The Effects Of Local Variable Optimisation In A C-Based Stack Processor Environment.

C. BAILEY, R. SOTUDEH, and
M. OULD-KHAOUA
SST, University of Teesside, Middlesbrough,
Cleveland, TS1 3BA, UK.
(email*: c.bailey@teesside.ac.uk)* ϕ see note below

ABSTRACT:

Stack based processors, traditionally associated with the FORTH language, have many advantages not reflected in contemporary RISC and CISC register file processors. However, current trends indicate that high level languages, such as C, may become important for future real time and distributed systems. If the advantages of stack processor hardware are to be retained, the issue of efficient HLL execution in a stack based context must be investigated.

Local Variable management, and its efficient support in hardware, are a prime concern in developing efficient stack based computation for HLL's. Recent work by Phil Koopman has suggested that local variables may be virtually eliminated by compiler optimisation techniques.

In this paper we present results from an initial investigation of Koopman`s intra-block algorithm and examine its impact on the data stack behaviour of compiled C, an area previously ignored. Consideration is also given to the application of generalised data scheduling in a FORTH context. The results are presented, and their implications for processor design are discussed.

1. INTRODUCTION

Stack based processors have many advantages that register file architectures cannot claim to match: Fast interrupt response, low context switch overhead, and minimal procedure call penalties all lend themselves to modular but efficient code execution. FORTH, a stack based language, exploits these features to the full, and has all the added advantages of a low level interpretative environment for control systems development.

Current trends indicate that high level languages, such as C, may become important for future real-time and distributed systems. Future real-time system environments may demand many of the concepts found both in C and FORTH. If the advantages of stack processor hardware are to be retained in future programming environments, the issue of hardware support for HLL features must be addressed. With this objective in mind, our research programme[1] has centred upon the design of a new stack processor, with enhanced HLL features[Bail93a],[Bail94a].

Our latest findings are concerned with an initial investigation of the impact of Koopman`s intra-block scheduling algorithm, and its effects on dynamic stack behaviour during program execution. The impact of scheduling is complex, and its efficiency is not simply a measurement of net local variable reduction.

2. INTRA-BLOCK SCHEDULING.

In a recent paper [Koop92], Phil Koopman proposed two mechanisms by which local variables may be eliminated from C-code compiled for a stack processing environment. The 'global' algorithm was applied only by hand, proposing no clear method for automation, hence we have yet to implement this technique. The 'intra-block' algorithm was however well described, and implementation for our architecture proved to be relatively easy.

Intra-Block scheduling is an algorithm that attempts to eliminate local variable references only within the scope of a basic block. This we define as being bounded by an assembler label and a block terminator, such as Exit, Branch, or Call. An example of code scheduling is given in Fig.1.

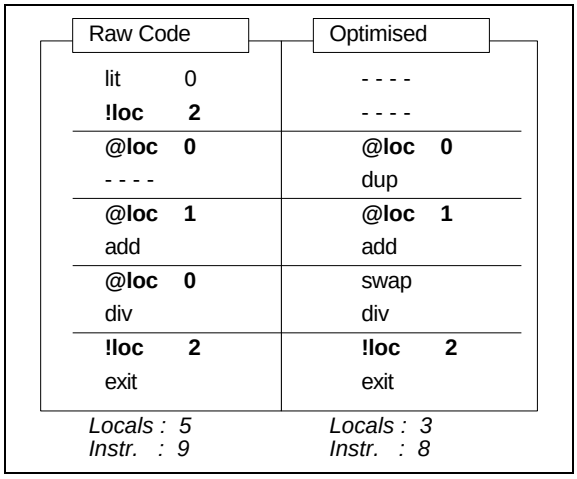

Fig.1, Example of local elimination for (a+b)/a.

Code is optimised in two ways: If two stores are made to the same local variable, yet no intervening use of the 1st stored value is encountered, then a 'dead store' exists. The first store is then replaced by a drop and later removed by a peephole optimiser.
If any local fetch is preceded by a fetch or store to the same local, then there is an opportunity to

This ref: Proceedings of the 1994 EuroForth Conference, Winchester, 5th - 8th September 1994. pp17-22
f note : the author may be now contacted at chrisb@cs.york.ac.uk

duplicate the item on the data stack at the point of the preceding reference by placing it some way down the stack. The second reference becomes redundant and is typically replaced by using swap or rot to bring the duplicate item to the top of stack.

