A Multithreaded Vector Co-processor

Bernard Goossens

IBP-LITP, Université Paris 7 Denis Diderot, 2 place Jussieu
75251 Paris cedex 05, France
email: bg@litp.ibp.fr

Abstract. A multithreaded vector co-processor design is described. It is intended to be placed with its private vector memory, on an expansion board, linked to the scalar processor and its cache-based memory hierarchy. The vector co-processor can run up to 8 vector tasks (threads) in parallel. Vector registers can be accessed either as independent sets of scalar values or as array sets. The Tomasulo's algorithm, simplified to keep the issue and termination logics simple in a multithreaded context, dynamically schedules the dependent instructions. A locking feature is provided to handle both the reductions and the complex recurrences in a vector form.

1 Introduction

In the 70's and 80's, vector supercomputers like the Cray were designed to handle data parallel computations more efficiently than on standard hardware. However, these architectures were very costly ones and in the 90's, commercial superscalar microprocessors have become very serious competitors for vector machines. Not that these new processors are as fast as their contemporaries supercomputers, but they do are as fast as 5 years old ones and more than ten times less costly. So the question is: has the superscalar technology rung the knell of vector computation? A key to the answer is to understand what essentially differentiates the two approaches. Vector hardware is designed to pull data from memory as smoothly as possible, fill the computing units pipelines and send results back to memory, whereas superscalar hardware is designed to provide enough scalar instructions to keep the issue unit busy.

To pull data smoothly from memory, a uniform memory access (UMA) structure must be employed rather than a hierarchical one based on caches. A large amount or all the data must be resident, so the memory size must be large. Then, the memory access time will also be large and to provide enough memory bandwidth to the computing units, first many parallel and interleaved accesses must be provided and second, vector registers are necessary to reduce the memory traffic. From the vector registers access time, the rate of the machine is derived. They provide operands to the computing units every cycle, so each computing unit that takes more than one cycle to deliver its result must be pipelined. Because read-after-write dependencies between vector instructions can be simply handled by chaining, no single scalar result by-pass from the pipelines outputs to their inputs has to be implemented. The last step of the design is to fix the vector machine language, with register oriented vector instructions. These multicycles instructions leave the processor fetch and decode units rather unbusy so that pre-fetching, branch prediction and superscalar issue hardware is irrelevant for vector code.

To adapt a superscalar design to high vector performance, we start from the instructions, which are scalar ones, and their issue logic. The cache-based memory structure is already imposed by the scalar computations that the processor must run efficiently. Data parallel computation is exhibited by the compiler through loop unrolling and software pipelining. From such prepared codes analysis, the computing units set is enhanced, extending more particularly the data cache ports set. The multiple instructions issue logic fixes the machine rate. As in the vector machine, any computing unit that cannot deliver its result in a single cycle is pipelined. Pipelines by-passes are essential to scalar computations but not used when independent vector data are treated. Scalar instruction fetch, decoding and issue includes a lot of useless work when applied to a set of instructions equivalent to a vector one.

So, firstly, in a certain sense, a vector machine performs well on vector computations and does its best on scalar ones and a superscalar machine has the reverse features. But none of the two is the best in both computation styles. Secondly, in a vector machine, the more the vector registers, the larger the machine cycle whereas in a superscalar machine, the higher the superscalar degree, the larger the machine cycle.

In this paper, we propose to separate vector computations from scalar ones (i.e. processes that do not manipulate vectors; the so-called non vectorizable part of vector code is taken in charge by the vector co-processor in a way explained at the end of section 3). We justify this separation by the fact that, as noted above, vector and scalar computations require rather different hardwares for memory structure, computing units and processor control.

Whatever the choice, superscalar or vector processor, data parallelism exploitation suffers from the non uniform distribution of the resources utilization. This is best illustrated by memory addresses conflicts leading to banks conflicts in a vector processor and to cache set conflicts in a superscalar one. In both cases, the full available bandwidth is not used. A similar situation arises when a functional unit has to handle multiple vector requests. Other units may stay idle while the requested one has to serialize all the computations. Duplicating functional units and memory banks or cache sets leads to unfrequently busy hardware which is bad both for cost and efficiency. Recent works on multithreading [7] [3] have shown that *simultaneous multithreading* (SMT) may somewhat enhance the distribution uniformity of the requests by sharing the resources and allocating them to multiple concurrent threads running in parallel. In the vector co-processor context, on each cycle one memory request per bank may be selected from different threads, for a full bandwidth filling. The same policy may also be applied to functional units allocation.

