
Conference or Workshop Item:
Crispin-Bailey, Christopher and Sotudeh, Reza (1993) Quantitative Assessment of Machine-Stack behaviour for better Computer Performance. In: 9th International conference on mathematical and computer modelling and scientific computing, 01 Jul 1993, USA.

Quantitative Assessment of Machine Stack Behaviour for Better Computer Performance

Article · January 1994

2 authors, including:

Reza Sotudeh
University of Hertfordshire
90 PUBLICATIONS **592** CITATIONS

SEE PROFILE

QUANTITATIVE ASSESSMENT OF MACHINE-STACK BEHAVIOUR FOR BETTER COMPUTER PERFORMANCE.

C. BAILEY & R. SOTUDEH

University of Teesside, SST, Middlesbrough, Cleveland, TS1 3BA, UK.
Email: c.bailey@teesside.ac.uk

ABSTRACT

This paper presents an experiment to quantify stack behaviour during execution of a range of complementary programs. Through better understanding of stack behaviour, further optimisations can be made, not only improving stack machine efficiency, but perhaps influencing future designs both in RISC and CISC technologies.

Mainstream technology has always been dominated by explicitly addressed register file architectures, with two clear philosophies predominant. The CISC school of thought demands complex instruction sets to reduce the semantic gap : 'More work for less code'. On the flip-side of the coin, RISC proponents believe simplicity and speed will succeed, even if more instructions are executed. The 'Stack machine' alternative, has been pushed to the back of the queue in terms of research and development.

Stack machines abandon traditional register file concepts, and with them, the need for register addressing in program code. Instead, operands are, by default, found at the top of stack. The benefits of reducing instruction size and functional complexity offer potential for comparable performance to that of RISC and CISC architectures. Quantitative assessment of stack behaviour will clearly demonstrate statistical and probabilistic examples of stack actions, helping to guide future designs.

KEYWORDS

stack machines, forth engines, stack behaviour, profiling, simulation, buffering, caching, bus bandwidth.

INTRODUCTION

Intel's recent introduction of the Pentium microprocessor is an illustration of the evolutionary forces at work in the field of microprocessor design. Its roots lie firmly in the CISC family of processors, and is a result of progressive enhancement of a successful design. RISC technology, meanwhile, has enjoyed a rapid growth in applications once thought to be exclusively CISC territory.

Both RISC and CISC use explicit register file access to minimise external bus dependency, thus sharing a common ancestry. But many embedded systems designers now see an alternative that offers a very attractive combination of high performance and low gate count within a simple well-defined architecture.

Accepted for publication in *Journal of Mathematical Modelling and Scientific Computing*, Volume 4, 1994, Bailey, C., Sotudeh, R.

The Stack Machine Philosophy

Basic stack machines have a simplicity which cannot be rivalled by RISC architecture. Whilst avoiding the penalties of a very small instruction set, semantic content is maintained. Yet hardware is reduced significantly. Reduced gate counts aid optimisation of layout for speed rather than size, and release space for on-chip resources, such as cache and interfacing support, hence minimising external interconnect delays for increased performance at the system level.

Abandoning the familiar explicitly addressed register file concept releases the stack machine from the burden of decoding register addressing information, and eliminates the need for register address fields. Thus, a theoretically higher operating frequency for both the decode and execute logic is realisable without a reduction in instruction flexibility.

A simplified stack architecture (Fig. 1) generally consists of two top-of-stack registers, which implicitly provide one or two operands for ALU actions, the result being fed back to 'TOS', whilst the new secondary operand is replaced from the stack space (which may be partly buffered on chip).

Although stack machines allow memory words to be utilised as primary or secondary operands for the ALU data sources, the reduction in operand addressing offers considerable simplification of hardware, without necessarily causing a proportional reduction in performance.

Much research has been done in the field of stack architecture, and it is clear that performance can be greatly enhanced by careful design of on chip stack buffering (Koopman, 1989; Flynn, 1992; Stanley et al, 1985). Buffering logic is an inherently simpler bus traffic minimisation strategy than an explicitly addressed register file, or caching in the data path.

