Trade-offs for Memory Bandwidth Reduction In Stack-Processor Design.

C. Bailey and R. Sotudeh.
SST, University of Teesside, Middlesbrough,
Cleveland, TS1 3BA, UK.

(email: c.bailey@teesside.ac.uk)

ABSTRACT

This paper presents an evaluation of techniques for reducing memory bandwidth overheads of stack-processors executing compiled C-code. The results build upon our previous work which investigated hardware concepts using modest benchmarks. Now, with access to better compilers and benchmarks, we can present a more extensive appraisal of this work.

Stack-based processors abandon the traditional register file concepts of RISC and CISC in favour of implicit stack based computation: an architecture ideal for FORTH execution. However, trends indicate increasing importance for HLLs (High-Level-Languages) such as C. If the advantages of stack processor hardware are to be retained for future technology, the issue of efficient HLL execution in a stack based context must be investigated thoroughly and new trade-offs identified.

Our research group's efforts have concentrated upon the design of a new stack-processor with particular attention to the implications of C-based code behaviour for stack machine design. Modelling of our 32-bit design is being conducted with VHDL design tools, and some simplified processor designs have already been synthesised.

Quantitative results presented for compiled C programs include stack depth analysis, instruction set effects, and the effects of local variable optimisation on bandwidth and stack behaviour. The application of stack-cache buffering and instruction packing techniques are also quantified and performance estimates reviewed. Including the effects of recently proposed software optimisations, which we have now investigated, has allowed us to identify previously un-recognised trade-offs of possible significance for future stack processor design.

KEYWORDS

Stack-machine, Bufferring, Stack-cache,Modelling, Memory-bandwidth, Optimisation.

INTRODUCTION

Mainstream processor designs are heavily biased towards register file computation. Whilst CISC technology attempts to increase the semantic value of each instruction, RISC designs lean towards simplification of instruction set and architecture for higher throughput as shown by Hennesey (1990). The concepts of stack based computation, dating back to the earliest years of computing, have largely been ignored. Examples include Hamblin (1957),Dent (1968), and Duncan (1977)

Although stack processors have much to offer, particularly in embedded systems and real-time control environments, they have failed to migrate to mainstream computing. Their specialisation toward FORTH programming in recent years has made them a poor choice where HLL languages are preferred, and with FORTH itself adopting new features akin to mainstream HLLs, the next generation of stack processors must take into account the evolving demands of their intended environments.

Increasing performance of stack processors is our major research objective. This has to be placed in the context of better support for HLL features, whilst maintaining the stack based computing philosophy and its inherently minimal processor logic.

Memory bandwidth is of particular importance to stack processor technology. Whereas CISC and RISC designs utilise cache heavily, and indeed rely upon it in meeting performance targets, the stack processor cannot always utilise such techniques. In embedded systems and real-time control, there are often demands for deterministic and predictable system behaviour. Cache increases overall performance, but the penalty is reduced system determinance as Koopman (1993) has shown.

We have concluded that widening the application areas for stack processor technology is critically linked to achieving optimal throughput for HLL-oriented language models, in a system where limited bandwidth and low memory costs are primary concerns. This paper presents results from a number of issues we have investigated and thier implications for better stack processor performance. A number of trade-offs are identified, and new mathematical models are presented.

STACK PROCESSOR TECHNOLOGY

Anyone who is familiar with the FORTH programming languages will be familiar with the concepts of stack based programming. A FORTH interpreter uses a virtual stack as an automatically extensible computation scratch-pad, which must be used in Last-in-First-out order. Items on the stack may however be exchanged in limited ways to permit stack re-ordering. It is therefore not surprising that many current stack processors are optimised for FORTH execution.

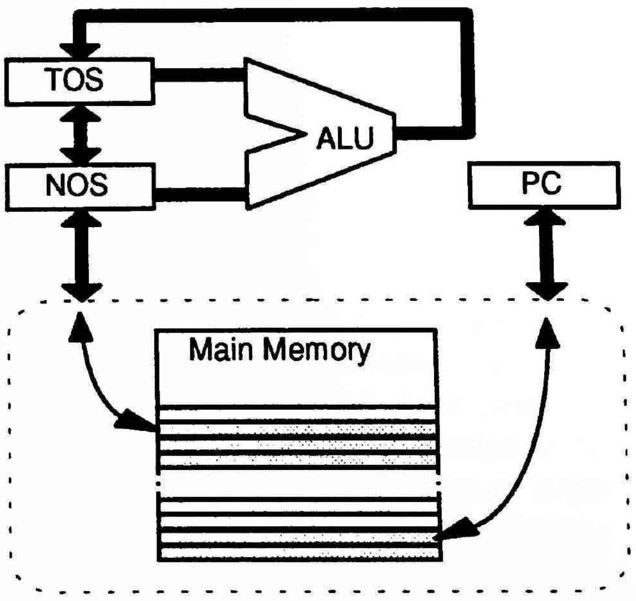

Fig.1. Simplified Stack Processor Model

A true stack processor implements a hardware model based upon dual stacks. The primary stack (the Data Stack) is for operand storage and manipulation, whilst the second (Return) stack holds program counter values and acts as auxiliary storage for call/exit events. Fig.1, shows a simplified stack processor.

By utilising a stack, the data storage is automatically managed as computation progresses. No register address fields are necessary since the majority of the operations act implicitly upon the Top-Of-Stack (TOS) and Next-On-Stack (NOS) items.

