PREDICTING THE SUSTAINED PERFORMANCE OF VECTOR PROCESSORS (*Preliminary Version*)

W. J. Paul. T. Rauber. U. Reeder

Informatik. Universität des Saarlandes
D-6600 Saarbrücken West Germany

Abstract: **A simple abstract vector processor is defined. The Livermore Loops, a common benchmark program consisting of 24 inner loops (kernels) of scientific code, are programmed on this machine. The flow of control in this benchmark program does not depend heavily on the input data. Therefore one can count quite accurately, how often each type of instruction of the abstract machine is used in each kernel. The results are listed in this paper. From this one can get good upper bounds for the runtime of real machines on the Livermore Loops. One only has to determine for each type of instruction of the abstract machine the time it takes to simulate it on the real machine. Some concrete processors are used as examples.**

1. Introduction. In [M81] one finds the following summary of a common problem faced by computer architects: "One factor that is usually of utmost concern to the architect is performance. ... Dealing with the execution efficiencies of of architectures (e.g. one wishes to compare a hypothetical architecture A with an existing system of architecture B) is not straightforward because of the following:

1. One cannot run benchmark programs, because architecture A exists only in thought.
2. One could build a simulation model (e.g. software simulator) ...
3. One could build a hardware implementation ...
4. The methods above are extremely costly and time consuming ways of obtaining data..."

What alternative ways are there to predict the performance of architecture A on benchmark programs while architecture A exists only in thought (or on paper) ? The first observation is, that the computer architect is clearly not interested in the *results* of the benchmark program when executed on architecture A (they are hopefully the same as those on an existing architecture) but only in the *runtime* of the program.

Unfortunately determining if an *arbitrary* program p with input data d on a general purpose architecture A terminates in t steps is a problem that is provably computationally hard. If you want to solve this problem on architecture B, and architecture A is powerful enough to simulate each instruction of B with a constant number of its own instructions (if it isn't forget architecture A), then this problem cannot be solved on architecture B with asymptotically less that t steps (it is exactly this problem which is used to establish the time hierarchy of complexity theory [HS65]. Thus determining the runtime of an *arbitrary* programs p by simulation of a run of program p is provably asymptotically the best one can do.

On the other hand there are plenty of programs p whose runtime one *can* predict without ever running them. Suppose p consists of a single loop which is executed n times , in this loop instructions $I_1, ..., I_s$ occur, each instruction I_j occurs a_j times, execution of I_j takes time t_j and determining if the loop has to be executed another time takes time s, then the runtime of p is $T = n * (s + \sum a_j t_j)$. If p is nested into another loop which is executed m times, then the runtime of the new program is $m * (s + T)$. Much more elaborate techniques than these are used in the theory of efficient algorithms in order to analyse the runtime of short model programs [AHU74].

What about real programs? Surprisingly enough the runtime of quite a few programs for numerical applications can be analysed very accurately in the most straightforward way. Examples include molecular dynamics code used in the Statistical Physics Project at the IBM Almaden Research Center in San Jose and the Livermore Loops, a well known benchmark program [McM84].

We can conclude that in addition to building a simulation model and building a prototype there is a third way of obtaining performance data about an architecture that exists only on paper. It has the advantage that one does not need programmers or hardware designers in order to use it. It shares with the other methods the disadvan-

tage of being time consuming. It is also utterly tedious.

There is yet another way to build a simulation model for an architecture A that exists only on paper. Suppose architecture B has an instruction set consisting of instructions $I_1, ..., I_s$. Suppose for each j architecture B needs r_j cycles to execute instruction I_j. Finally suppose that during execution of benchmark program p on architecture B for each j the relative frequency of instruction I_j is α_j and that alltogether L instructions are executed. If c_B is the cycle time of (a hardware realisation of) architecture B, then B will execute program p in time

$$t_B = c_B*L*(\sum \alpha_j r_j).$$

Now suppose for each j architecture A can simulate instruction I_j of B with s_j cycles. If c_A is the cycle time of (a planned hardware realisation of) architecture A, then A can execute program p in time

$$t_A = c_A*L*(\sum \alpha_j s_j).$$

Hence

$$t_A = t_B * (c_A/ c_B) * (\sum \alpha_j s_j)/(\sum \alpha_j r_j).$$

This is of course the way in which one predicts the performance of a RISC processor A on a benchmark with known instruction mix in terms of the performance of the processor B which has the original instruction set.

In this paper we combine the last two methods in order to develop a simple way of predicting the performance of processors on the Livermore Loops. In section 2 we define an *abstract vector machine* B. In section 3 we show for kernel 1 of the Livermore Loops how to program it on the abstract machine and how to analyse the runtime of B on these kernels. A complete analysis can be found in [Re88]. In section 4 we list the results of a similar analysis for most of the kernels. We list for each kernel i (with the exception of i = 16) and for each type of instruction I_j of B the number a_{ij} of times instruction I_j is executed when B executes our program for kernel i.

Now suppose A is a real machine. Then one way to implement the Livermore Loops for machine A is simply to simulate our programs for machine B instruction by instruction (a compiler which does this would of course have to know the methods we use for vectorizing the loops). If for each j machine A can simulate instruction I_j of B in time t_j , then for each kernel i the simulation of machine B by machine A will run for this kernel in time $\sum a_{ij} t_j$.

In order to give an impression of the accuracy of the predictions which can be made in this way we will in section 5 simulate abstract machine B on a CRAY -1 and to compare the predictions with the published performance measurements. Because machine B will not be a pure vector machine (it has RAM instead of vector registers) some extra information will be gathered during analysis of the various kernels in situations where a vector is accessed several times. but each time with a different offset relative to other vectors (in a RAM this amounts only to a change of a base address, in a true vector machine the vector must be shifted by the offsets within the vector registers).

