Scheduling Vector Straight Line Code on Vector Processors

C. W. Keßler*
Graduiertenkolleg Informatik der Universität Saarbrücken
Saarbrücken, Germany

W. J. Paul, T. Rauber†
Computer Science Department, University Saarbrücken
Saarbrücken, Germany

Abstract

We present an algorithm to schedule basic blocks of vector three-address-instructions. This algorithm is suited for a special class of vector processors containing a buffer (register file) which may be partitioned arbitrarily into vector registers by the user. The algorithm computes the best ratio of vector register spilling to strip mining, taking the vector length and the buffer size into consideration, as well as several machine parameters of the target architecture. We apply the algorithm to groups of vector instructions within a basic block that are quasiscalar, i.e. all vectors occurring in the group must have one fixed length L.

1 Introduction

For scalar processors register allocation is widely accepted as one of the most important optimizations in compiler construction. In [1] Allen and Kennedy claim that register allocation is even more important on vector processors with vector registers. They argue that by an effective use of the vector registers, the system performance can be enhanced by a factor of 3^1, so vector register allocation is the single most important optimization. This justifies a great effort in improving the use of the vector registers. Allen and Kennedy claim that strategies for scalar register allocation are of little use in vector register allocation because they do not take into consideration that the vectors may not fit into the vector registers. This argument is true, but we think that when considering basic blocks a scalar strategy (e.g. the one described in [7]) is useful provided that it is followed by an adaption algorithm which is presented in this paper.

We first give some basic definitions:

In general the front end of a compiler generates an intermediate code representation of the source program, often in the form of *three-address-instructions*,

*research partially funded by the Leibniz program of the DFG
†research partially funded by DFG, SFB 124

[1] This can be explained by saving `Load` and `Store` operations when holding intermediate results and operands in vector registers.

to which code optimization can be applied. A sequence of three-address-instructions which can only be entered via the first and only be left via the last statement is called a *basic block*. The data dependencies in a basic block can be described by a *directed acyclic graph (DAG)*. Evaluating this DAG means ordering the nodes v of the DAG such that each node appears after all its sons in the evaluation, i.e. an *evaluation* is a topological order of the DAG.

A mapping *reg* from the set V of the DAG nodes to the set of registers is called a (consistent) *register allocation* for a given evaluation A if for all nodes u, v, $w \in V$ the following holds: If u is a son of w, and v appears in A between u and w, then $reg(u) \neq reg(v)$.

Each evaluation A requires a certain number $m(A)$ of registers. We call

$$m(A) = \min_{reg \text{ is reg. alloc. for A}} \{ \max_{v \text{ appears in } A} \{reg(v) + 1\}\}$$

the *register need* of the evaluation A. The problem of finding an *optimal* evaluation A_{opt} with minimal register need

$$m_{opt} = m(A_{opt}) = \min_{A} m(A)$$

is NP–complete (see [16]). A heuristic which produces fairly good evaluations with little register need for a scalar DAG in linear time has been presented in [7]. An improved complete search algorithm will be given in a later paper (see [8]).

In general it is advisable to minimize the register need m while maximizing register usage for registers $1, ..., m$ in order to get a good register allocation. For our considerations, however, it is much better to keep register usage as *uneven* as possible, as suggested in [7, 8]. This yields the same register need as an allocation with even register usage, but now registers with low usage will be very good spilling candidates. That will be utilized by the optimization algorithm described in this paper.

Here we adjust a given evaluation A for a vector basic block to a special class of vector processors by taking into account some important machine constraints and architecture parameters. We present the target processor in section 2. The third section gives two different strategies and a combination of them which map the symbolic registers to the limited buffer of the processor. By applying the combination to an example DAG we find an interesting tradeoff–effect in section 4. Finally we generalize this in the fifth section and close with a look to the applicability of our results to real vector processors.

2 An abstract machine model of a vector processor

In order to maintain generality we do not consider any special vector processor but an abstract model $\mathcal{MV}$ (see Fig. 1) representing a wide class of real vector processors. A typical representative of this class is the SPARK 2.0 (see [2, 3, 15, 11, 4]) developed at university Saarbrücken. Since this is the only real vector processor we know all the exact machine parameters about that are essential for our optimizations, we will carry out the example computations in this paper for this processor. Nevertheless, the results are still valid for several other vector processors, as indicated in section 6.

Omitting control logic and adressing units we describe here only the main architecture components of $\mathcal{MV}$ which are relevant for our optimization ideas:

- The vector floating–point unit (**VFPU**) of the processor which is divided into d pipeline stages executes the arithmetical operations (e. g. addition, subtraction, multiplication). We suppose the execution of 1 stage to take 1 machine cycle. So a scalar operation needs $d + 1$ machine cycles from filling in the operands up to the result being available. A vector instruction of length L needs exactly

  $$t_{COMP}(L) \;=\; L \;+\; d$$

  machine cycles. d is called *pipeline depth*; typical values for d are between 3 and 10 (cf. [14]).

- The floating–point memory (**FM**) is a very fast buffer (usually realized by a static RAM register file) for M floating–point words; the FM may be arbitrarily partitioned into *vector registers*. Because of the high cost, the FM is not very large, e. g. $M = 4096$ for the SPARK 2.0 or $M = 512$ for the *INTEL Hypercube VX* (see [5]). The data which do not fit into the FM must reside in the main memory (MM).

- **processor bus:** We require the bandwidth of the connection between VFPU and FM being large enough to ensure that two operands and one result can be transported from FM to VFPU (respectively vice versa), i.e. in one machine cycle the following operations may be executed:
  - computation of 1 result element in the VFPU
  - transport of 2 values stored in the FM to the VFPU as operands
  - transport of 1 result value from the VFPU to the FM.

