Instruction complexity and implicit execution architectures: orthogonality, optimization, and VLSI-Design.

Chris Bailey and Reza Sotudeh

University of Teesside,
Borough Road, Middlesbrough,
TS1 3BA, United Kingdom

Email: c.bailey@tees.ac.uk

ABSTRACT

With the rapid emergence of Java as a machine independent programming environment, the stack-based processor model has entered a renaissance of interest after a long period of indifference from the general research community. The implicit execution mode employed in Java byte-code interpretation demands optimal execution of code on a stack-based virtual (or physical) machine model.

One efficient solution is to utilise a stack-based microprocessor paradigm in which the associated native code mirrors the semantics of the object byte-code. This approach already has a past-history that spans the era of ALGOL through to present day FORTH-Engine implementations. However, the development of virtual and/or native semantic content of these environments has suffered a haphazard evolution, influenced more by convenience than by formalised concepts of stack-oriented execution. As a result, instruction-set schemes for stack based processing models have suffered from a lack of orthogonality that suits human programmers, but complicates automated code generation and optimisation.

In this paper, we propose a classification scheme, a scaleable and symmetrical model for stack manipulation, which allows the degree of instruction complexity to be specified and the orthogonality of those functions to be visualised.

Using a virtual machine simulator, VHDL models of CPU designs, and analysis of synthesised VHDL models, we show that the effects of varied degrees of instruction complexity have a clear and quantifiable impact on program efficiency, system performance, and VLSI logic characterisation.

INTRODUCTION

Stack machines, processors which employ an implicitly addressed operand stack in place of an explicitly addressed register-file, offer fast interrupt handling, minimal parameter passing overheads, and very high code density. Whilst mainstream acceptance has never featured significantly in this class of architecture, recent attention has been brought to focus by the sudden arrival of JAVA technology. Previous developments in this area have often been driven by the requirements of human programmers, and attempts to 'optimise' instruction set efficiency by adding special instructions. Whilst this may make life easier for hand coding and allows semantic mapping of source code, such as FORTH, onto machine level instructions, it does not possess any formal structure or substance upon which to classify or evaluate such architectures, or their performance.

Quantitative assessment demands identifiable and expressable terms to define architectural complexity, and comparative studies are hindered when each architecture appears to be unique. Existing architectures such as FRISC3 (Hayes 1989), ShBoom (Turley 1996a), and PicoJava (Turley 1996b) appear poles apart in many respects, yet have much commonality.

In this paper we evaluate instruction set features common to many stack processors, and thus expose a method of classification that permits the degree of complexity of a stack architecture to be specified in simple numerical terms. We are then able to identify a sub-classification of stack manipulation operators that distinguishes functionality, and consequently present a scaleable and orthogonal model which permits quantitative studies of performance-versus-complexity for the stack-architecture paradigm.
.
Through the use of code optimisation tools, architectural simulators, and VHDL logic synthesis, this paper presents a series of evaluations that identify trade-offs for code optimisation, execution efficiency, and CPU logic latency, as a function of architectural complexity. In doing so we are able to evaluate complexity as a hardware/software trade-off, and present an evaluation for overall performance of a stack-based processor system.

MODERN STACK PROCESSOR SCHEMES

Mainstream stack processor technology falls into two groups, the 'FORTH engines' developed to serve the needs of real-time embedded systems, and the new JAVA architectures, that are rapidly being adopted in network computers, set-top boxes, and so-on (Turley 1996a). In spite of continued FORTH development, the stack manipulations that are provided therein have remained relatively stable, reflecting the human-oriented approach to interpreted source-code construction. Typical stack manipulations that are found in stack processor instruction sets are represented below.