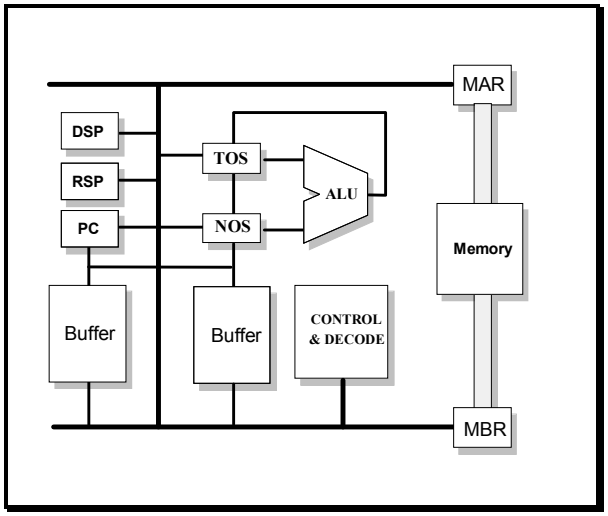

Fig. 1. Simple Stack Machine Diagram

At the University of Teesside, the computer architecture research group is involved in the design of a stack architecture for embedded systems applications, optimised for execution of 'C' code. (Bailey, 1993). It was considered essential to explore behaviour of machine stacks, whilst executing a collection of programs, and identifying the best approach toward optimising on chip support for stack oriented computation. An initial study of four programs was performed to provide a basis for more detailed work planned.

PERFORMANCE FACTORS

Definition of machine stack behaviour embodies many complex attributes, each of which has its own implications for machine performance and behaviour, and each dependant upon the machine architecture's interaction with the executing program. Critical factors for stack machine performance are outlined below.

Stack Size

Of the various stack attributes, the size of computation stack(s) during program execution is of particular importance. The size of a machine stack varies dynamically with program execution, with short term fluctuations and long term trends. No two programs are the same in this respect. However, the general behaviour of smaller programs bear similarities to sub-procedures within larger programs. When the number of intermediate products exceed the number of on chip storage elements then machine performance suffers, as the overhead of managing a 'virtual register set' becomes significant. This is manifested as 'stack spilling' on a stack machine, or register spilling/swapping on a register file architecture.

Functional Specialisation

Functional specialisation of the stack hardware also has a bearing upon performance issues, which is apparent from the graphs of Fig. 2, Fig. 4, and in many of the results presented here. The behaviour of the 'data-stack' primarily used for computational elements, and that of the 'Return-stack', used for return addresses and loop counters, may be quite different. Hence any optimised design may have to take into account the individuality of each on-chip resource- a fact which has been ignored in many such studies.

A more representative qualification of stack magnitude would be a statistical probability function, based upon as wide a sample of programs as possible, to give both behavioural scope, and generalised modelling. Knowledge of the general stack-size probability allows a machine architect to gauge the correct amount of on chip resources to be allotted for intermediate products.

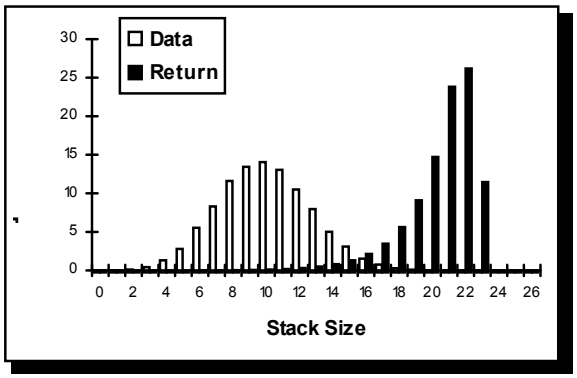

Fig. 2. Typical stack size probability functions for Fibonacci recursion

The correct sizing of register file, or 'stack buffer' in the case of a stack machine, will reduce memory traffic significantly. But performance gains should not be assumed to exhibit linear proportionality: a program which maintains a stack depth of 100 or less elements for 80% of program execution, would not achieve a doubling in performance if a buffer size were similarly doubled from 100 to 200 elements.

Dynamic Stack Modulation

We have seen that a probabilistic assessment of stack size can give static evaluation of the needs of a stack buffering mechanism which would apply to a typical program viewed as a whole. But this does not reflect the dynamic changes taking place within the program stacks during execution.

The data and return-stacks display an overall trend in terms of size during program execution. However a high frequency 'stack noise' component of stack movement (data-stack in Fig. 4. for instance) can be quantified by processing of profiled program data to reveal a probability of stack depth fluctuation of 'n' elements. These short term changes may be positive for stack growth or negative for contraction (zero represents constant depth).

A graph of relative frequencies of stack depth fluctuation is shown in Fig. 3. We chose a composite plot of the data-stack behaviour in this case, representing the combined behaviour of four programs studied.

The 'atomic' fluctuation probability shows the effect of single instructions, whilst the culminative plot indicates frequency of stack depth 'runs'- sequences of pops or pushes, with intervening static depth regions permitted. Hanoi is also individually plotted for comparison.