- The main memory (**MM**) is slower than the FM in general, but large enough to store all data occurring in the source program. It is connected to the FM via the

- **memory bus:** The data transmission rate of this connection is dominated by the access time of the slower MM. We express this by the

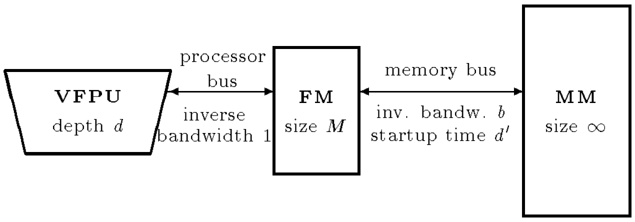

Figure 1: Block diagram of the abstract vector processor $\mathcal{MV}$: floating–point data paths

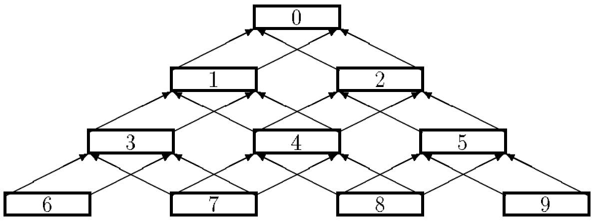

Figure 2: A (quasiscalar) 4–pyramid (4 leaves)

inverse bandwidth[2] b of the memory bus. Furthermore each vector move needs some *startup time* d'. In general the time needed for moving a vector of length L from FM to MM or vice versa is

$$t_{MOVE}(L) = t_{LOAD}(L) \; = \; t_{STORE}(L) = L \cdot b \; + \; d'.$$

Typical values for b are e.g. 1 (Spark 2.0) and e.g. 3 for d' (Spark 2.0). Moving a vector a from MM to FM is denoted by `Load`(a), from FM to MM by `Store`(a).

3 Evaluating quasiscalar vector DAGs

Definition: Let (V, E) be a DAG and $\Lambda = (L_1, \ldots, L_{|V|})$ be a list of positive integer values that associates a length L_v to each node $v \in V$. We call $G = (V, E, \Lambda)$ a vector DAG (short: *VDAG*). The VDAG G is called *quasiscalar* if all nodes have equal length L and if the VDAG has only one root. To make things not too complicated we first consider only basic blocks in which the operands of each instruction are addressed successively. The generalization to the case that the operands are addressed by constant stride > 1 is handled in section 7.

If we have an arbitrarily large register file we can evaluate quasiscalar VDAGs in the same way as scalar DAGs.

We can imagine a quasiscalar VDAG as a stack of L equally structured scalar DAGs which represent the data dependencies of the respective vector components. From the definition of a quasiscalar VDAG we easily obtain an algorithm to test whether a given VDAG G with n nodes is quasiscalar in time $O(n)$.

3.1 Strip mining

Quasiscalar VDAGs have the handy property that the evaluation of a certain (e.g. the i–th) component of the root only depends on the i–th component of the other nodes, i.e. it does not matter which components are combined and in which order the component–DAGs are evaluated.

[2]The inverse bandwidth is the number of machine cycles needed to transport a value from MM to FM or vice versa.

node	6	7	3	8	4	1	9	5	2	0
in reg.	0	1	2	0	3	1	2	4	0	3

Table 1: An evaluation for the 4–pyramid

In the following we will see that sometimes it is necessary or appropriate to evaluate a quasiscalar VDAG not by a single phase (handle whole vectors) but to split the evaluation in T phases. Each phase consists of an evaluation of a strip of the DAG with the *strip length*

$$str = \left\lceil \frac{L}{T} \right\rceil .$$

In the jth phase $(1 \leq j < T)$ for all nodes only the interval $[(j-1) \cdot str+1 \; : \; j \cdot str]$ is evaluated. In the Tth phase the rest interval $[(T-1) \cdot str+1 \; : \; L]$ is evaluated; we call

$$lstr = L - (T-1)str$$

the *strip rest length*.

This strategy known as *strip mining* (see [1]) is important if the required buffer size $m \cdot L$ of an evaluation (we need m vector registers of length L each) exceeds the number M of available registers. Then it seems advisable to use

$$T = \left\lceil \frac{m \cdot L}{M} \right\rceil \quad \text{strips.}$$

Example: Consider the 4–pyramid in Fig. 2. Let the node length be $L = 2000$. We need five vector registers of length 2000 to evaluate the DAG, thus 10000 floating–point words altogether. If we have e.g. only a FM size of $M = 4096$, we must divide the evaluation in two strips of length $str = 667$ and one of length $lstr = 666$, such that the evaluation of one strip requires at most $667 \cdot 5 = 3335 \leq 4096$ memory words. Inserting the remaining target machine parameters $d = 6$, $b = 1$ and $d' = 3$, we obtain for the SPARK 2.0 exactly 20144 machine cycles to evaluate the three strips. Additionally we must take into account some time (SPARK 2.0: 4 cycles) to handle the strip counter, thus 20148 cycles altogether, including an overhead of 148 cycles.

3.2 Spilling

For simplicity in this section we consider scalar DAGs. However, the considerations can also be applied to quasiscalar VDAGs.