Dup	*(duplicate top of stack),*	**Drop**	*(pop and discard top stack item)*
Swap	*(exchange two top items)*	**Rot**	*(circulate top three items)*
Over	*(copy 2nd item to top)*	**Tuck**	*(push copy of top under second item)*

Java technology is primarily driven by the Java Virtual Machine model, in which a very small set of stack manipulations are offered. The Java virtual machine provides only five single-word operations:-

Dup	*(duplicate top item)*	**Pop**	*(drop topmost item)*
DupX1	*(duplicate top, place under 2nd)*	**Swap**	*(exchanged top 2 items)*
DupX3	*(duplicate top, place under 3rd)*		

Whilst FORTH and Java virtual machines augment their basic schemes with double-word operations, such that dup2X1 in the Java-VM duplicates two topmost items rather than one, an efficient architecture would provide a word-length that makes such double operations infrequent and potentially redundant where a RISC approach is being adopted. We therefore concentrate upon single cell operators only in this paper.

	Java VM	FORTH	FRISC-3	ShBoom
Dup	●	●	●	●
Drop	●	●	●	●
Nip		●		
Drop 2nd		●	●	
Drop 3rd				
Drop 4th				
Over		●	●	
Tuck	●	●	●	
DupX3	●			
Swap	●	●	●	●
Rotate 3		●	●	●
Rotate 4				

TABLE 1, Stack Manipulations for selected models

Silicon implementations of the Java VM model are further limited in practice. ShBoom for instance provides only Dup, Drop, Rev (revolve 3 stack items), and XCG (Exchange two top stack items) It is interesting to note that ShBoom's stack manipulators are more characteristic of a FORTH engine than a Java influenced design in spite of its current marketing.

When one chooses to analyse the stack operators that are available to a series of machines, as shown in Table 1, it becomes apparent that a philosophy of orthogonality and scaleability is conspicuous by its absence. However, it can be seen from Table 1 that there appears to be no agreement upon the requirements for effective stack manipulation. In the absence of any method of architectural classification, the issue of relative performance evaluation is quite complex.

A NEW CLASSIFICATION TECHNIQUE FOR STACK MANIPULATIONS

Having identified a need for a clearer understanding of stack manipulations, a demographic projection of instructions in terms of degree-of-complexity was visualised, in which an operation which requires hardware access only to the topmost item would be classified as being of degree-1, whilst a degree-4 operation would require hardware access to the 4th item in the operand stack.

By presenting this classification as an expanding shell of 'degrees of complexity', the authors are able to present the 'bulls-eye diagram' concept of Fig. 1, where operators which are efficiently implemented in current FORTH engines are plotted. It can be seen that generally, the FORTH operators are restricted to the 3rd degree, and not in a uniform fashion. Applying the same technique to the Java VM model yields Fig. 2. Comparing Fig.1 and Fig.2, suggests that FORTH has evolved orthogonal attributes whilst the Java VM has adopted scaleability. This is quite logical if one considers the manual/automated code generation styles which they pertain to in practice.