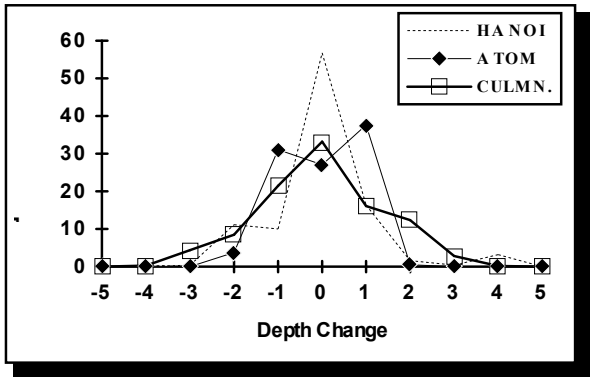

Fig. 3. Comparison of single operator stack effects, with culminative effects.

The results for unbounded stack-depth fluctuations would seem to be approximately predictable from the atomic depth fluctuation measurements. However, there is reason to suggest that stack-depth change is not a simple 'random walk' function, as assumed by Hasagawa et al, (1985), but may have an element of code specific behaviour (Note Hanoi, around the region in Fig. 3. for a depth change of -1 or -2, for instance). This had previously been suspected from studies by (Hayes 1989), and may also be observed in Fig. 4. The composite results described tend to dampen this subtle effect, where the overall behaviour of the stack appears to conform to a simple probability model, but when small portions of the program are examined in detail, there is a likelihood of a run of pops or pushes being more significant than expected.

Again, the individuality of separate stacks must be respected. The data-stack, whose main role is as a 'virtual register set', tends to be highly modulated, whilst the Return-stack tends to be more subdued, as its primary use is storage of return addresses and loop operands- generally active at 'basic block' boundaries. This may be observed in Fig. 4.

Programs with moderately sized procedures should tend to create 'bursts' of activity on the return-stack, whilst the data-stack changes continuously. This is blurred somewhat in reality by the tendency of programmers to shuffle data between the data and return-stacks, which may be confirmed by study of instruction execution frequencies (Koopman, 1989). A third stack for activation records (a likely option for stack machines), provides a further functional specialisation.

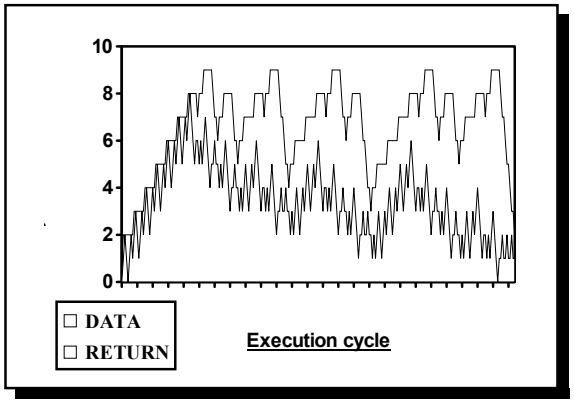

Fig. 4. Example of differing temporal attribute of Data and Return-stack

Buffer and Memory Coherency

The nature of computation (on a stack machine in particular), causes the 'top of stack' elements to exhibit a high degree of dynamism, whilst the 'deep stack' elements tend to remain invariant over short time scales at least. Hence, when a machine stack is monitored during execution of a program, it is not surprising to note that some stack elements are altered quite frequently, whilst others are less active. This can be inferred from the execution frequencies of profiled programs, and has been observed in our simulations.

Activity Spectrum of Top-of-Stack Elements

A study of the occurrence of changes in buffer contents would provide a model for evaluating buffer strategies, and guide the 'tuning' of algorithmic parameters. Quantifying those alterations is relatively straightforward. Program profiles are produced recording changes detected in stack elements after each and every atomic operation. They may then be tabulated to reveal an 'activity spectrum'.

Programs which heavily rely on stack manipulations such as 'swap', 'rot', and 'Dswap', which exchange two, three, or four stack elements respectively, will show an activity spectrum which extends across several stack elements. Inefficient 'Roll' stack manipulation extends activity beyond those 'shallow' stack elements into deep stack territory.

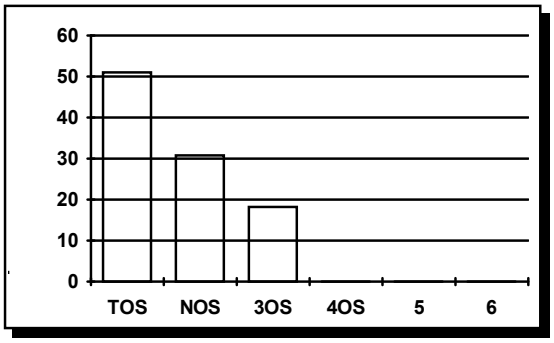

Fig. 5. Relative probability of change for top of stack elements (subject: Towers of Hanoi)

The example in Fig. 5. shows the relative frequency of change for stack elements during the execution of 'Towers of Hanoi'. If this were a general result, it would appear desirable to optimise hardware activities associated with those top three or four elements, which is in fact common practice in stack machine design, minimising the need for multi-port 'register file techniques' for on chip stack buffers.
In reality, the coherency of a given stack buffer element at any moment must take into account the movement of elements within a buffer. An element may move back and forth within the buffer as the stack grows and contracts, according to the buffer strategy used. A true assessment of coherency must take into account this interaction, which requires simulation of buffer elements with profiled program data. The impact of cycle stealing and tagging on buffer coherency, in combination with the underlying buffering policy, have significant effects upon memory traffic,(Stanley et al, 1985). A comprehensive study would be required in order to assess the interactive behaviour of such mechanisms.