Some vector co-processors have been designed in the past, mainly as extensions to existing scalar machines. The IBM 3090VF [5] is a special unit taking in charge the execution of vector instructions fetched and decoded by the scalar part of the machine. The Vax vector architecture [1] has the same function in the Vax 9000 machine but it is able to keep a limited set of multiple vector instructions, mainly to take advantage of the pipelines and the vector registers through efficient chaining. But none of them has a separate instruction path allowing a separate execution of scalar and vector programs.

However superscalar processors are less efficient than vector ones on vector code, it is worth mentioning two of them that have introduced some interesting features to enhance vector computations. First, the MultiTitan is a superscalar FPU [4] in which instructions have a length field indicating the number of steps to be performed. A length-1 instruction is scalar and a length-n ($n > 1$) one is a vector instruction. Moreover, the register file is vectorized and instructions can designate any vector register entry, both for sources and destination. In a certain way, vector instructions are converted into multiple scalar ones. The computing units and the load/store ones are built as in a traditional superscalar machine (load/stores use the memory hierarchy and computing units have the lowest possible latency to achieve a high scalar performance). This is why it is more considered a superscalar processor with vector facility than a true vector processor. Second, Chiueh [2] studies multithreading in the context of vector processing. He stresses the fact that multithreading may serve to fill the pipelines empty slots due to dependent vector instructions and operators latencies. Threads are obtained by software pipelining applied by the compiler to the vector loops. The compiler computes instruction issue scheduling (the one that leaves a minimum pipeline empty slots). The paper shows that multithreading particularly enhances traditionally non-vectorizable code like recurrences. However, a major disadvantage of compiler-based issue scheduling is that changing operator latencies (e.g. critical path improvement) changes the schedule, implying a recompilation.

Next section introduces the vector processing board and its relationships with the scalar processor and the scalar memory hierarchy. Section 3 describes the multithreaded vector co-processor. Section 4 gives some performance hints from a simulation of the SAXPY vector loop and a vector sum reduction.

2 The vector processing board

The vector co-processor is intended to be placed on an expansion board (see Fig. 1), along with its private vector memory. The board communicates with either the scalar processor or the scalar memory hierarchy. It may also serve as a DMA controller to load and store data to and from mass-storage devices. A vector task is started from the scalar processor, by writing parameters and task address in the co-processor registers. Once this step is finished (ending with the task address placed in one of the available PCs), the vector thread starts. The vector co-processor fetches vector code from the scalar processor data memory hierarchy. Once fetched, opcodes are kept in an internal instruction cache (one private instruction cache per thread). During computation, the vector co-processor interacts with its private vector memory. If necessary, exchanges between the vector and the scalar memories may take place (vector data I/O). An *end* instruction stops the vector thread which interrupts the scalar processor that may read scalar results or status directly from the co-processor registers. The different threads work asynchronously, some computing, some being engaged in vector loading or storing and possibly some staying idle.

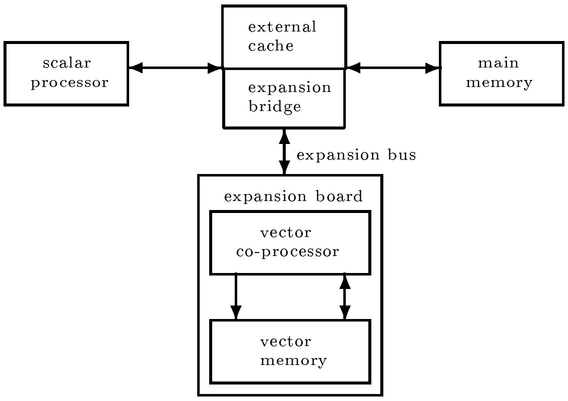

Fig. 1. The vector processing expansion board

Vector memory can be viewed as a system resource, allocated and freed by the processes upon system calls requests. This has some advantages over a compiler allocation that would lead to a virtual/physical mapping, possibly placing a vector into non contiguous pages. A part of the vector memory can be reserved for local needs, e.g. extending each thread register set. The threads may be started either by different processes or by a single one (e.g. computing independent parts of a vector result).

3 The vector co-processor

The co-processor is composed of 3 main units: the expansion bus interface unit, the vector memory busses interface unit and the vector processor unit. In the following we will only focus on the vector processor unit.

