

# Parallel Graph Reduction with the $\langle \nu, G \rangle$ -machine

Lennart Augustsson and Thomas Johnsson

*Department of Computer Science  
Chalmers University of Technology  
S-412 96 Göteborg, Sweden*

Email: *augustss@cs.chalmers.se* and *johnsson@cs.chalmers.se*

## Abstract

We have implemented a parallel graph reducer on a commercially available shared memory multiprocessor (a Sequent Symmetry<sup>TM</sup>), that achieves real speedup compared to a fast compiled implementation of the conventional G-machine. Using 15 processors, this speedup ranges between 5 and 11, depending on the program.

Underlying the implementation is an abstract machine called the  $\langle \nu, G \rangle$ -machine. We describe the sequential and the parallel  $\langle \nu, G \rangle$ -machine, and our implementation of them. We provide performance and speedup figures and graphs.

## 1 Introduction

Compiled graph reduction, as embodied in the G-machine and the Lazy ML compiler [Aug84, Joh84] has proved to be rather an efficient way to implement lazy functional languages on conventional machines. In this paper we report our results on extending these compilation techniques for parallel computers. We have implemented a parallel graph reduction system, a modified parallel G-machine, in a commercially available shared memory multicomputer, a Sequent Symmetry<sup>TM</sup>. Using 15 processors we have achieved a speedup ranging from 5 to 11, compared to our ‘conventional’ G-machine implementation of Lazy ML.

It would be possible to design a parallel G-machine by simply extending the sequential one to allow multiple executing threads of control instead of a single one, and a stack for each thread of control instead of a single stack – this has been done in e.g. [RHH<sup>+</sup>88, ANT88, Bur88]. In these abstract machine designs, creation of a new computation process is accompanied by the creation of a new stack for the process. In implementing such a machine, we are faced with the prospect of having to implement arbitrarily many, arbitrarily deep stacks, giving us a ‘cactus stack’ structure that might be hard to implement efficiently.

Yet, in the heap management of e.g. the G-machine, we already have a general and efficient memory allocation ma-

chinery. So why not simply put the stacks on the heap? If we do that, with a large number of processes each with a fair size stack, we run the risk of using up the heap space for stacks only. The solution to this problem seems to be to reduce the size of these stacks.

Also, consider the following fact: In the G-machine, when reduction of a function application starts, the arguments of the application nodes are moved to the stack, and when reduction is finished, the result is moved back to the heap by updating the (root) apply node of the redex with the root of the result. This seems like a lot of unnecessary data movement, especially since the arguments could in principle be accessed from the heap each time (albeit at the penalty of some extra memory references).

So rather than handling arbitrarily large ‘stack nodes’, we instead want to allocate ‘stack frames’, i.e., small stacks for single activations of EVAL — in effect, a (vector) apply node and a stack frame becomes the same thing. This scheme also has the advantage that there is less synchronisation when the evaluation of a node is about to start; with binary application nodes the spine from the evaluated node and downwards have to be locked before evaluation can start (see [CJ86]), in our case it is just the frame that has to be locked.

In our implementation of the G-machine, heap allocation is done simply and efficiently by incrementing a heap pointer — this is just as fast as allocating a new frame on a stack [Joh86] (Appel argues similarly [App87]). A risk we face with such a scheme is a slowdown due to the fact that the garbage collector would have to be invoked more often — however, our preliminary measurements indicate that this is not such a serious problem in practice.

Figure 1 shows a frame node in our abstract machine, called the  $\langle \nu, G \rangle$ -machine (for reasons that will become obvious later). A frame node is built to represent a suspended function application, and so holds the arguments of the application, plus a pointer to the code for the function. But since the frame node also serves as work space during the reduction of the function application, there may be some extra space for temporary variables allocated in the frame node when it is created. Finally, there is also a ‘dynamic link’ field, which during reduction of the application hold a pointer to the frame node from which evaluation was called.

Figure 2 shows how control is transferred when reduction is initiated with an EVAL instruction. Execution always occurs in the frame marked as ‘the current point of reduction’ (a triangular blob in the figures). At EVAL this point is moved, and the dynamic link pointer is set as appropri-

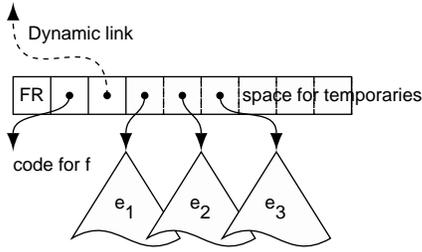


Figure 1: A frame node.

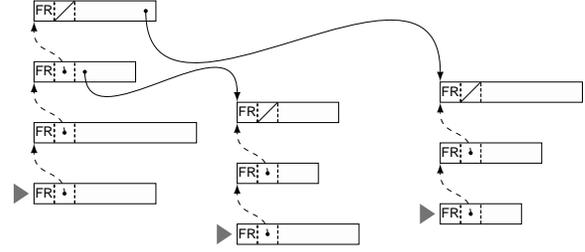


Figure 3: The frames in a cactus stack.

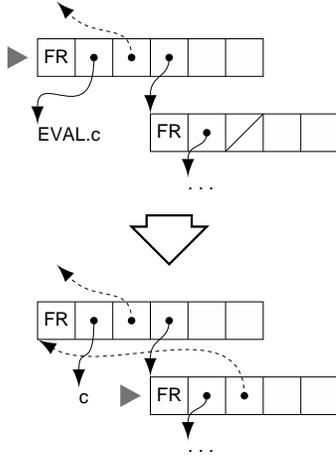


Figure 2: Initiating EVAL.

ate. Thus, recursive calls of EVAL links together individual frames to form a stack of frames.

We have adopted the spark model for introducing parallelism in the machine [CJ86]. The programmer writes spark annotations in the program, which when evaluated advise the system that certain expressions may be evaluated should there happen to be a processor available. There is no violation of semantics if the advise is not taken or if the spark message is lost altogether. If parallel evaluation of an expression has not started when the value is needed, the process needing the value evaluates it itself.

In the parallel  $\langle \nu, G \rangle$ -machine there will be one point of reduction for each evaluating process. Figure 3 shows an example configuration with three evaluating processes.

## 2 Technical details of the sequential abstract machine

In order to highlight the essential features of the  $\langle \nu, G \rangle$ -machine, we describe how to compile and execute programs

of the following very simple form:

$$\begin{aligned}
 p & ::= f_1^{m_1} x_{11} \cdots x_{1m_1} = e_1 \\
 & \vdots \\
 & f_n^{m_n} x_{n1} \cdots x_{nm_n} = e_n \\
 e & ::= x e_1 \cdots e_m & (m \geq 0) \\
 & \quad | f_i^k e_1 \cdots e_m & (m \geq 0) \\
 & \quad | c_k e_1 \cdots e_m & (m \geq 0) \\
 & \quad | \text{case } e \text{ in } c_{k_1} v_1 \cdots v_m : e_1 \parallel \cdots \text{end}
 \end{aligned}$$

That is, a program is a set of functions assumed to be the result of super-combinator abstraction [Hug82] or lambda-lifting [Joh85]. We also assume that the value of the program is the value of an application of one of those functions, “main” say, to some argument to the program (we ignore the further details of program startup). Expressions consists of parameters  $x_i$  possibly applied to some arguments, global functions  $f_i^k$  possibly applied to some arguments, constructor expressions, and case expressions with simple patterns. Constructors are distinguished by their constructor number and their arity, and we assume that constructor are always applied to a number of arguments equal to their arity. Thus, values in programs are either constructor values or functions.