It should also be noted that, since the stack cells on chip are separate register cells, there is no need for the complexity of multi-port addressable register files, a considerable advantage in minimisation of logic complexity.

Any stack growth is accommodated by spilling successive items through the on-chip stack cells to main memory, where a stack pointer tracks the top of the auxiliary stack space. This implies a heavy bandwidth requirement for stack transfers to and from memory (known as 'stack spilling'), but techniques exist to limit this overhead as we shall see in later sections of the paper.

SCOPE OF THIS PRESENTATION.

In this paper we collate findings from a series of related studies, Bailey et al (1994a, 1994b, 1995a), in order assess the problems of stack processor performance, and identify future directions. Figure 2, below, serves as a useful illustration of the key areas we will examine in this paper. This shows the general memory bandwidth utilisation of a series of benchmarks, broken down into its components.

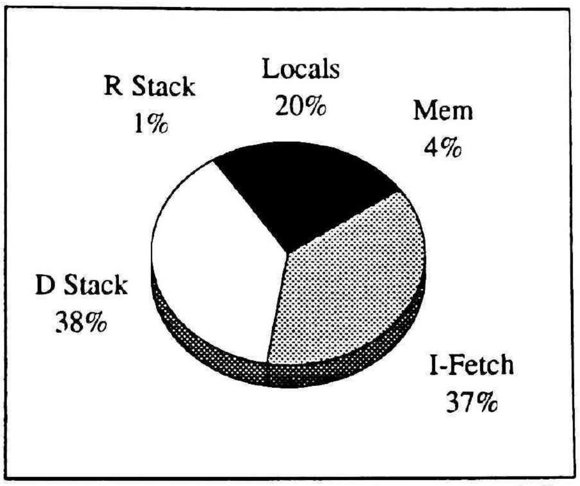

Fig.2. Memory traffic For Stack Based C-code.

The benchmark code presented throughout this paper was based upon a set of C source programs which were then compiled into the stack based code of a proposed machine architecture (UTSA) and then simulated with tools developed by our research group. It was clear to us that there are three major areas of stack processor performance to be addressed: stack spilling, local variable access, and instruction fetch bandwidth.

Data Stack Spilling.

By far the largest source of memory traffic was data stack spilling, with a minor contribution from the return stack. Here we have investigated buffering issues within the context of C-code rather than the previous FORTH-oriented studies. Since the majority of stack processor instructions lead to stack growth or shrinkage, the overhead for stack spilling is often in the region of 0.9 to 1.0 memory accesses per instruction: a severe handicap to any idea of single cycle execution. It is well known that this can be marginalised using buffering, and it is generally assumed that this will in practice be minimised by stack buffers of 16 to 32 cells.

The issue of buffering was found to be more complex than previously thought, as we indicated in Bailey (1995a), with new trade-offs being identified that will ultimately apply to the vast majority of programming languages targeted for stack processor execution.

Local Variables.

An area essential for our studies was the support of local variables, now a feature of new FORTH standards as well as a cornerstone of many mainstream HLL`s. It is clear that after

removing stack spill traffic from the distribution model of Fig.2, local variables represent a major bottleneck to performance.
Several 'stack-scheduling' techniques are proposed in Koopman (1992) to reduced the impact of local variable references.

We have implemented some of those post-compilation optimisation techniques and evaluated performance for our UTSA architecture. Our work progresses beyond Koopman's original study by assessing the impact that stack scheduling has upon execution characteristics and dynamic behaviour of those programs. Trade-offs encountered when taking into account hardware attributes such as stack buffering and instruction set architecture are apparent.

Instruction Fetch Bandwidth.

Partial reduction of local variable traffic leaves one final barrier to higher performance: the stack machine cannot execute instructions faster than they can be fetched from memory. Instruction fetch bandwidth ultimately limits performance in the absence of cache. Normally we would simply make the memory faster. And, if cost was a factor (it usually is), then cache would provide memory speed-up without inflating system cost. However the main application areas of stack processors are still firmly rooted in embedded systems and real-time control environments. Often, here, cache is prohibited in order to strictly maintain completely deterministic and predictable system behaviour as discussed in Koopman (1993). Here the multi-level cache hierarchies of the current-generation PC would be met with a cry of contempt, if not despair !.

When cache cannot be used to increase memory bandwidth then other mechanisms must be found to enhance performance. We have investigated the concept of instruction packing, where several compact stack processor instructions may be packed into a single memory word, and fetched within a single cycle.

Our results show that stack processors can achieve high dynamic packing densities even with naively generated compiler code. This reduces fetch overheads whilst releasing memory bandwidth for data fetches and local variable access. We find that executing 3 instructions for every 2 memory cycles is typical even including explicit data fetch/store overheads.

STACK BUFFERING HARDWARE.

Stack buffers are in some respects similar to cache, but are considerably simpler in form and have much more modest requirements in terms of logic and silicon area. The sequential nature of an implicitly addressed stack means that there is no need for address fields in the buffer 'cache'. Instead, a stack pointer typically keeps track of the memory area corresponding to the interface between CPU stack cells and main memory over-spill. Fig.3, shows a more detailed view of s stack processor, with buffered stacks.

By introducing a non-linear relationship between stack depth modulations and the associated memory transfers that would normally be generated (which are largely redundant) the buffer eliminates most of the stack spill traffic.

Demand-Fed Buffering.