As in the MultiTitan FPU [4], our co-processor does not include any separate scalar unit. Instead, scalar instructions are interpreted as vector length-1 instructions. Their source and destination may be any vector register (one of them may serve as the scalar register file). The vector unit has 8 vector registers per thread, of 64 entries each. With 8 threads, this represents 32KB of data storage for the registers. Such a choice implies enlarged instructions opcodes, with each source and destination specifying both the vector register number and the entry aimed by the first step of the instruction. Because fetch bandwidth is low for this co-processor, opcodes can be extended to 64 bits which leaves enough place to encode three operands (including a length), a destination and an operation. Figure 2 shows the format of all the co-processor instructions where *vop* is the vector operation (arithmetic or logical operation, load, store or control flow), *vd* the vector destination (8 vector registers), *ed* the first destination entry in the *vd* vector register ($ed < 64$), $vs1$ the vector left source, $es1$ the first $vs1$ entry, $vs2$ the vector right source, $es2$ its first entry, *vl* the vector register containing the length and *el*, its entry. The left and right sources may be either scalar ones or vector ones (s/v bit). They may also be constants (k/r bit), as the length field. The destination is scalar if the length is 1, vector otherwise. The *reduction* bit (bit 55) is used to indicate if the vector operation is a reduction one or not (explained in more details later; the main difference between reductions and other operations is that the destination is a scalar accumulator rather than a vector). The *lock* bit (bit 45) is used for reduction and recurrence vector instructions (explained later). Tables 1 and 2 give some examples of different vector and scalar instructions.

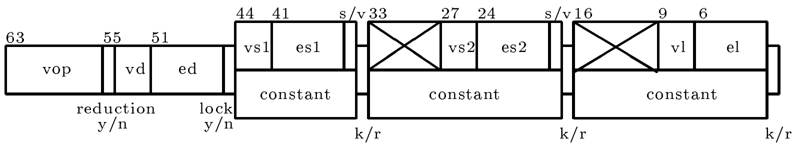

Fig. 2. The instruction format

instruction	semantic
V1=V2+V3	R[1][0..63]=R[2][0..63]+R[3][0..63] $ed = 0$ and $length = 64$ implied
V1=V2+1	R[1][0..63]=R[2][0..63]+1 $s2$ is a constant
V1=V2+S3	R[1][0..63]=R[2][0..63]+R[0][3] $s2$ is a scalar register, $vs2 = 0$ implied
S0=S1+S2	R[0][0]=R[0][1]+R[0][2] scalar instruction; $length = 1$ implied $vd = vs1 = vs2 = 0$ implied
S[7][0]=S[7][1]+S[7][2]	R[7][0]=R[7][1]+R[7][2] scalar instruction; $length = 1$ implied

Table 1. Instructions examples

instruction	semantic
V1[2]lock=V1[1]+V1[0]	R[1][2..63,0,1]=R[1][1..63,0]+R[1][0..63] recurrence
V1lock+=0,V2	R[1][0]=sum(R[2][0..63]) reduction; $length = 64$ implied vs1 (0): accumulator entry number
V1lock+=V3[7],V2[4]	R[1][R[3][7]]=sum(R[2][4..63,0..3]) reduction; $length = 64$ implied vs1 (V3[7]): accumulator entry number
V1=V2+V3,32	R[1][0..31]=R[2][0..31]+R[3][0..31] immediate $length = 32$
V1=V2+V3,S0	R[1][0..l]=R[2][0..l]+R[3][0..l] indirect $length = l = R[0][0]$
V1=M[S1]	R[1][0..63]=M[A+0..A+63] load from $base$ $A = R[0][1]$ $stride = 1$ and $length = 64$ implied
M[S1],S2=V1	M[A+0*S2..A+63*S2]=R[1][0..63] store to $base$ $A = R[0][1]$ $stride = R[0][2]$, $length = 64$ implied
M[S1],V2=V1	M[A+V2[0]..A+V2[63]]=R[1][0..63] store to $base$ $A = R[0][1]$ vector $stride = R[2][0..63]$ $length = 64$ implied
B S1=S2,1024	if (R[0][1]==R[0][2]) goto 1024 length field serves as jump address

Table 2. More instructions examples

The fetch part of the vector processor pipeline (see Fig. 3) has 8 parallel instruction paths (one per thread). For each thread a single instruction is fetched per cycle, from its private instruction cache. For each step of a vector instruction, the path from its decoding to its execution is separated in two phases: the *issue* and the execution *start*. The first step is issued with the vector instruction (the vector instruction issue allocates a reservation station). Each time a step is started, the next one is issued, until the length has been reached.