Let $G = (V, E)$ be a DAG. *Spilling* a node $v \in V$ means to hold its value not in a register but in the main memory. As a consequence, before each occurrence of v as an operand in an evaluation A of G, the current value of v must be loaded into a free register (recall that the target machine can handle only operands residing in registers, not in the main memory), and after the occurrence of v in A as a result, we must store the value into the main memory. For scalar evaluations spilling becomes unavoidable if the target machine has less registers than required. However it is a difficult problem which nodes should be spilled and which ones should not.

We will present a simple algorithm to decide which nodes to spill in section 4. Before presenting the algorithm we look at a special case of spilling, the splitting of the DAG into several subDAGs. This special case is well suited to demonstrate how strip mining and spilling may be combined.

3.3 DAG splitting

For certain DAGs with very simple geometric structure, e.g. for the k–pyramid, it might be advisable to determine the spill nodes by splitting the given DAG G into several subDAGs S_i.

Instead of generating an evaluation A for G we compute an evaluation A_i for each S_i and put the pieces together to obtain an evaluation A' for G. By doing this we hope that

- the maximal register need $\max_i \{m(A_i)\}$ of the A_i is considerably smaller than the register need $m(A)$ of the original evaluation A, and that
- the additional `Load` and `Store` instructions required when concatenating the A_i in order to get A' do not become too numerous.

Here we consider *canonical segmentations* which are defined as follows:

Definition: A segmentation $\{S_1, \ldots, S_y\}$ of a scalar DAG $G = (V, E)$ into y segments $S_1 = (V_1, E_1), \ \ldots \ , S_y = (V_y, E_y)$ is called *canonical*, if:

1. $V_i \subset V$, $E_i \subset E \cap (V_i \times V_i)$ for all i, $1 \le i \le y$.
2. the S_i are edge–disjoint, i. e. $E_i \cap E_j = \emptyset$ for $i \ne j$, $1 \le i, j \le y$.
3. S_i has only one root w_i for all i, $1 \le i \le y$.
4. S_i is a weakly connected component[3] for all i, $1 \le i \le y$; in particular there exists a path in E_i from each node $v \in V_i$ to w_i.
5. There is no path $u, \ldots, v, \ldots, w$ in E such that $u, w \in V_i$ and $v \in V_j - V_i$ for $i \ne j$, $i, j \in \{1, ..., y\}$.
6. There exists a permutation $\pi : \{1, ..., y\} \rightarrow \{1, ..., y\}$, such that a path $(w_{\pi(i)}, \ldots, w_{\pi(j)})$ exists in E for all $i < j$.

Example: The segmentation $\{S_1, S_2\}$ in Fig. 3 is canonical (with $\pi(1) = 2$, $\pi(2) = 1$), the segmentation in Fig. 4 too (there π is the identity on $\{1, ..., 5\}$).

By construction the V_i are not disjoint for $y > 1$, i. e. there are nodes belonging to several (at least two) segments which are evaluated in exactly one segment S_i and used in the others. We call such a node a *bridge* of S_i.

Example: Consider Fig. 3. Nodes 5 and 8 are bridges since they are defined in S_2 and used in S_1. In S_1 they are leaves. Evaluating the segmented

[3] A weakly connected component of a directed graph G is a subgraph $G' = (V', E')$ of G where every two vertices are joined by a path in which the direction of each edge is ignored.

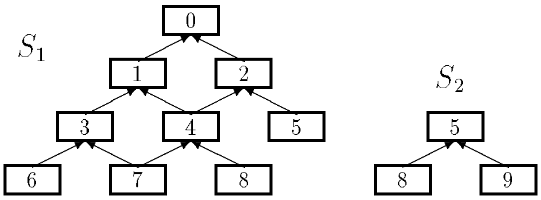

Figure 3: A 4–pyramid, canonically segmented into two segments

DAG $G = S_1 \cup S_2$ means: load nodes 8 and 9, compute node 5 and store it (node 8 needs not be stored again since it is already in the memory). In order to evaluate S_1 node 8 must be loaded again, and also node 5 (see Tab. 2). The evaluation in two segments requires only four registers; without the segmentation five registers would have been needed (cf. Tab. 1). Provided a FM size of $M = 4096$ and vectors of length $L = 2000$, this means that the DAG can be evaluated in 2 strips of length 1000. For the SPARK 2.0 this yields a run time of 26117 machine cycles.

We make the following observations:

- Since segmentation can be considered as a special form of spilling, the bridges can be considered as spilled nodes.
- Let S_j be a segment in which a bridge v is not evaluated but used as operand. Then v is a leaf in S_j.
- If a bridge is not the root of a segment, then its outdegree in S is at least 2. So we have to introduce at least one additional `Load` instruction for each bridge.
- The root w of the DAG G is also the root $w_{\pi(y)}$ of the topmost segment $S_{\pi(y)}$.

We get the **evaluation** A' of a canonically segmented DAG G by generating evaluations A_i for the segments S_i and then adding new `Load` and `Store` instructions to the sequence $A'' = (A_{\pi(1)}, \ldots, A_{\pi(y)})$: We insert a `Store`($v$) after the first occurrence of an inner bridge v in A'' (called definition point) and we insert a `Load`(v) before each occurrence of a bridge v as operand in A'' (called use point). As result we obtain A'.

Remarks: Because of points 5 and 6 in the definition of a canonical segmentation, the definition point for a bridge v in A'' appears before its use points, thus A'' is consistent and so is A' too. — Since a bridge does not need to reside in the same register all the time, it gets a register number in each A_i it is occurring in; thus the A_i are mutually independent and register allocation can start new from register no. 0 in each A_i.