A common choice in stack processor design is the demand fed approach, which has a linear buffer block. This can accommodate a certain degree of stack growth or contraction without becoming empty or full. A small internal pointer keeps track of the buffer level, and spills/fills are generated when full/empty situations are encountered. This topic was examined by Koopman (1989), and Hayes et al (1987). Such a scheme is shown in Fig.4.

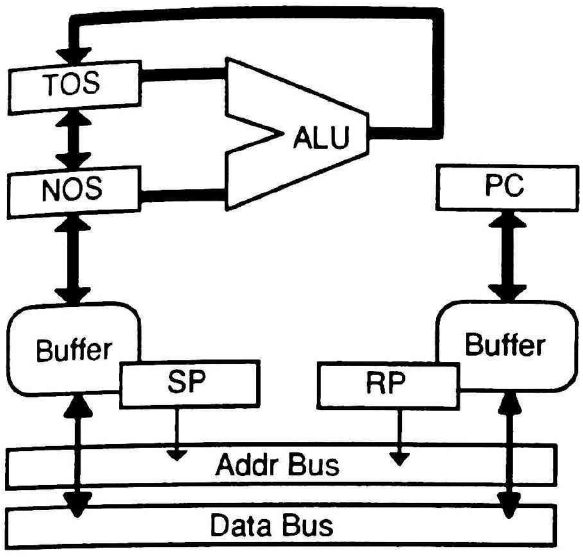

Fig.3. Buffered Stack Processor System

Whilst stack depth change remains within the range of the buffer, no spilling to memory occurs. However as soon as the buffer capacity is exceeded, the buffer must access main memory, spilling the bottom buffer item to the main memory stack space.

The stack typically encounters short term depth changes that lie within the buffer capacity, whilst following longer term trends that are 'tracked' at a penalty of occasional spills to and from memory.

The Tagged Push-Through Buffer.

Whilst the demand-fed buffer is somewhat simpler than a randomly addressable cache, it still requires indexing of items within a small storage file which is addressed by buffer index pointer(s). From our research we have identified a possible alternative buffering system which may be applicable to optimisation of stacks in general, and could have comparable performance to the demand fed buffer. A much simpler buffer scheme can be envisaged if we abandon the idea of index pointers in the buffer space, and instead use an n-bit wide shift register for stack buffer contents.

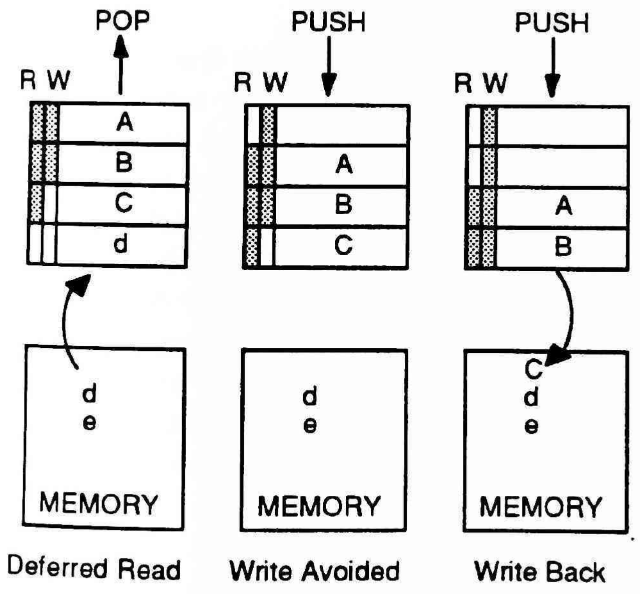

Fig.5. Tagged Push-Through Buffer.

We propose a word-wide shift register arrangement through which stack items may be pushed and popped. This would result in memory transfers to maintain buffer continuity after each push or pop. However, with the addition of read-in and write-back tags, the redundant memory transfers can be avoided, as is illustrated in Fig.5.

When a POP occurs at the top-of-stack, the buffer contents shift up by one cell. However, rather than immediately reading in a value to fill up the vacant cell in the shift-block, we simply clear the read-in flag of that cell to indicate that its contents are not present. The read is deferred, and will only take place if that item is popped right to the top of the buffer, a rare occurrence as buffer size is increased.

Conversely, If an item is read into the buffer its write-back-tag bit is cleared. If any change takes place, the tag bit is set. If and when the item is pushed back into memory, the transfer is only issued if the tag bit indicates a change in contents. If the write-back-tag remains clear, then its contents are already in memory.

We have established that the top of stack items change with ever decreasing frequency as the stack depth increases, such that TOS changes for the majority of operations, whilst the third and fourth stack items change for perhaps 10% of operations executed. The hardware requirements of the push through buffer are somewhat different from demand fed. We require only an n+2 bit wide shift register, where n is the word-size of the stack in bits.

The main memory locations corresponding to the top and bottom of buffer are tracked by the stack pointer itself (bottom), and a stack-pointer offset address equal to the depth of the buffer. With a paged stack memory management scheme, the offset addition will be between 8 and 12 bits for the most likely architectural options. We shall compare the performance of the two buffering strategies, under various conditions, in later sections of this paper.

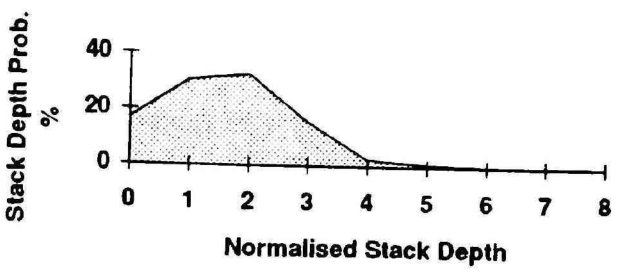

Fig.6. Normalised Stack Depth Probability.