In section 6 we give a brief overview over the SPARK project, a parallel computation project which is conducted jointly between the IBM Almaden Research Center in San Jose and the University of Saarbrücken. We present the results of performance analyis with the method of this paper for the processor developped in this project.

2. Definition of an abstract vector machine. We define a machine B with 6 units which communicate via a common bus (see fig. 1). The units are BU (branch unit), XPU (fixed point unit), FPU (floating point unit), DMA (direct memory access). MM (main memory) and I/O (input/output). The fixed point unit XPU and the floating point unit FPU generate condition codes XCC(<,=,>) and FCC(<,=,>) which belong to the last numbers computed in these units . The branch unit BU can perform jumps which depend on these condition codes.

An instruction I of B consists of 2 parts (IB, IZ) where IB is an instruction for the branch unit and IZ is an instruction for one of the units XPU, FPU, DMA or I/O. Instruction I is executed in the following way: with the condition codes computed in the previous instruction BU executes part IB of instruction I, i.e. it fetches a new instruction. The unit corresponding to IZ executes part IZ of instruction I; if IZ is an instruction for the XPU resp. the FPU, then a new condition code XCC resp. FCC is computed. In a sense this abstract machine has an instruction pipe of depth 2.

2.1. The **branch unit** consists of a sequencer and a memory BM, which holds the program for machine B or parts thereof (see fig. 2.a). The sequencer contains an instruction counter IC. The branch unit can perform the following instructions:

continue: $IC_{new} := IC_{old} + 1$;

fetch instruction IC_{new}

jump (Z. cond. i) where Z ∈ {XPU, FPU}, cond ∈ {<,≤,=,≥,>,≠} and i is an address in BM:

IC_{new} := i if condition cond holds for the condition code selected by Z

IC_{old} + 1 otherwise;

fetch instruction IC_{new}

2.2 The **fixed point unit** consists of a fixed point processor and a memory XM capable of storing fixed point numbers (see fig. 2.b). For all i we denote by XM(i) the fixed point number stored at address i of XM.The fixed point processor has 2 registers: an accumulator ACC and an index register IR. The fixed point unit can perform the following operations:

load# i	ACC := i
load i	ACC := XM(i)
loarir i	ACC := XM(IR+i)
loadir+ i	ACC := XM(IR+i); IR := IR + 1
irload# i	IR := i
add i	ACC := ACC + XM(i)
sub i	ACC := ACC - XM(i)
and i	ACC := ACC ∧ XM(i) bitwise
shr i	ACC := ACC shifted right by i bits
shl i	ACC := ACC shifted left by i bits
store i	XM(i) := ACC
storeir i	XM(IR+i) := ACC
storeir+ i	XM(IR+i) := ACC; IR := IR+1
load ir	ACC := IR
store ir	IR := ACC

2.3. The **floating point unit** consists of a pipelined floating point arithmetic/logic unit FALU of depth d, a memory FM capable of storing floating point numbers and fixed point numbers, a vector index memory VIM capable of storing indices of vectors stored in FM, a vector mask memory capable of storing mask bits and 4 address generators AG1, ..., AG4 (see figure 2.c). For all i we denote by FM(i) the number stored at address i of FM, by VIM(i) the index stored at address i of VIM and by VMM(i) the mask bit stored at address i of VMM.

The floating point unit performs only (generalized) vector operations. Each vector operation has a *lenght* n. We allow n = -1, which means, that the lenghth of the operation is determined by the accumulator ACC of the XPU. Scalar operations must be specified as vector

Fig. 1

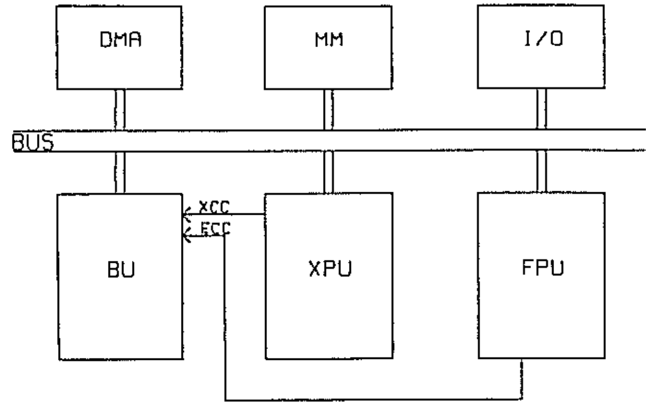
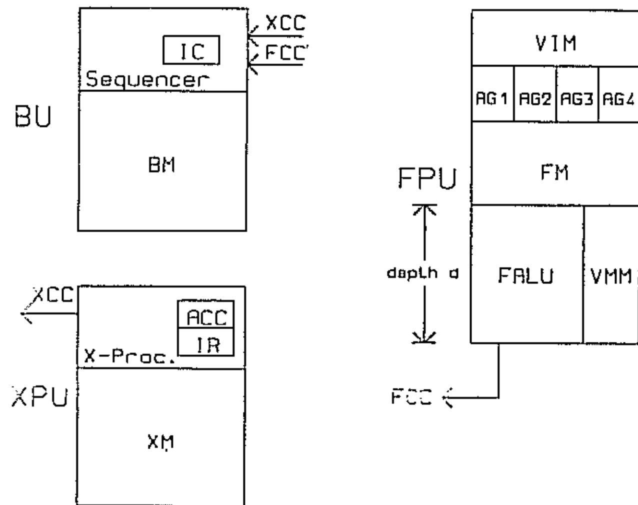


Fig.2



operations of length 1. The condition code FCC which is set after the execution of an instruction refers to the last component of the vector computed in that instruction (in this abstract model FCC is not of much use unless the machine works in scalar mode).

The addresses for the (generalized) vector operations of the FPU are provided by the address generators. Between 2 and 4 address generators are involved in each vector operation. Each vector operation of the FPU can be specified in the form (FOP, n, IAG1, ..., IAGs) where $s \in \{2, 3, 4\}$. FOP specifies the operation to be performed, n is the length of the operation and for each i IAGi is an instruction for address generator AGi. The instructions IAGi have the form (mi, bi, si) where mi specifies the *mode*, bi the *base address* and si the *stride* for address generator AGi.

