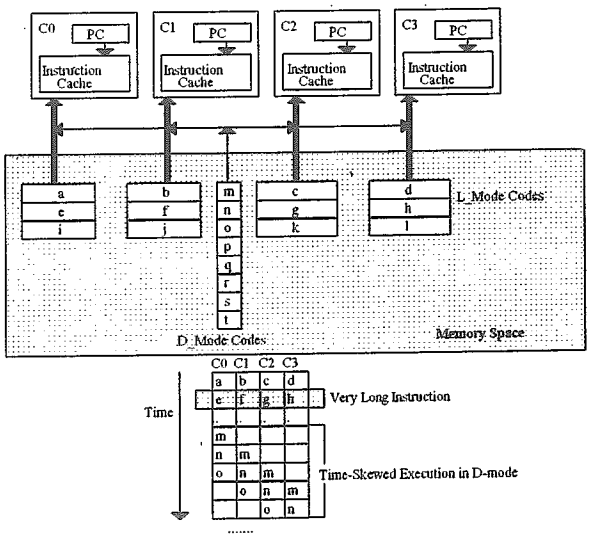

Fig. 5. Two Execution Modes of HVLIW

The second mode is called the Duplication mode, abbreviated as D-mode. In this mode, all clusters will execute identical instructions, for different loop iterations, concurrently. It should be noted that each cluster can still exploit very fine grain parallelism within an iteration instance because there are four functional units in each cluster. In the D-mode, the HVLIW system acts almost like an SIMD machine. The only difference is that there are N program counters in HVLIW but only one in an SIMD.

For the smooth execution of D-mode, a scheme called time-skewed execution[15], as shown in Fig.5, is adopted to reduce the demand on bandwidth due to possible burst data traffic and to avoid the use of expansive non-blocking network. It distributes the same operation regularly into consecutive time slots.

In L-mode, operations that should be executed at the same time are scheduled in the same long instruction. When there is a need to change into D-mode, the compiler can generate proper branch operations as the last L-mode long instruction. Fig.6 shows the events necessary for switching from L-mode to D-mode. By executing the normal JMP instructions simultaneously, the clusters are forced into D-mode. To switch from D-mode to L-mode, the reverse operations can be arranged by the compiler.

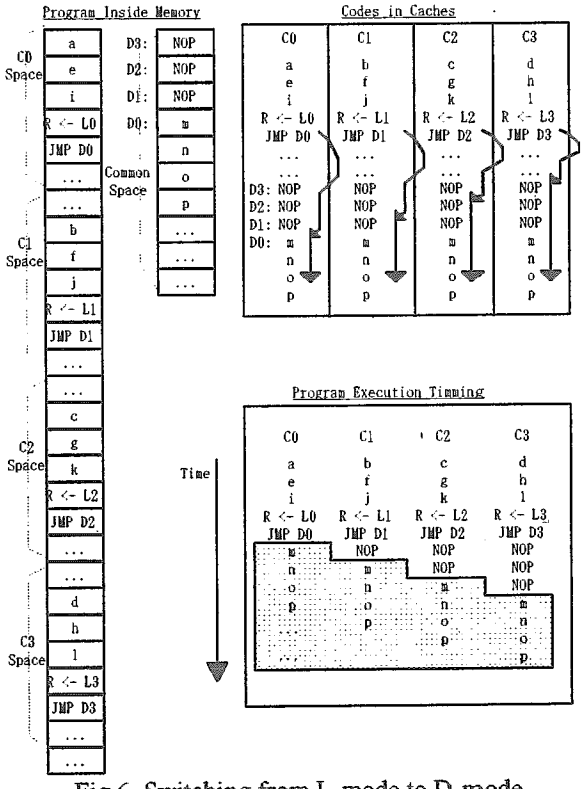

Fig.6. Switching from L-mode to D-mode

On-Chip Memory Hierarchy

If a memory access is not completed in one instruction cycle, the pipeline will be stalled for the

access and thus the performance will be seriously degraded. A general solution of this problem is to insert a high speed data cache between the processors and the main memory. However, this is not suitable for the HVLIW machine. The efficiency of a cache depends mainly on spatial locality and parallel execution reduces the spatial locality. Furthermore, a significant waste of bandwidth can result if a whole block is fetched when the execution is vector-like and the stride is greater than one.

Since the compiler can predict the reference behavior for most instants of program execution, using a line of buffers to store the data expected to be referenced soon is a feasible alternative to provide fast reference.