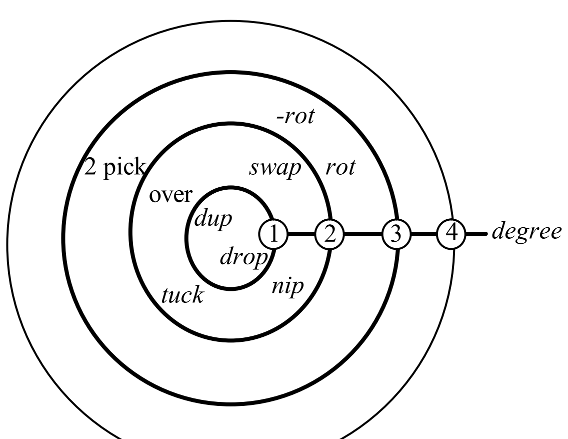

Fig.1 FORTH Operators classified by Degree of Complexity

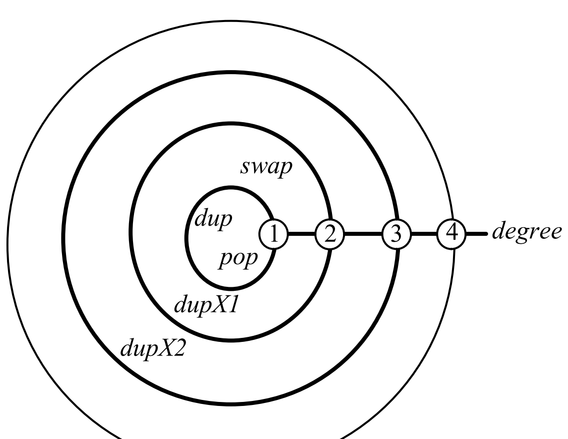

Fig.2, Java VM Operators Classified by Degree of Complexity

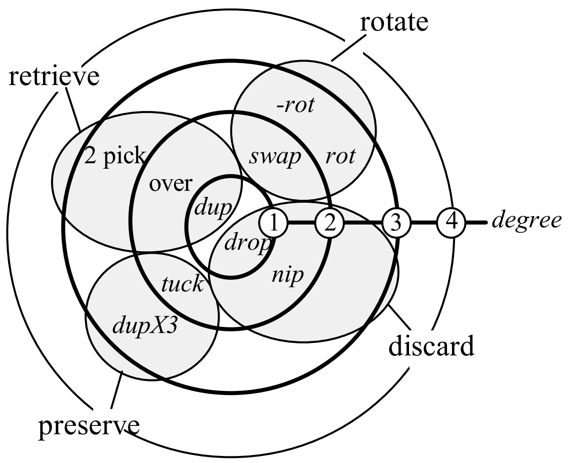

Fig. 3 - Instruction groups for FORTH and JAVA VM Combination.

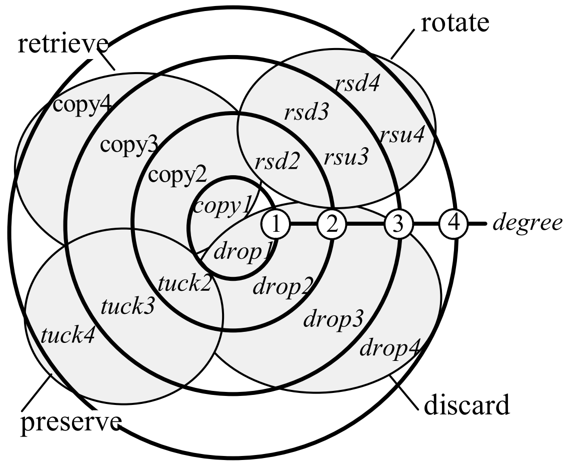

Fig. 4 - Newly proposed scaleable and orthogonal instruction scheme.

A SCALEABLE AND ORTHOGONAL STACK MANIPULATION SCHEME

Whilst there is a clear distinction between orthogonality and scaleability, it has to be asked why should we not combine both attributes, and in doing so what would the consequences be ?. To answer such questions it was necessary to further develop the presented model of stack processor characterisation. It proved possible to categorise stack-manipulation operations into four groups - each with a particular yet inter-related function. By combination of the FORTH and JAVA VM models, the bulls-eye diagram of Fig.3, is arrived at, highlighting the four groups.

The functional groups are classed as follows: **'Preserve'** forces a copy of the current TOS (top-of-stack) to be placed at the specified position in the stack, **'Retrieve'** copies an item from a specified position in the stack, and thus has an oppositional relationship to Preserve. **'Rotate'** causes circulation of stack cells to the specified depth, effectively bringing an item to the top of stack, whilst **'Discard'** operators permit an identified stack cell to be removed. Examination of Fig.3 indicates that there are unexploited areas of the instruction scheme and it is apparent that, once filled, these extra operations would complete a scaleable and orthogonal scheme for stack manipulation. Such a scheme is presented in Fig. 4, with revised nomenclature for operators in each class. Thus, in Fig. 4, we find a scaleable and orthogonal instruction set scheme for stack manipulations, which is both uniform and quantifiable in terms of complexity.