During a vector operation of length n each address generator AGi involved in that operation produces as a function of IAGi a stream of addresses ai = (ai[1], ..., ai[n]):

mi = f (fixed): ai = (bi, ..., bi)

mi = v (vector): ai = (bi, bi+ 1, ..., bi+n-1)

mi = cs (constant stride) ai = (bi, bi+ si, ..., bi +(n-1)si)

mi= i(indexed): ai = (VIM(bi), VIM(bi +1),..., VIM(bi +n-1))

For each i we allow bi = -1 in which case the value of XM(i) is used.

Addressing modes f and v are called *standard* addressing modes. If two or more address generators work in indexed addressing mode, then they all must use the same base address b. This is a restriction in generality, but more flexibility would not help here, it would only make it harder to simulate the abstract machine B on concrete machines A.

2.3.1 **Simple 3-address operations** (FOP, n, IAG1, IAG2, IAG3)

fadd:
FM(a3[i]) := FM(a1[i]) + FM(a2[i]) for all $i \in \{1, ..., n\}$
fsub, fmul, fdiv: similarly

If AG2 works in fixed mode and FM(b2) = 0 the FALU simply passes FM(a[i]). If AG1 works in vector mode and AG3 works in indexed mode a scatter operation is performed. If AG1 works in indexed mode and AG3 works in vector mode a gather operation is performed.

2.3.2 **operations which compute masks or index vectors** (FOP, n, IAG1, IAG2, IAG3):

for each $rel \in \{ <, \leq, =, \geq, \neq \}$:

$madd_{rel}$ VMM(a3[i]) := 1 if FM(a1[i]) + FM(a2[i]) rel 0
0 otherwise

$msub_{rel}$, $mmul_{rel}$, $mdiv_{rel}$ similarly

$iadd_{rel}$
```
L:=0;
for i=1 step 1 to n do
if FM(a1[i]) + FM[a2[i]] rel 0
        then VIM[b3+L] := i;
        L:=L+1;
        fi
    od;
ACC := L
```

$isub_{rel}$, $imul_{rel}$, $idiv_{rel}$ similarly

Addressing modes of AG1, AG2 and AG3 must be standard.

2.3.3. **merge by mask**: (mrg, n, IAG1, IAG2, IAG3, IAG4)

FM(a3[i]) := FM(a1[i]) if VMM(a4[i]) = 1
FM(a2[i]) otherwise

Addressing modes standard

2.3.4. **vector sum** (vs, n, IAG1, IAG3)
FM(b3) := $\sum$ FM(a1[i])

Addressing mode of AG3 constant

2.3.5. **scalar product** (sp, n, IAG1, IAG2, IAG3)
FM(b3) := $\sum$ FM(a1[i]) * FM(a2[i])

Addressing mode of AG3 constant

2.3.6. **multiply and add** (ma, n, IAG1, IAG2, IAG3, IAG4)
FM(a3[i]) := FM(a1[i]) * FM(a2[i]) + FM(a4[i])

This is a special case of chaining.

2.3.7. **convert** (FOP, n, IAG1, IAG3)

xtof FM(a3[i]) := FM(a1[i])
conversion of fixed point to floating point

ftox similarly

2.3.8. **fnoop**. The FPU works in internal cycles. Each of the address generators generates one address per cycle. Generally storing to FM(a3[i]) is done $d_{op} \leq d$ cycles

after reading from FM(a1[i]) and FM(a2[i]) is done where d_{op} depends on the operation which is being performed. Generally d_{op} = d. except during chaining. when it is done $d_* + d_+ \approx 2d$ cycles later. Operation fnoop lasts 1 cycle. It starts no new floating point operations. It must be used when one waits for the partial results of vector operations op whose computation was started less than d_{op} cycles ago.

2.4. The DMA controls moves of blocks of data between the various memories of the abstract machine B. Instructions for the DMA come in 2 flavours:

2.4.1. dmove(n. scr. dest. bscr. bdest)

Here src (source) and dest (destination) specify any of the memories of the machine. This instruction moves n data stored at locations bsrc. .... bsrc + n - 1 of the source memory to locations bdest. .... bdest + n -1 of the destination memory. We allow n = -1. which means that n is determined by ACC. We allow bsrc = -1 resp. bdest = -1 which means that bsrc resp. bdest are determined by XM(5) resp. XM(6).

2.4.2 imove (n. src. dest. bf. IAGi)

imove(n. MM. FM. bf. IAGi) gathers data from locations ai[1]. .... ai[n] of MM into locations bf. .... bf+n-1 of FM.

imove(n. FM. MM. bf. IAGi) scatters data from locations bf. .... bf+n-1 of FM into locations ai[1]. .... ai[n] of MM.

This concludes the definition of the abstract machine B. It is not completely defined (what happens if the DMA moves data from FM to VMM ?. what about I/O ?). It is much messier than abstract machines from theoretical computer science. It is also much less messy than a real machine. Like machines from theory it is not meant to be built but as a vehicle of analysis. It is still in preliminary form. but it already allows to make predictions which are rarely off by more than 25 %.

3. Analysis of kernel 1.

```
DO 1 L = 1. LP
DO 1 K = 1. N
1     X(K) = Q + Y(K) * (R * Z( K+10 ) + T * Z( K+11))
```

Generally variable LP indicates how often the kernel is executed during a run of the benchmark program. Large values of LP give the rates of MFLOPS if the kernels are encached. The case LP = 1 is most sensitive to the bandwidth of the bus between FM and MM. The analysis presented here is only for this case.

Variable N indicates vector length and is initialised outside of the kernels to some ugly value. The value for kernel 1 happens to be 1001.