Previous investigation was limited to an analysis of local variable counts, and their reduction by scheduling, yet showed an impressively high level of redundant local variable elimination.

3. GENERALISED SCHEDULING: FORTH?

Although the intra-block scheduling concept was proposed as a means of optimising local variables in a C-oriented programming environment, generalisations may be made which are applicable to FORTH and indeed any stack based target code.

The benefit of scheduling is to eliminate repeated memory access to unchanged data values. We may generalise this to include both statically declared variables, and constants, not just variables local to a C procedure. FORTH variables that are statically declared can be scheduled in a similar fashion to a C-code local, and long constants may also benefit from this approach. An example is given in Fig.2.

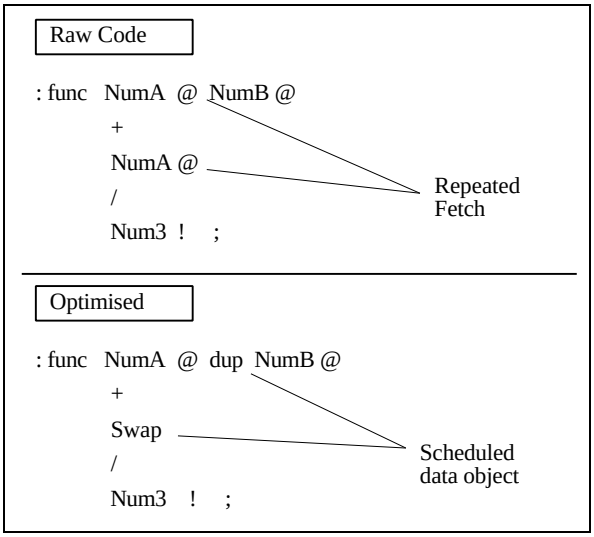

Fig.2, Scheduling ((a+b)/a) in a FORTH context.

Repeated use of the statically declared variable NumA allows application of scheduling. Constants and long literals can be treated similarly with potential benefits for speed and code density.

This is of course a trivial case, optimised intuitively by most FORTH programmers. However, complex situations may result in missed opportunities, which a scheduling tool might exploit. Perhaps this is even more likely with global scheduling techniques.

Even if we assumed that a human programmer will always better an automated algorithm the effects of this scheduling upon data stack behaviour, from an architectural viewpoint, do not normally enter into the mind of the programmer when evaluating an optimisation. The result, in a real system, may not be as expected once the potentially adverse effects on stack behaviour are accounted for.

4. SCHEDULING TRADE-OFFS

Trade-offs exists with this algorithm that require some understanding of stack processor hardware. Since most stack processors maintain some top-of-stack registers on chip, and minimise stack register spillage by use of stack buffers [Bail93b],[Stan87], then keeping copies of local variables on the data stack may not cost anything in terms of memory references. In contrast, a direct read or write to a local variable, which would typically be held in a memory-resident third stack, incurs a memory access penalty for every occurrence and a corresponding loss of performance.

It may seem that placing local variables on the data stack at all costs would eliminate all memory-resident local variable accesses. However, we must be careful to avoid making arbitrarily deep stack accesses commonplace, this cannot be supported in hardware without complication of stack processor data paths, and could result in verbose and inefficient code. Hence all scheduling opportunities must limit themselves to a scope of reference equal to that of the discrete top-of-stack register set, which is four registers in the case of the University of Teesside Stack Architecture (UTSA).

5. THE TEESSIDE IMPLEMENTATION OF INTRA-BLOCK SCHEDULING.

Our research group constructed a 3 phase optimiser for intra-block scheduling. The phases are:

- *Local Fetch Elimination.*
- *Dead Store Elimination.*
- *Peephole Optimisation.*

Dead store removal must come after local fetch elimination, in order to preserve the maximum number of optimisable references for fetch optimisation. Optimisable variable pairs are ranked according to Koopman`s criteria. Ranking is repeated after each individual optimisation.