### Compilation rules

To simplify the presentation further, the pointer data area in the nodes are treated as a stack (although in the implementation they are used as a set of locations allocated ‘randomly’). Hence, the  $\langle \nu, G \rangle$ -machine is presented as a stack machine, and the presentation of it here follows the presentation of the G-machine in [Joh84], by giving compilation rules and instruction transition rules.

Thus, compilation of the above language to  $\langle \nu, G \rangle$ -code is described in terms of five compilation schemes:

- $\mathcal{F}[[f x_1 \cdots x_n = e]]$  the top-level scheme, giving code for a function definition,
- $\mathcal{R}[[e]]\rho n$  code to return the value of the expression  $e$ , where  $\rho$  is a local environment, i.e., a mapping from variables to their location on the stack, and  $n$  is the current depth of the stack,
- $\mathcal{E}[[e]]\rho n$  code to compute the value of  $e$  to canonical form and leave a pointer to it on the top of the stack,

(0)	$\mathcal{F}[[f x_1 \cdots x_n = e]]$	$= \mathcal{R}[[e]] [x_1 = n, \dots, x_n = 1] n$	
(1)	$\mathcal{R}[[x e_1 \cdots e_m]] \rho n$	$= \mathcal{S}[[x, e_1, \dots, e_m]] \rho n; \text{EVAL}; \text{DO}$	$(m > 0)$
(2)	$\mathcal{R}[[x]] \rho n$	$= \mathcal{S}[[x]] \rho n; \text{EVAL}; \text{RETURN}$	
(3)	$\mathcal{R}[[f_i^k e_1 \cdots e_m]] \rho n$	$= \begin{cases} \mathcal{S}[[e_1, \dots, e_m]] \rho n; \text{MKFRAME } c_{f_i} k; \text{EVAL}; \text{DO} & (m > k) \\ \mathcal{S}[[e_1, \dots, e_m]] \rho n; \text{JFUN } c_{f_i} & (m = k) \\ \mathcal{C}[[e_m]] \rho n; \dots \mathcal{C}[[e_1]] \rho (n + m - 1); \text{MKCAP } c_{f_i} k; \text{RETURN} & (m < k) \end{cases}$	
(4)	$\mathcal{R}[[c_k e_1 \cdots e_m]] \rho n$	$= \mathcal{C}[[c_k e_1 \cdots e_m]] \rho n; \text{RETURN}$	
(5)	$\mathcal{R}[[\text{case } e \text{ in } c_{k_1} x_1 \cdots x_m : e_1 \parallel \dots \text{end}]] \rho n$	$= \mathcal{E}[[e]] \rho n; \text{CASE } (l_1, l_2, \dots);$ $l_1 : \text{SPLIT}; \mathcal{R}[[e_1]] \rho [x_1 = n + 1 \cdots x_m = n + m] (n + m)$ $\dots$	
(6)	$\mathcal{E}[[f_i^k e_1 \cdots e_m]] \rho n$	$= \mathcal{C}[[f_i^k e_1 \cdots e_m]] \rho n; \text{EVAL}$	$(m = k)$
(7)	$\mathcal{E}[[c_k e_1 \cdots e_m]] \rho n$	$= \mathcal{C}[[c_k e_1 \cdots e_m]] \rho n;$	
(8)	$\mathcal{E}[[x]] \rho n$	$= \text{PUSH } (n - \rho(x)); \text{EVAL}$	
(9)	$\mathcal{C}[[f_i^k e_1 \cdots e_m]] \rho n$	$= \begin{cases} \mathcal{C}[[e_m]] \rho n; \dots \mathcal{C}[[e_1]] \rho (n + m - 1); \text{MKFRAME } c_{f_i} m; & (m = k) \\ \mathcal{C}[[e_m]] \rho n; \dots \mathcal{C}[[e_1]] \rho (n + m - 1); \text{MKCAP } c_{f_i} k; & (m < k) \end{cases}$	
(10)	$\mathcal{C}[[c_k e_1 \cdots e_m]] \rho n$	$= \mathcal{C}[[e_m]] \rho n; \dots \mathcal{C}[[e_1]] \rho (n + m - 1); \text{MKCONSTR } k m;$	
(11)	$\mathcal{C}[[x]] \rho n$	$= \text{PUSH } (n - \rho(x))$	
(12)	$\mathcal{S}[[e_1, \dots, e_m]] \rho n$	$= \mathcal{C}[[e_m]] \rho n; \dots \mathcal{C}[[e_1]] \rho (n + m - 1); \text{SQUEEZE } m n;$	

Table 1: Compilation rules to  $\langle \nu, G \rangle$ -machine code.

$\mathcal{C}[[e]] \rho n$  code to construct the graph for  $e$  and leave it on the top of the stack,  
 $\mathcal{S}[[e_1, \dots, e_m]] \rho n$  code to rearrange the stack for a tail call, so that the contains pointers to the graphs of the expressions  $e_1 \cdots e_m$  (and those only); in this paper  $\mathcal{S}[[\ ]]$  has a rather simpleminded formulation, and pushes all expressions on the stack and then moves them all down to the base of the stack using the SQUEEZE instruction — in an actual implementation only those locations that actually needs to be updated are actually meddled with.

The compilation rules are given in table 1. It would seem that there are a number of cases missing in the compilation schemes — however, these are transformed away prior to  $\langle \nu, G \rangle$ -code generation. This has to do with how we deal with higher order functions in the  $\langle \nu, G \rangle$ -machine. To simplify the implementation, a frame node must always hold a number of arguments equal to the arity of the applied function. In other words, a frame node always represents an application  $f^k e_1 \cdots e_k$  where  $f^k$  is a global function of  $k$  args. This means that wherever we want to *build* an application, it must be transformed into this form. In other cases, i.e. if the function is of unknown arity (a variable) or a global function where the arity does not match the number of arguments, ‘lambda-lifting’ is done. For example,  $\mathcal{C}[[x e_1 e_2]] \rho n \Rightarrow \mathcal{C}[[f' x y]] \rho n$  with a new global function  $f' x y = x e_1 e_2$  if  $x$  and  $y$  are the free variables of the expression  $x e_1 e_2$ . In this manner, all the difficult cases are pushed into one place, the  $\mathcal{R}[[\ ]]$  scheme, which can handle all the cases using the DO instruction which performs a general tail call. Similarly,  $\mathcal{C}[[\text{case} \cdots]] \rho n$  is lambda-lifted in the same manner since we have no direct way to build a representation of an unevaluated case expression. Further for the sake of simplicity, we ignore  $\mathcal{E}[[\text{case} \cdots]] \rho n$  — it would be similar

to  $\mathcal{R}[[\text{case} \cdots]] \rho n$ , with code in each branch to evaluate the expression, adjust the stack, and jump to code succeeding the case.

## The abstract machine

The nodes that can occur in the graph are of the following type:

- CAP  $a c s$  A function application with too few arguments, thus unreducible. The code for the function is  $c$ , the function has arity  $a$ , and the available arguments are  $s$ . Since there are too few arguments the size of  $s$  is less than  $a$  ( $\#s < a$ ).
- FRAME  $c d s$  This node represents a function applied to exactly the right number of arguments. The code for the function is  $c$  and the arguments are  $s$  ( $\#s = \text{arity of the function}$ ). During reduction of the frame node the  $d$  field holds a pointer to the calling frame (the ‘dynamic link’). When the frame node is created this field is set to  $\emptyset$  (nil).
- CONSTR  $k s$  A node representing a constructor value,  $k$  being the constructor number, and  $s$  the list of arguments. CONSTR nodes are used to represent lists, integers etc.