Let us have another look at Tab. 2. We observe that register usage is suboptimal here: Reloading node 8 may be avoided since segment S_2 has only register need 3 while S_1 has register need 4, thus one register remains unused during the first part of the evaluation and could be used to hold the value of

node	8	9	5	`Store` 5	6	7	3	`Load` 8	4	1	`Load` 5	2	0
in reg.	0	1	2	—	0	1	2	0	3	1	2	0	3

Table 2: Evaluation of the segmented 4–pyramid

node 8. In general it may be possible to eliminate some `Load`s and `Store`s by first assigning *relative* register names separately for each segment (e. g. pairs $\langle segment - name,\ reg. - name\rangle$) and then deciding which bridge nodes may be held in the remaining free registers of a segment (i. e. which pairs $\langle S_i, r\rangle$ and $\langle S_{i+1}, r'\rangle$ should be coalesced). After that, the pairs are rearranged and mapped onto the real register names. — However, VDAGs with such a regular structure really do not occur in real programs. Furthermore, the optimization idea sketched above refers only to those segments that have lower register need than the maximal one. Even for the k–pyramid this situation occurs only at the last segment (if at all). Thus we omit these optimizations for simplicity and because of the little significance in practice. On the other hand, the simple optimization of spill code mentioned at the end of section 5 is much more useful instead.

3.4 Combination of strip mining and DAG splitting for the k–pyramid

Now we will combine strip mining with DAG splitting, at first for the example of the k–pyramid. Each of the $n = k(k+1)/2$ nodes of the k–pyramid should be evaluated exactly once.

For this special DAG e. g. a *left first* evaluation (see [6]) yields the minimal register need of $k+1$ vector registers.

Now, if $m \cdot L = (k+1)L > M$, i.e. the number m of the required vector registers is higher than the number of available vector registers, we must either spill some of the registers or virtually increase the number of vector registers by strip mining with several strips.

In order to get the optimal ratio of strip length to spilling rate which minimizes the run time of the evaluation on the target machine, we construct an algorithm which combines both possibilities:

Imagine we apply strip mining in order to obtain x vector registers ($3 \leq x < m$) each having a length of $str := \lfloor M/x \rfloor$ words. With x vector registers we can evaluate a $(x-1)$–pyramid or segments[4] of a greater pyramid of width $x-1$, as given in Fig. 4; there we have $k = 14$ and $x = 5$. The number of segments is $\lceil (k-1)/(x-2) \rceil$, the number of strips is $T := \lceil L/str \rceil$.

Here we suppose that the segmentation of the k–pyramid is given; the segmentation in Fig. 4 is canonical and seems to be the most favorable with respect to the number Br of bridges and to the number A_{Br} of "bridge chains"[5]:

$$A_{Br} := \left\lceil \frac{k-1}{x-2} \right\rceil - 1$$

[4] We obtain these segments by an appropriate canonical segmentation of the k–pyramid.
[5] A bridge chain is a chain of bridges that separates two segments.

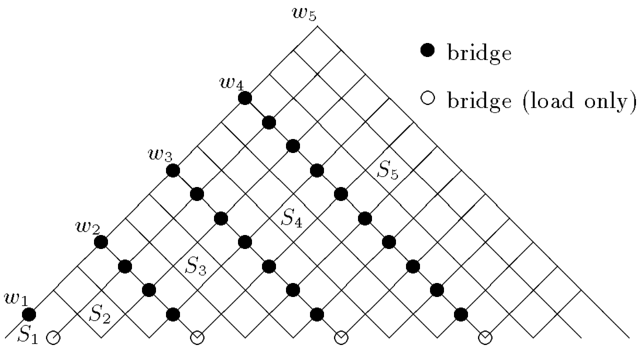

Figure 4: A 14–pyramid, splitted in segments of width $x - 1 = 4$

$$Br := \sum_{i=1}^{A_{Br}} (k - i(x-2))$$

A_{Br} bridges are leaves. Leaf values need not be stored because they are already present in the main memory. So we need Br additional `Load`s and $Br - A_{Br}$ additional `Store`s in order to handle the bridge values in the main memory.

Thus we obtain the following procedure:

---

program *pyramid*
Input: k–pyramid, number x of vector registers
Output: an evaluation of the k–pyramid
Method:
(1) {divide the k–pyramid canonically into segments
    $S_1, S_2, \ldots$ and determine *str*, T, A_{Br}, Br}
(2) **for** *strip* **from** 1 **to** T
(3) **do for** i **from** 1 **to** $A_{Br} + 1$
(4)   **do** {generate evaluation A_i for segment S_i}
(5)   {let π be the permutation
            to the canonical segmentation}
(6) $A'' = (A_{\pi(1)}, \ldots, A_{\pi(A_{Br}+1)})$;
(7)   {insert a `Load` instruction before each
            use point of a bridge in A''}
(8)   {insert a `Store` instruction after each
            definition point of a bridge in A''}
(9)   {return the resulting evaluation}
        od
    od

We now want to compute the run time t of the evaluation generated by *pyramid* which depends on the strip length *str*, the strip rest length *lstr*, the number of strips T and the number of additional move instructions *moves*. We have to evaluate $T - 1$ strips of length *str*, each strip having computation cost $t_{COMP}(str)$ per compute operation and move cost $t_{MOVE}(str)$ per move operation. Then we have to evaluate the rest strip of length *lstr*. $comps = k(k-1)/2$ is the number of compute instructions (i.e. the number of inner nodes which is constant for fixed k) and $moves = k + 2Br - A_{Br}$ is the number of `Load` and `Store` instructions. Let $t_{loop}(T)$ denote the cost for handling the strip loop[6] which depends on the number of strips.