In most cases each optimisation replaces a single local reference with 1 or 2 manipulation

This ref: Proceedings of the 1994 EuroForth Conference, Winchester, 5th - 8th September 1994. pp17-22
f note : the author may be now contacted at chrisb@cs.york.ac.uk

instructions. However, with several optimisations within a basic block, the new instructions often cancel each other out after peephole optimisation.

As a result of applying efficient peephole optimisation, we found that program size increased by only 1 or 2 % for static code length, with net increases in dynamic instruction counts ranging from 2 to 8 % of total instruction path length.

6. SCOPE OF ASSESSMENT.

Of eight programs tested originally, two proved unoptimisable with intra-block scheduling. A third, the ackerman benchmark, had too much stack data to include in our graphs. One unoptimisable program was kept in the set to avoid biasing results. The remaining programs studied here are:

- *Bubble sort of 100 ranked integers.*
- *Eratosthenes Sieve algorithm.*
- *Fibonacci Recursion (20th number).*
- *Image Smoothing of 100´100 pixels.*
- *Factorial Recursion (10th factorial).*
- *Empty Loop (unoptimisable).*

Although our test programs are rather limited they were all automatically generated by a C compiler[2], and contained a reasonable range of local variable useage. We found an average 3:1 ratio of fetch to store to be typical. The initial distribtion of locals as a percentage of static code is shown in Fig.3.

The results were analysed in a number of ways, reflecting the direct and indirect effects of scheduling. Dynamic execution statistics were gathered using the group`s UTSA simulator. The following sections discuss these results and their implications for machine design.

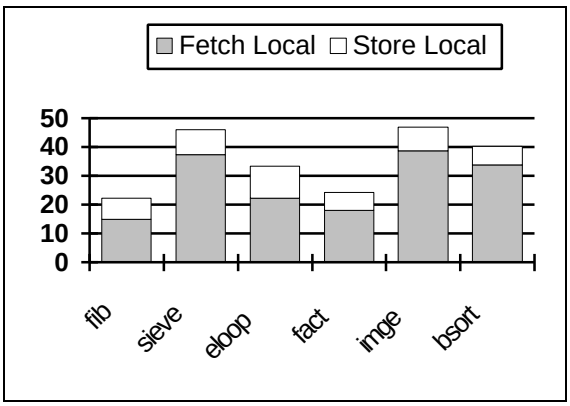

Fig.3. Locals as a percentage of program code.

7. DIRECT EFFECTS OF SCHEDULING.

7.1 Static Variable Utilisation.

The most direct measurement of the effects of intra-block scheduling can be seen in Fig.4, which shows the static reduction in local variable occurrence.

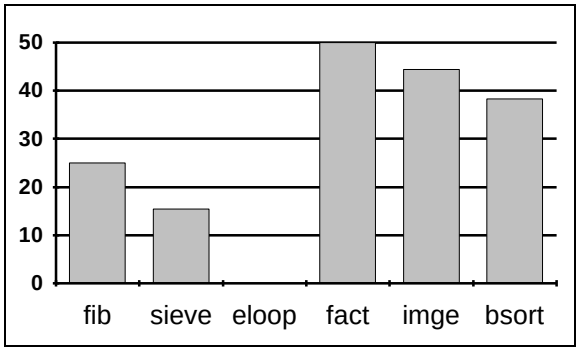

Fig.4 Reduction in locals for static code.

Results are not as impressive as Koopman`s own results (60-80% reduction), but this is both dependant upon the initial compiler output quality, and the selection of benchmarks. Setting this aside, the impact of intra-block scheduling appears to be significant in some cases at least.

7.2 Dynamic Local Utilisation.

A more realistic measure of performance of the algorithm is to estimate the reduction in local variable references actually issued, rather than a simple count of thier occurrence in a program file. (see Fig.5).

As might be expected, the overall reduction in local variable references was not identical to static figures. The effect of loops and repeated procedure calls distort the true utilisation of locals. Dynamic counts of local utilisation show a 10-60% reduction.