SCOPE OF ASSESSMENT

In order to have a complete understanding of machine stack behaviour, it is clearly necessary to study a large, and wide-ranging set of programs. The data and information presented in this paper is based on a small sample of programs of limited scope, and as such, cannot be claimed to be definitive. The studies performed so far are intended as an initial investigation of stack behaviour.

The initial study examined 'Fibonacci recursion', the '8-queens' problem, the '7 Towers of Hanoi', and Eratosthenes 'Sieve', implemented in FORTH. The program listings may be found in [Code,a-d]. Three of the programs are recursive, whilst 'Sieve' is iterative. This biases the results somewhat, but ensures fairly aggressive exercise of both Data and Return-stacks.

Quantifying Techniques

The initial study was performed with a number of simple measurement tools developed by the research group. The program to be studied was first loaded into a customised Forth interpreter [MPE] - a 'FORTH Profiler', which then executed the program whilst outputting execution statistics to data files for later processing and trace-driven buffer simulation. The final aspect of the quantitative assessments discussed in this paper relates to the role of buffering in stack related traffic management, which can be reduced significantly.

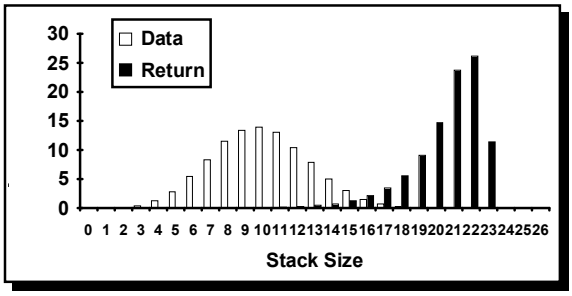

Fig. **6. Stack depth probabilities for '22nd Fibonacci Recursion'.**

Fibonacci Recursion : Data and Return-stack depths were measured for the Fibonacci recursion, calculating the 22nd Fibonacci number. The Fibonacci series consists of a numerical progression, where each new number is the sum of the previous two Fibonacci numbers. This was implemented with a simple doubly-recursive algorithm which repeatedly calls itself until solved.

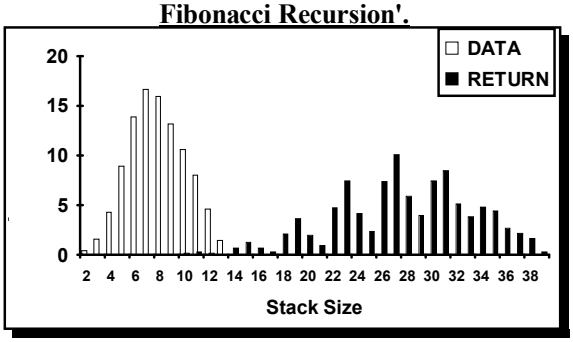

Fig. 7. Stack depth probabilities for '8-Queens'

As can be seen from the graph Fig. 6., the data-stack depth has a well-defined probability distribution, with approximately 75% of stack depth within a range of 5 values. The return-stack, which has a skewed distribution, also spends 70 to 80 % of its time within a 5 value range.

8-Queens : The '8-queens' profile reveals a similar story for the data-stack to that of 'Fibonacci'. A definite probability distribution is visible (Fig. 7.). But the return-stack behaviour is clearly of a completely different nature to that of the Fibonacci recursion- another case of functional specialisation. The complex return-stack activity of '8-queens' can be explained when the program code is examined. The '8-Queens' program studied consists of 6 procedures, with dynamic basic block sizes ranging from 2 to 14 instructions, and operates recursively. The return-stack activity is a mixture of nested procedure calls, program looping, and recursion. As such, '8-Queens' is more typical of general program behaviour than Fibonacci.

Towers Of Hanoi : The Towers of Hanoi program shows an almost opposite behaviour to that of '8-queens' in terms of stack depth. Here the return-stack is a clear probability distribution, whilst the data-stack is a broadly spread out distribution curve augmented by second order activities. (Fig. 8.).

The broad spread of data-stack depth indicates that towers in this particular implementation, maintains a very large set of intermediate values on the data-stack until resolving the computation. The program contains graphical output routines using ASCII characters, and this would account for some of the spread in data-stack size.