COMPLEXITY VERSUS CODE OPTIMISATION

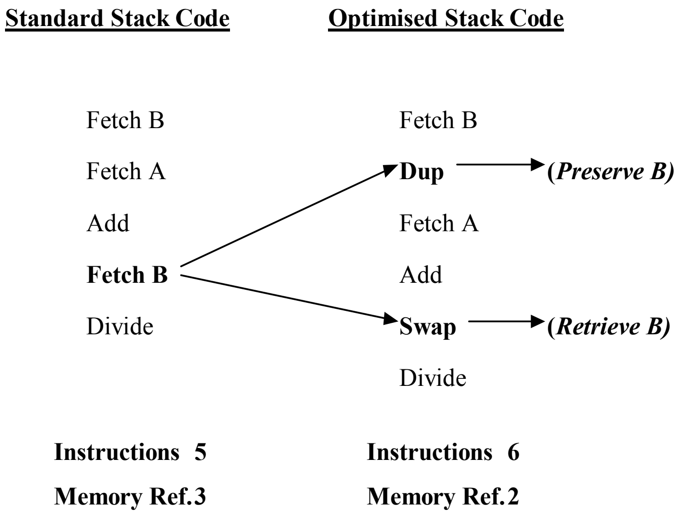

Fig. 5, Example of Variable Scheduling.

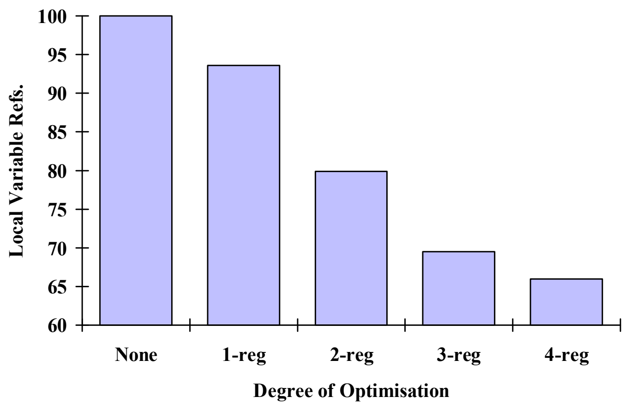

Fig. 6, Local optimisation versus Complexity.

Now that a basis for quantifying complexity has been introduced, it is possible to explore the performance trade-offs that relate to the presented scaleable and orthogonal scheme. Here we will consider the impact of varying degrees of instruction-set complexity upon the ability to perform code optimisation, and also evaluate the implications for CPU logic latency in each case.

Local-variable optimisation techniques have recently been developed to reduce the excessive use of operand reloading that takes place due to the inherently destructive action of stack-based computation. This has allowed recovery of ground lost to mainstream architectures, which have had the advantage of register optimisation (Chow 1984). Recent studies suggest that gains delivered are significant (Koopman 1992, Bailey 1995a, Maierhofer 1997).

There now seems to be growing evidence that comprehensive studies which showed stack processor technology at a disadvantage (based on the state of research at that time), are now in need of review (Flynn 1992, Bailey 1996).

Local-variable optimisation relies upon stack manipulation to copy items currently at the top of stack, to points further down the stack space, where they can be retrieved or allowed to 'float' to the top of stack through natural stack movement. This eliminates the need to reload short-term-invariant operands which are consumed during destructive computation.

It has now been proven that results are significantly affected by the degree of complexity found in the stack manipulation scheme being utilised (Bailey 1995b). The trade-off between architectural complexity and successes of code optimisation is presented in Fig. 6. Although the trend of Fig. 6 shows increasing architectural complexity driving enhanced optimisation, a diminishing return is also apparent which suggests that gains are unlikely to be significant beyond the 4th degree. This can be blamed upon the scope of Koopman's technique to allow optimisation only within basic block boundaries (i.e. intra-block optimisation). New developments such as 'inter-block scheduling' are expected in the next few years, and will no doubt make better use of deeper stack space. Gains for larger degrees of complexity will no-doubt become significant enough to warrant further investigation at that point.