Stack Behaviour and Buffer Performance

Buffer performance is dependant both upon stack depth behaviour, buffer size, and the algorithm applied. The dynamic behaviour of the data stack is illustrated in Fig.6, which represents the normalised stack depth profile for a set of C benchmarks compiled to stack code. The nature of compiler generated code is quite different from that produced by a good FORTH programmer. Rather than keeping useful data items on the data stack during a series of computations, the compiler tends to start with an empty stack, fetches the associated operands, then computes and stores the results, leaving the stack empty once again.

This system will use the minimum number of stack cells, a subject well researched by Bruno *et al* (1975), but is not necessarily the optimal choice for absolute performance. We shall see later that optimisation of the raw code produces better use of the stack, but not without changing the behaviour of the stack. The two stack buffer strategies of interest were simulated, and were found to perform as in Fig.7., which reveals the relationship between increased buffer size and reduced stack spill traffic. On logarithmic scales the characteristics are approximately straight lines, implying an exponential form of behaviour.

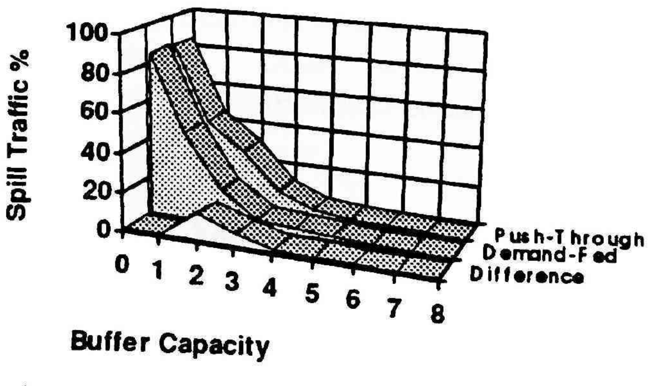

Fig.7. Stack Buffer Performance Comparison.

A Mathematical Model of Buffer Behaviour.

We have found that the buffer spilling behaviour can be approximated by an exponential decay formula of the form given by equation 1.

$$S_{(x,b)} = s \times e^{\omega\, b} \qquad (Eqn.\ 1)$$

Here, $S_{(x,b)}$ represents the spill traffic for a buffer algorithm denoted by 'x', with a buffer size of 'b' cells. The damping efficiency 'ω' determines the rate at which traffic is reduced when buffer size is increased. It was determined from analysis of the data for Fig.7, that the damping factors for the two algorithms were -0.773 and -0.715 (for Demand-fed and Push-through buffers respectively). The baseline traffic was measured at $s = 0.846$ for the benchmarks considered, indicating that 85% of instructions caused a change in data stack depth. Hence the two algorithms can be represented by:-

$$S_{(Demf,b)} = 0.846\, e^{-0.773\, b}$$
$$S_{(Pthr,b)} = 0.846\, e^{-0.715\, b}$$

Comparative Efficiency

A comparison of Demand-Fed and Push-Through buffering (as shown in Fig.7) indicates that the demand fed algorithm is marginally better for small buffers than the tagged-push-through buffer, this is reflected in the ω constant. With larger buffer capacities, both algorithms meet the objective of eliminating the majority of stack spill traffic. If hardware considerations such as logic complexity are important, then it may be that the simpler push-through-buffer is a better choice.

LOCAL VARIABLE MANAGEMENT.

The second area of attack, in reducing bus bandwidth requirements, centres upon the reduction or elimination of local variable traffic during computation. Local variable transfers are a significant problem whenever high-level language based computation is encountered, and usually arises from the residence of local variables in main memory.

In our UTSA design, we have decided upon maintaining locals in a third stack, held in main memory. This 'Frame Stack', can be addressed by offset addressing relative to the frame pointer. It follows that, whenever a computation takes place, the contents of the operands (being held in local variables) are fetched from the external frame stack to the internal data stack. After computation, the result is written back to the corresponding local variable in main memory. This is clearly going to generate a large amount of variable traffic, as was apparent in the traffic breakdown of Fig.2.

Local Variable Redundancy.

After studying a suite of benchmarks, we found the dynamic distribution of locals as in Fig.8. We have found that the typical distribution of locals in an executing program to be such that local variable fetches outweigh local variable stores by 3 or 4 fetches to every store.

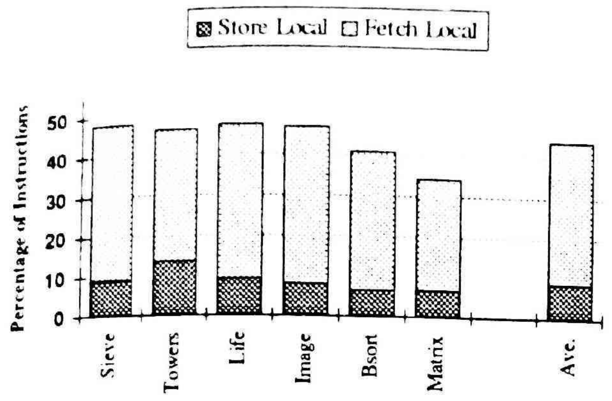

Fig.8. Percentage of locals in executed code.