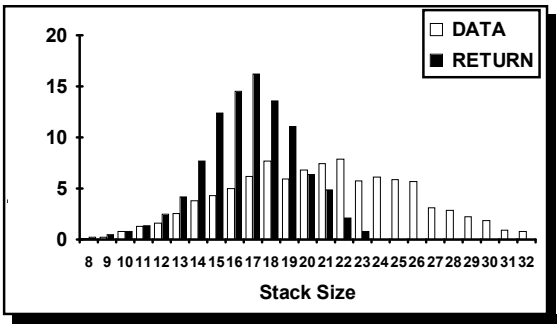

Fig. 8. Stack depth probabilities for 'Towers of Hanoi'.

Sieve Of Eratosthenes : The SIEVE program was solved iteratively, and has no procedural modularity. As a monolithic program block, which utilises loop structures without recursive methods of computation. the return-stack activity is virtually nil in terms of depth analysis. The data-stack action is also very simple in comparison to the other programs surveyed.

RESULTS

It is clear from examination of even this small set of programs, that stack depth is highly program dependent. A general model for stack depth would not be at all accurate if based on the small set of programs surveyed here. A large sample of programs would be more representative, but could only hope to be accurate for the application area targeted.

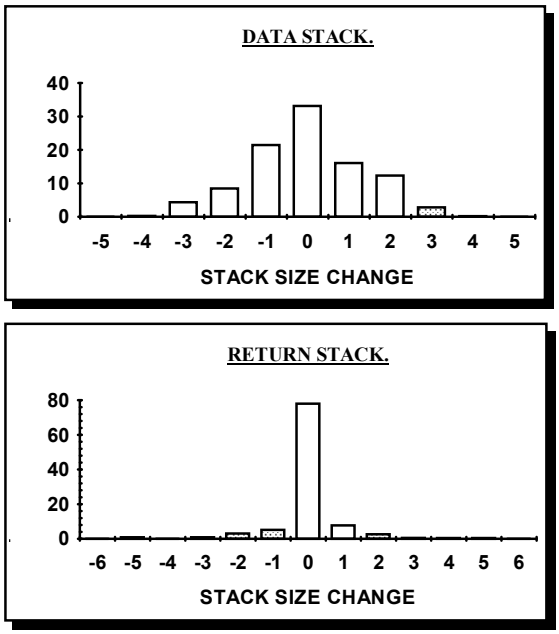

Fig. 10. Composite Models for Probability Of Stack Size Change.

Composite Model, Culminative Effects

A general model of stack behaviour may be derived by combination of the stack size change characteristics of the set of programs profiled. The graphs of Fig.10, illustrate this. Combination of the results reveals that the data-stack has a predisposition to remain static, although this is much less pronounced than the return-stack.

The two stacks together remain static for at least 50% of instructions executed. This is not because they are not in use, but can be partially attributed to the fact that a number of operations consume as many stack operands as they generate. 'Load', for example, takes a single address from the stack, and replaces it with a single data item fetched from memory.

Design strategies may be influenced by these results. The data-stack buffer, with its wider range of size changes, may benefit more readily (if at all) from set-associative-cache or cut-back-k-buffer strategies for example, but the return-stack may not offer improvement enough in performance to justify devoting silicon
to such a concept.
The data-stack buffer behaviour is more dynamic than the return-stack, and has a breadth of change which is related to the amplitude of stack noise, described previously. The noise amplitude is of the order of three or four elements, which agrees with the temporal attributes of Fibonacci for example (Fig. 4).

Prediction of Stack Behaviour from 'Atom' Depth Fluctuations

When similar composite models are plotted (Fig. 11) for the atomic operator effects upon stack depth (the modulation observed for single isolated machine operations), we find that the vast majority of larger depth changes are caused by repeated sequences of single stack reducing or expanding operations, with minimal occurrence of large stack depth excursions attributed to single operations. The probability of a stack depth change of 'n' in a given number of cycles, may be approximated by

using probability theory with the Fig.ures used to plot these graphs. For example, the probability of a stack growth of 1 word in a single instruction is 0.376 (37%) as derived from the atomic depth change graph data. But the absolute probability of two consecutive instructions both causing stack growth of 1 (a depth growth of two) may be calculated approximately as:

$$(P_{grw1} \times P_{grw1}) + (P_{grw2} \times P_{static})$$

or :

$$(0.37 \times 0.37) + (0.005 \times 0.27) = 0.14 \ (14\%)$$

The stack machine design being considered by the research group allows up to three instructions to be executed for a single overlapped instruction word fetch (Bailey, 1993a). This leaves only two free bus cycles to accommodate stack spills without penalty.