HARDWARE TRADE-OFFS : COMPLEXITY VERSUS LATENCY

The majority of recent work has concentrated upon the effectiveness of the scheduling algorithms in dealing with the occurrence of local-variables in stack-based code. Research has extended this work to include the effect upon execution time, using hypothetical or specific machine architectures (Bailey 1996, Maierhofer 1997). However, there are no previous examples of examination of processor logic latency, and its contribution to performance as a function of varying degrees of complexity. Perhaps this is not surprising in some respects - without scaleability and orthogonality, such a study would struggle to apply terms of reference in any generalised way. But here we are been able to apply our newly proposed scaleable and orthogonal stack manipulation scheme to this problem, and evaluate machine complexity in terms of logic latency and gate utilisation.

When one examines the instruction scheme proposed in Fig. 4, it becomes apparent that the number of stack manipulation operators increases in a roughly linear fashion from a non-zero baseline as the degree of complexity is increased. An architecture with a single degree-of-complexity would have only two operators, dup and drop. A 2nd degree architecture has six operators, a 3rd degree architecture has 11, and a 4th degree yields 16 instructions (See Fig 7).

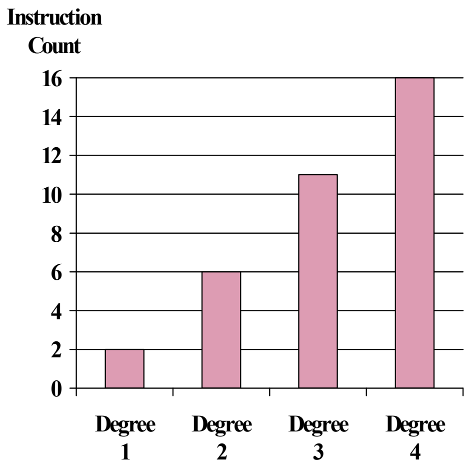

FIG. 7, Stack Operators Vs complexity

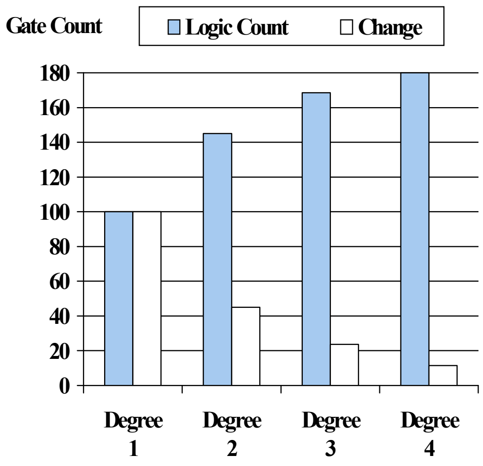

FIG. 8, Logic Gate Utilisation Vs Complexity

Careful examination of Fig. 8 reveals an interesting mathematical characteristic. Apart from the unexpected implication that increased complexity can be accommodated by progressively reducing logic penalties[1], it appears that the increase in logic gate utilisation approximates to Equation. 1.

[1] Addittional complexity does not add much 'new logic' into the system at each increment, since the VHDL optimisation tools make full use of recombination of existing logic. This explains the diminishing penalty.

It might be thought that more complexity is better, after all Eqn 1 suggests that an architecture with a degree of 8 only increases logic utilisation by 6% compared to that of a 4th degree architecture. However, an overriding concern of greater significance has yet to be considered in our analysis;- the impact of logic latency.

$$\sum_{n=1}^{n=d} 2^{(1-n)}$$

Eqn. 1, Gate utilisation as a function of architectural complexity.
Where d = chosen degree of complexity.

Taking measurements for ALU propagation from source to destination cells in the hardware stack, we find that logic latency increases in a non-uniform manner as complexity is increased. This is illustrated in Fig. 9. What was not initially suspected, was that latency would increase in a stepped fashion, such that changes from degree 1-to-2 or 3-to-4 have little effect upon propagation delays in the synthesised circuits.