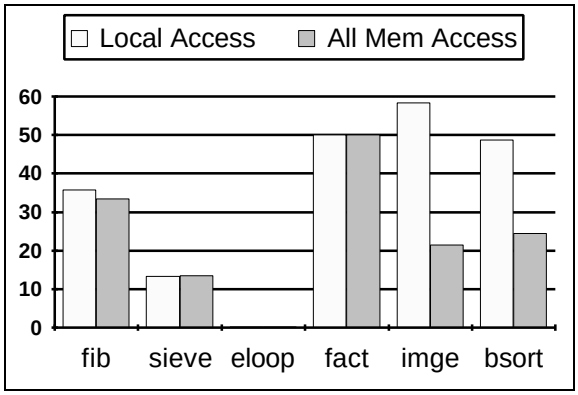

Fig.5. % Dynamic reduction locals/memory refs.

This ref: Proceedings of the 1994 EuroForth Conference, Winchester, 5th - 8th September 1994. pp17-22
f note : the author may be now contacted at chrisb@cs.york.ac.uk

Also shown in Fig.5, is the reduction in overall memory access, which are partly local variable references, but also array and data accesses.

7.3 Speed-up In Program Execution.

The ultimate goal of scheduling locals is to speed up program execution. We observed this execution speed-up for five of the six benchmarks, as indicated in Fig.6. Empty-loop was the exception, being unoptimisable with the intra-block algorithm.

Stack buffer effects were not included in the simulations, but a stack buffer of 16 elements, often quoted as a good choice [Hayes87], would in any case eliminate buffer spills in the cases presented.

Execution speed-up ranged from 5-27%, being highly dependant upon the number of locals available for optimisation, and the size and frequency of basic blocks in the program code.

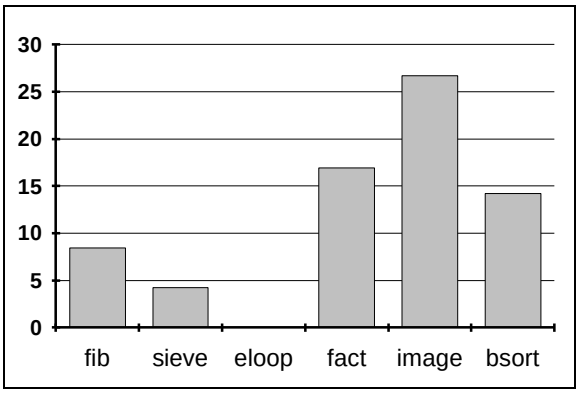

Fig.6. % Speed increase.

Programs with long complex basic blocks gain more than the short simple blocks of programs such as "empty-loop". More complex benchmarks including C library code could show more significant gains.

Also our compiler was found to be inefficient in its code structuring, with frequent unconditional branches to code fragments that could be removed by re-ordering the basic blocks generated.

Compiler techniques such as in-line code expansion of small basic blocks would increase the opportunity for intra-block scheduling, possibly without significant increases in code size once peephole optimisation is applied.

However, we still feel that some form of global scheduling is ultimately essential to maximise the efficiency of C-code.

8. INDIRECT EFFECTS OF SCHEDULING.

Phil Koopman`s original study of local variable scheduling was limited to an analysis of the algorithm itself, presenting results of its effectiveness in reducing local variable frequencies.

The absolute effects of local variable reduction are a clear indication of the effectiveness of the algorithm in its own right, but do not necessarily imply an equal performance gain. By changing the nature of program execution, we may neccesitate a re-appraisal of hardware for maximum throughput.

In our previous publications we have emphasised studies of dynamic machine-stack behaviour, and presented figures for the general behaviour of programs in a stack based computational environment [Bail93b],[Bail94a]. It therefore seemed appropriate to examine the effects of variable scheduling on the dynamic run-time behaviour of the program code at a low level.

Several measurements are of importance in terms of optimum stack processor design, we shall now discuss them.

8.1 Data Stack Depth Probability.

The characteristic of data stack depth is important in understanding the behaviour of a buffered stack system. A high probability of a narrow band of depth values would represent stack depth variations that are easy to accommodate in a small stack buffer. Conversely, a widespread distribution of stack depths would imply that larger transitions in stack depth are likely, leading to poorer performance for stack buffers.