The probability of more than two stack spills or fills in three cycles may be assessed from the model:

Prob of at least 2 spill or fill = 35.49%
Prob of > two spill or fill = 12.25%

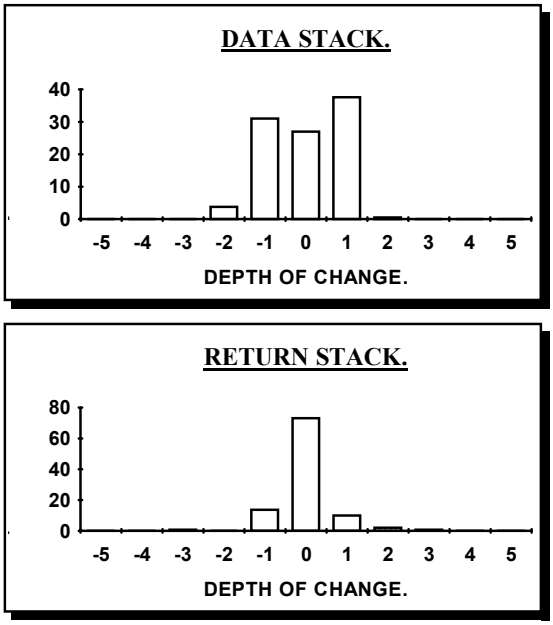

Fig. 11. Stack depth change for individual operations (Atoms).

Clearly the CPU will have maximum permissible bus demands for 35% of machine cycles, and will require extra stack management cycles for 12% of cycles, which entails stalling the ALU for a cycle- an undesirable situation.
The solution is to implement on-chip buffering. A RISC machine might adopt an overlapping register window approach while CISC technology might use cache. But the stack machine utilises the far simpler 'stack buffer' concept to alleviate such performance bottlenecks.

STACK BUFFER SIMULATIONS

The stack traffic in a un-buffered stack machine would be about 70% in total, roughly 30% each for spills and fills (if the program set sampled were representative). But by adding a buffering mechanism to the stack traffic path we may reduce the bus traffic to less than 1% even with a small buffer (16 to 32 elements), This is discussed in the remaining pages of the paper. (See also Koopman, 1989; Hayes; Hayes et al, 1989; Fraeman et al, 1987)

On-chip buffering effectively implements a special case of the Harvard architecture. Any access accommodated by the stack buffers will not necessitate an access to main memory, hence the main memory buses are available for overlapped instruction fetches, and explicit data transfers such as loads and stores. Provided the buffer does not 'miss' frequently when accessed, then there are effectively two separate bus paths for data or program code transfer, and the penalties of stack spilling are minimised as a result. Three buffering algorithms were chosen for simulation, each being quite different. One was an artificial algorithm chosen to assess potential improvement from a worst-case scenario viewpoint. The other two were practical algorithms which have been investigated or used previously.

Demand Fed Buffer

When the buffer is full, the bottom element is pushed into main memory. Only when the buffer becomes empty is a read from memory resultant for stack contraction (Koopman, 1989). Once emptied the buffer will not fill up again until new data is created or pushed onto the stack. This results in very poor coherency between memory and buffered elements, since almost all buffer elements are 'new' data not previously resident in main memory stack space.

Cut Back k Buffering

The Cut-back-K algorithm is well understood and has been mathematically studied (Hasagawa et al, 1985). The basic principle is to write or read more data than is demanded, in order to diminish the future demands for memory transfers. The basic demand fed algorithm described, can be modified to perform a 'read in' of 'k' stack elements when the buffer is empty, rather than just a single element. Similarly, a full buffer condition would result in 'k' elements being spilled to memory, to create more potential 'free growth space' in the stack buffer.

Simple Buffer

The final algorithm maintains a permanently full buffer potentially reducing 'buffer misses' for random (arbitrary) stack element access. A push or pop always demands a memory transfer unless the stack is small enough to be retained completely in the buffer. This is clearly not going to reduce memory transfers, but can be justified when cycle stealing, and write back tagging are used with the buffer, improving performance and allowing favourable comparisons with other algorithms.

Minimisation of 'buffer misses' for arbitrary stack access might compensate for the worse buffer spilling performance, particularly in an architecture which utilises the stack for 'virtual register file' techniques.