ORIGINAL CODE		OPTIMISED CODE	
lit	4	lit	4
fp-		fp-	
!loc	2	rsu3	
!loc	1	tuck2	
!loc	0	rsu4	
lit	2	dup	
@ loc	0	rsu3	
@loc	1	lit	2
mul		rsu3	
@loc	0	mul	
@loc	2	rsd3	
mul		rsd4	
add		tuck3	
@loc	1	mul	
@loc	2	add	
mul		rsd4	
add		rsd3	
mul		mul	
!loc	3	add	
@loc	3	mul	
lit	4	lit	4
fp+		fp+	
exit		exit	

Fig.9. Stack Based Code to calculate Surf(x,y,z)= 2x{(x× y)+(x× z)+(y× z)}.

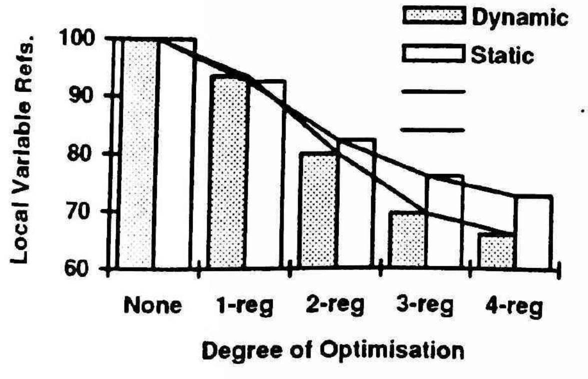

Fig. 10. Stack-cell access Vs Local Reduction.

This result implies that local variables are frequently utilised in a non-destructive fashion, whilst being modified relatively infrequently. This short-term invariance of the variables allows us to make duplicates of their contents on the data stack, which may later be used in preference to external memory access.

Figure 9 illustrates the level of improvement that can be achieved using this technique, showing a compiler-generated subroutine fragment before and after intra-block scheduling.

It should be apparent in the case of Fig.9. that all locals (@loc and !loc) are eliminated from the code without lengthening the instruction count. In the origional un-optimised code tthere were 11 local variable references. In general, a few residual locals remain, but execution times significantly improve.

Effectiveness of Intra-Block Scheduling.

In order to assess the performance of Koopman's technique, we produced a series of measurements to indicate its effectiveness in removing local variables from benchmarks. The results extended Koopman's work by varying the degree of optimisation possible to reflect the restrictions in top-of-stack access that may occur in differing stack processor architectures. This is illustrated in Fig.10.

Figure 10 shows clearly the effects of permitting increasing cell accessibility at the top of stack.: As we increase stack accessibility more local variables are eliminated. The effect is particularly visible for 2 and 3 stack-cell access, but, for 4 stack-cell accessibility and beyond there is a rapidly diminishing return. The increased complexity of hardware supporting deeper stack access may make this undesirable.

Impact On Stack And Buffer Behaviour

An important factor, and one which was previously ignored, is the effect that intra-block scheduling has upon stack behaviour. By radically altering the use of the data stack by the program code being executed, we inevitably cause major alterations in that programs stack behaviour, as can be seen in Fig.11.

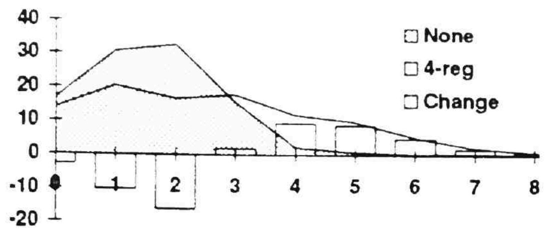

Fig.11. Normalised Stack Depth Probabilities.

Even though a small reduction in baseline traffic (from $s = 0.846$ to $s_0 = 0.786$) was observed, it was found that application of intra-block scheduling resulted in poorer buffer performance. The stack buffer's damping efficiency for a given size of buffer was reduced because the stack behaviour it attempts to dampen, has altered. Table-1 emphasises the change in the buffer behaviour. We have found that the effect of intra-block scheduling upon stack behaviour is to increase stack depth range, with a typical doubling of depth variation during program execution. As a result, the stack buffers have a harder task to perform in reducing stack spill traffic, and this results in degraded stack buffer performance, as illustrated in Fig12a and 12b.

We can now represent the degraded stack buffer performance by the models given below, and hence begin to mathematically quantify the degrading effects of local variable optimisation on buffer performance.

	Unoptimised	Optimised
Demand Fed	$\omega = -0.773$	$\omega_0 = -0.592$
Push-Thru	$\omega = -0.715$	$\omega_0 = -0.566$
Base traffic.	$s = 0.846$	$s_0 = 0.786$

Table-1. Buffer Model Parameters.

$$S(Demf,b_0) = 0.786\, e^{-0.592\, b}$$
$$S(Pthr,b_0) = 0.786\, e^{-0.566\, b}$$

Reduction in Buffer Efficiency.

Having measured the buffer performance both before and after applying local variable reduction, we can make a comparison between the approximations of the mathematical models in terms of thier ω values:

Demand fed:

$0.773 \div 0.592 = 1.30$ (i.e., 30% worse).

Push-Through:

$0.715 \div 0.566 = 1.26$ (i.e., 26% worse).

If we normalise the baseline traffic to 1.0 then we may verify this by equating the buffer efficiency for a given value of b.