To control the different lengths pipelines, we implement a simplified version of the Tomasulo's algorithm [6]. In the superscalar implementation of the Tomasulo's algorithm, each pipelined computing unit has a set of reservation stations attached to it. Each cycle, instructions to be issued must be routed to free reservation stations in the set of those belonging to the needed computing units. In a multithreaded design, the number of instructions competing for station allocation is linearly proportional to the number of threads. Moreover, the number of computing units is slightly larger than in a single threaded design (however, thanks to thread sharing, it is less than linearly proportional to the number of threads). Hence, the number of reservation stations is also larger. This means

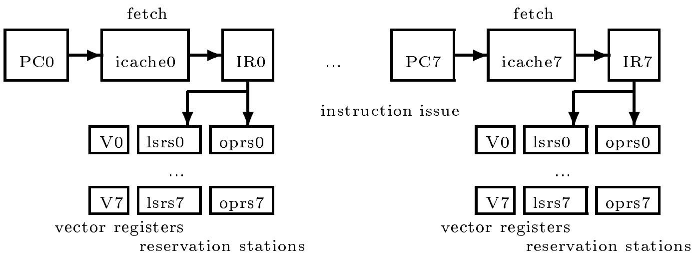

Fig. 3. The fetch, and instruction issue pipeline stages

that the routing of issued instructions to their allocated stations must be properly organized in order not to widen the processor cycle. For this reason, we attach the reservation stations to the vector registers instead of the computing units. Each register has two stations: one for loads and stores (*lsrs* on Fig. 3) and another for any inter-register operation (*oprs* on Fig. 3). Then, instruction issue may be handled in parallel for all the threads.

However, the routing problem has just been moved from the issue phase to the step start one. The step start logic is pipelined (see Fig. 4), with each thread having a private start cycle. This pipeline is composed of two sets of operand buffers, one for load/stores steps (one buffer per output bus) and one for the computing units (one buffer per unit). During its private start cycle, each thread adds in free buffers its ready to start steps operands to those gathered in the preceding cycles from higher priority threads. When the last start cycle is through, all the gathered steps start their execution simultaneously.

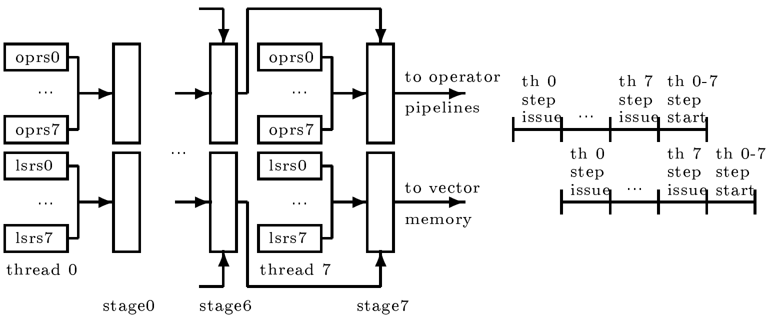

Fig. 4. The step start pipeline stages

A consequence of attaching the reservation stations to the vector registers is that when two instructions modify the same register (e.g. S1=S2+S3; S1=S1*2), the second one is not issued before the first one has leaved its *oprs* reservation station. For vector instructions, this means that the second vector operation has to wait until the first one has ended. A way around this problem is to use a different vector register for the second computation (i.e. V1=V2+V3; V4=V1*2). If the second instruction is the same as the first one because it is the next iteration in a loop (e.g. loop: V1=V2+V3; ... ; if (cond) goto loop), its issue will be delayed until the first instruction has left the reservation station, i.e. when its last step has been started. It seems that a pool of reservation stations intended to be shared by all the instructions would perform better. It would allow next iteration to be started as soon as all the instructions steps in the loop, that read the V1 first entry written by V1=V2+V3, have been started (then, the V1 first entry may be overloaded). But then, all the reservation stations would be rapidly saturated by very dependent vector instructions coming from different loop iterations, making multithreading useless. Vector instructions coming from multiple threads tend to use various resources but vector instructions coming from different iterations in a single thread do not. So, we have to choose between single thread and multiple threads instruction parallelism.

To simplify the connection between the computing units pipelines outputs and the vector registers, some more modifications to the Tomasulo's algorithm are added. Each vector register is assigned two write ports ($wp1$ and $wp2$ on Fig. 5; one write port for the $lsrs$ station and another one for the $oprs$ station) and two read ports ($rp1$ and $rp2$ on Fig. 5). The write ports are composed of a buffer and a pointer to one of the vector register entries. An incoming result, that goes along with its vector register number and register entry number, is first written to the buffer and then (next cycle) copied in the register entry.