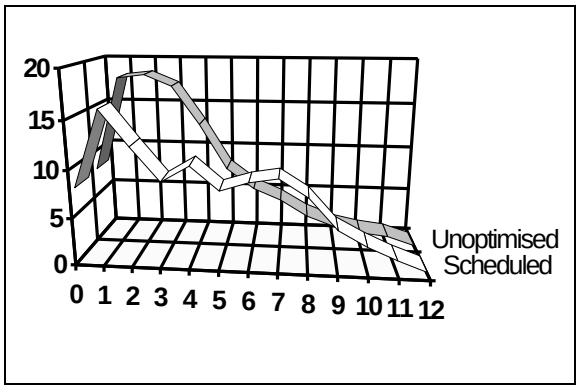

Fig.7. Stack depth Probability

Simulating each of our chosen programs, using our own UTSA simulator tool, allowed us to measure the stack depth of each program after each

This ref: Proceedings of the 1994 EuroForth Conference, Winchester, 5th - 8th September 1994. pp17-22
f note : the author may be now contacted at chrisb@cs.york.ac.uk

instruction cycle before and after applying intra-block scheduling. The results are plotted in Fig.7.

We found that the probability of given stack depth for unoptimised C-compiler output had a graceful curve, with the probability of stack depth decaying with increased size. The major part of program execution was spent in a stack depth region of 0 to 5, the initial peak of the 'unoptimised' curve (Fig.7).

Application of local variable scheduling to the C-code modified the stack depth characteristics considerably. The overall trend remained downward, but the distribution of stack depths is much broader and more significant for large depths.

8.2 Impact on stack depth change.

The probability of the data stack being a certain depth was considered in section 8.1, now we consider the probability of a change in stack depth during program execution. Two measurements were made: the average effect of individual instructions (atoms), and the effects of a run of stack operators that increase or decrease stack depth over a series of instructions (spans).

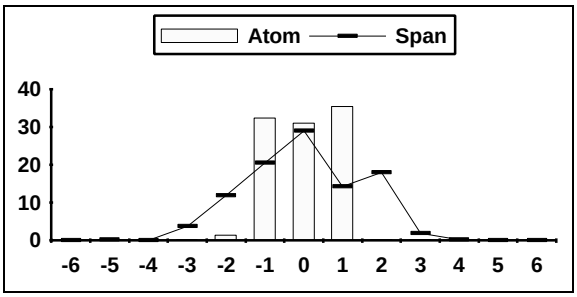

Fig.8a, Stack depth changes for Un-scheduled code.

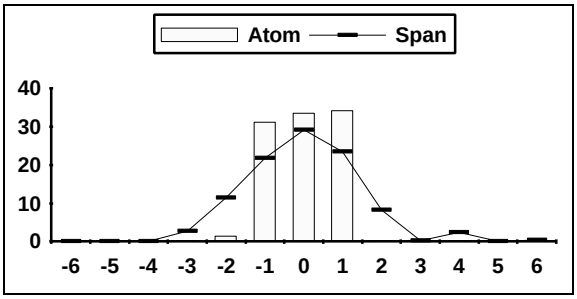

Fig.8b, Stack depth change after Scheduling.

In Fig.8a we can see that the atom changes in stack depth consist almost exclusively of stack depth changes of ±1 element. The atom depth change probability for the scheduled code (Fig.8b) is quite similar, but the number of operations increasing stack depth has reduced in favour of operators that have no net effect on stack depth, such as swap.

The span of stack depth changes tells a more convincing story of stack scheduling effects. Whereas the unoptimised code of Fig.8a shows a distinct excess of stack depth changes with a magnitude of two, the optimised code (Fig.7b) shows a far reduced occurrence of that magnitude of stack depth change.

The change in stack depth spans can be explained: If, for example, a local fetch and a literal are executed the result is a stack depth change of +2. If a local is scheduled on the stack already then stack depth will change only by +1, as the literal is pushed to stack. This is not always the case, but occurs frequently enough to have the marked effect on the stack characteristics shown in Figs 8a & b..