We describe vector operations within the FPU in a programming language which is a vector extension of PASCAL. The notation is rather obvious as is the translation into instructions of machine B.

i) Load constants Q. R and T from MM into FM (3 dmoves of length 1).

ii) Load Z[11 : 1012] and Y[1 : 1001] from MM into FM (2 dmoves of lengths 1002 resp 1001).

iii) ZWSP[1:N] := R*Z[11: N+10]
(vector operation fmul of length 1001. all addressing modes std).

iv) X[1:N] := T*Z[12: N+11] + ZWSP[1: N]
(chaining. length 1001. addressing modes std). We observe that in a machine with vector registers shifting vector Z within the vector registers by an offset of 1 would be required between operations iii) and iv).

v) X[1:N] := Y[1:N] * X[1:N] + Q
(chaining, length 1001. addressing modes std).

vi) move X from FM to MM
(dmove of length 1001).

In a real machine FM (or the part of the machine which simulates FM) is not arbitrarily large. and vectors of length 1001 may very well be too large to fit. The vector registers of a CRAY-1 for example have only length 64. Thus moves of long vectors between MM and FM have to be broken into moves of smaller vectors. There might also be problems with storing intermediate results. In order to get a handle on these problems we associate with each of our programs for machine B the following parameters:

s: the maximal number of scalars (input. intermediate or result) held at any time in memory FM. In the example above s equals 3.

vr: the maximal number of vectors (input, intermediate or result) held at any time in memory FM. In the example above vr equals 4.

bl: the largest number β such that for all result vectors X and for all indices i component X[i] is computed only from components Z[j] of input vectors or intermediate

vectors Z with index $j = i + \beta$. Input and output is seen with respect to FM, i.e. an input vector is one that is loaded from MM into FM. An output vector is one that is written from FM to MM. In the above example bl = 0.

bu: the smallest number γ such that for all result vectors X and for all i component X[i] is computed from components Z[j] with $j \leq i + \gamma$. In the example above bu equals 11.

b: bu - bl. In the example above b equals 11.

sh: a vector of integers indicating the offsets of shifts which would be necessary in a pure vector machine. In the example above sh equals (1).

The results of this analysis as well as results of similar analysis for the other kernels (except kernel 16) are summarized in section 4.

4. Summary of performance analysis for the abstract machine.

If in any kernel nothing is said about b and sh, then there is no shift and b equals zero. If b but not sh is given then sh = (b).

Kernel 1

fixed point 0
branches 0
floating point (total 5005)

#	FOP	n	IAG_1	IAG_2	IAG_3	IAG_4
2	ma	1001	std	std	std	std
1	mul	1001	std	std	std	

DMA

#	n	src	dst	m_{src}	m_{dst}
1	1002	mm	fm	std	std
1	1001	mm	fm	std	std
1	1001	fm	mm	std	std
3	1	mm	fm		

s = 2, vr = 3, b = 1

Kernel 2

This kernel can be programmed in two different ways:
fixed point 53
branches 30
a)
floating point (total 388)

#	FOP	n	IAG_1	IAG_2	IAG_3	IAG_4
1	ma	i	cs	cs	std	
1	ma	j	cs	cs	std	
6	mul	1				
6	add	1				

one operation for each $i \in \{50, 25, 12, 6, 3, 1\}$
$j \in \{49, 24, 11, 5, 2\}$
if $j < d$ and in scalar operations wait for the results

DMA

#	n	src	dst	m_{src}	m_{dst}
1	101	mm	fm	std	std
1	198	mm	fm	std	std
1	97	fm	mm	std	std

b = 101, stride = 2

b)
floating point (total 388)

#	FOP	n	IAG_1	IAG_2	IAG_3	IAG_4
1	ma	i	std	std	std	
1	ma	j	std	std	std	
6	mul	1				
6	add	1				

one operation for each $i \in \{50, 25, 12, 6, 3, 1\}$
$j \in \{49, 24, 11, 5, 2\}$
if $j < d$ and in scalar operations wait for the results

DMA

#	n	src	dst	m_{src}	m_{dst}	
4	i	mm	fm	cs	std	$\forall i \in \{50,25,12,6,3,1\}$
1	i	fm	mm	std	std	$\forall i \in \{50,25,12,6,3,1\}$

b = 0, stride = 2
in both cases s = 0, vr = 3

Kernel 3

fixed point 0
branches 0
floating point (total 2001)

#	FOP	n	IAG_1	IAG_2	IAG_3	IAG_4
1	sp	1001	std	std	std	

DMA

#	n	src	dst	m_{src}	m_{dst}
2	1001	mm	fm	std	std
1	1	fm	mm		

s = 1, vr = 2

Kernel 4

fixed point 12
branches 3
floating point (total 1194)

#	FOP	n	IAG_1	IAG_2	IAG_3	IAG_4
3	sp	5	std	std		
3	sp	194	std	std		
6	add	1				

DMA

#	n	src	dst	m_{src}	m_{dst}
3	199	mm	fm	std	std
1	199	mm	fm	cs	std
3	1	fm	mm		

stride = 5
s = 1. vr = 2

Kernel 5

fixed point 0
branches 1001 (tests of a for loop)
floating point (total 2000)

#	FOP	n	IAG_1	IAG_2	IAG_3	IAG_4
1000	add	1				
1000	mul	1				

DMA

#	n	src	dst	m_{src}	m_{dst}
1	1	mm	fm		
2	1000	mm	fm	std	std
1	1000	fm	mm	std	std

s = 1. vr = 3

Kernel 6

fixed point 196
branches 64 (tests of a for loop)
floating point (total 3843)

#	FOP	n	IAG_1	IAG_2	IAG_3	IAG_4
1	sp	i	std	cs		
63	add	1				

one scalar product for each $i \in \{1, \ldots, 63\}$

DMA

#	n	src	dst	m_{src}	m_{dst}
1	i	mm	fm	std	std
1	64	mm	fm	std	std
1	63	fm	mm	std	std

stride = -1
s = 1. vr = 2

Kernel 7

fixed point 0
branches 0
floating point (total 15920)