A state in the sequential abstract machine is described by a pair  $\langle \nu, G \rangle$ , where  $G$  is the graph being reduced, i.e., a mapping from node names to nodes, and  $\nu$  is the current point of reduction, as explained in the introduction. In the transition rules in table 2 we write a state in the machine like for example

$$\langle \nu, \{ \nu : \text{FRAME } c s d \} \rangle$$

both to the left of  $\Rightarrow$  pattern matching fashion, and to the right of  $\Rightarrow$  to indicate state change. Nodes in the graph are written within  $\{ \}$ . The parts of a state not shown in a transition rule are not affected by the transition.

## Instruction transition rules

Table 2 shows the instruction transition rules for the instructions emitted by the compilation schemes.

The EVAL instruction, rules (1s), (2s) and (3s), reduces the node on the top of the stack (we always mean of the currently active frame, omitted henceforth) to canonical form, i.e., either a CONSTR node or a CAP node. If the node to be reduced is a frame node, rule (1s), (also shown in figure 2) the point of reduction is moved to the frame node to be reduced, setting the dynamic link pointer to the old frame (this field must be nil ( $\emptyset$ ) previously, otherwise we have a circular data dependency). EVAL on a CAP node or a CONSTR node is a no-operation, since these node are already canonical.

The DO instruction is by far the most complicated one; it performs a general tail call to the function on the top of the stack (assumed to be evaluated to a CAP node), with the rest of the stack containing the arguments. There are three cases to consider, depending on how the total number of arguments in the CAP node and the stack compares with the arity of the function (the  $a$  field in the CAP node).

First consider the case when the number of arguments of the CAP node and the frame node together are less than the arity of the function represented by the code of the CAP node, case (4s). Then the frame node is turned into a CAP node, holding all the arguments, and a return to the calling frame is performed.

Next, consider the case when the total number of argument is exactly equal to the arity of the function, case (5s), hence the application is reducible. Then the arguments of the CAP node are simply lifted up to the frame node, and execution continues with the code for the function.

Finally, consider the case when the number of arguments is greater than the arity of the function, case (6s). Since a FRAME node cannot contain ‘too many’ arguments, the arguments of the CAP node cannot simply be lifted up to the frame node as in the previous case. Instead, a new frame node is constructed with the first  $a_1$  arguments of the list  $s_1@s$  (@ denotes stack concatenation;  $a_1$  is the arity of the function,  $s_1$  are the arguments of the CAP node, and  $s$  are the rest of the arguments from the frame node). Then this node is reduced by a call to EVAL, where the result must be a new CAP node. Finally, DO is executed again.

A ‘conventional’ tail call to a function of known arity and with exactly the right number of arguments (which is by far the most common way a tail call is made) is done with a JFUN instruction, case (7s), which jumps to the code for the called function.

The RETURN instruction, case (8s), copies the node pointed to from the top of the stack (assumed to be a canonical node) onto the current frame node, and then a return is done by moving the reduction pointed to by the  $d$  field of the frame node.

The MKFRAME instruction, case (9s), constructs a frame node, with the arguments taken from the top of the stack.

The instructions MKCAP  $c_f a m$  and MKCONSTR  $k m$  work analogously to MKFRAME, we omit the description of these for brevity.

The PUSH instruction, case (10s), takes a pointer from the stack and puts it on the top of the stack. The SQUEEZE  $m n$  instruction, case (11s), slides the top  $m$  elements of the stack to the base of the stack. The CASE instruction, case (12s), performs a case jump, depending on the constructor number of a CONSTR node. The SPLIT instruction, case (13s), brings up the pointer from a constructor node to the top of the stack.

## 3 Technical details of the parallel abstract machine

The  $\langle \nu, G \rangle$ -machine described so far performs sequential graph reduction in a manner similar to the G-machine, the basic difference being that in the  $\langle \nu, G \rangle$ -machine individual stack frames are held in the graph. We have done this modification to pave the way for parallelism. We now show the modifications to the sequential machine that has to be made to turn it into a parallel one.

In the functional program, parallelism is introduced with a **spark** annotation<sup>1</sup>,

$$e ::= \text{spark } x e$$

the intention being that the expression bound to the variable  $x$  may be evaluated if there is a processor available to do it. The following two compilation rules apply:

$$\begin{aligned} \mathcal{R}[\text{spark } x e]_{\rho n} &= \text{PUSH } (n - \rho(x)); \text{SPARK}; \mathcal{R}[e]_{\rho n} \\ \mathcal{E}[\text{spark } x e]_{\rho n} &= \text{PUSH } (n - \rho(x)); \text{SPARK}; \mathcal{E}[e]_{\rho n} \end{aligned}$$

There is no corresponding  $\mathcal{C}[\ ]$  rule, instead such expressions are ‘lambda lifted’, i.e., turned into a function applied to the free variables of the expression. Thus, sparking happens when the expression in which the spark occurs is evaluated.

A state in the parallel machine is described by a triple  $\langle N, S, G \rangle$  where  $N$  is a *set* of reduction points (instead of a single one, as before). Reduction is taking place at these points concurrently.  $S$  is the ‘spark pool’, a set of node pointers. It holds nodes that have been sparked, but for which evaluation has not yet started (described further below). We also introduce a new node type FRAME\*:

FRAME\*  $c s$  is a ‘newborn’ frame node, i.e., one which an EVAL has not yet started reducing.

Table 3 shows the transition rules for the parallel  $\langle \nu, G \rangle$ -machine (trivial differences to the sequential machine are not shown.)

Rule (1p) describes the SPARK instruction: it puts a pointer to a node into the spark pool  $S$ , possibly to be picked up later by a free processor.

We change the meaning of the instruction MKFRAME, so that it now creates a FRAME\* node, rule (2p).

The rule for EVAL, rule (3p), has to be changed slightly from the sequential case: EVAL is ‘activated’ only if the node in question is a FRAME\*, i.e., one for which evaluation

<sup>1</sup>In LML the concrete syntax is  $e\{\text{SPARK } x\}$ , see appendix A.