We feel that this can be explained when one considers the multiplexing requirements to select an operand from the top of stack, where an odd number of selections results in a redundant opportunity for an additional data path to be multiplexed. The redundant opportunity is exploited when the complexity is increased to the next level, with only small additional increments due to miscellaneous decoding effects of each additional sub-set of instructions to the scheme.

The stepping effect, and our suggested explanation, implies that a step will also be seen when moving from 4th to 5th degrees, but that the next step will not be observed until moving to a degree of 9, due to the Log_2 nature of the synthesised multiplexing-hierachy, where n levels can handle 2^n selections.

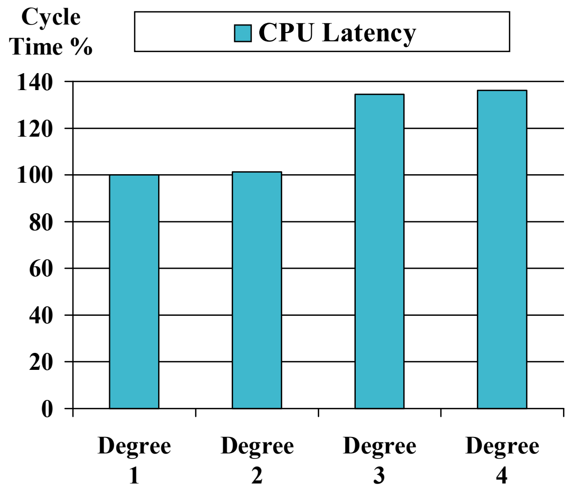

Fig. 9, Critical path latency, as a function of complexity.

OVERALL PERFORMANCE

In our previous work we have sought to evaluate optimisation techniques in the context of a holistic system view, taking into account not only the gains introduced by applying optimisation techniques, but the penalties that those techniques have upon other aspects of system behaviour such as stack-caching and memory bandwidth (Bailey 1995b). This evaluation has allowed us to produce detailed mathematical models through which to explore stack-processor behaviour (Bailey 1995a). We have also found that the best expectations for such optimisation techniques are tempered by the degradation of other key system characteristics, but overall performance had indicated a clear gain in execution time as a result.

It is now possible to combine the final factor into the analysis and quantify the full impact of architectural complexity against overall system performance, without ignoring interacting elements such as code optimisation and stack cache behaviour, complexity and speed. In our earlier work we presented overall execution-time profiles for a number of benchmarks, including stanford-mips benchmarks[2].It was found that overall performance gains could be as large as 25 % for relative execution time, with a couple of cases where performance was made worse (Fig. 10).

Once the identified increase in CPU cycle times is included in the analysis, the true overall performance is somewhat worse than expected. Our results have indicated that whilst 1st and 2nd degree architectures perform exactly as expected from the previous study, attempts to increase complexity in order to support more aggressive optimisation have a large cost. Instead of larger benefits, the gains are slashed (by the impact of logic latency) to the mid 90 % range, offering very little for all the effort invested in hardware and software optimisation. Table 2 summarises the data for utilisation, latency and relative performance.

One positive note of caution has to be stressed at this point. UTSA exploits only 56 core instructions, and would no-doubt have a larger base-line critical path latency if a more comprehensive 'commercially viable' instruction set architecture were developed. Perhaps in a fuller UTSA implementation the impact of increased complexity would be smaller overall .

CONCLUSIONS

Undoubtedly, the issue of performance in stack processor systems has become more complicated in recent years, but the need for clear terms of reference, and quantitative assessment has never been greater. Our results show that complex issues such as hardware-software trade-offs can be tackled, and successfully investigated, but indicate that some optimisation techniques may not be as good as previously thought. Future research must simplify and clarify the science of stack processor technology in order to understand and enhance the actual technology and techniques employed.