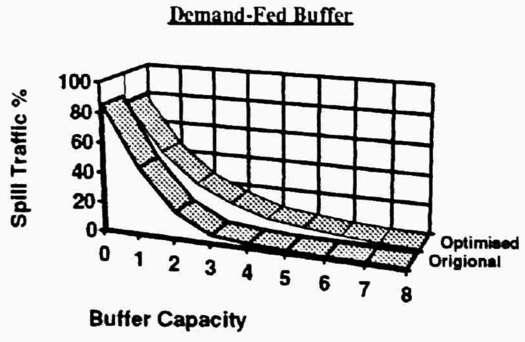

Fig.12a. Effect of Intra-block Scheduling on Demand-Fed Buffer.

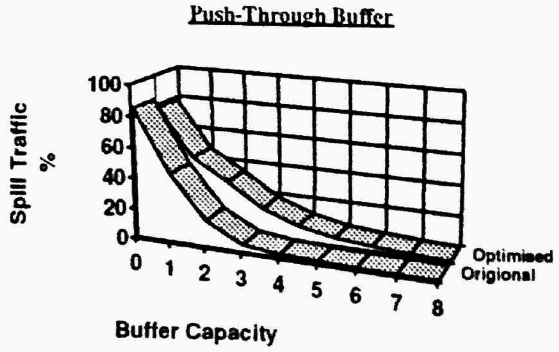

Fig.12b. Effect of Intra-block Scheduling on Push-Through Buffer.

For example, the demand fed buffer with b= 2 delivers performance of :-

$$S(demf,2) = s \, . \, e^{-0.773\,.\,(2)} = 0.213\,.\,s$$

In order to achieve similar performance with an equal workload after intra-block scheduling, we have suggested a buffer 1.3 times larger, hence:-

$$S(demf,2.6) = s \, . \, e^{-0.592\,.\,(2.6)} = 0.214\,.\,s$$

Thus, it appears that the relationship for traffic damping is verifiable, and a degradation in buffer efficiency of 30% due to the change in stack behaviour is quantified correctly by our model. Naturally, we may only select buffers of size 2 or 3 in practice rather than 2.6, but the implication is that any demand-fed buffer would need to be 30% larger in order to maintain equal performance after applying intra-block scheduling. Since the baseline traffic reduces slightly (by about 8%), this slightly offsets the reduced buffer efficiency, so that the actual increase in spill traffic may not be as great as might be envisaged.

The effects of Intra-Block scheduling clearly have an influence over other performance factors, and whilst the effects may not be particularly significant in that case, the application of global scheduling [Koop92] would have a more pronounced effect, and will no-doubt be found to have more significant influence over the behaviour of buffers in stack based systems.

INSTRUCTION FETCH REDUCTION.

In order to reduce the final bus bandwidth bottleneck - instruction fetch overhead, we chose to investigate ways of exploiting the minimal length of stack based instructions in a 32 bit data width architecture.

If instruction fetch overheads were significantly reduced, the result would be an architecture that could execute several instructions per memory cycle without cache - provided that stack spilling and local variable traffic had been dealt with as discussed previously.

Packed instruction word schemes.

By packing several short stack operations in each 32 bit word it was expected that several instructions would be executed for every instruction-word fetch issued.

This concept has been investigated in a RISC context by Bunda (1993) and Patterson (1985), but failed to generate substantial savings due to RISC's long instruction lengths. However, the trade-off discussed by Bunda for register-window size Vs. code density (hence reduced instruction fetch overhead) does not translate to the implicit operand addressing scheme of the stack processor, so more significant gains can be made with this concept when using stack processor technology as the core architecture.

UTSA Instruction set format.

The full set of word formats for our proposed 'University of Teesside Stack Architecture' are illustrated in Fig.13. One format permits up to three operations to be packed in a memory word, implying a peak throughput of 3 times the external memory bandwidth.

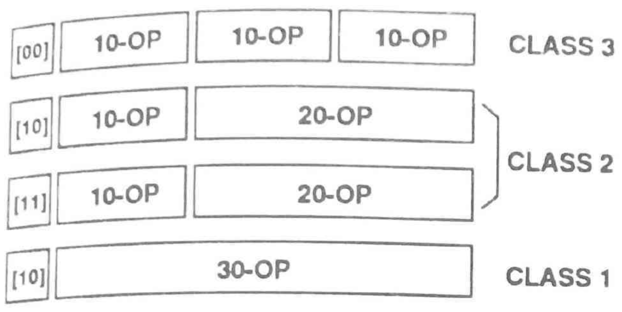

Fig. 13. Instruction packing for 32-bit word.

In practice, longer instructions are occasionally required, to support long literals (constants) for example, or absolute memory addresses. This means that the code density will rarely reach 3 instructions per word in practice.

The following results indicate the effectiveness of the instruction packing scheme for both static (Fig.14) and dynamic (Fig.15) code density.

The static code density only reflects the mapping of opcodes onto memory words, not the efficiency of the instruction fetch scheme. Dynamic code density does reflect this factor however, and represents the number of instructions executed (including any nops) for each word fetched.

The figures show that the static code density is quite impressive, reaching an average of over 2.7 instructions packed in every word. The dynamic figure shows, however, that the number of instructions executed for every word fetched is somewhat lower - an average of 2.2 instructions per word.

The reduced density of dynamic measurements reflect the fact that basic blocks may end in the middle of an instruction word, while looping and procedure call frequencies weight some localised characteristics of the program in preference to others.

Word Alignment Vs. Code Density.

One area where trade-offs have been identified is in the area of word alignment. The issue is whether we should make calls and jumps word-aligned, or permit a jump to be made to individual instructions within a memory word.