(1s)	$\langle \nu, \left\{ \begin{array}{l} \nu : \text{FRAME (EVAL.c)} d (\nu_1.s) \\ \nu_1 : \text{FRAME } c_1 \emptyset s_1 \end{array} \right\} \rangle$	$\Rightarrow$	$\langle \nu_1, \left\{ \begin{array}{l} \nu : \text{FRAME } c d (\nu_1.s) \\ \nu_1 : \text{FRAME } c_1 \nu s_1 \end{array} \right\} \rangle$	
(2s)	$\langle \nu, \left\{ \begin{array}{l} \nu : \text{FRAME (EVAL.c)} d (\nu_1.s) \\ \nu_1 : \text{CAP } a_1 c_1 s_1 \end{array} \right\} \rangle$	$\Rightarrow$	$\langle \nu, \left\{ \begin{array}{l} \nu : \text{FRAME } c d (\nu_1.s) \\ \nu_1 : \text{CAP } a_1 c_1 s_1 \end{array} \right\} \rangle$	
(3s)	$\langle \nu, \left\{ \begin{array}{l} \nu : \text{FRAME (EVAL.c)} d (\nu_1.s) \\ \nu_1 : \text{CONSTR } k s_1 \end{array} \right\} \rangle$	$\Rightarrow$	$\langle \nu, \left\{ \begin{array}{l} \nu : \text{FRAME } c d (\nu_1.s) \\ \nu_1 : \text{CONSTR } k s_1 \end{array} \right\} \rangle$	
(4s)	$\langle \nu, \left\{ \begin{array}{l} \nu : \text{FRAME (DO.c)} d (\nu_1.s) \\ \nu_1 : \text{CAP } a_1 c_1 s_1 \end{array} \right\} \rangle$	$\Rightarrow$	$\langle d, \left\{ \begin{array}{l} \nu : \text{CAP } a_1 c_1 (s_1 @ s) \\ \nu_1 : \text{CAP } a_1 c_1 s_1 \end{array} \right\} \rangle$	$(\#s + \#s_1 < a_1)$
(5s)	$\langle \nu, \left\{ \begin{array}{l} \nu : \text{FRAME (DO.c)} d (\nu_1.s) \\ \nu_1 : \text{CAP } a_1 c_1 s_1 \end{array} \right\} \rangle$	$\Rightarrow$	$\langle \nu, \left\{ \begin{array}{l} \nu : \text{FRAME } c_1 d (s_1 @ s) \\ \nu_1 : \text{CAP } a_1 c_1 s_1 \end{array} \right\} \rangle$	$(\#s + \#s_1 = a_1)$
(6s)	$\langle \nu, \left\{ \begin{array}{l} \nu : \text{FRAME (DO.c)} d (\nu_1.s) \\ \nu_1 : \text{CAP } a_1 c_1 s_1 \end{array} \right\} \rangle$	$\Rightarrow$	$\langle \nu, \left\{ \begin{array}{l} \nu : \text{FRAME (MKFRAME } a_1 c_1 \text{.EVAL.DO.c)} d (s_1 @ s) \\ \nu_1 : \text{CAP } a_1 c_1 s_1 \end{array} \right\} \rangle$	$(\#s + \#s_1 > a_1)$
(7s)	$\langle \nu, \left\{ \nu : \text{FRAME (JFUN } c_f.c) d s \right\} \rangle$	$\Rightarrow$	$\langle \nu, \left\{ \nu : \text{FRAME } (c_f) d s \right\} \rangle$	
(8s)	$\langle \nu, \left\{ \begin{array}{l} \nu : \text{FRAME (RETURN.c)} d (\nu_1.s) \\ \nu_1 : N \end{array} \right\} \rangle$	$\Rightarrow$	$\langle d, \left\{ \begin{array}{l} \nu : N \\ \nu_1 : N \end{array} \right\} \rangle$	
(9s)	$\langle \nu, \left\{ \nu : \text{FRAME (MKFRAME } c_f m.c) d (\nu_1 \dots \nu_m.s) \right\} \rangle$	$\Rightarrow$	$\langle \nu, \left\{ \begin{array}{l} \nu : \text{FRAME } c d (\nu'.s) \\ \nu' : \text{FRAME } c_f \emptyset [\nu_1; \dots; \nu_m] \end{array} \right\} \rangle$	
(10s)	$\langle \nu, \left\{ \nu : \text{FRAME (PUSH } m.c) d (\nu_0 \dots \nu_m.s) \right\} \rangle$	$\Rightarrow$	$\langle \nu, \left\{ \nu : \text{FRAME } c d (\nu_m \nu_0 \dots \nu_m.s) \right\} \rangle$	
(11s)	$\langle \nu, \left\{ \nu : \text{FRAME (SQUEEZE } m n.c) d (\nu_1 \dots \nu_m \nu'_1 \dots \nu'_n.s) \right\} \rangle$	$\Rightarrow$	$\langle \nu, \left\{ \nu : \text{FRAME } c d (\nu_1 \dots \nu_m.s) \right\} \rangle$	
(12s)	$\langle \nu, \left\{ \begin{array}{l} \nu : \text{FRAME (CASE } (l_0 l_1 \dots) \dots l_i : c) d (\nu'.s) \\ \nu' : \text{CONSTR } k_i s' \end{array} \right\} \rangle$	$\Rightarrow$	$\langle \nu, \left\{ \begin{array}{l} \nu : \text{FRAME } c d (\nu'.s) \\ \nu' : \text{CONSTR } k_i s' \end{array} \right\} \rangle$	
(13s)	$\langle \nu, \left\{ \begin{array}{l} \nu : \text{FRAME (SPLIT.c)} d (\nu'.s) \\ \nu' : \text{CONSTR } k s' \end{array} \right\} \rangle$	$\Rightarrow$	$\langle \nu, \left\{ \begin{array}{l} \nu : \text{FRAME } c d (s' @ s) \\ \nu' : \text{CONSTR } k s' \end{array} \right\} \rangle$	

Table 2: Instruction transition rule for the sequential  $\langle \nu, G \rangle$ -machine.

---

(1p)	$\langle \{\nu\} \cup N, S, \left\{ \nu : \text{FRAME (SPARK.c)} d (\nu'.s) \right\} \rangle$	$\Rightarrow$	$\langle \{\nu\} \cup N, \{\nu'\} \cup S, \left\{ \nu : \text{FRAME } c d s \right\} \rangle$
(2p)	$\langle \{\nu\} \cup N, S, \left\{ \nu : \text{FRAME (MKFRAME } c_f m.c) d (\nu_1 \dots \nu_m.s) \right\} \rangle$	$\Rightarrow$	$\langle \{\nu\} \cup N, S, \left\{ \begin{array}{l} \nu : \text{FRAME } c d (\nu'.s) \\ \nu' : \text{FRAME}^* c_f [\nu_1; \dots; \nu_m] \end{array} \right\} \rangle$
(3p)	$\langle \{\nu\} \cup N, \left\{ \begin{array}{l} \nu : \text{FRAME (EVAL.c)} d (\nu_1.s) \\ \nu_1 : \text{FRAME}^* c_1 s_1 \end{array} \right\} \rangle$	$\Rightarrow$	$\langle \{\nu_1\} \cup N, \left\{ \begin{array}{l} \nu : \text{FRAME } c d (\nu_1.s) \\ \nu_1 : \text{FRAME } c_1 \nu s_1 \end{array} \right\} \rangle$
(4p)	$\langle N, \{\nu\} \cup S, \left\{ \nu : \text{FRAME}^* c s \right\} \rangle$	$\Rightarrow$	$\langle \{\nu\} \cup N, S, \left\{ \nu : \text{FRAME } c \dagger s \right\} \rangle$
(5p)	$\langle N, \{\nu\} \cup S, \left\{ \nu : \text{FRAME } c d s \right\} \rangle$	$\Rightarrow$	$\langle N, S, \left\{ \nu : \text{FRAME } c d s \right\} \rangle$
(6p)	$\langle N, \{\nu\} \cup S, \left\{ \nu : \text{CAP } c a s \right\} \rangle$	$\Rightarrow$	$\langle N, S, \left\{ \nu : \text{CAP } c a s \right\} \rangle$
(7p)	$\langle N, \{\nu\} \cup S, \left\{ \nu : \text{CONSTR } k s \right\} \rangle$	$\Rightarrow$	$\langle N, S, \left\{ \nu : \text{CONSTR } k s \right\} \rangle$
(8p)	$\langle \{\dagger\} \cup N, \{\nu\} \cup S, \left\{ \right\} \rangle$	$\Rightarrow$	$\langle N, S, \left\{ \right\} \rangle$

Table 3: Transition rules for the parallel  $\langle \nu, G \rangle$ -machine. Only nontrivial differences to the sequential machine are listed.

has not yet been started (by another process): There is no reduction rule for EVAL on a FRAME node; this means that that process ‘blocks’ until the node has become canonical. Just as in the sequential machine, EVAL on a canonical node is a no-operation (not shown in table 3).