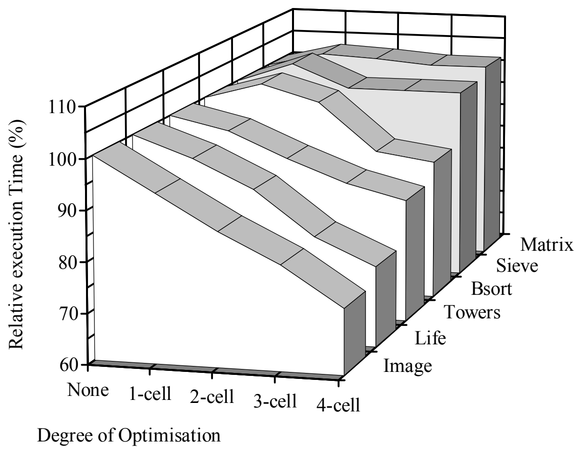

Fig. 10 Execution time Vs complexity.

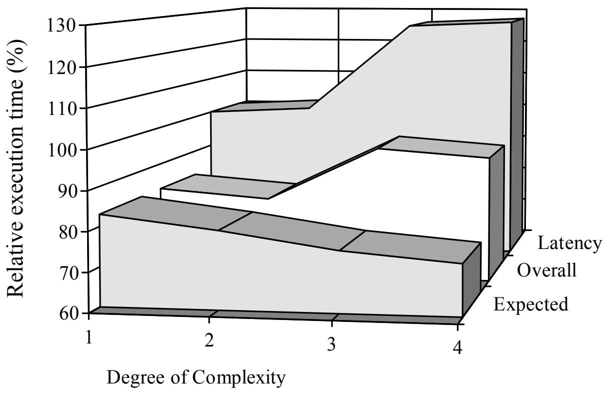

Fig. 11, Overall Execution Time, with CPU latency

	d=1	*d=2*	*d=3*	*d=4*
Gate Utilisation	1.0	1.45	1.69	1.80
Logic Latency	1.0	1.01	1.35	1.36
Overall Perf.	1.0	0.95	1.14	1.12

TABLE 2, utilisation, latency, and performance

REFERENCES

(Bailey 1995a) Bailey, C., Sotudeh, R. *'The Effects of Intra-Block Scheduling in a Stack Processor*

[2] Modifications were made only to circumvent limitations in our compiler tools, such as handling multi-dimensional arrays as single dimensional arrays.

Environment'. Proc. Rochester Forth Conference, Rochester, USA, June 1995.

(Bailey 1995b) Bailey, C., Sotudeh, R., (1995). *'Trade-offs for Memory Bandwidth Reduction in Stack Processor Design'*, Proc. of the 10th ICMCM, Boston, USA, July 1995.

(Bailey 1996) Bailey, C. *'Optimisation techniques for stack-based architectures'*. Ph.D. Thesis, July 1996, University of Teesside, Middlesbrough, UK.

(Chow 1984) Chow, F., Hennesey, J., (1984). *'Register Allocation by Priority-Based Coloring'*. Proc. of ACM SIGPLAN 1984 Symp. on Compiler Construction, SIGPLAN Notices, Vol. 19, No. 6, pp 222-232.

(Hayes 1989) Hayes, J., R., and Lee, S., C., *'The architecture of the SC32 Forth Engine'*. JFAR.

(Flynn *et al.* 1992) Flynn, M., J., and Mulder, H., M., (1992). *'Processor architecture and data buffering'*. IEEE Trans. on computers, Vol. 41, No. 10, October 1992, pp 1211- 1222.

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

(Maierhofer 1997) Maierhofer, M, Ertl, A. *'Optimised Stack Code'*. Deutche Forth-Tagung, 1997.

(Turley 1996a) Turley, J. *'New embedded CPU goes ShBoom'*. Microprocessor Report, April 15, 1996, Vol 10, No 5, pages 1, 6-10.

(Turley 1996b) Turley, J, 'Sun Reveals First Java Processor Core'. Microproc. Report, Oct. 28, 1997, p28-31.