Two result busses are provided for each thread. The thread-i $ls[i]$ bus is distributed to the 8 thread-i $lsrs$ write ports and the $op[i]$ bus feeds the 8 thread-i $oprs$ write ports. For each thread, its ls bus comes from the selection of the vector memory input data busses. It is required that at most one load may be started per thread per cycle, so that at most one memory bus delivers a data per thread per cycle (no conflict is possible on any ls bus). Conflicts may occur for accessing an op bus. If two results for the same thread are delivered in the same cycle, they are serialized on their op bus. The delayed results freeze their operator pipeline. Thus, only two writes may be performed per thread per cycle. This greatly favours multithreading and it also greatly simplifies the connection between the functional units and the vector registers.

When an instruction is issued, read ports are dynamically allocated to its reservation station (e.g. V1=V2+V3: one read port in vector register V2 and another in vector register V3 are allocated; V1 $oprs$ reservation station points to V2 and V3 ports). Figure 5 shows the stations and ports connections established during a SAXPY loop execution (only the four involved registers are presented: V0=M(Sx); V1=M(Sy); V2=V0*Sz; V3=V1+V2; scalar values coming from constants or from scalar registers are read when the instruction is issued and placed in the allocated reservation station; they do not consume read ports).

The read ports pointers are updated each time the reading instruction issues a new step. The write ports pointers are fixed by the incoming results. Their write buffers serve both to simplify the vector register write access pattern and to allow an easy propagation of an incoming result to any of the two read ports.

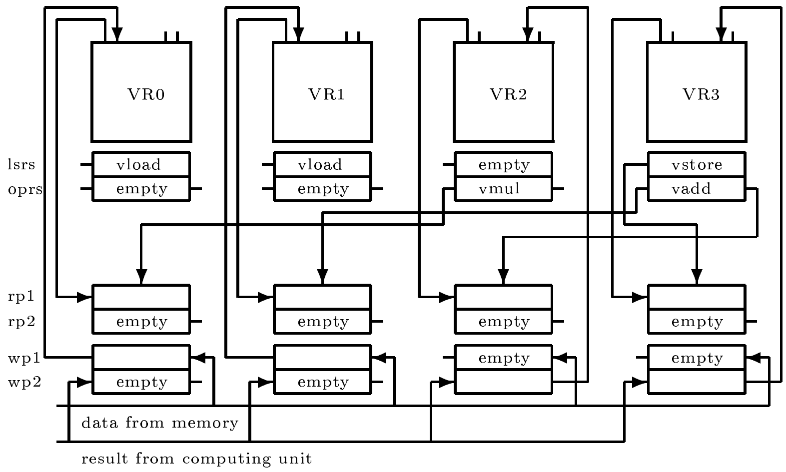

Fig. 5. The reservation stations and read ports allocation for a SAXPY run

No renamed registers are provided in our design. They are replaced by a single state bit indicating whether a register entry is *up-to-date* or *being-computed*. The state bit is set to *being-computed* as soon as an instruction writing the entry is issued. It is reset when the result is written. It may not be set if it is not in the reset state. Hence, a step may not be issued if its destination state bit is set. This prevents write-after-write hazards. For write-after-read hazards, one more precaution is necessary: a step may not be issued if its destination is requested by any of the two read ports (it means that the writing vector instruction has caught the reading one; it must not go beyond it). To prevent read-after-write hazards, a step may not be started if one of its sources is *being-computed*.

A reservation station and its linked read ports are released when the last step of the vector instruction it contains has been issued.

Branch instructions stall the fetch until the branch condition is known. Such instructions occur mainly at the end of vector loops. In the loop case, the next instruction to issue after the branch is the loop first vector instruction, starting next iteration. It is conflicting with the same instruction issued at the beginning of the current iteration if it is still running. In such a case, fetch stall should not cost any cycle. Moreover, during fetch stall, other threads continue fetching and issuing instructions.