8.3 Implications and trade-offs for stack-depth.

The stack-depth probability results imply that a choice of a stack buffer with a capacity of 8 elements, based upon the characteristics of un-scheduled C-code, could be a mistake. The application of local variable scheduling could adversely effect stack behaviour, and degrade stack buffer performance. Some of the reduced memory cycles gained by eliminating local accesses could be lost again due to more frequent stack buffer spills.

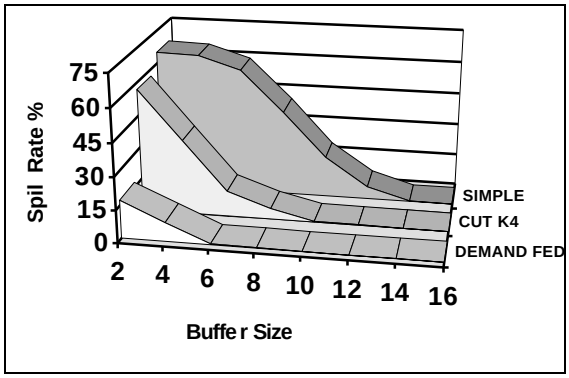

Fig.9,Buffer algorithm performance.

A stack buffer chosen to be 16 elements deep would be sufficient to accommodate both conditions shown in Fig.7, but this trade-off is further complicated by the question of increased task switching latencies, where smaller buffers are more desirable.

Fig.9, shows three buffering algorithms we originally assessed in [Bail93b], and their effect on bus traffic spilling ratios with increasing buffer size. Choice of algorithm, and buffer size may be affected by aspects of C-code and its optimisation which would not normally be observed (using FORTH).

This ref: Proceedings of the 1994 EuroForth Conference, Winchester, 5th - 8th September 1994. pp17-22
f note : the author may be now contacted at chrisb@cs.york.ac.uk

9. CONCLUSIONS.

We have investigated a limited set of benchmarks in the context of efficient code execution in a stack based architecture. However, the techniques and stack effects are tacitly applicable to FORTH and stack based programming environments in general.

The speed-up and reduction in locals achieved by intra-block scheduling is modest but valuable in creating an optimum environment for C-code execution. But this apparent gain must be weighed carefully against the impact it has on the data-stack.

Scheduling is not the clear cut issue that we may have assumed when we began this study, but a complex one with many implications for machine design and subtle trade-offs for performance. A study of global scheduling, and its effects, would be valuable. This more aggressive approach may show further pronounced changes in stack behaviour.

Future work we hope to carry out will permit us to study global scheduling effects, and investigate extensions to Koopman`s techniques. With access to a new and comprehensive compilation platform by 1995, we hope to follow up our initial work with a more comprehensive study and it`s implications for stack processor design.

REFERENCES

[Bail93a] Bailey, Investigation of Stack Machine Design for Efficcient HLL Support. Proc. of 1993 Rochester FORTH Conference, June 1993, U.S.A.

[Bail93b] Bailey, C. Quantitative assessment of machine stack behaviour for better performance. Proceedings of the ICMCM 1993 Berkerley California, USA.

[Bail94a] Bailey, HLL Enhancement For Stack Based Processors. Short-Note Proc.of EuroMicro-94, Liverpool, England, Sept 5th-8th 1994.

[Hayes87] Hayes, Fraeman, Williams, Zaremba; A 32-Bit FORTH Microprocessor, Proc. of 1987 Rochester FORTH conference.

[**Koop92]** Koopman, P. A preliminary exploration of optimised stack code generation. Proc. of 1992 Rochester FORTH conference.

[Stan87] Stanley, T., J., Wedig, R., S. (1987). A performance analysis of automatically managed top of stack buffers. Proc. 14th Int. Symp. on Computer Architecture, June 1987, pp 272-281.

ACKNOWLEDGEMENTS

[1]. This research is Sponsored by MicroProcessor Engineering Ltd, Southampton, England.

[2]. The C compiler used for our study was developed at the University of Teesside.

This ref: Proceedings of the 1994 EuroForth Conference, Winchester, 5th - 8th September 1994. pp17-22

f note : the author may be now contacted at chrisb@cs.york.ac.uk