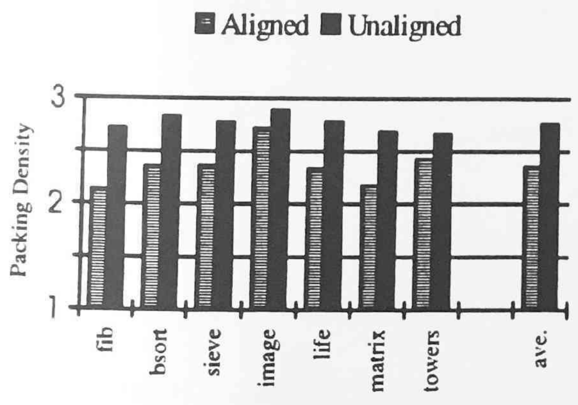

Fig.14. Static Packing Densities.

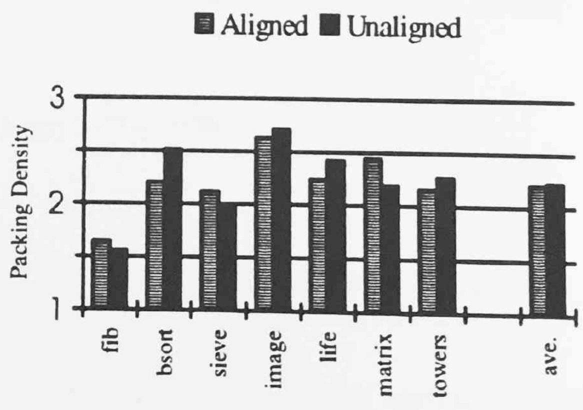

Fig.15. Dynamic Packing Ratios.

It can be seen from Fig.14, that forcing program flow to take place on word boundaries results in a substantial decrease in static code density, from 2.7 to 2.3 instructions per word. The unaligned code mapping was found to produce 20% smaller program memory requirements than aligned code. When dynamic code density figures are examined (Fig.15) it is clear that unaligned code displays no consistent advantage over aligned code. The average gain for the whole benchmark suite was almost nil.

Without a more comprehensive set of benchmarks, it seems that we cannot make firm conclusions about the benefit of word-aligned program code. But our results at least serve to indicate that the application of optimisations to improve dynamic code density would be worth investigating.

Word alignment Trade-offs

Since it seems to be clear that program size is reduced by using unaligned code, then there may be some justification for using it. Conversely, non-aligned code seems to deliver comparable dynamic performance, yet also implies larger address range capability and simpler address generation logic. Additional arguments, such as cache efficiency may also play a part in the evaluation process. We are concerned with improving systems where cache is not acceptable, but other applications may have quite different operating requirements.

A more detailed investigation would help to resolve these questions. Examining both hardware and software factors whilst accounting for optimisation strategies targeted toward increased dynamic code density (hence optimising for execution speed) would help to quantify these trade-offs. This is beyond the scope of this paper, but may be addressed in future studies.

OVERALL PERFORMANCE GAINS

Now that we have examined the three areas which were identified as critical for our proposed stack processor performance, we can quantify the effects of each optimisation on the system, and assess the gains likely for combination of these strategies.

Mathematical Performance model.

Let us first propose a mathematical model in order to represent the memory bandwidth requirements of a stack processor system, represented by equation 2, which assumes overlapping fetch-execute cycles.

$$M = \frac{1}{\iota_f} + S_{D(x,b)} + S_{R(x,b)} + m_L + m \qquad \text{(Eqn. 2)}$$

Where	ι_f	= Dynamic instruction density,		
	$S_{D(x,b)}$	= Data stack spill Traffic,	$S_{R(x,b)}$	= Return stack spill Traffic,
	m_L	= Local Variable Access/Instr,	m	= Other Mem Access/Instr.

We have already introduced the stack buffer spilling function $S_{(x,b)}$, which represents the data stack spilling overhead for a given stack. However, the stack processor is typically a dual stack system, hence we now have two stack spilling terms, one for data stack traffic, and another for return stack traffic.

In practice return stack baseline traffic is quite small with $s = 0.1$ being typical, so that even minimal buffers will eliminate most return stack spilling. The return stack is not subject to any trade-offs pertaining to intra-block scheduling. We have measured the key characteristics for a simulated stack processor system, with a series of C compiled benchmarks, as shown in Table-2.

Evaluation of speedup

From the above values we can assess the performance of the imagined stack processor for a specific condition, for example, a demand fed buffer with size b=8, without local variable optimisation gives:-

Symbol	Value
ι_f	2.20
S(demf,b)	$0.846\ e^{-0.773\ b}$
S(demf,b$_o$)	$0.786\ e^{-0.592\ b}$
m_L *(Before Optimisation)*	0.409
m_L *(After Optimisation)*	0.269
m	0.08

Table-2. Key Characteristics.

$1/\iota_f$	= 0.4545	
$S_{D(Demf,8)}$	= 0.00174	
$S_{R(Demf,8)}$	= 0.00021	
m_L	= 0.4090	
m	= 0.0800 +	
Total M	= 0.945	*(model projection)*

Hence the memory bandwidth requirement would be M = 0.945. In other words the CPU issues 1.06 instructions for each memory access, barely achieving single-memory-cycle execution.

Now, after applying local variable scheduling, we may identify the new performance of the system, as shown in the second projection.