Rules (4p) to (7p) shows what happens to the node pointer in the spark pool. A pointer to a node can be taken from the spark pool for reduction (by an otherwise idle processor, but that is not visible at this level of abstraction). If the node is a FRAME\* one, rule (4p), then no-one else has started to reduce the node, so the node is changed into a FRAME and reduction of it is commenced. If the pointer taken out from the spark pool points to a FRAME node, rule (5p), then some other process has already started to reduce this node, so this pointer is discarded; similarly, if the node is a CAP or CONSTR node (rules 6p and 7p), the node cannot be reduced further and the pointer to it is also discarded.

The ‘dynamic link’ pointer † put into the  $d$  field of the FRAME node when reduction is commenced, indicates that when this frame node has been reduced, the process reducing it should die (rule 8p).

All the other instruction work exactly as in the sequential machine.

## 4 Implementation

An abstract machine is a good way to get a handle on language implementation complexity in general. If the abstract machine is going to be useful for a real implementation it should not be too far from the actual target machine. Both the G-machine and the  $\langle \nu, G \rangle$ -machine were designed with ordinary von Neumann computers in mind.

The mapping of the two components of the abstract machine to a concrete is quite simple: the graph is stored in a garbage collected heap and the node under reduction is a pointer into the graph that is kept in a machine register. The code for the functions are not kept in the nodes, of course, it is instead located in memory and the nodes only contain a pointer to the next instruction and while executing the code pointer is naturally kept in the pc in the processor.

In the abstract machine the FRAME nodes contain a stack. The  $\langle \nu, G \rangle$ -machine has been designed so that the maximum size of this stack is often easy to compute at compile time. This is a major reason why we decided to disallow ‘excess’ arguments (i.e. more arguments than the arity of the function) in frame nodes, as it greatly simplifies this computation. If we, for a moment, disregard the tail call instructions (JFUN and DO) it is quite easy to compute the maximum size of the stack that is going to be used in a FRAME node. The size is simply the maximum value that the stack depth parameter in the compilation schemes ever has. The JFUN instruction makes it somewhat more complex to determine the maximum size (remember that a tail call re-uses the old frame), but it is still possible. In this case the maximum size is the maximum size of all the frames for functions that can be reached via tail calls using JFUN. Whenever a FRAME node is created it is created with its computed maximum size. However, if the function contains a DO instruction it is much more difficult to compute the maximum size. It would be possible to use a very pes-

simistic worst case analysis (by looking at all the functions in the program), but we use a different scheme. In case of a DO instruction the node may have to change its size, this is done by allocating a new node and remaking the old node into an indirection node. Here Lester’s analysis technique [Les89] would be eminently applicable — see section 6.

Since the actual stack depth is known at all points during compilation there is no need to keep a stack pointer into the node; the pointer to the FRAME node itself,  $\nu$ , is sufficient. All the stack positions are simply fixed offsets into the node. Hence, it is important that the initial stack in a FRAME (i.e., before reduction begins) contains exactly the number of arguments that is needed for the functions to reduce, if excess arguments were kept in the stack it would not be possible to use fixed offsets anymore; neither would the maximum frame size be easy to compute.

A FRAME node has the following fields (the flags used for parallel evaluation are explained further in the next section):

tag	this field holds various flags and it identifies this node as a FRAME node. The flags indicate that: <ul style="list-style-type: none"> <li>• the node is already evaluated (this flag is not set for FRAME nodes; only for CAP and CONSTR nodes).</li> <li>• the node is under evaluation (corresponds to the difference between FRAME* and FRAME).</li> <li>• the node tag and/or <math>d</math> field are being manipulated (to guarantee atomicity of EVAL and RETURN).</li> <li>• the node has processes awaiting its evaluation. If this flag is set the <math>d</math> field points to the first of these processes and the last of them contains the real <math>d</math> value.</li> </ul>
pc	a pointer to the next instruction to be executed. When a node is the node under reduction this value is (of course) not kept in the node itself, but instead in the program counter in the CPU. It is restored and loaded when reduction leaves and enters the node.
$d$	a pointer to the previous node under reduction (just as in the abstract machine).
args	a number of words that initially contain the arguments to the function and some empty words that are used during reduction.

A CAP node has the following fields:

tag	flags and identification
pc	pointer to code for the function of this partial application.
arity	the function arity.
fsize	the size of the frame this function needs.
args	the arguments of the partial application.

A CONSTR node has the following fields:

tag	flags and identification
cno	constructor number.
args	subparts of the constructor.

The CAP node does not need to have any extra space because it will never be rewritten, but it does need to have

information about the size of the frame that is needed for the function since the DO instruction may have to create a new FRAME. The CONSTR node simply contains a tag, the constructor number, and the subcomponents.

The mapping of the abstract instructions to actual machine instructions is mostly straightforward. The PUSH instruction simply becomes a move, and the SQUEEZE becomes a series of moves. Memory allocation is done by keeping a pointer to the next free location in the heap. The allocating instructions (MKFRAME, MKCONSTR, and MKCAP) check for heap overflow (and invokes that garbage collector in that case) and then simply fills to heap area and change the pointer to the next free location. The EVAL instruction should do nothing for canonical nodes and should move the point of reduction for a FRAME node. This can be accomplished by a single bit in the tag that specifies if the node is canonical or not. If it is not canonical the current pc and  $\nu$  are saved (in the old and new frames respectively) and the new ones loaded instead.

In the abstract machine the RETURN instruction changes the contents of a FRAME node. The real RETURN instruction either remakes the contents of the node or remakes it into an indirection node depending on whether it is preceded by and EVAL or not. It then restores the old  $\nu$  and program counter and returns to the caller. The DO instruction is without any doubt the most complicated one to implement. Fortunately only one copy of the code for it is needed, and a DO is translated into code that loads the stack size into a register and jumps to this code in the run time system. This instruction is also not used very often (see the performance section below). There are three different cases that DO has to handle: too few, too many, and the right number of arguments. In the case of too few and the right number of arguments the size of the node that the FRAME should become may be larger than its actual size. In this case a new node is allocated and the old one is rewritten into an indirection. This is the only place where the size of a node changes and a new node has to be allocated. The DO instruction looks complex, but since it is not used very often (most tail calls executed are through JFUN, see below) and the code for it is not very long the actual time spent in it is less than 20% of the total time.

Our current implementation of the  $\langle \nu, G \rangle$ -machine is quite simple and the code that it produces is far from optimal. The FRAME nodes can be made smaller by re-using unused frame entries (live variable analysis) and with careful register allocation.

## Implementation of the parallel machine

Our implementation of the parallel  $\langle \nu, G \rangle$ -machine is on a shared memory multi-processor, a Sequent Symmetry<sup>TM</sup>. The Sequent has one global address space and there is no local memory.<sup>2</sup> This kind of machine is the simplest for doing parallel graph reduction since there is much less need to try to keep locality. Furthermore, the Sequent Symmetry has the ability (through special instructions) to use any cell of the memory as a an atomic lock, a fact that we have taken advantage of.

---

<sup>2</sup>There is in fact a cache for each CPU, but that is invisible to the programmer.

There are two new problems that have to be solved when mapping the parallel  $\langle \nu, G \rangle$ -machine to a real machine: handling atomic instructions and waiting.