For each HVLIW cluster there are up to two memory references per instruction cycle. In D-mode, N cycles, one for each cluster, will be required for each memory reference in a loop, or there will be bus conflicts. For a matched memory system in which the memory access time is equal to the reference cycle of a cluster, up to $2*N^2$ words of data would be accessed, and the bus must have a bandwidth of 2*N words per cycle.

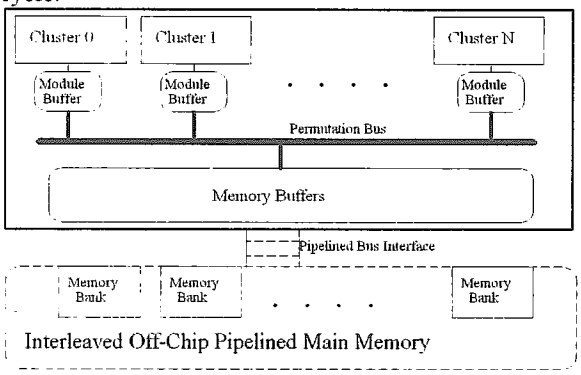

Fig.7. Memory Hierarchy in HVLIW

As a result, a high speed two-level buffer system[15] is proposed instead of a 2-level cache, as shown in Fig.7. The first level is a line of module buffers, one in front of each cluster. A module buffer will catch the 2*N words of data to be referenced by the respective cluster. This allows the clusters to perform one-cycle LOAD/STORE operations. The second level is called memory buffer. It is inserted between the module buffers and the memory banks to hold the $2*N^2$ words of data. A permutation bus 2*N words wide connects the two levels of buffers. This memory hierarchy is named a 3-stage memory.

With the support of time-skewed execution, the memory hierarchy can satisfy the demand of the HVLIW system at a cost lower than that of a conventional parallel memory system. Furthermore, a dynamic storage scheme is applied in our design for an interleaved memory to avoid memory bank conflicts.

IV. Theoretical Analysis

In order to evaluate the features of HVLIW, a theoretical analysis is performed. Since the main difference between HVLIW and ILP machine is the method for exploiting parallelism from vector loop, we shall first present a simplified comparison for the performance characteristics between the D-mode execution in HVLIW and the traditional ILP machines. Then the performance model of HVLIW is modified to include considerations on 3-stage memory as well as on non-vector codes. From the analysis results for HVLIW model and ILP machines, the efficiency of the HVLIW architecture is clearly shown.

In the following discussion, T represents the total number of operations in a program, and S represents the number of stages in a pipeline. Since HVLIW is constructed by a line of N clusters, the clusters of ILP machine is defined by N VLIW processors each with K F.U.s for an equitable comparison.

Evaluation for Vector Loop Execution

For verifying the efficiency of D-mode execution, the execution time spent for vector loop execution in a program is estimated. Assume that the ratio of vector codes in a program is R_l, then there exist $T* R_l$ operations that can be scheduled for execution in the D-mode. Since each iteration is scheduled to a dedicated cluster with K F.U.s, a parallelism with degree μ that is less than or equal to K can be extracted from an iteration. Let I stands for a positive integer and $I = \left\lfloor \frac{K}{\mu} \right\rfloor$, then I iterations can be arranged into a cluster for full use of the potential structural parallelism supported by cluster. As a result, the number of cycles spent for the D-mode execution is:

$$C_l = \frac{T*R_l}{N*I*\mu} \quad \text{.....(1)}$$

Let us define two additional parameters:

O_i: operations needed for the initialization of a vector loop execution.

N_l: the number of vector loops in a program.

If there are four operations in an instruction word, then , for the worst case, a total of N*(N-1)/2 *K operations slots could be wasted during mode switching. The slots spent for the aforementioned time-skewed execution are N*(N-1)/2*K*2 for each pair of mode switching. The delay, C_2, caused by loop initialization and time-skewed penalty can be calculated as :

C_2= Cycles for Loop Initialization + Cycles for Time-skewed Penalty

$$= \frac{O_i * N_l}{N*I*\mu} + \frac{N*(N-1)*K*N_l}{N*I*\mu} \quad \text{.....(2)}$$

Finally, the total number of cycles required by D-mode execution can be estimated by the sum of C_1 and C_2. In equation (2), O_i stands for the number of initialization operations. Although O_i is difficult to estimate, it should be proportional to the size of registers in each cluster. Therefore, we say that O_i should never be greater than 32*N, if each cluster is equipped with 32 registers. N_l is another parameter that is difficult to determine. We can only assume that N_l is a small constant value and independent to the other parameters.

