Parallel Evaluation of Functional Programs: The $\langle \nu, G \rangle$-machine approach (Summary)

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

For a number of years this author, together with Lennart Augustsson, have been developing fast implementations of lazy functional languages, based on graph reduction, for ordinary (sequential) computers. Our approach can be summarised very briefly as follows.

Our ideas stem from Turner's S, K, I standard combinator reduction approach [Tur79]. But instead of using a standard, fixed set of combinators, a compiler transforms the program into a new set of specialised combinators, or 'super-combinators' [Hug82]. This transformation process is called *lambda lifting* [Joh85]. Each of these super-combinators are then compiled into machine code for the machine at hand, this code implements the corresponding graph rewrite rule. In other words, the compiler constructs a specialised, machine-language coded combinator interpreter from each program. However, rather than compiling each combinator into machine code directly, we first compile them into code for an abstract machine, the *G-machine* [Joh84]. Also, rather than letting the code rewrite the graph for a combinator application into the graph of the right hand side of the combinator definition, quite a lot of improvements to this scheme is possible. The G-machine is a convenient abstraction for expressing these improved compilation schemes.

An overview of the techniques used in our compiler for Lazy ML can be found in [Aug84] and [AJ89b]. The compilation of pattern matching into efficient code is described in [Aug85]. Our method of lambda lifting is described in [Joh85], and the G-machine is described in [Joh84]. The approach to machine code generation used in the Lazy ML compiler is described in [Joh86].

Parallel computers, consisting of dozens, hundreds, or thousands of processors connected to either a shared memory or a message passing network, are now becoming available on the marketplace. Recently, we have done work on extending the G-machine techniques to perform parallel graph reduction on such computers [AJ89a], to obtain real speedup compared to the sequential implementation.

It is possible to modify the sequential G-machine into a parallel one straighforwardly, by having multiple threads of control each with one or two stacks, all of which perform graph reduction in a common graph — this is the shared memory model. Such systems have been designed and implemented by Maranget [Mar91] and [Geo89]. This is also the approach taken in the GRIP project [Jon87, JCS89].

However, there are some properties of the standard G-machine that made us want to try a different approach for a parallel implementation. Firstly, in the G-machine, when reduction of a function application starts, the arguments of the application (either in the

E. H. L. Aarts et al. (eds.), *Parle '91 Parallel Architectures and Languages Europe*

form of a chain of binary application nodes, or a vector application node) are moved to the stack, and when reduction is finished the result is moved back into the heap by updating the root application node with the value of the function application. This seems like a lot of unnecessary data movement when the datum could have been accessed from the node in the first place (this reasoning has nothing to do with parallelism, of course). Secondly, the prospect of having to manage a cactus stack was not very appealing, we wanted something simpler.

In the machines we would like to consider, and in particular the machine we have implemented our parallel graph reducer on, the Sequent Symmetry™, a memory reference into the heap has the same cost as a reference into a stack, since they reside in the same (shared) memory. Thus the cost of moving a word to the heap while building a node is the same as pushing a word into the stack.

Thus, in the abstract machine we have designed, called the $\langle \nu, G \rangle$-machine, function applications are represented by *frame nodes*. A frame node holds the arguments of the function application and a pointer to the code for the function being applied, but in addition also contains enough space for temporaries needed for reduction of the function application. Figure 1 shows what happens when EVAL is called: the 'current point of reduction' is moved to the frame node to be evaluated, and a 'dynamic link' field is set to point back to the frame which called did the EVAL. Thus, instead of an ordinary stack we have a linked list of stack frames. In the parallel case, we have many points of reduction. For further details of the abstract machine, see [AJ89a].

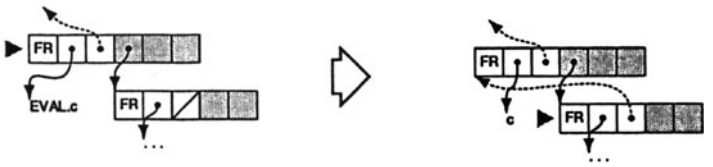

Figure 1: Calling EVAL to reduce a frame node.