In the abstract machine all instructions are atomic, but on a real machine some of them will be implemented by several instructions. In the case where the concurrently executing processes may interact atomicity has to be guaranteed. This interaction can only take place during manipulation of the FRAME node (except for the spark pool, see below). The EVAL instruction examines a node and takes some action according to the node type and RETURN changes the node type. It is only the execution of these two instructions (operating on the same node) that need exclusive access to the node. To get exclusive access to a node there is a flag bit in the tag. Before any manipulation of the tag or  $d$  part of a node the processor must have exclusive access. This is guaranteed by doing test-and-set on the flag bit and doing busy-wait until the lock is acquired. The busy-waiting is in fact very seldom done, since the time during which the lock is set is very short. Once the EVAL instruction has acquired the lock it changes the node from a FRAME\* to a FRAME node, saves the old  $\nu$ , releases the lock, and starts executing the code for the function.<sup>3</sup>

The RETURN instruction acquires the lock, updates the tag, and releases the lock. If there are any processes suspended, waiting for this node to become evaluated, they are awakened (see below).

The second problem, the waiting, occurs because in the abstract machine a process stops doing reductions when trying to evaluate a FRAME node. In the abstract machine this is simply expressed by having no reduction rules involving EVAL on a FRAME node. The corresponding simple solution on a real machine would be to use busy-wait until the node has been reduced. Busy-waiting in this case is totally unacceptable since the waiting can be arbitrarily long. If we have a limited number of processors this can lead to deadlock at worst, or a terrible slowdown at best.

Thus, we need a process suspension mechanism so that the process can be waiting, but the processor can continue doing something else. When a process is created, i.e., when an idle processor takes a node from the spark pool, it gets a process control block. This is a node in the heap that is used during process suspension, it contains space to save relevant process status (essentially just  $\nu$ ) and also a link field for making lists of processes. The condition that awakens a process is that a node becomes evaluated. This means that we need some efficient way to wait for a certain node to change status. The  $d$  field in the FRAME nodes actually doubles as a queue header. If there are any processes waiting for a node to become evaluated these processes are hanging on a list, created by linking together the control blocks for the processes, starting from the  $d$  field. The fact that a node has a queue at all is recorded in yet another bit in the tag. The original contents of the  $d$  field of the node is stored in the last process control block.

Process suspension, which occurs when evaluating a FRAME node, simply takes the  $d$  field and stores it in the link field of the process control block and then sets the queue bit in the tag field. Process awakening, which takes place

---

<sup>3</sup>In reality it is a little more complex than this since the node may not be a FRAME node anymore once the lock is acquired.

when updating a node with a queue, takes the list of processes, puts them on the run queue, and then restores the original contents of the  $d$  field.

There may be more reduction points than there are actual processors. This is handled in the standard way by having a queue of runnable processes, and having the processors alter between them. For the moment we have a very simple round robin scheduling algorithm. If the run queue is empty a node is picked from the spark pool, a process is created for it, and the process is added to the run queue.

Both the suspension and awaking operation take a fixed small time to execute (around 15 machine instructions each). Suspension and awaking are in fact quite rare events; the normal execution of the EVAL and RETURN instructions are disturbed very little by the fact that they may occur (2 instructions each). The additional cost of having the parallel machine is mainly in the locking that is needed during EVAL and RETURN. Both of these instructions need about 5 machine instructions extra to take care of the locking.

If there is going to be any parallelism in the program SPARK instructions have to be executed. These also take time (of course), but we have tried to keep this as short as possible. The current implementation makes use of the fact that sparks are advisory and it does not matter if sparks are lost. This means that insertion in and extraction from the spark pool does not have to be guarded by any mutual exclusion. Not having mutual exclusion means that sparks may be lost (because of simultaneous write in the spark pool) or duplicated (during extraction), but this can only affect the run time behaviour of the program, not the final result.

The current implementation uses time slicing to guarantee that all reduction points make progress. We would like to make sure that the normal order branch of the computation proceeds at full speed, but this requires priorities and handling these incurs more overhead. A future implementation may contain priorities.

LML programs (which is what we compile) are allowed to do input from the terminal and from other devices, and this means that a processor could block for an undetermined amount of time in a read operation. This is avoided by using the suspension mechanism in much the same way as described above and using interrupt based input (which is available under BSD 4.3 UNIX).

Memory allocation is done by having each processor allocate a chunk of the heap space when it runs out of heap. It then allocates from this chunk until it has been consumed. This means that there is no contention for allocating memory. When the whole heap has been consumed (which is detected by a processor trying to allocate another chunk) there is a need for garbage collection. This is signaled to all the other processors; when they are all synchronised a collection is performed. The current collector is a completely sequential copying collector, but we plan to use a concurrent collector along the lines in [AEL88].

## 5 Performance

We have made a number of measurements to see where the bottlenecks are in the  $\langle \nu, G \rangle$ -machine and then tried to avoid or speed up these. The statistics were gathered with ordinary sequential reduction (except for enqueue/dequeue),

running small and some fairly large programs such as the compiler itself. Some observations:

- Most of the EVAL instructions (about 70%) try to evaluate something on canonical form and should do nothing. This has led us to implement EVAL by a test of the canonical bit and a jump to out-of-line code if there is need for evaluation. This means that in most cases there is no break in the control flow which is important for many modern pipelined machines.
- There are few DO instructions (about 15%) executed compared to the total number of executed tail calls. This means that the implementation of it is not so critical. With the simple scheme for determining the node size when a DO instruction is present the node size has to be increased (requiring allocation of a new node) in about 70% of all DO instructions. Since the DO instructions are relatively infrequent the time spent in DO (including the time for allocation of bigger nodes) is typically less than 5%.
- It is very rare for a node to have a queue ( $< 0.05\%$  of all RETURN), this fact has led to an implementation where there is straight line code for the common case and a call to a dequeuing routine in the run time system in case of a queue.
- It is also rare for a process to become suspended ( $< 0.1\%$  of all EVAL), this fact has also led to straight line code for the common case and a call for the rare case.
- When a program with sufficient parallelism (i.e., enough to keep all processors busy all the time) is run typically less than 10% of the processor time is spent in overhead. The overhead includes process switching, process creation, and searching the spark pool for work.
- There is almost no contention in accessing the run-queue (which needs mutual exclusion). The time spent waiting for it is less than 0.01%.

The timings for the programs the tables and figures do not include garbage collection time because the current collector is both inefficient and sequential (nevertheless GC seldom exceeds 30% of the total time). The current implementation of the  $\langle \nu, G \rangle$ -machine has about the same speed as the conventional G-machine implementation used in the old LML compiler (as can be seen from table 4). We expect that using the same degree of sophistication in code generation as the old implementation will speed up the  $\langle \nu, G \rangle$ -machine by at least 25%.

A better code generator will speed up the execution for several reasons. Using registers instead of stack locations is faster per se, but it also gives smaller frames. Smaller frames speeds up both allocation and garbage collection. The current implementation uses 2-3 times more memory than the old G-machine. With a better code generator this should decrease significantly.