Assume that the average number of effective operations that can be scheduled in a cycle by ILP scheduling is σ, and that there are σ_c additional communication operations to be added into a cycle for cross-cluster references. The total number of cycles required can be estimated as:

$$C_3 = \frac{T*R_l}{\sigma}, \quad \sigma + \sigma_c \le N*K \quad \text{..... (3)}$$

The relation between σ and σ_c can be further considered. Assume that each operation needs two operands for execution and that the operands are provided by preceding calculations of clusters. Also assume that the scheduling algorithm is modified to arrange an instruction into a dedicated cluster only under the situation that there is at least one operand already existed in the cluster. Then the possibility for needing communication to obtain the other operand is $\left(1-\frac{K}{\sigma}\right)$. Thus the expect number of communications in a cycle, σ_c, is $\sigma*(1-\frac{K}{\mu})$. By assuming the usage of operation slots is perfect, that is $\sigma+\sigma_c = N*K$, the value of σ can be calculated by equation (3) and the result is $\sigma = \frac{(N+1)*K}{2}$.

Table 1. Parameters for Analysis

N(Number of processors)	4
R_l(Ratio of vectorized codes)	0.1~0.9
N_l(Number of vector loops)	10
K(Number of functional units)	2,4,8,16
T(Total operations)	1000000
μ(Achieved ILP from iteration)	1~16
O_i(Operations for initialization)	32N

By setting reasonable values to the parameters, we can compare the performance characteristics of these two types of execution. With the set of parameter values in Table 1, the speedup of D-mode execution over ILP scheduling is shown in Fig. 8.

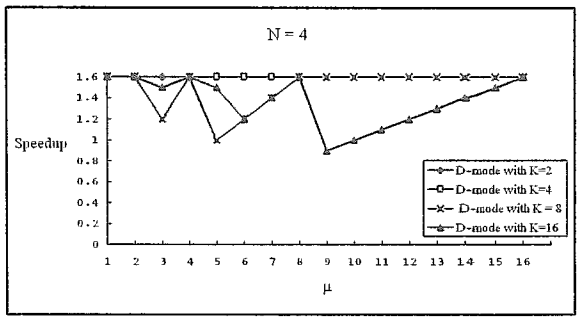

Fig.8. Speedup of D-mode Execution over ILP Scheduling

The lines on Fig.8 indicate that the speedups are clearly affected by the pair of parameters, structural parallelism (K) and extracted parallelism (μ). Although most of speedup values are 1.6, the worst case of speedup is degraded to 0.9 when (K, μ) are (16,9). The reason for the degradation is caused by a mismatch of the pair which leads the usage of effective operations per cycle to become lower than that in the ILP machine. The degradation can be eliminated by reducing the value of μ from 9 to 8, which would in turn make a recovery on the speedup to 1.6. We believe that the speedup would actually be higher than 1.6 since the cycles spent for time-skewed penalty and loop initialization can be overlapped and the value of σ may be lower than the estimated optimal.

The analysis described above is simplified for easy understanding. However, from the analysis above, we conclude that whenever a vector loop is suitable for SIMD type execution, D-mode is an easier and at least as effective way for execution.

Performance Models for HVLIW/ILP Machines

In the following discussion, analytical models are built for the HVLIW and ILP machines in Fig.4 and Fig.1, respectively. Since the method for exploiting the parallelism of non-vector codes are similar in both machines, the effects of inter-cluster communication, speculative execution and branch penalty, etc. are not considered in the modeling for simplication reasons. A main point of the analysis is on the effects of memory hierarchy. Since HVLIW is equipped with two-level buffer, the memory hierarchy of ILP machine is equipped, for fairness, with two-level caches with the same probability of miss rate and miss penalty.

The cycles spent for ILP machine are estimated first. Assume that the length of the pipelines of all the functional units are equal to S. Then the number of cycles needed to execute the program, excluding delays caused by branches and/or memory references, can be estimated using the following equation, where

μ means the average parallelism extracted from machine codes.

$$C_4 = C_3 + (S + \frac{T*(1-R_l) - Min(\mu, N*K)}{Min(\mu, N*K)}) \quad \text{.....(4)}$$

For the delay caused by branch operations and memory references, let us assume the following parameters:

R_r: the ratio of memory operations in the entire program.

R_{ml}: the miss rate of first-level cache.

R_{m2}: the miss rate of second-level cache.

D_{ml}: the number of pipeline stages stalled due to a first-level cache miss.

D_{m2}: the number of pipeline stages stalled due to a second-level cache miss.

