7Y-3

A 1.1 M LIPS (i.e. MHz) Flat GHC Interpreter for the Hitachi Supercomputer S-820

Martin Nilsson and Hidehiko Tanaka *
The University of Tokyo

Abstract: We will describe some optimizations of an implementation of Flat GHC For the Hitachi S-820/80 supercomputer. The implementation performs about 1.1 million process reductions per second, for a concatenate-type benchmark.

1 Introduction

One of the interesting features of logic programming languages is that interpreters for these languages often can be expressed as very compact loops. Our research is based on the idea that *if we can express the language interpreter as a few vectorizable loops*, we might get a very efficient supercomputer implementation.

We compile Flat GHC to a lower-level language Fleng, for which it is relatively easy to write a vectorizable interepreter [1], [2], [3].

There is an empirical guideline for Prolog implementations which says that the inference frequency (LIPS number) is very approximately $f/1000$ for an interpreter, where f is the number if machine instructions per second (MIPS number). If we take the peak performance as the MIPS number for the supercomputer, we obtain a "dream" value of about 3 MHz inference frequency for the S-820 supercomputer. Since this computer has 12 pipelines, several of which are dedicated to multiplication, etc., which is not useful for our implementation, the actual number of useful pipelines is something like four to eight. This reduces the dream value to 1-2 MHz.

Our Fleng interpreter comes quite close to this value, by performing about 1.1 million process reductions per second on the S-820.

The interpreter we describe is an improvement of an earlier interpreter. We will briefly mention the main optimizations we have made, benchmark results, and some other implementation data.

2 Optimizations

Almost all optimizations involve unification. This might not be so strange, since almost all of the code *is* unification. All optimizations maintain the 100% vectorization ratio.

- Separation of Head and Body Unification
  The old implementation tried to share as much code as possible. For instance, most of the code for head and body unification was shared. In the new implementation, these have been separated, allowing different optimizations for each case.
- More Compact Clause Representation
  Earlier not only data, but also clauses, were represented as binary trees. They are now represented as linear records. This speeds up AND- and OR-reduction, which need fewer memory references.
- Indexing
  Head unification now looks at the type of the first argument to see if some quick decision is possible.
- Separate First Unification Stage
  The first stage of unification has been taken out, and is handled specially. This allows a kind of "secondary indexing." Also, stack frames do not have to be created for unifications which succeed quickly.
- Unification first Sequential, then Parallel
  Unification is first performed serially, and if successful, it is split up in several parallel parts. This also builds upon the idea of indexing: The success of head unification is usually decided early. The purpose of the later stages are more important for parameter transfer.
- Recognize First Occurrence of Variables
  Unification recognizes the first occurence of head variables, and binds them immediately, without unnecessary dereferencing.
- The Unification Stack Area is Reset Automatically
  Stack frames used by unification are automatically thrown away after unification finishes. This

*Hidehiko Tanaka Lab., Dept. of Electr. Engineering, Univ. of Tokyo, Bunkyo-ku, Hongo 7-3-1, TOKYO 113

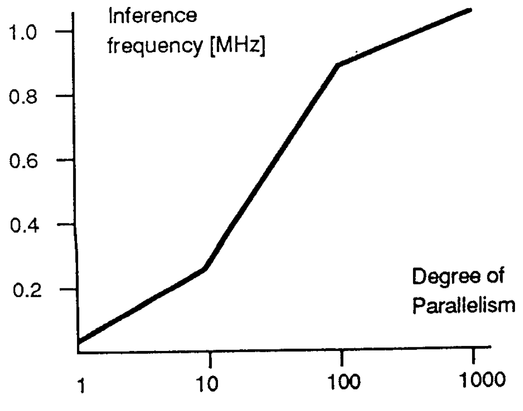
has the side effect that suspended processes must restart unification from the beginning when they are activated.

- SIMD Simplifies Distributed Unification
  We realized that SIMD computers substantial simplification of distributed unification:

  In distributed unification, we have the problem of binding a variable to another, with a directed binding. Unless we are careful, it might happen that several variables which are bound asynchronously are bound in a circle. This is not acceptable, but avoiding it is hard; if we try to lock both variables, we risk deadlock. Another way is to impose a permanent ordering on all variables, and make sure that variable bindings are bound in the direction specified by the ordering. This scheme can work, but leads to substantial overhead.

  For an SIMD implementation, *we only need a temporary ordering during the binding of the variables.* After binding, the ordering may safely be forgotten. No cycles can be created by binding variables, assuming that they have been fully dereferenced before binding. (This is not obvious, but is perhaps most easily seen by observing that the fact that variables are bound in tree structures is an invariant.)

3 Benchmark Results



For a simple benchmark, consisting of a number of parallel **concatenate** calls, run on the Hitachi S-820/80, the optimized interpreter achieved approximately 1.1 million process reductions per second. The new version is about twice as fast, but almost twice as large, as the old version. The interpreter cycle contains 40 loops, which represents a doubling compared to the non-optimized version. Most of this code is for unification.

The degree of parallelism for this benchmark was 1000. The cost per degree of parallelism is approximately 80 bytes, i.e. 20 positions of queue storage.

The ratio of vector processor time to scalar processor time goes up to about 20:1 for 1000 parallel processes. This shows that the processing time is clearly dominated by vector processing.

4 Conclusions

We have shown that a vector parallel supercomputer can execute Flat GHC with a speed competing with that of the fastest compiled implementations.

5 Acknowledgments

This work was supported by the Japanese Ministry of Education, and the Swedish National Board for Technical Development. We have benefited very much from discussions with members of the Special Interest Group of the Inference Engine at the university, and with members of the Parallel Programming Systems Working Group at ICOT. We are especially grateful for vivid discussions with Kanada-san and Tatsuguchi-san.

References

[1] Nilsson, M. and Tanaka, H.: *Fleng Prolog - The Language which turns Supercomputers into Prolog Machines.* In Wada,E. (Ed.): Proc. Japanese Logic Programming Conference. ICOT, Tokyo, 1986. p 209-216. Proceedings also published as Springer LNCS 264.

[2] Nilsson, M. and Tanaka, H.: *The Art of Building a Parallel Logic Programming System.* In Wada,E. (Ed.): Proc. Japanese Logic Programming Conference. ICOT, Tokyo, June, 1987. p 155-163. Proceedings also to appear as Springer LNCS.

[3] Nilsson, M. and Tanaka, H.: *Converting FGHC Clauses with Guards into Clauses without Guards.* In Information Processing Soc. of Japan Workshop on Programming Languages no. 17, July 1988. (In bad Japanese).