As we can see the speedup is not linear for all programs, but levels off. We are still investigating the exact reasons for this, but a likely cause is that the bus gets saturated, since our measurement indicate that the idle and overhead

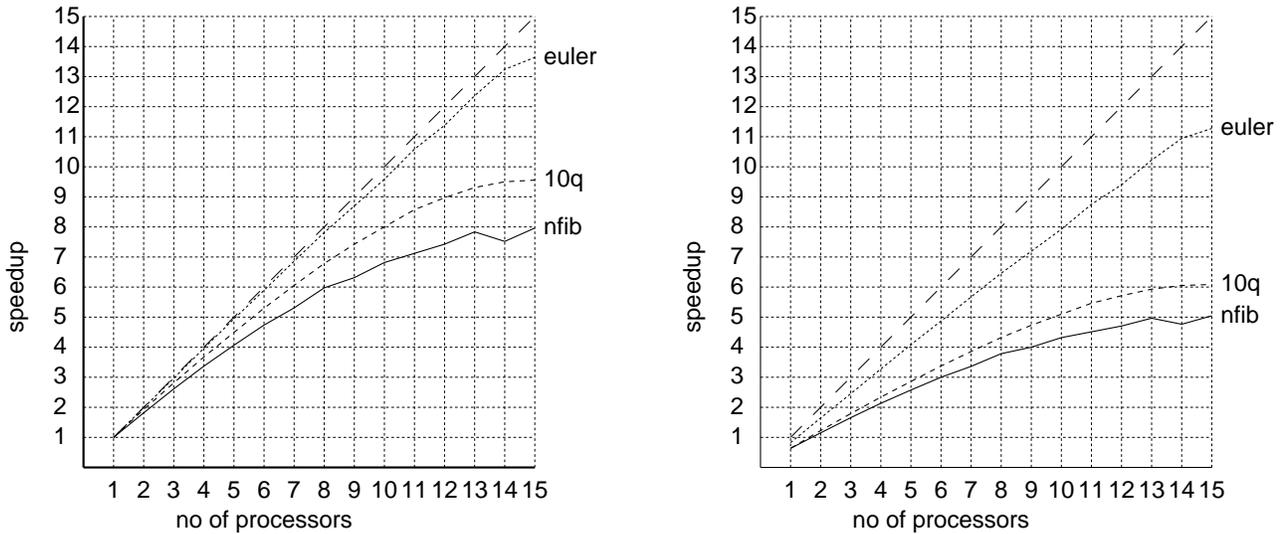


Figure 4: Speedup graphs for three benchmark programs. The left graph shows speedup relative to one processor, the right graph shows speedup relative to the old G-machine in the LML compiler.

time is less than the time that is lost. For conventional languages (e.g. C) this saturation does not happen until more processors are running. The  $\langle \nu, G \rangle$ -machine has less locality due to the many heap allocations, which is further accentuated by the fact that heap memory is used instead of a stack. A better code generator which utilises registers more will alleviate this somewhat.

The absolute performance of the  $\langle \nu, G \rangle$ -machine is given in table 4.

The ‘nfib’ program is the usual test program for function calls. It generates very much parallelism, and as can be seen from in table 4 the time consumed by the spark operation is very significant. Nfib only performs arithmetic and function calls and is thus a poor benchmark program.

The ‘euler’ program computes the Euler totient function (in the most naïve way) for the first 1000 numbers. The numbers are summed to avoid outputting a long list. This program behaves in a very nice way; essentially every new process that is created produces another spark that will give rise to a new process. This means that the program produces the exact “right” amount of parallelism, all processors have work to do, but there are few excess sparks produced. Each process that is created is relatively large, so process creation overhead is less critical. This program also shows a very nice speedup. Euler does a fair amount of creation and inspection of lists in addition to arithmetic.

The ‘10q’ program is a variant of the eight queens problem, but with ten queens. This program is something in between the two others; it does not produce the excessive amount of parallelism that ‘nfib’ does, neither does it adapt itself to the number of processors as ‘euler’. The ‘10q’ program does a lot of data structure handling and relatively little arithmetic.

The measurement for the graphs were made using real time clocks; the table show processor time. Since the machine was not completely quiescent during the measurement the accuracy is not as good as it could be.

## 6 Related work

There have been several proposals for parallel graph reduction architectures, we will mention those that resembles ours most.

Our abstract machine, the  $\langle \nu, G \rangle$ -machine, is very similar to the HDG-machine [KLB89], a distributed graph reducer intended for a transputer network. They have also arrived at a model the frame nodes in the heap, with a linked list of stack frames instead of a monolithic stack. However, they permit conventional ‘unrestricted’ use of the stack in the frames. In this project, Lester [Les89] have devised a sophisticated analysis technique, based on abstract interpretation, to determine a maximum size of the stack in each frame, also in the presence of higher-order functions, so that stack overflow cannot occur.

As mentioned in section 4, a major reason for us to disallow excess arguments in frame nodes in the  $\langle \nu, G \rangle$ -machine was to simplify the computation of the maximum size of stack in the frame. Still, there was a pitfall: the DO instruction, which applies an ‘unknown’ function object and thus requires a frame of unknown size – here we resort to a runtime test machinery possibly allocating a new bigger frame node, remaking the old frame node into an indirection node. Here Lesters analysis technique could be used to determine a frame size also in this case, so that it will never be necessary the use indirections. All the implications of this analysis technique to the  $\langle \nu, G \rangle$ -machine is not yet clear — it

	nfib		program			
			euler		10q	
G-machine	36.8	(1.00)	104.9	(1.00)	46.3	(1.00)
$\langle \nu, G \rangle$ -machine	42.4	(1.10)	106.4	(1.01)	65.3	(1.41)
$\langle \nu, G \rangle$ -machine+synch.	49.9	(1.29)	128.4	(1.22)	73.9	(1.60)
$\langle \nu, G \rangle$ -machine+synch.+SPARK	62.1	(1.61)	126.9	(1.21)	72.7	(1.57)

- G-machine: The old G-machine.  
 $\langle \nu, G \rangle$ -machine: The purely sequential  $\langle \nu, G \rangle$ -machine.  
 $\langle \nu, G \rangle$ -m.+synch.: The  $\langle \nu, G \rangle$ -machine with the full synchronisation machinery of the parallel machine added, but without the program doing any SPARK, using only one processor.  
 $\langle \nu, G \rangle$ -m.+synch.+SPARK: The parallel  $\langle \nu, G \rangle$ -machine, i.e., with the program doing SPARKS, but using only one processor.

Table 4: Execution time in seconds of the G-machine and the  $\langle \nu, G \rangle$ -machine for the three benchmark programs. Figures in brackets is the slowdown factor compared to the G-machine.

is possible that we may want to redesign the  $\langle \nu, G \rangle$ -machine further in the light of this work.

The  $\langle \nu, G \rangle$ -machine, with its stacks in the graph, is very similar in spirit to the packet rewriting model of both the ALICE machine [Dar81] and FLAGSHIP [WW87]. The use of the stack in the frame node of the  $\langle \nu, G \rangle$ -machine can be seen as rewriting of the frame packet until its canonical form has been reached. A difference is that with the  $\langle \nu, G \rangle$ -machine, we use ‘stock hardware’, whereas in the the ALICE and FLAGSHIP projects they design specialised hardware.

Goldbergs Alfalfa and especially Buckwheat [Gol88b, Gol88a] is akin to our machine, but the aim of this work was more to try out automatically generated parallelism than to generate good code. Goldberg uses C as an intermediate language and runs the programs on a somewhat slower machine (with much slower locking primitives). These factors taken together results in an implementation that is about two orders of magnitude slower than ours.

It is interesting to compare the spark model, where the sparks are advisory, with the model used by Goldberg, where spawn commands always create a new process. Goldberg has ‘hot-spot’ problems with the run-queue, since this is a resource that has to be shared by all processors and it is accessed every time a new process is created. Using the spark model this problem is absent. The spark pool is also a shared resource, but it does not have to be guarded by locks. Accessing the run-queue then becomes quite rare (only at process creation, suspension, and resumption).