It should be noted that the memory reference will lead to a situation where only D_{ml}/ D_{m2} operation slots be stalled. In the model, there are Min(μ, N*K)/σ effective slots exist in an instruction cycle. Then the total delay can be briefly defined as:

C_5 = Cycles for memory references =

$$T*R_r*(\frac{(1-R_l)*R_{ml}*(D_{ml}+D_{m2}*R_{m2})}{Min(\mu, N*K)} + \frac{R_l*R_{ml}*(D_{ml}+D_{m2}*R_{m2})}{\sigma}) \quad \text{.....(5)}$$

The total number of cycles required by an ILP machine can be estimated by the sum of C_4 and C_5.

The number of cycles needed by HVLIW, again excluding delays caused by memory delay, can be estimated by the following equation:

$$C_6 = C_1 + C_2 + (S + \frac{T*(1-R_l) - Min(N*K, \mu)}{Min(N*K, \mu)}) \quad \text{.....(6)}$$

In order to consider the delays caused by the 3-stage memory preload/poststore operations, let us assume that the miss rate and miss penalty of 3-stage memory are the same as those in the two-level caches, as described above, and that P represents the number of data words that can be packaged in a preload/poststore instruction. Now, the penalties caused by the 3-stage memory preload/poststore operations can be estimated by equations (7) and (8), for the D-mode and the L-mode, respectively.

$$C_7 = (\frac{T*R_l*R_r}{N})*(\frac{D_{m2}*R_{m2}+D_{ml}}{2*N})*R_{ml} \quad \text{.....(7)}$$

$$C_8 = (\frac{T*(1-R_l)*R_r}{Min(\mu, K*N)\ /\ K})*(\frac{D_{m2}*R_{m2}+D_{ml}}{P})*R_{ml} \quad \text{.....(8)}$$

The total number of cycles required by HVLIW can be estimated by the sum of C_6, C_7 and C_8.

Analysis Results

We first define a standard model by setting values to the parameters as shown in Table 2. As a second step, the ratio of vector codes(R_l) is tuned from 0.1 to 0.9 and the average ILP parallelism(μ) is varied from 2 to 16. The resultant speedups of HVLIW over ILP machine are listed in Table 3.

Table 2. Parameters for Standard Model

N	4	S	4
N_l	10	R_r	0.25
D_{ml}	1	D_{m2}	10
P	K*N/4	T	1000000
K	4	O_i	32N
R_{ml}	0.05	R_{m2}	0.1
D_m	N	D_m	N

Table 3 shows that HVLIW can always gain a better performance than ILP machine. This situation becomes more evident when the vectorized portion is increased. Fig.9 shows the curves of the speedups by varying vector rate. From the curves that proportional to the growth of vector rate, we can conclude that HVLIW inherits not only the feature of fine-grained parallelism, but also that of the data parallelism.

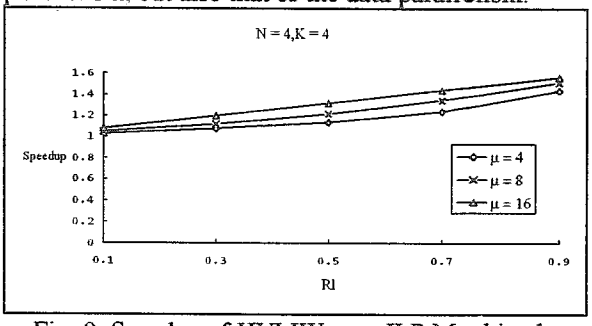

Fig. 9. Speedup of HVLIW over ILP Machine by Varying Vector Rate

The maximum speedup can be achieved is about 1.567 when both the vector rate and the ILP parallelism are high enough. It should be noted that it is mainly the SIMD type execution which provides the HVLIW an edge over the ILP machine. When the execution for non-vector codes is included, the benefit gained may become less obvious. It is the reason about the degradation of the speedup than the value estimated for D-mode execution with 1.6 speedup.

The 3-stage memory is designed not only for reducing the complexity of interconnection network from cross-bar to bus connection, but also for promoting the efficiency of on-chip memory hierarchy. We find that the performance of 3-stage memory is 2.534~5.811 times better than two-level cache for different (μ,R_l) pairs.