By substituting the values for *str*, *lstr*, T, *moves*, *comps*, t_{COMP} and t_{MOVE}, we see that t only depends on the pyramid size k, the vector length L, the number x of vector registers and on the machine parameters M, d, b and d':

$$\begin{aligned} t &= t(str, lstr, T, moves, comps) \\ &= (T-1) \cdot [comps \cdot t_{COMP}(str) \\ &\qquad\qquad + moves \cdot t_{MOVE}(str)] \\ &\quad + comps \cdot t_{COMP}(lstr) \\ &\quad + moves \cdot t_{MOVE}(lstr) \\ &\quad + t_{loop}(T) \\ &= t(x, k, L, M, d, b, d'). \end{aligned} \tag{1}$$

In the following we suppose k, L and the target machine parameters to be constant, so the evaluation A generated by *pyramid* depends only on x and so does t.

Now we must only determine the number $x = x_{opt}$ of registers that minimizes the run time $t(x)$ of $A(x)$. The next section describes how this can be done.

4 The Tradeoff–Effect for the k–pyramid

At this time we take the architecture parameters of the target machine into consideration.

Function $t(x)$ is not very handy because of the ceil and floor operations. Furthermore t is not differentiable. Therefore we simplify $t(x)$ such that it can be differentiated; we set

$$str \approx lstr \approx M/x, \quad T \approx L/str \approx Lx/M,$$

$$A_{Br} \approx \frac{k-1}{x-2} - 1$$

$$Br = A_{Br}k - (x-2)\sum_{i=1}^{A_{Br}} i \;\approx\; \frac{k^2 - kx + 2k - x + 1}{2(x-2)}$$

i. e. we neglect the ceil and floor operations. We obtain

$$moves = k + 2Br - A_{Br} \approx \frac{k^2 - k}{x-2}$$

[6] In the following we choose $t_{loop}(T) = T + 1$; that corresponds to the loop costs for the SPARK 2.0.

and the approximate run time t becomes

$$t(x) \approx \overbrace{\tfrac{Lx}{M}}^{\approx T} \left[\overbrace{\tfrac{k(k-1)}{2}}^{=comps} \left(\tfrac{M}{x} + d\right) + \overbrace{\tfrac{k^2-k}{x-2}}^{\approx moves} \left(b\tfrac{M}{x} + d'\right) \right]$$

neglecting the loop costs $t_{loop}(T)$ for simplification. Thus

$$t(x) \approx Lk(k-1) \left[\frac{1}{2} + \frac{d \cdot x}{2M} + \frac{b}{x-2} + \frac{d'}{M} \frac{x}{x-2} \right].$$

We differentiate this approximation of t with respect to x and obtain

$$\frac{dt(x)}{dx} \approx Lk(k-1) \left[\frac{d}{M} - \frac{b}{(x-2)^2} - \frac{d'}{M} \frac{2}{(x-2)^2} \right].$$

The necessary condition for a minimum is $dt(x)/dx = 0$, i. e.

$$(x-2)^2 - \frac{bM}{d} - 2\frac{d'}{d} = 0.$$

That yields

$$\boxed{x_{opt} \;\approx\; 2 \pm \sqrt{\frac{bM + 2d'}{d}}} \tag{2}$$

For control we form the second derivation:

$$\frac{d^2 t(x)}{(dx)^2} \approx 2Lk(k-1) \frac{b + 2d'/M}{(x-2)^3}.$$

By inserting the values for x_{opt} we see that the approximation of the function $t(x)$ is convex at the position $x_{opt,1} = 2 + \sqrt{\frac{bM+2d'}{d}}$, so $x_{opt,1}$ is a minimum position whereas $x_{opt,2} = 2 - \sqrt{\frac{bM+2d'}{d}}$ is a maximum position.

From formula 2 we conclude the following:

1. x_{opt} is independent of the vector length L (the small term depending on L got lost by neglecting $t_{loop}(T)$) and the size k of the pyramid.[7]
2. The smaller the buffer size M, the more registers will be spilled (x_{opt} moves towards the origin).
3. The greater the pipeline depth d, the smaller becomes x_{opt}, the more registers will be spilled.
4. The greater the inverse bandwidth b, the greater x_{opt} (less spillings).
5. d' is in practice not significant for x_{opt} because for existing vector processors (see section 6) the term $2d'$ is too small compared with the term bM.

[7] The value of t at position x_{opt} depends on the ratio of k to M.

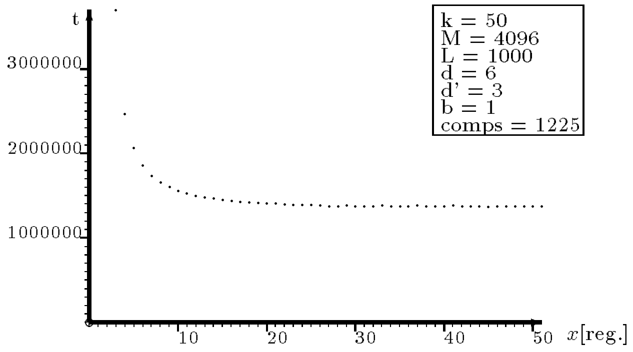

Figure 5: For the SPARK 2.0 with the whole FM available we have no clear minimum since M is large enough. The rapidly growing curve for very small values of x is caused by the immense bridge number and the communication overhead involved. For $x = 3$ all nodes are bridges, for $x = 4$ still nearly half of them. The net amount of the run time is $nL = (comps + k)L = 1275000$ cycles, the rest are startup times, loop– and communication costs.