$1/\iota_f$	= 0.4545	
$S_{D(Demf,8)}$	= 0.00689	*(increased)*
$S_{R(Demf,8)}$	= 0.00021	
m_L	= 0.2690	*(decreased)*
m	= 0.0800 +	
Total M	= 0.8106	*(decreased overall)*

We find that the system performance is now 1.23 instructions executable in every memory cycle, or a speedup of 16 % by using local scheduling, with no need for cache.

Performance projection and verification.

The series of curves in Fig.16 shows the relative performance of several machine/software configurations as projected by the mathematical models introduced. Such theoretical performance projections mirror measurements made on our CPU simulator, verifying the mathematical model. For example, we find in simulation that a 100 ns memory cycle has the potential to deliver 12 Mips with packed instruction fetch and local variable optimisation.

Such theoretical performance projections mirror measurements made on our CPU simulator, verifying the mathematical model. For example, we find in simulation that a 100 ns memory cycle has the potential to deliver 12 Mips with packed instruction fetch and local variable optimisation. The above simulation figures agree well with the result for the D-M-O model (Demand-fed Multi-fetch Optimised) in Fig.16, for example, where 0.81 memory references per instruction are predicted with adequate buffers

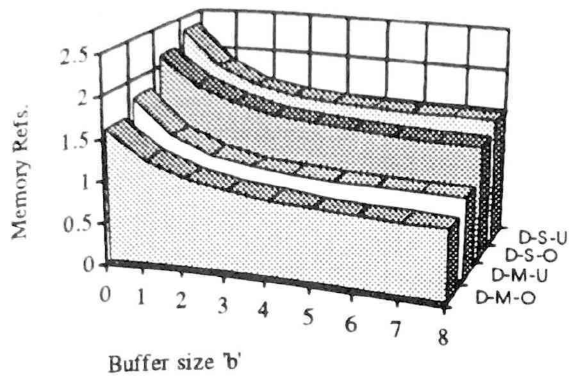

Fig 16. Performance Projections by Model.

D = Demand Fed Buffer
S/M = Single/Multi Fetch
U/O = Un-optimised/Optimised Locals

CONCLUSIONS.

We have presented a number of methods and techniques for improving stack processor performance under the limited conditions of embedded systems environments. Most of these techniques were previously investigated in empirical terms, making formal comparisons difficult. Our paper has presented a number of techniques, illustrated trade-offs in each case, and attempted to represent those effects and trade-offs with a mathematical modelling approach, hence allowing a numerical evaluation of trade-offs and performance gains associated with optimisation.

The ultimate result of the work presented here is to combine mathematical components, reflecting various aspects of machine optimisation, and therefore present an overall model of performance. Our model agrees with simulator output, and allows quantitative comparisons to be applied to performance data, rather than the previous empirical ad-hoc approach found in previous publications. Current work is concentrating upon VHDL processor models, which should allow us to further validate our findings at the hardware level, and allow lower level trade-offs to be investigated with consideration given to gate-delay and silicon-area trade-offs.

ACKNOWLEDGEMENTS.

1. A new compiler is being developed by Rowley Associates, UK, and MPE Ltd.
2. We wish to acknowledge the support given by Microprocessor Engineering Ltd, Southampton. UK, both as a research sponsor, and for technical consultation.
3. The research programme was additionally supported by a UK Govt. SERC CASE Award for the period 1992-1995.

REFERENCES.

Bailey, C., Sotudeh, R. (1994) HLL Enhancement of Stack-Based Processors. EuroMicro Journal of Microprocessing and Microprogramming Vol. 40, 1994, pp 685-688.

Bailey, C, Sotudeh, R. (1994) The effects of Local Variable Optimisation on a C-based Stack Processor Environment. Proc. of the 1994 Euroforth Conference, Winchester Nov 1994.

Bailey, C, Sotudeh, R. (1995a) The Effect of Intra-Block Scheduling In a Stack Processor Environment. Proc. of the 1995 Rochester Forth Conference on emerging Technologies. Rochester, USA. June 1995.

Bruno, J, Lassagne, T. (1975). The Generation of Optimal Code For Stack Machines. Journal of the ACM, July 1975, Vol 22, No 3, Pages 382-396.

Bunda, J., Fussell, D., Jenevin, R., Athas, W. (1993). 16-Bit vs. 32-Bit Instructions for Pipelined Microprocessors.

Hauck, E,A, Dent, B, A. (1968). Burroughs B6500/B7500 Stack Mechanism. Proc. AFIPS SJCC.

Duncan, F. G. (1977). Stack Machine Development: Australia, Great Britain, and Europe. IEEE COMPUTER.

Hamblin, C, L. (1957). An Addressless Coding Scheme Based on Mathematical Notations. Proc of the W.R.E Conf. on computing, Salisbury, South Australia, June 1957.

Hennessy, J, L, Patterson, D, A. (1990) Computer Architecture: A Quantitative Approach. Morgan & Kaufmann Publishers, Inc. ISBN 1-55860-069-8.

Koopman, P. (1989) Stack Computers - The New Wave. Ellis-Horwood Press. ISBN 0-7458-0418-7.

Koopman, P. (1992). A preliminary exploration of optimised stack code generation. Proc. of 1992 Rochester Forth Conference. Rochester, USA, June 1992.

Koopman, P. (1993) Perils of the PC Cache. Embedded systems Programming, May 1993, pp 26-34.

Patterson, D. A. (1985). Reduced instruction set computers', Comms. of the ACM v28 No1, pp8-21.