The GRIP [CJ86, Jon87] featured the spark model for parallelism. The GRIP machine has special purpose hardware for global memory and communication. The division into local and global memory puts constraints on the design that we do not have. In [JS89] a variation of the G-machine is described; the problem of boundless stacks are handled by breaking the stack into segments. This gives better space locality and less space overhead than our approach, but it requires more checking of stack boundaries and more data movement of the arguments.

## 7 Conclusions

The implementation technique we have described here works fine for a shared memory machine. Unfortunately these machines are not very extensible since it requires constant access time to all parts of memory. An approach for large scale parallelism would have to take the communication cost into consideration.

We have already mentioned many avenues for further research. To summarise:

The target code generation can be improved considerably, to make better use of machine registers and the locations in the frame, rather than the current simpleminded stack regime.

Using Lester’s analysis technique [Les89], one can get better approximations for upper bounds on frame sizes.

Parallelism is introduced in our system by the programmer annotating the programs. We have not yet addressed the problem on how to automatically place such annotations. A suitable starting point for this would be the serial combinators [GH85].

Our experience so far with the  $\langle \nu, G \rangle$ -machine implementation is limited to a few small benchmark programs. More detailed measurements are needed to find out the reasons for the non-linear speedup for some programs, and more practical experience in general of parallelizing functional programs are needed.

## 8 Acknowledgements

We would like to thank the Programming Methodology Group and especially the Multi-Fraction for many pleasant Thursday afternoon cakes. In particular, we thank Mikael Rittri and Staffan Truvé for reading and commenting on this paper. Mikael Rittri also suggested the Euler benchmark. The Swedish Board for Technical Development (STU) provided funding for part of this work.

## References

- [AEL88] Andrew W. Appel, John R. Ellis, and Kai Li. Real-time Concurrent Collection on Stock Multiprocessors. In *Proceedings of the SIGPLAN '88 Conference on Programming Language Design and Implementation*, pages 11–20, June 1988.
- [ANT88] L. Augustsson, C. Nilsson, and S. Truvé. The PG-machine: a Machine for Parallel Graph Reduction. Technical Report Memo 57, Department of Computer Sciences, Chalmers University of Technology, S-412 96 Göteborg, 1988.
- [App87] Andrew W. Appel. Garbage collection can be faster than stack allocation. *Information Processing Letters*, 21(4):275–279, 1987.
- [Aug84] L. Augustsson. A Compiler for Lazy ML. In *Proceedings of the 1984 ACM Symposium on Lisp and Functional Programming*, pages 218–227, Austin, 1984.
- [Bur88] G.L. Burn. A Shared Memory Parallel G-machine Based on the Evaluation Transformer Model of Computation. In *Proceedings of the Workshop on Implementation of Lazy Functional Languages*, volume PMG report ???, Göteborg, September 1988. Programming methodology Group. To appear.
- [CJ86] C. Clack and S.L. Peyton Jones. The Four-Stroke Reduction Engine. In *Proceedings of the 1986 ACM Conference on LISP and Functional Programming*, pages 220–232, 1986.
- [Dar81] J. Darlington. Alice: A Multi-Processor Reduction Machine for the Parallel Evaluation of Applicative Languages. In *Proceedings 1981 Conference on Functional Languages and Computer Architecture*, Wentworth-by-the-Sea, Portsmouth, New Hampshire, 1981.
- [GH85] B. Goldberg and P. Hudak. Serial Combinators: Optimal Grains of Parallelism. In *Proceedings 1985 Conference on Functional Programming Languages and Computer Architecture*, pages 382–399, Nancy, France, 1985.
- [Gol88a] Benjamin Goldberg. Buckwheat: Graph Reduction on a Shared-Memory Multiprocessor. In *ACM Conference on LISP and Functional Programming*, 1988.
- [Gol88b] Benjamin F. Goldberg. *Multiprocessor Execution of Functional Programs*. PhD thesis, Yale University, Department of Computer Science, 1988.
- [Hug82] R. J. M. Hughes. Super Combinators—A New Implementation Method for Applicative Languages. In *Proceedings of the 1982 ACM Symposium on Lisp and Functional Programming*, pages 1–10, Pittsburgh, 1982.
- [Joh84] T. Johnsson. Efficient Compilation of Lazy Evaluation. In *Proceedings of the SIGPLAN '84 Symposium on Compiler Construction*, pages 58–69, Montreal, 1984.
- [Joh85] T. Johnsson. Lambda Lifting: Transforming Programs to Recursive Equations. In *Proceedings 1985 Conference on Functional Programming Languages and Computer Architecture*, Lecture Notes in Computer Science 201, Nancy, France, 1985. Springer Verlag.
- [Joh86] T. Johnsson. Code Generation from G-machine code. In *Proceedings of the workshop on Graph Reduction*, Lecture Notes in Computer Science 279, Santa Fe, September 1986. Springer Verlag.
- [Jon87] S. L. Peyton Jones. GRIP: A Parallel Graph Reduction Machine. In *Proceedings of the 1987 Conference on Functional Programming Languages and Computer Architecture*, Portland, Oregon, September 1987.
- [JS89] S.L. Peyton Jones and Jon Salkild. The Spineless Tagless G-machine. In *Proceedings of the 1989 Conference on Functional Programming Languages and Computer Architecture*, 1989.
- [KLB89] H. Kingdon, D. R. Lester, and G. L. Burn. The HDG-machine: A Highly Distributed Graph Reducer for a Transputer Network. Technical Report 123, GEC Hirst Research Centre, East Lane, Wembley, Middlesex HA7PP, UK, March 1989.
- [Les89] David R. Lester. Stacklessness: compiling recursion for a distributed architecture. In *Proceedings of the 1989 Conference on Functional Programming Languages and Computer Architecture*, 1989.
- [RHH<sup>+</sup>88] M. Raber, T. Hemmel, E. Hoffman, D. Maurer, F. Müller, H. Oberhauser, and R. Wilhelm. Compiled Graph Reduction on a Processor Network. *Informatik-Berichte*, GI/ITG Tagung Paderborn, 1988.
- [WW87] P. Watson and I. Watson. Evaluating Functional Programs on the FLAGSHIP Machine. In *Proceedings 1987 Functional Programming and Computer Architecture*, pages 80–97, 1987.

## A The benchmark programs

```
let rec nfib n =
  if n < 2 then 1
  else let r1 = nfib (n-1)
        and r2 = nfib (n-2)
        in (r1 + r2 + 1){SPARK r2}
in nfib 30
```

The nfib benchmark program

---

```
let rec map f [] = []
|| map f (x.xs) =
  let r = map f xs
  and v = f x in
  (v.r){SPARK r v}
in
let rec gcd x 0 = x
|| gcd x y = gcd y (x%y)
in
let relprime x y = gcd x y = 1
in
let euler n = length (filter (relprime n) (fromto 1 (n-1)))
in
  sum (map euler (fromto 1 1000))
```

The euler benchmark program

---

```
let rec
  concmap f [] = []
|| concmap f (a.b) =
  let r = concmap f b
  in (f a @ r){SPARK r}
in
let nsoln nq =
  let rec ok [] = true
  || ok (x.l) =
    let rec safe x d [] = true
    || safe x d (q.l) = x ~= q & x ~= q+d & x ~= q-d & safe x (d+1) l
    in safe x 1 l
  in
  let rec gen 0 = [[]]
  || gen n =
    concmap (\b.(filter ok (map (\q.q.b) (fromto 1 nq)))) (gen (n-1))
  in length (gen nq)
in nsoln 10
```

The 10q ('ten queens') program

---