These considerations are confirmed by the diagrams of Fig. 5 and Fig. 6. We call the existence of a local minimum of $t(x)$ *tradeoff–effect*, because strip length and spilling rate are traded off against each other in order to minimize the run time of the evaluation.

5 How to take advantage of the Tradeoff–Effect

We have computed the run time t (in machine cycles) of program *pyramid* for a 50–pyramid according to formula 1 and have varied all machine parameters. This results in a lot of $x/t(x)$–diagrams. Two of them are printed in Fig. 5 and 6. We make the following observations:

- The tradeoff–effect is most remarkable for a great pipeline depth d, cheap communication (especially a small b) and in particular for a small buffer size M with respect to the pyramid size k.
- If the register file is large enough even a deep pipeline does not make a difference; we have no clear minimum.
- A higher inverse bandwidth b attenuates the tradeoff–effect.
- The vector length L is unimportant and will be resolved cheaply by strip mining.
- The move startup time d' is unimportant.
- The curves are not smooth, several local minima are possible. A strict minimum search via (iterative) zero point determination of t' is not possible.

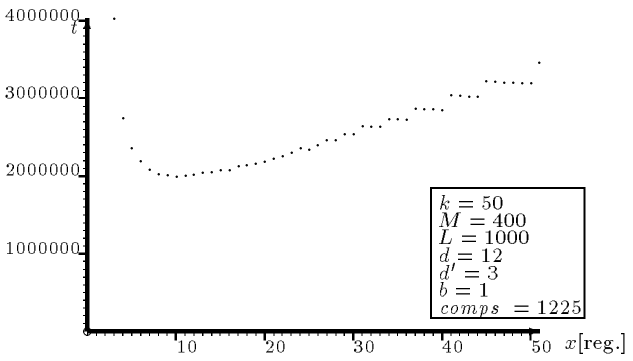

Figure 6: A deep pipeline increases the computation costs; the communication is cheap. If we suppose a small FM we get a remarkable tradeoff–effect: When using $x_{opt} = 10$ vector registers of length $\lfloor M/x_{opt} \rfloor = 40$ instead of $m = 51$ vector registers of length 7 we save about 43% of the run time.

> So there remains *linear search* as it is done by the postprocessor algorithm described in the next section.

So far we have used the k–pyramid as an example VDAG because of its regular structure which in particular enables splitting the DAG easily into segments of equal register need and minimizing the run time function analytically. This shows the existence of a minimum in many cases. However, real VDAGs do generally not have such a regular structure and thus DAG splitting cannot be applied. In this case we replace DAG splitting by the more general spilling method of subsection 3.2. This results in a new algorithm described in the following subsection.

5.1 The Postprocessor algorithm

For a given evaluation A with a register need of m symbolic vector registers of length L each, the following algorithm computes the best ratio of strip length str to the number x of registers used.

In opposition to the spilling decisions in the last section we now do not spill *single nodes* but *whole (vector) register contents* (i. e. in general several nodes at one time). The ideal spilling candidates are those (vector) registers which are referenced least.

The nodes to spill are determined by an algorithm which runs after the generation of the evaluation A (therefore we name it "postprocessor").

When spilling registers we get the following problem: For the registers to be spilled the evaluation generator allocated symbolic registers that now do no longer correspond to any physical register but to a main memory location. Since spilling was not provided in the original evaluation we must introduce $E \leq 2$ *communication registers* with the same size as the other vector registers. These are required because both operands used by an operation might be spilled. Two additional vector registers are sufficient since we need the value of a spilled node only for the time of one compute operation such that the result — if it is spilled too — can also be computed into one of the communication registers; because the VDAG is quasiscalar, there are no data dependency problems.

Furthermore, given an evaluation A, we compute for each register $i = 1, \ldots, m$ two lists of pointers to three-address-instructions in A:

$$\forall i = 1, \ldots, m: \quad Def_i = \{d_{i,1}, \ldots, d_{i,\theta_i}\}$$

$$\forall i = 1, \ldots, m: \quad Use_i = \{u_{i,1}, \ldots, u_{i,\mu_i}\}$$

where the $d_{i,j}$ point to three-address-instructions that compute a value into register i (i.e. $d_{i,j}$ is a definition point for register i) and the $u_{i,j}$ point to three-address-instructions using the contents of register i (i.e. $u_{i,j}$ is a use point of register i).

$move_i = \theta_i + \mu_i$ is the number of read and write accesses to the symbolic register i.

Moreover, for a three-address-instruction z let $node(z)$ denote the DAG node which is evaluated by z. Let $spilled[1...m]$ be a boolean array.

The postprocessor starts with building a sorted list $sortedreg[1 \ldots m]$ which contains the symbolic registers $1, \ldots, m$ in ascending order of $move_i$.

Next we determine the point x_0 up to which *one* communication register suffices, and after which we really need two, when spilling the registers in the order given by $sortedreg[1, ..., m]$. This is done by the following function in linear time:

```
(1) function last_point_one_extra_reg: integer;
(2) for i from 1 to m do spilled[i] = FALSE od;
(3) for z from 1 to m
(4) do i = sortedreg[z];
(5)     for all d_{i,j} ∈ Def_i
(6)     do spilled[reg(node(d_{i,j}))] = TRUE od;
(7)     for all u_{i,j} ∈ Use_i
(8)     do if spilled[reg(lson(node(u_{i,j})))]
              and spilled[reg(rson(node(u_{i,j})))]
(9)        then return m − z + 1 fi
        od
    od
    end last_point_one_extra_reg;
```