#	FOP	n	IAG_1	IAG_2	IAG_3	IAG_4
8	ma	998	std	std	std	std

DMA

#	n	src	dst	m_{src}	m_{dst}
1	1001	mm	fm	std	std
2	995	mm	fm	std	std
2	1	mm	fm		
1	995	fm	mm	std	std

b = 6, sh = (1, 1, -5, 1, 1, -3)
s = 2, vr = 4

Kernel 8

fixed point 31
branches 2
floating point (total 7128)

#	FOP	n	IAG_1	IAG_2	IAG_3	IAG_4
12	add	99	std	std	std	
6	mul	99	std	std	std	
12	ma	99	std	std	std	std
12	ma	99	std	std	std	std

DMA

#	n	src	dst	m_{src}	m_{dst}
12	101	mm	fm	cs	std
9	1	mm	fm		
6	99	fm	mm	std	cs
3	99	fm	mm	std	std

b = 2. stride = 5
s = 2. vr = 6

Kernel 9

fixed point 0
branches 0
floating point (total 1717)

#	FOP	n	IAG_1	IAG_2	IAG_3	IAG_4
1	add	101	std	std	std	
8	ma	101	std	std	std	std

DMA

#	n	src	dst	m_{src}	m_{dst}
10	101	mm	fm	cs	std
8	1	mm	fm		
1	101	fm	mm	std	cs

stride = 25
s = 1, vr = 3

Kernel 10

fixed point 0
branches 0
floating point (total 909)

#	FOP	n	IAG_1	IAG_2	IAG_3	IAG_4
9	add	101	std	std	std	

DMA

#	n	src	dst	m_{src}	m_{dst}
1	101	mm	fm	cs	std
1	909	mm	fm	cs	std
1	1010	fm	mm	std	cs

stride = 25
s = 0. vr = 2

Kernel 11

fixed point 0
branches 1000 (tests of a for loop)
floating point (total 1000)

#	FOP	n	IAG_1	IAG_2	IAG_3	IAG_4
1000	add	1	std	std	std	

You'll always have to wait for the results.

DMA

#	n	src	dst	m_{src}	m_{dst}
1	1001	mm	fm	std	std
1	1000	fm	mm	std	std

s = 0. vr = 2

Kernel 12

fixed point 0
branches 0
floating point (total 1000)

#	FOP	n	IAG_1	IAG_2	IAG_3	IAG_4
1	add	1000	std	std	std	

DMA

#	n	src	dst	m_{src}	m_{dst}
1	1001	mm	fm	std	std
1	1001	fm	mm	std	std

b = 1
s = 0. vr = 1

Kernel 13

fixed point 960
branches 192 (tests of a for loop)
floating point (total 640)

#	FOP	n	IAG_1	IAG_2	IAG_3	IAG_4
7	add	64	std	std	std	
3	ftox	64	std		std	

DMA

#	n	src	dst	m_{src}	m_{dst}
4	64	mm	fm	cs	std
7	64	mm	fm	i	std
4	64	fm	mm	std	cs
1	64	fm	mm	std	i
4	64	fm	vim	std	std

stride = 4
s = 0. vr = 6

Kernel 14

fixed point 0
branches 0
floating point (total 16016)

#	FOP	n	IAG_1	IAG_2	IAG_3	IAG_4
7	add	1001	std	std	std	
2	add	1001	i	std	i	
1	ma	1001	std	std	std	std
2	ftox	1001	std		std	
3	xtof	1001	std		std	

DMA

#	n	src	dst	m_{src}	m_{dst}
4	1001	mm	fm	std	std
2	1001	mm	fm	i	std
9	1001	fm	mm	std	std
2	1001	fm	vim	std	std

s = 0. vr = 4

Kernel 15

fixed point 90
branches 5
floating point (total 10940)

#	FOP	n	IAG_1	IAG_2	IAG_3	IAG_4
5	$msub_<$	100	std	std		
5	$msub_<$	99	std	std		
15	$msub_<$	98	std	std		
25	mrg	98				
20	mrg	99				
5	mul	99	std	std	std	
5	mul	98	std	std	std	
5	ma	99	std	std	std	std
5	ma	98	std	std	std	std
5	div	99	std	std	std	
5	div	98	std	std	std	
5	move	1				
1	move	100	std		std	
5	sqrt	99	std		std	
5	sqrt	98	std		std	

b = 1

DMA

#	n	src	dst	m_{src}	m_{dst}
3	606	mm	fm	std	std
3	1	mm	fm		
2	505	fm	mm	std	std
1	101	fm	mm	std	std

or

#	n	src	dst	m_{src}	m_{dst}
18	101	mm	fm	std	std
3	1	mm	fm		
11	101	fm	mm	std	std

b = 1. there are shifts for three vectors
s = 3. vr = 8

Kernel 16

We do not understand this kernel.

Kernel 17

fixed point 0
branches a) 400
b) 200

floating point
a) total: 1100

#	FOP	n	IAG_1	IAG_2	IAG_3	IAG_4
400	move	1				
400	add	1				
350	mul	1				

b) total: 800

#	FOP	n	IAG_1	IAG_2	IAG_3	IAG_4
300	move	1				
350	add	1				
150	mul	1				

You'll always have to wait for the results.