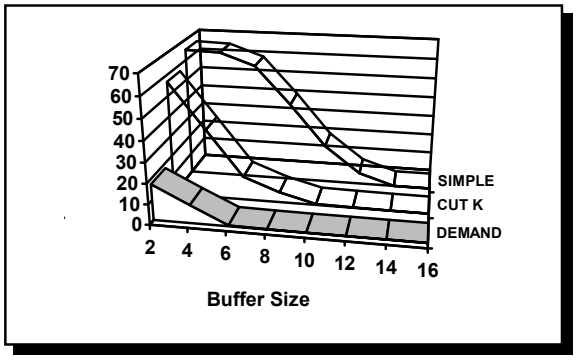

Fig. 12. Data-stack: Basic algorithms

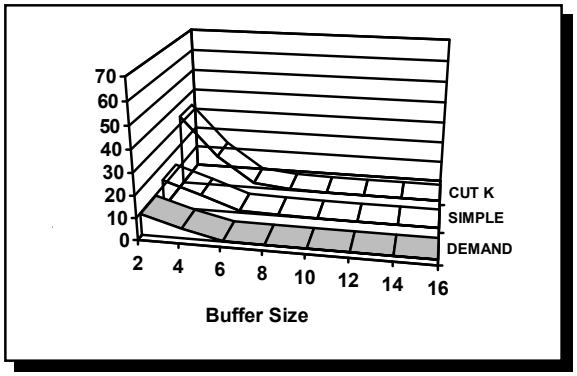

Fig. 13. Data-stack: Optimised algorithms

RELATIVE PERFORMANCES

The graphs in Fig. 12 and 13, illustrate a performance comparison for the three basic algorithms discussed above, showing basic data-stack performance, and optimised performance, respectively. A single program , '8-Queens' is shown here. Although all four programs were studied, a composite model of such a small program set would only confuse the issue. Return-stack simulations were also performed. Of the three basic algorithms given in Fig. 12 and 13, the simple buffer algorithm is the worst performer, which is not surprising since it performs no buffering as such (gains are only made as the buffer becomes larger than the stack). The 'cut back k' strategy, with k=4, provides slightly better performance but is inefficient compared with the third, 'demand fed' algorithm.

Once a full optimisation strategy is applied to the algorithms, the performance of each algorithm is altered considerably, a 75% reduction of 'simple buffer' traffic, and almost 50% reduction in the demand fed buffers memory traffic are shown in Fig 12 & 13. Now we find that Cut-back-K has improved a little, but the 'demand fed' and 'simple buffer' strategies are almost equalised in performance terms (Fig. 13).

It is interesting to consider that an unoptimised approach would lead a designer to choose the demand fed buffer for obvious reasons. But, when optimisation is applied, it is possible to choose demand fed or simple buffer strategies depending upon other (now relatively more significant) factors.

In the case of the return-stack analysis, a similar situation arises, but in this case, the simple buffer provides marginally better performance than the demand fed approach, which reinforces the earlier comments about functional specialisation. The next set of graphs show the individual effects of adding cycle stealing, write back tagging, and a combined approach, to each of the basic algorithms.

The results indicate the differing aspects of memory traffic optimised by each enhancement. Write back tagging naturally reduces buffer write behaviour. Cycle stealing tends to improve read behaviour, but at the cost of worse write performance. Only by combining the operations, do the full benefits of each approach become attainable.

In some cases it can be seen that write back tagging offers little improvement, in conjunction with demand fed buffering for example (Fig. 14c). This is because of the method used for the underlying buffer algorithm, which is not sympathetic to the write back tagging strategy. In other cases we see that cycle stealing does not (in isolation) offer significant improvements (Fig. 14b for instance).

By increasing the number of valid elements present in a buffer with cycle stealing, the chance of an empty buffer reduces, but at the same time a full buffer condition is more likely. So we see a reduction in memory reads, but increased memory writes, depreciating the potential for improvement.