Then the following program fragment is executed which simulates the successive spilling of the symbolic registers in the order given by *sortedreg*. After each spill we compute the run time of the resulting computation with formula (1).

```
(1a) m_opt = m;  moves = number of leaves;
(1b) t_opt = t(str, lstr, T, moves, comps);
(1c) x_0 = last_point_one_extra_reg;
(2)  for x from m − 1 downto ⌊M/L⌋
(3)  do i = sortedreg[m − x];
(4a)     moves = moves + move_i;
```

(4b) **if** $x \geq x_0$ **then** $E = 1$ **else** $E = 2$ **fi**;
(5) **if** $\lfloor M/(x+E) \rfloor > \lfloor M/(x+E+1) \rfloor$
(6) (∗ found candidate for local minimum: ∗)
(7a) **then** $str = \lfloor M/(x+E) \rfloor$;
(7b) $T = \lfloor L/str \rfloor$;
(7c) $lstr = L - (T-1)str$;
(8) $t_{new} = t(str, lstr, T, moves, comps)$;
(9a) **if** $t_{new} < t_{opt}$
(9b) **then** $t_{opt} = t_{new}$; $m_{opt} = x$ **fi**
fi
od
(∗ now spill reg. $sortedreg[1, \ldots, m - m_{opt}]$: ∗)
(10) **for** z **from** 1 **to** $m - m_{opt}$
(11) **do** $i = sortedreg[z]$;
(12) **for** all $d_{i,j} \in Def_i$
(13) **do** {insert a `Store` after instruction $d_{i,j}$} **od**
(14) **for** all $u_{i,j} \in Use_i$
(15) **do** {insert a `Load` before instruction $u_{i,j}$} **od**
od
(16) {renumber the remaining registers
from 1 to m_{opt} and use the new
register numbers in the modified evaluation}

$moves$ contains the number of vector move instructions required. $moves$ is initialized with the number of leaves in the DAG because we need a `Load` instruction for every leaf. t_{opt} contains the best run time found so far, m_{opt} contains the corresponding number of vector registers.

In lines (2) to (9) the spilling of the registers in the order given by $sortedreg$ is simulated in order to find the number m_{opt} of vector registers which yields minimal cost t.

The condition in (5) prevents unnecessary computations since increasing $moves$ in (4) without a gain in strip length can never lower the cost t.

By adding E extra registers to x respective $x+1$ in lines (5) and (7a) the communication registers are taken into account.

According to (9) m_{opt} is the number of registers giving the cost minimum t_{opt}. The actual spilling with inserting the `Load` and `Store` instructions into the evaluation is done in lines (10) to (15). In line (16) a consecutive numbering of the remaining registers is rearranged. The final number of registers used is $m_{opt} + E$.

Let us consider the run time of the postprocessor algorithm: Generating the lists Def_i and Use_i takes time $O(n)$ altogether since the original evaluation A has exactly n instructions. Since the register need can never be greater than n, there exist at most $2n$ of these lists. The total length of all lists Def_i and Use_i, $i = 1, ..., m$, cannot exceed $3n$ since each instruction in A has at most two operands and one result, thus at most two `Loads` and one `Store` have to be additionally generated for each instruction in A. It follows that the $move_i = \theta_i + \mu_i$ are natural numbers within $\{1, \ldots, 3n\}$. Thus we can apply *bucket sort* in order to sort the symbolic registers according to $move_i$ for which linear time suffices.

The body of the first loop (lines (3) to (9)) is executed in constant time; the loop (lines (2) to (9)) is executed less than m times. In line (2) x only needs to decrease down to $\lfloor M/L \rfloor$ since each further spilling can not lower the cost any more because the strip length str has already reached the whole vector length L.

The second loop (lines (10) to (15)) also runs in linear time because at most $3n$ move instructions are inserted. The same fact implies that renumbering the modified evaluation containing now at most $4n$ instructions can be done in linear time. We conclude that the whole postprocessor has run time $O(n)$.

5.2 Application in practical use

We have included the postprocessor in an optimizing vector PASCAL compiler for the SPARK 2.0 processor. We observe that with the whole FM with $M = 4096$ floating–point words available even for DAGs with rather high register need (ca. 30) no register was spilled. That corresponds to our observations at the example of the k–pyramid.

The tradeoff–effect becomes remarkable if only a small part of the FM is available for the evaluation of the DAG. This often occurs in practice because other vectors not used in the DAG may be stored in the FM too.

The implementation of the basic block optimizer of the vector PASCAL compiler developed at our institute consists of four parts:

1. a transformer which tests a given vector DAG constructed in a previous stage from a basic block (see [10]) on being quasiscalar. If the test is successful, the transformer translates the VDAG into a simple scalar DAG. If the test fails, the basic block is passed unchanged to the code generator.
2. a randomized heuristic evaluation generator for scalar DAGs (see [7])
3. the postprocessor
4. another transformer reconstructing a basic block from the modified evaluation to be passed to the code generator.

Some of the `Load` and `Store` instructions inserted by the postprocessor may be redundant in the final evaluation.[8] But these are filtered out by a simple peephole optimizer in linear time.

6 Real Vector Processors

Not all vector processors occurring in practice fulfill the preconditions given by $\mathcal{MV}$. For example, the CRAY–2 does not fit into our model since it has vector registers of a fixed length (64), i. e. the vector memory is prepartitioned such that the postprocessor could not vary the strip length at all.

The INTEL i860 has no real FM but only 32 scalar registers (32 bit wide) that can be combined to hold $M = 16$ 64–bit words.[9]

[8] E.g. if an inner node u and its father v are spilled and v is computed immediately after u then one operand `Load` of v can be saved because the value of u is still in a register.