Many other observations can be made by tuning values of different parameters. For one example, we find that the packing ability of compiler for 3-stage memory (P) , will obviously influence the

Table 3. Speedups of HVLIW Over ILP Machine

	μ=2	μ=4	μ=6	μ=8	μ=10	μ=12	μ=14	μ=16
$R_l = 0.1$	1.023	1.031	1.039	1.047	1.054	1.061	1.068	1.075
$R_l = 0.3$	1.046	1.074	1.099	1.122	1.144	1.163	1.181	1.198
$R_l = 0.5$	1.083	1.137	1.182	1.219	1.25	1.277	1.300	1.321
$R_l = 0.7$	1.153	1.240	1.300	1.345	1.378	1.405	1.426	1.444
$R_l = 0.9$	1.339	1.439	1.488	1.516	1.535	1.549	1.559	1.567

efficiency of 3-stage memory in non-vector execution. When the value of P is tuned to 1, 2, 4 and 8, the efficiencies over two-level cache are changed from 0.25, 1, 2.5 to 5, respectively. For another example, when the latency of off-chip memory access (D_{m2}) is increased to 20, the performance of 3-stage memory decreases from 2.534~5.811 to 1.523~4.2.

V. Concluding Remarks

With the trends of microprocessor design in mind, we have proposed in this paper a new type of parallel architecture that combines the features of both ILP and SIMD machines. By using special execution modes, this architecture can exploit both instruction level parallelism and loop level parallelism. A special on-chip 3-stage memory hierarchy is proposed. From the results made by theoretical analysis, we expect that the parallel architecture will have a gain in performance of up to 1.567 times over traditional ILP machine.

We have reached the first step of our design goals so far. There are at least four directions of research that can be studied in the future: the inclusion of MIMD style execution, the design of speculative execution in multi-cluster environment, the design of a fast inter-cluster communication network, and, above all, the implementation of an effective compiler.

References

[1] E. McLellan, "The Alpha AXP Architecture and 21064 Processor," IEEE Micro, pp. 36-47, June 1993.

[2] S. P. Song *et al.*, "The PowerPC 604 RISC Microprocessor," IEEE Micro, pp. 8-17, October 1994.

[3] D. Alpert and D. Avnon, "Architecture of the Pentium Microprocessor," IEEE Micro, pp. 11-21, June 1993.

[4] J. L. Hennessy and D. A. Petterson, Computer Architecture: A Quantitative Approach, 2nd edition, *Morgan Kaufmann Publisher, Inc.*, 1996.

[5] P. P. Chang, N. J. Warter, S. A. Mahlke, W. Y. Chen, and W. W. Hwu, "Three Architectural Models for Compiler-Controlled Speculative Execution," IEEE Trans. Comput., Vol. 44, No. 4, pp.481-494, April 1995

[6] M. D. Smith, M. S. Lam and M. Horowitz,"Boosting beyond static scheduling in a Superscalar Processor," Proc. 17th Annual Int. Symp. Computer Architecture, pp. 344-255, 1990.

[7] S. W. Keckler and W. J. Dally,"Processor Coupling: Integrating Compile Time and Runtime Scheduling for Parallelism," Proc. 19th Annual Int. Symp. Computer Architecture, pp. 202-213, 1992.

[8] M. J. Flynn,"Some Computer Organizations and Their Effectiveness ", IEEE Trans. Comput., 21(9), pp.948-960, 1972

[9] A Wolfe and J. P. Shen,"A Variable Instruction Stream Extension to the VLIW Architecture," Proc. of the 4th International Conference on Architectural Support for Programming Languages and Operating Systems, pp.2-14, April 1991.

[10] G. S. Sohi, S. E. Breach, T. N. Vijaykumar,"Multiscalar Processors," Proc. 22th Annual Int. Symp. Computer Architecture, pp. 414-425, 1995.

[11] R. M. Tomasulo, "An Efficient Hardware Algorithm for Exploiting Multiple Arithmetic Units," IBM Journal, pp.25-33, Jan. 1967.

[12] G. R. Beck, D. W. L. Yen, "The Cydra 5 Minisupercomputer: Architecture and Implementation," *The Journal of Supercomputing*, vol. 7, no. 1/2, pp. 143-180, 1993.

[13] J. A. Fisher, "The VLIW Machine: A Multiprocessor for Compiling Scientific Code," IEEE Computer, pp. 45-53, July 1984.

[14] M. S. Lam, "Software Pipelining: An Effective Scheduling Technique for VLIW Machines," Proc. ACM SIGPLAN Conf. Prog. Lang. Design and Implementation, pp. 318-328, 1988.

[15] L. Wang, C. T. Kan, J. W. Jou, "A Dual-Mode Homogeneous VLIW Computer and Its Memory Hierarchy," Proceeding of the International Computer Symposium, Taiwan, pp.457-464, December 1994.