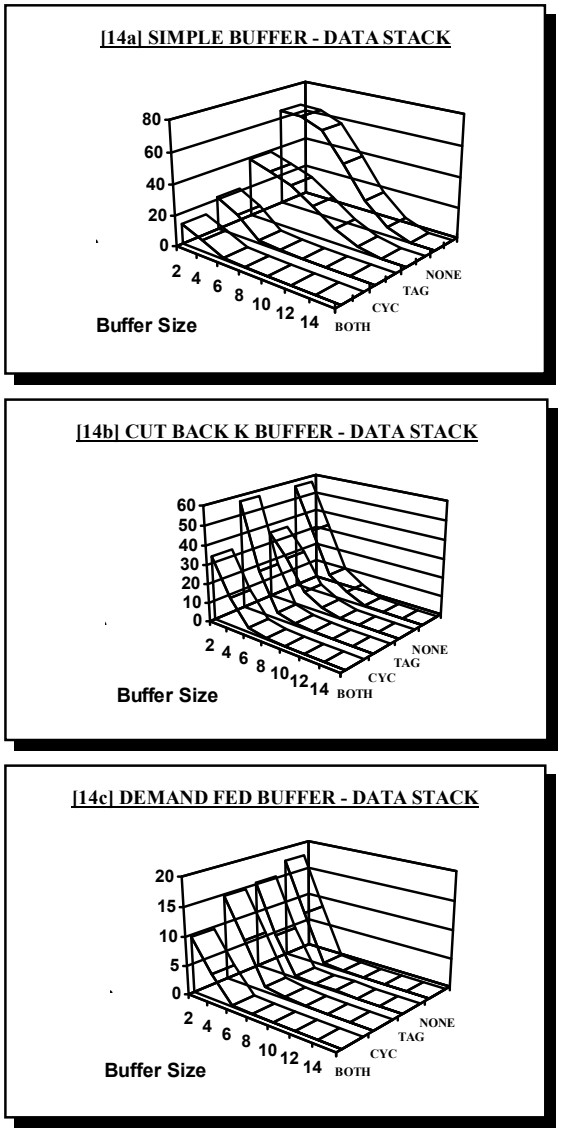

Fig. 14 a,b,c .Effect of Optimisation algorithms.

Clearly cycle stealing, or write back tagging can not be applied arbitrarily to any algorithm in isolation and respond with constant results. However, the combination of the two enhancements allows 'the best of both worlds'.

It is notable that the cycle stealing mechanism actually increases stack 'spills' or writes to memory. Our investigations have attempted to completely fill the buffer when cycle stealing is applied. However it has been suggested that optimum stack buffer capacity would maintain a small 'breathing space', (an

empty region of several elements), the size of which would be sympathetic with the stack noise amplitude, and hence accommodate momentary stack depth changes within this region.

CONCLUSIONS

By studying a small set of programs, the research group has not provided a definitive assessment of stack behaviour. However, the study is a step toward a more comprehensive study, which is the next stage of our research strategy, and has helped to set the scene for more complex analysis. Only by understanding the demands of truly representative programs can performance penalties be identified, and solutions provided.

The wider aims of the research are to develop a highly optimised 32-bit stack machine, starting from basics without inheriting the specialisations of 'Forth Engines'. Implementation and simulation tools provided by a full VHDL design route will allow advanced concepts to be modelled. A synthetic benchmark would have many advantages to offer in respect to such architectural assessments.

Finally, the aim of the buffer simulations has been successful, in that the workspace management overheads (stack management in this case), have been shown to be reduced very significantly, by use of relative simple optimisation techniques, making the stack machines standing amongst the competition of RISC and CISC technology more realistic.

The stack machine philosophy offers minimal silicon outlay, and yet achieves respectable levels of performance in the appropriate environments. It is no accident that aspects of stack machine strategy are now finding their way into architectures such as the Pentium (Alpert et al, 1993). By careful application of optimisation techniques it should be possible to prove that stack machines are no longer an exotic 'FORTH spin-off', but a real and valid concept, worthy of serious development and research.

ACKNOWLEDGEMENTS

We wish to acknowledge the ongoing support and encouragement provided by our industrial partner, MicroProcessor Engineering (MPE) Ltd (UK).

REFERENCES

Alpert, D., D. Avnon (1993). Architecture of the Pentium Microprocessor. IEEE Micro, June 1993. pp11-21.

Bailey. C. (1993a). Investigation of stack machine design for efficient high level language support. Proceedings of the Rochester Forth Conference, June 1993.

Flynn, J., H. Mulder (1992). Processor Architecture and Data Buffering. IEEE Transactions on Computers, Vol 41, No. 10. (Oct 92).

Fraeman, M., E., J. R. Hayes, L. R. Williams, T. Zaremba., A 32 Bit Forth Microprocessor. Proceedings of the Rochester Forth Conference 1987. pp. 39-48.

Hasagawa, M., Y. Shigei. (1985). High speed top of stack scheme for VLSI processor: a management algorithm and its analysis. Proc. 12th Symp. on Comp Architecture, June 85, pp. 48-54.

Hayes, J., R., S. C. Lee. (1989). The Architecture of the SC32 Forth Engine. Journal of Forth Applications and Research. 1989.

Koopman. P. 'Stack computers: the new wave'. (Ellis Horwood). ISBN 0-7458-0418-7. 1st Ed. 1989.

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

MPE. (Microprocessor Engineering Ltd). 'Pinc Power-forth' Interpreter V3.06.

[Code-a] 'Sieve' found in book: 'Interpretation and instruction path co-processing', MIT press, ISBN 0-262-04107-3

[Code-b] 'Fibonacci' found in: 'SC32 Stack chip Microprocessor' (data sheet), Silicon Composers, Inc. 210 California Ave., CA 94306.

[Code-c] 'Towers of Hanoi' based on version in 'FORTH DIMENSIONS', Vol II, No. 2, page 32.

[Code-d] '8-queens' based on version in 'FORTH DIMENSIONS', Vol II, No. 1. page 6.