Vector recurrences *lock* their destination vector register entries to prevent read-after-write hazards. More precisely, the lock is a counter. Each time a step s is issued (step $s-1$ is started), the step $s+1$ destination entry counter is incremented. It is decremented when the step $s+1$ is started. When the last vector entry before the vector recurrence starts is initialized, the initializing instruction that is marked as part of a recurrence (*lock* bit in the opcode) locks the next vector entry. This lock will be removed by the vector recurrence instruction first step start. For example:

V0[0]lock=x; locks R[0][1]	computes	A[0]=x;B[0]=y;
V1[0]lock=y; locks R[1][1]		for (i=1;i<65;i++){
V0[1]lock=V2[1]*V1[0],64;		A[i]=C[i]*B[i-1];
V1[1]lock=V3[1]+V0[0],64;		B[i]=D[i]+A[i-1];}

Any step that has to read or write a locked entry, except if it belongs to the lock owner instruction, is not started until the lock is removed (but it may be issued if its destination is not *being-computed*). First-order linear recurrences do not need the locking mechanism (e.g. V0[0]=0;V0[1]=1;V0[2]=V0[0]+V0[1],98 computes the Fibonacci values F(0) to F(99) without locking).

Reduction operations use the same locking technique. However, in a reduction the destination entry is fixed for all the steps. Hence, each step issue increments the fixed destination lock counter and in the same cycle, each step start decrements it. The counter is set to 1 when the reduction destination is initialized. The vector reduction instruction issue sets it to 2 (this is why it is a counter and not a single bit) and then, it remains invariant until the last step is issued. This last issue fixes the lock counter to 1. The last step start decrements it to 0 and the lock is removed.

With these features, almost any code is vectorizable and multithreading helps to fill the pipelines empty slots produced by reductions and recurrences dependencies.

4 Some performance hints

We are presently building a simulator of the vector co-processor. Yet, we have experimented it on the hand-coded SAXPY loop. Table 3 left part gives the floating point operations per cycle (FPC) rate achieved for 3600 entries vectors. The simulated co-processor had 4 adders, 2 multipliers and 8 memory accesses (with 8 half word data busses, i.e. a total of 256 data bits). Two sets of latencies were applied: a *slow* one with 8 cycles for the adder, 10 cycles for the multiplier and 28 cycles for the memory; a *fast* one with 3 cycles for the adder, 4 cycles for the multiplier and 8 cycles for the memory. The *slow* memory latency applies to actual multi-banks SDRAM chips and the *fast* memory latency corresponds to SRAM chips.

The results show that the performance increases quite linearly with the number of threads. Moreover, if we compare the *slow* result column with the *fast* one, we can see that multithreading works well even for long latency operators. The

SAXPY						vector sum					
threads	slow	fast	threads	slow	fast	threads	slow	fast	threads	slow	fast
1	0.25	0.25	5	1.04	1.21	1	0.06	0.09	5	0.31	0.45
2	0.49	0.50	6	1.33	1.39	2	0.12	0.18	6	0.37	0.54
3	0.73	0.74	7	1.40	1.55	3	0.19	0.27	7	0.43	0.63
4	0.95	0.98	8	1.53	1.74	4	0.25	0.36	8	0.50	0.72

Table 3. SAXPY and vector sum simulation results

co-processor could be placed on a PCI-bus compatible expansion board with cheap SDRAM components giving a low cost vector processing facility that can be used in a personal computer.

We have also tested the effects of multithreading on reductions. Table 3 right part gives the FPC performance achieved for the sum of 8000 entries of a vector (in the multithreaded case, each thread would compute an equal share of the total job and save its result in memory; the final sum timing concerning up to 8 terms is not included in the measure). We can see that however the performance is lower than for SAXPY, the linear progression still holds.

References

1. B. Bhandarkar and R. Brunner: Vector extensions to the VAX architecture. CompCon spring, 1990, pp120-126.
2. T. Chiueh: Multithreaded vectorization. 18th ISCA, 1991, pp 352-361.
3. B. Goossens and D.T. Vu: Multithreading to improve cycle width and CPI in superpipelined superscalar processors. 2nd IEEE ISPAN symposium, 1996, pp36-42.
4. N.P. Jouppi, J. Bertoni and D.W. Wall: A unified vector/scalar floating-point architecture. 3rd ASPLOS, 1989, pp134-143.
5. A. Padegs *et al.*: The IBM system/370 vector architecture: design considerations. IEEE transactions on computer, vol. 37-5, 1988, pp509-520.
6. R.M. Tomasulo: An efficient algorithm for exploiting multiple arithmetic units. IBM J. of Res. and Dev., vol. 11-1, 1967, pp25-33.
7. D.M. Tullsen, S.J. Eggers and H.M. Levy: Simultaneous multithreading: maximizing on-chip parallelism. 22nd ISCA, 1995, pp392-403.