To be fair, the stack model has some advantages too. The spineless G-machine [BRJ88], which offers a more general and efficient tail call mechanism than the 'standard G-machine' in particular when dealing with higher order functions, requires essentially an arbitrarily big stack. However, Lester [Les89] has devised an analysis technique based on abstract interpretation, to determine the maximum size a stack might have under the 'spineless' evaluation regime. Thus it would be possible to merge the $\langle \nu, G \rangle$-machine model with the 'spineless' model of execution, by allocating a frame node of the required maximum size.

Both the stack model and the frame node model have their advantages, and it is too early to nominate an overall winner.

So far, to introduce parallelism in the LML programs the programmer has to write *spark* annotations [CJ86] in the programs explicitly. The spark annotation is advisory: if there is a processor available then it may evaluate the sparked expression, otherwise the process really needing the value will evaluate it itself.

Code generation now works rather differently from the way it was described in [AJ89a]. Code is generated by first translating the combinators into three-address form, with liberal use of temporary names. We illustrate this with the code for the combinator `f x y z = x y`, which is:

```
funstart f 3        start of f, which takes 3 args
load  t0,0(nu)      load z from frame into t1
load  t1,1(nu)      load y from frame into t2
load  t2,2(nu)      load x from frame into t3
move  t1,t4
move  t4,t5
store t5,0(nu)      store the arg y of the application into the frame
eval  t2            evaluate x, the function
move  t2,t3
do    t3,1          tail call, function is x, one arg in frame
```

The code for a combinator starts by loading all arguments into temporaries from the frame node. Then in the example above the argument y of the tail call, is moved into the current frame at the location of the last argument; the function x of the tailcall is evaluated into function form, and finally the tail call is performed with the general tail call instruction do.

This 'raw' code is then subjected to various improvement transformations; for instance, the loads and stores are moved around to minimise the number om live variables across eval. Finally, temporaries are bound to machine registers. The resulting code is:

```
funstart f 3        start of f, which takes 3 args
load  r0,2(nu)      load x from frame into register r0
eval  r0            evaluate x, the function, in r0
load  r4,1(nu)      load y from frame into register r4 ...
store r4,0(nu)      ... and store y, the arg of the application into the frame
do    r0,1          tail call, function is x, one arg in frame
```

From this code the actual machine code is generated. A notable feature of the generated code is that we have abandoned the method of coding the tag as a pointer, either to a table (as described in [Joh86]) or to code directly, as in the spineless tagless G-machine [JS89]. Instead, the tag word contains various tag bits. The reason comes from two observations: firstly, most of the time when doing eval the node is allready canonical — according to measurements 80% of the time is typical. Secondly, in most modern architectures with instruction prefetch, it is rather costly to break the sequential flow of control. We therefore implement eval with code that tests a canonical-bit in the tag field of the node to be evaluated; if canonical the next instruction is executed, and only if it is not canonical does a jump occur to code that performs the actual call to the eval routine. The call to eval is surrounded by code that stores and reloads the content of live registers. Our

implementation of the parallel $\langle \nu, G \rangle$-machine is for the Sequent Symmetry, a bus-based shared memory machine. The architecture supports up to 30 processors connected to the bus; our machine has 16 processors. This machine has some features that helps very much in the implementation of the parallel $\langle \nu, G \rangle$-machine, for instance, any cell in the memory can be used as an atomic lock.

At the moment of writing this, a new garbage collector is being tested [Röj91]. It is an improved version of the Appel-Ellis-Li garbage collector, which is an efficient real-time copying garbage collector which runs concurrently with the mutator processes. Röjemo has extended it also to collect processes which have become garbage.

Since the publication of [AJ89a] we have improved the performance somewhat due to the improved code generation method, as described briefly above. The improvement is

about 25% for code purely sequential code, but for parallel programs the improvement is less than that – depending on how big a proportion of the time is spent in activities like syncronisation, task switching etc. Figure 2 shows the current speedup charts for three benchmark programs. Garbage collection time is not included in these figures.