DMA

#	n	src	dst	m_{src}	m_{dst}
301	1	mm	fm		
300	1	fm	mm		

s = 12. vr = 0

Kernel 18

fixed point 208
branches 15
floating point (total 21780)

#	FOP	n	IAG_1	IAG_2	IAG_3	IAG_4
80	add	99	std	std	std	
65	ma	99	std	std	std	std
10	div	99	std	std	std	

DMA

#	n	src	dst	m_{src}	m_{dst}
41	101	mm	fm	std	std
30	101	fm	mm	std	std

b = 2. sh = (1. 1)
s = 2. vr = 8

Kernel 19

fixed point 0
branches 202
floating point (total 606)

#	FOP	n	IAG_1	IAG_2	IAG_3	IAG_4
404	add	1				
202	mul	1				

You'll always have to wait for the results.

DMA

#	n	src	dst	m_{src}	m_{dst}
2	101	mm	fm	std	std
1	101	fm	mm	std	std

s = 2. vr = 2

Kernel 20

fixed point 0
branches a) 2000
b) 4000
herein 1000 tests of a for loop
floating point
a) total 17000

#	FOP	n	IAG_1	IAG_2	IAG_3	IAG_4	
6000	add	1					
4000	mul	1					
3000	div	1					
2000	add	1					(a) only
2000	move	1					(a) only

You'll always have to wait for the results.

DMA

#	n	src	dst	m_{src}	m_{dst}
6002	1	mm	fm	std	std
1000	1	fm	mm	std	std

s = 12. vr = 0

Kernel 21

fixed point 1309
branches 650 (tests of a for loop)
floating point (total 15625)

#	FOP	n	IAG_1	IAG_2	IAG_3	IAG_4
625	sp	25	std	cs		
1	add	625	std	std	std	

DMA

#	n	src	dst	m_{src}	m_{dst}
3	625	mm	fm	std	std
1	625	fm	mm	std	std

stride = 25
s = 0. vr = 3

Kernel 22

fixed point 0
branches 0
floating point (total 405)

#	FOP	n	IAG_1	IAG_2	IAG_3	IAG_4
1	add	101	std	std	std	
1	mul	101	std	std	std	
1	mul	1				
2	div	101	std	std	std	
1	exp	101	std		std	

DMA

#	n	src	dst	m_{src}	m_{dst}
3	101	mm	fm	std	std
1	1				
2	101	fm	mm	std	std

s = 1. vr = 3

Kernel 23
fixed point 4000
branches 500 (tests of a for loop)
floating point (total 5445)

#	FOP	n	IAG_1	IAG_2	IAG_3	IAG_4
15	ma	99	std	std	std	std
1485	add	1				
1485	mul	1				

You'll have to wait for the scalar results.
DMA

#	n	src	dst	m_{src}	m_{dst}
32	101	mm	fm	std	std
5	101	fm	mm	std	std

s = 2. vr = 4

Kernel 24
fixed point 0
branches 2000 (1000 tests of a for loop)
floating point (total 1000)

#	FOP	n	IAG_1	IAG_2	IAG_3	IAG_4
1000	add	1				

DMA

#	n	src	dst	m_{src}	m_{dst}
1	1001	mm	fm	std	std
1	1	mm	fm	std	std

5. Simulating abstract machine B on a CRAY-1.

This section is based on the description of the CRAY-1 computer in [HB84] and [Ru78]. Where these descriptions are incomplete we guess certain parameters. The section serves on one hand as an illustration how to get predictions for real machines from the above analysis. On the other hand the real performance figures of a CRAY-1 are known. Thus we will be able to compare predicted and real performance.

Simulation of abstract machine B is done in the following way:

Scalars stored in FM are stored in the scalar registers. This works because $s \leq 8$ in all kernels. Vectors stored in FM are stored in the vector registers. This works because $vr \leq 8$ in all kernels. Shifting a vector within the vector registers by a positive offset λ is treated like a vector operation on length λ (with a pipeline depth 0). Shifting by negative offsets is done by loading the vector again from MM.

The vector registers have length L = 64. Operations on vectors of length n > L in a program with parameter b = β are broken into $\mu = \lfloor n/(L-\beta) \rfloor$ chunks of length $L - \beta$. This concerns both operations within the FPU as well as moves of data between FM and MM. The penalty for not having vector registers of length n is $\mu - 1$ extra startup times for each such operation. Whenever two alternative ways to program a kernel are indicated in section 4 we choose the one with a small value of b.

We estimate time in units of 12.5 ns cycles.

Fixed point operations are very rare in the Livermore loops, so a good estimate is not important. We count 1 cycle per fixed point operation.

If a machine has an instruction pipe of depth di, then a branch operation other than continue may take between 1 cycle (if successful instruction prefetch has been done) and di cycles. The classical situation where prefetch is easy is to test for the end of a loop. We count 1 cycle for simulating a test performed in the branch unit of B.

Plain vector operations of length $n \geq 64$ are accounted for by s + n where s is a startup time (assumed to be 4 in operations other than chaining, where 2 + the depth of the first operand pipe is added).

We only account for pipe depth if it is indicated in the table of section 4 or if n = 1 (scalar mode). We use pipe depth 6 for addition, subtraction, 7 for multiplication, 13 for ma, 14 for 1/y and sqrt(y) and 21 for division (chaining 1/y with multiplication, startup time 4 + 2 + 14 = 18).

Scalar products are computed with the method described in [BH84], which is accounted for by 4 + 2 + 7 = 13 (startup using chaining) + n + 7*(4+6) (adding in scalar mode 8 partial sums) = 83 + n. Vector sum is a special case of scalar
product, but no chaining is necessary. We account for it by 74 + n.

Merging two vectors as indicatd by a mask vector is

done in two passes. We account for it by 2*(4 + n). Pipe depth is 0. Operations which compute mask bits are accounted for with 4 + n. Pipe depth is that of the arithmetic operation involved.

Vector operations of machine B which use addressing modes cs or i must are simulated in scalar mode. Depending on the operation performed they are accounted for by $(4+d_{op})*n$.

Simulating a dmove operation of length $n \leq 64$ is accounted for by s+n where s is assumed to be 4. An imove with addressing mode i must be handled as n moves of length 1. We account for this by n*(s+1) = 5n.