[9] The hardware cache of the i860 is not accessible to the user.

In spite of these, the vector extension *VX* of the *INTEL Hypercube* (see [5]) fulfills our constraints: From the 16 Kbyte FM[10] 4 Kbyte ($\Rightarrow M = 512$ words) are available for the user. The inverse bandwidth b is 1 (cycle time 300 ns per addition or single precision multiplication). Unfortunately the values of d and d' are not available.

The postprocessor is also applicable to the vector node of the SUPRENUM (see [9]): The vector start addresses[11] of the vector registers are realized by pointers into the FM which contains up to $M = 8192$ words. — Another $\mathcal{MV}$–like vector processor is the Fujitsu VP–200 with the same FM size. — Furthermore, according to recent announcements, the vector node of the new CM–5 (see [17]) seems to be another good representative of $\mathcal{MV}$.

Unfortunately we do not have available all the required exact parameter values of all these machines. That is why we preferred our SPARK 2.0 processor for the example computations in this paper.

7 General VDAGs

The postprocessor can easily be adjusted to VDAGs which contain vectors addressed by a constant stride $s > 1$ provided that the single components do not interdepend and the effective vector length[12] is the same for all DAG nodes.

However some constructs of a vector programming language cause the resulting VDAG to be not quasiscalar, i.e. there are data dependencies between the single component DAGs. Such constructs are e.g. vectors addressed by index vectors, reduction operations with vector operands and scalar result (e.g. scalar product) or vector operations that use several VDAG nodes as a single operand.

This more general case seems to be rather hard to handle; a simple heuristic (but far from optimal) algorithm to generate evaluations for general VDAGs is given in [6]. However if the (not quasiscalar) VDAG is a tree then the algorithm in [12] computes an optimal evaluation of the tree in time $O(n \log n)$. Further research should yield sufficiently good heuristics which modify these evaluations too for the case that the required register space does exceed the buffer size M.

8 Acknowledgement

We thank Robert Giegerich for his comments on this paper.

References

[1] J.R. Allen, K. Kennedy: *Vector Register Allocation.* Rep. COMP TR 86-?, Rice University, April 1986.

[10] The FM of the Hypercube VX is realized by SRAMs with an access time of 100 ns; it can store 2048 words of 64 bit each.

[11] Of these there exist indeed only 8, but by holding addresses in the local memory of the node (while taking into account an additional memory access) we can simulate an arbitrarily great number of vector registers.

[12] The effective vector length of a vector of length L addressed by constant stride s is $\lfloor L/s \rfloor$.

[2] D. Auerbach, W.J. Paul, A. Bakker, C. Lutz, W. Rudge, F. Abraham: *A Special Purpose Parallel Computer for Molecular Dynamics.* Journal of Physical Chemistry, 1987, 91, 4881.

[3] A. Formella: *Entwurf, Bau und Test eines Vektorprozessors mit parallel arbeitenden Operationseinheiten, Teil 1.* Master thesis, 1989, Universität Saarbrücken.

[4] A. Formella, A. Obé, W.J. Paul, T. Rauber, D. Schmidt: *The SPARK 2.0 System — a Special Purpose Vector Processor with a VectorPASCAL Compiler.* Proceedings of the Twenty–fifth Annual Hawaii International Conference on System Sciences (HICSS–25) 1992.

[5] INTEL Corp.: *A Technical Summary of the iPSC/2 concurrent supercomputer.* Proc. of the Third Hypercube Conference, ACM 88.

[6] C.W. Keßler: *Code–Optimierung quasiskalarer vektorieller Grundblöcke für Vektorrechner.* Master thesis, 1990, Universität Saarbrücken.

[7] C.W. Keßler, W.J. Paul, T. Rauber: *A Randomized Heuristic Approach to Register Allocation.* Proceedings of PLILP91 Third symposium of programming Language Implementation and Logic Programming 1991, Passau (Germany), Springer LNCS Vol. 528, 195–206.

[8] C.W. Keßler, T. Rauber: *On the Complexity of Contiguous Evaluations.* Submitted (1992).

[9] H. Kammer: *The SUPRENUM Vector Floating Point Unit.* Proceedings 2nd International SUPRENUM Workshop, 1988.

[10] J. Lillig: *Konstruktion vektorieller DAGs für einen Vektorpascal–Compiler.* Program manual 1989, unpublished, Universität Saarbrücken.

[11] A. Obé: *Entwurf, Bau und Test eines Vektorprozessors mit parallel arbeitenden Operationseinheiten, Teil 3.* Master thesis, 1989, Universität Saarbrücken.

[12] T. Rauber: *Ein Compiler für Vektorrechner mit optimaler Auswertung von vektoriellen Ausdrucksbäumen.* Ph. D. thesis, 1990, Universität Saarbrücken.

[13] T. Rauber: *An Optimizing Compiler for Vector Processors.* Proc. ISMM International Conference Parallel and Distributed Computing and Systems, New York 1990, Acta press, 97–103.

[14] U. Reeder: *Die Vorhersage der Leistung von Vektorrechnern.* Master thesis, 1988, Universität Saarbrücken.

[15] D. Schmidt: *Entwurf, Bau und Test eines Vektorprozessors mit parallel arbeitenden Operationseinheiten, Teil 2.* Master thesis, 1989, Universität Saarbrücken.

[16] R. Sethi: *Complete register allocation problems.* SIAM J. Comput. **4**, 1975, 226–248.

[17] *The Connection Machine CM–5 Technical Summary*, Thinking Machines Corporation, Cambridge, Massachusetts, 1991.