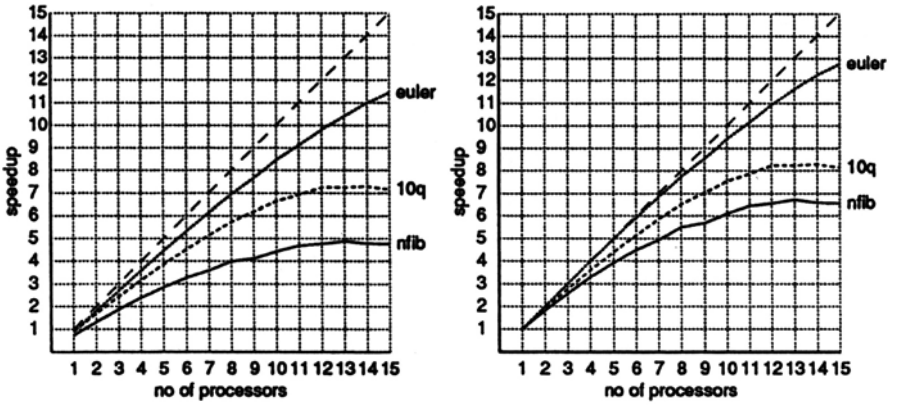

Figure 2: Speedup graphs for three benchmark programs: The left graph shows the speedup relative to one processor, the right graph shows the speedup relative to the 'standard G-machine' in the LML compiler.

References

[AJ89a] L. Augustsson and T. Johnsson. Parallel Graph Reduction with the $\langle \nu, G \rangle$-machine. In *Proceedings of the 1989 Conference on Functional Languages and Computer Architecture*, pages 202–213, London, England, 1989.

[AJ89b] L. Augustsson and T. Johnsson. The Chalmers Lazy-ML Compiler. *The Computer Journal*, 32(2):127 – 141, 1989.

[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.

[Aug85] L. Augustsson. Compiling Pattern Matching. In *Proceedings 1985 Conference on Functional Programming Languages and Computer Architecture*, Nancy, France, 1985.

[Aug87] L. Augustsson. *Compiling Lazy Functional Languages, Part II.* PhD thesis, Dept. of Computer Science, Chalmers University of Technology, Göteborg, Sweden, November 1987.

[BRJ88] G. Burn, J. Robson, and S. Peyton Jones. The Spineless G-machine. In *Proceedings of the 1988 ACM Symposium on Lisp and Functional Programming*, Snowbird, Utah, 1988.

[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.

[Geo89] Lal George. An abstract machine for Parallel graph reduction. In *Proceedings of the 1989 Conference on Functional Programming Languages and Computer Architecture*, London, Great Britain, 1989.

[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.

[JCS89] S. L. Peyton Jones, C. Clack, and J. Salkild. High-Performance Parallel Graph Reduction. In *Proceedings of PARLE'89 Parallel Architectures and Languages Europe (Vol I)*, volume LNCS 365, pages 193–206. Springer-Verlag, June 1989.

[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.

[Joh87] T. Johnsson. *Compiling Lazy Functional Languages.* PhD thesis, Dept. of Computer Science, Chalmers University of Technology, Göteborg, Sweden, February 1987.

[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, U.S.A, 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*, London, Great Britain, 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*, London, Great Britain, 1989.

[Mar91] Luc Maranget. GAML: a Parallel Implementation of Lazy ML. Technical report, Department of Computer Sciences, INRIA Rocquencourt, BP 105, 78153 Le Chesnay CEDEX, France, 1991.

[Röj91] Niklas Röjemo. A Concurrent Garbage Collector for the $\langle \nu, G \rangle$-machine. Technical report, Department of Computer Sciences, Chalmers University of Technology,S–412 96 Göteborg, February 1991.

[Tur79] D. A. Turner. A New Implementation Technique for Applicative Languages. *Software—Practice and Experience*, 9:31–49, 1979.