The main memory of the CRAY-1 is organized into groups each consisting of 16 banks. Cycle time of the main memory (1 m-cycle) is 50 ns = 4 cycles. Stepping through such a memory with constant stride st delivers or consumes data at a rate that depends only on σ = st mod 16. The worst case is $\sigma = 0$, in which case only 1 data item per m-cycle can be moved. Two data items per m-cycle can be moved if $\sigma = 8$. In all other cases four or more data items per m-cycle can be moved. More cannot be consumed or delivered by one vector register in 1 m-cycle. Adding startup time s = 0 we account for a move of n data items $4 + n*\tau(\sigma)$ where $\tau(0) = 4$, $\tau(8) = 2$ and $\tau(\sigma) = 1$ for all other values of σ.

Using this simulation for the operations of abstract machine B which are listet at the table in section 3 we get the following estimate for the number of cycles, in which a CRAY1 can run kernel 1:

2 * (13 + 1001) for ma
+ 4 + 1001 for mul
+ 3*4 + 2*1001 + 1002 for moves of vectors
+ 3*5 for scalar moves
= 6064 .

Shifting once by 1 costs 4 + 1 = 5.

Startup times for all vector operations (including shift and move) are t = 2*13 + 4 + 3*4 + 4 = 46. Parameter b = 1, thus $\mu \leq \lceil 1002/(64 - 1) \rceil = 16$. Penalty is $(\mu-1)*t = 690$. This gives a total execution time of 6754 cycles = 81 048 ns. The total number of floating point operations is 5005. Thus we have a sustained performance of $5005 / 81\,048 * 10^9 = 61\,753\,500$ floating point operations per second.

In table 1 we compare the MFLOPS rates obtained in this way with the values published in [McM84].

Table 1 : predicted MFlops for CRAY 1

Kernel	predicted MFlops	published MFlops
1	60.0	80.5 (+)
2	27.1	15.4 (*)
3	35.8	53.9 (+)
4	36.9	36.2
5	4.3	4.6
6	29.5	3.4 (*)
7	78.9	86.4
8	78.4	63.0
9	50.0	81.2
10	23.2	29.2
11	3.8	2.8
12	25.1	22.6
13	9.8	3.3 (*)
14	21.4	7.7
15	55.1	3.1 (*)
17	6.3	7.8
18	67.0	56.7
19	6.0	5.9
20	6.7	9.6
21	17.2	22.8
22	30.2	41.7
23	10.2	9.4
24	4.7	2.3

(*) prediction better than actual rate, because of vectorisation by hand (+) prediction worse than actual rate, because abstract machine B (in its present form) does not allow vector operation and moves between FM and MM to occur simultaneously

6. The SPARK Project. The roots of this work go back about 4 years. In the winter of 83/84 an effort was started at the IBM Research Lab in San Jose to build a massively parallel computer for molecular dynamics computations which (depending on the number of processors) was to be 10 to 1000 times faster than mainframes available at the time (IBM 3081-K) [A85]. Since 1986 a joint study on massively parallel computers for numeric applications is conducted between the IBM Almaden Research Center in San Jose and the University of Saarbrücken.

Matching on numerical computations the speed of a 3081-K with a design based on a microprocessor was out in 1984 and is tough today. Building a network capable of interconnecting many thousands of microcomputers is hard too. Thus the decision was made to develop a pipelined custom node processor based on avilable VLSI chips. In order to run molecular dynamics code particu-

larly well hardware support for scatter and gather operations was provided. The architecture of this processor (SPARK, San Jose Processor ARray Kernel) is sketched in [A85] and A[87].

Extensions for programming languages like FORTRAN or PASCAL were sketched, which raise the level of abstraction of these languages roughly to that of abstract machine B above. In particular a command was added which for arrays a and b computes in ascending order the sequence of indices i such that a[i] rel b[i] where rel is as in section 2.3.2. Using techniques similar to those of this paper the performance of this particular machine for some of the kernels of the Livermore Loops were predicted [MP84]. Results of an analysis of all kernels (except kernel 16) for this processor which were obtained by simulation of abstract machine B are listed in table 2. Work for a compiler for SPARK is currently being done both in San Jose and Saarbrücken.

Table 2 : predicted MFlops for SPARK

Kernel	predicted MFlops
1	4.6
2	2.7
3	3.9
4	3.2
5	0.6
6	2.0
7	5.3
8	5.1
9	4.3
10	2.8
11	0.6
12	3.0
13	1.5
14	3.8
15	2.1
17	0.9
18	5.0
19	0.7
20	0.4
21	1.5
22	0.8
23	1.0
24	0.6

A first prototype of SPARK with a peak performance of 6 MFLOPS on 32-bit floating point numbers was built and programmed in San Jose. It runs as backend of an IBM-AT. Molecular dynamics code (programmed in assembly language) is executad at 1.7 time the speed of a 3081-K [A87].

A second prototype of SPARK with a planned peak performance of 10 MFLOPS on 64-bit floating point numbers is currently being debugged in Saarbrücken. Sustained performance is almost exactly 10/6 times that of the older machine. It is slightly less than that, because in the new design the relative bandwidth of the bus is smaller than in the older machine (which results in a much simpler DMA). With the methods presented here it was literally found out overnight, that the penalty in performance for this would be small.

We now sketch the architecture of the massively parallel machine. We start with the user's view of a machine with P processors. At compile time the user defines a virtual communication network in form of a graph. The nodes of the graph are the available processors, and there is an edge from processor S to processor S' if at some time during the computation processor S sends data to S'. Good communication networks for scientific code are small collections of grids which are glued together (to cover nicely say the body and wings of an airplane).

The user generates a serial program ps and P parallel programs pp(S), one for each processor S. One processor (subsequently called *king*) executes the serial program in addition to his parallel program. The parallel programs are used like subroutines of the serial program, whose lokal variables retain their values when a subroutine is left and reentered subsequently. The serial program can access the local variables of the subroutines, both one by one or by a broadcast write or by a "broadcast read" (in which case the results from the various processors are OREd together).

An important restriction is that the parallel part of the communication proceeds in strict *compute-communicate*-rounds. During *compute* all processors compute. One waits until all are done. During *communicate* they exchange arrays of data via the links of the virtual communication network. This strict communication pattern does not lead to loss of efficiency in molecular dynamics simulations. Moreover it does not cause loss of efficiency in amazingly many further numerical applications for parallel machines. It has the advantage of leading to an extremely simple operating system and to startup times for interprocessor communication in the order of a few microseconds.

An operating system which supports this type of parallel programming was implemented in Saarbrücken on a system of 6 microcomputers that are interconnected by a crossbar. An extension for languages like PASCAL or FORTRAN which allows the user to define virtual networks and to define systems of programs which work in this way on massively parallel computers is sketched in [HP87]. A compiler for an extension of

PASCAL is under construction in Saarbrücken.

The overall structure of the hardware of the parallel machine is shown in figure 3. All processors communicate with each other via a Benes-permutation network [B65]. If the virtual communication network is defined by any graph of degree d, then communication along all edges of the virtual network can be simulated with d rounds of communication of the permutation network [GP83].

The king processor communicates with all other processors via a control bus. Communication from the king to the other processors on this bus is in broadcast mode. This allows to realize broadcast write and to give to all processors the start signal of a communication phase. Data sent from the processors to the king on this bus are ORed together. This allows to realize "broadcast read" and to determine the end of computation phases.

The king also tells the permutation network, when it is being reconfigured for a new permutation. Routing information is provided by all processors simultaneously via the data paths.

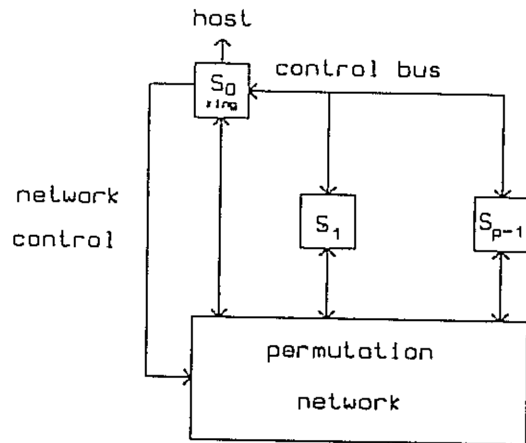
For small numbers of processors it makes sense to use a crossbar instead of a permutation network. The crossbar can comfortably be built out of PALs. A parallel computer consisting of 10 SPARKs interconnected in this way is under construction at the IBM Almaden Research Center in San Jose. The crossbar has 16 bit planes.

For large numbers of processors one uses a recursive construction. The elementary building blocks of permutation networks in the classical construction are 2x2-switches. This leads to a depth of $2 \log_2 P - 1$ for networks interconnecting P processors (bad for startup time) and to P times as many wires or traces on printed circuit boards per bit plane (bad for reliability). Fortunately the classical construction can be extended to work with arbitrary sxs-switches as basic building blocks [LPV81]. The depth then becomes $2 \log_s P - 1$. For P = 256 and s = 16 one gets a simple three stage network.

In the proof of the fact that the classical construction indeed produces a permutation network several routings are provided for each permutation. This can be exploited in order to make the network tolerant against failures of single basic building blocks in each bit plane. The network can then be reconfigured to work *without loss of performance*. Switching in spare processors in place of faulty processors on permutaion networks is easy too and communication does not become slower [PP85].

A gate array realizing a basic 16x16 switch which supports construction of such a fault tolerant system has been designed in Saarbrücken. A network for 256 processors is under construction.

Fig. 3



7. References.

[A85] D. J. Auerbach et al.: A highly parallel computer for molecular dynamics simulations. Mat. Res. Soc. Symp. Proc. Vol. 63, 1985, 219-224, Materials Research Society.

[A87] D.J. Auerbach et al.: A special purpose parallel computer for molecular dynamics: motivation, design, implementation and application. J. Physical Chemistry, 1987, 91, 4881-4890.

[AHU74] Aho, Hopcroft and Ullman: The design and analysis of computer algorithms. Addison Wesley, 1974.

[B65] V. Benes: Mathematical theory of connecting networks and telephone traffic. Academic: New York, 1965

[GP83] Z. Galil and W. J. Paul: An efficient general purpose parallel computer. J. ACM 30, 360-387, 1983.

[HB84] K. Hwang and F. A. Briggs: Computer architecture and parallel processing. McGraw-Hill, 1984.

[HP87] D. Hinz und W. J. Paul: Über Parallelrechner für numerische Anwendungen und ihre Programmierung. PIK 10 (1987) 3, 161 167.

[HS65] Hartmanis and Stearns: On the computational complexity of algorithms. Trans. Am. Math. Soc., 117, 288-306, 1965.

[LPV81] G. Lev, N. Pippenger and L. Valiant: A fast parallel algorithm for routing in permutation networks. IEEE trans. Comput. C-30, 93-100, 1981.

[M81] G. J. Myers: Advances in computer architecture, 2nd ed. John Wiley and Sons, 1981.

[McM 84] F. H. McMahon: Fortran Kernels: Mflops; file modification date 29 Feb 84. Lawrence Livermore National Laboratory.

[MP84] A. Munshi and W. J. Paul: A language for SPARK and Barnburner. Unpublished memo, 1984.

[PP85] W. J. Paul and N. Pippenger: Parallel computers coupling a permutation network, IBM technical disclosure bulletin Vol.28, No 7, 3050-3051, 1985.

[Re88] U. Reeder. Diplomarbeit.

[Ru78] R. M. Russel: The CRAY-1 Computer System. Comm. ACM, Vol. 21, 63-72, 1978.