

# Inter-Boundary Scheduling of Stack Operands: A preliminary Study

Chris Bailey  
Dept. of Computer Science,  
University of York,  
Heslington, York, Yo10 5DD

[Chrisb@cs.york.ac.uk](mailto:Chrisb@cs.york.ac.uk)

## Abstract

Stack-processors, which abandon register files and instead work directly on the stack content have recently enjoyed a resurgence of interest in conjunction with higher level languages. However, the way in which local variables are handled by typically stack-targeted compilers leaves much room for improvement. These issues are not restricted to stack architectures using a C-compiler, although this is the focus here.

The elimination of troublesome local variable references has been investigated in several studies [**Koop92**], [**Maierhofer97**], with some interesting trade-offs for stack instruction set design noted [**Bailey97**]. These studies have so far been limited to optimisation of memory dependencies only within each basic-block of object code.

Further development would have benefits in such areas as real-time systems where caching may be restricted, and memory penalties increased, whilst any move toward instruction-level parallelism in a stack-based environment would be severely handicapped if these issues are not first alleviated.

This paper presents a new technique for optimisation of local-variable references, “inter-boundary scheduling”, which extends the scope of optimisation beyond the basic-block boundary and allows local variable content to remain on the computation stack even when control-flow operations are encountered. Results presented suggest that the method can eliminate 10 to 20% of local variable references remaining after the existing intra-block techniques are applied. Data presented is based upon compilation of C source to a stack-based target architecture, however the method is open to application in wider areas including FORTH.

## 1. Introduction

The zero-operand computational model has received a rather unfair reputation as an outdated and inefficient technology which has had its day. However, many of the criticisms can now be either shown to be misconceived, or overtaken by improvements in stack-processor technology. Stack buffers, for example, can eliminate the argument that stacks extending into main memory create stack-thrashing and traffic bottlenecks [**Bell70**]. Meanwhile, the complaint that stack-based programs spend too much time manipulating stack content [**Amdahl64**], can largely be addressed through a more orthogonal stack management scheme [**Bailey97**].

However, as old arguments have been swept aside, new problems have emerged. In particular the need to effectively support referencing of operands such as local variables has been an obstacle in achieving higher performance with HLL’s such as C. The issue may to some extent be minimised in FORTH systems, since the hand-coding for most FORTH code tends to assist in optimisation of stack usage. However, the interest in automatically generated FORTH from front-end tools, and the work being conducted in areas such as C-to-Forth compilation, suggest that this is an area which needs further thought.

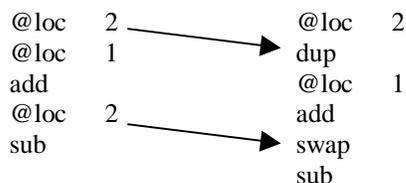
Recent progress has been made, with the development of algorithms for optimisation of repetitive local variable references within basic blocks of code (i.e. linear blocks of instructions bounded by call, or branch events). Whilst work by Phil Koopman indicated substantial performance benefits for this limited ‘intra-block’ scheduling [**Koop92**], there remain ‘unoptimisable’ local references which span basic block boundaries, and so lie beyond the scope of that method. Koopman alluded to hand-coded optimisations across block boundaries, and indicated further gains would be possible. Therefore an algorithm capable of

performing this task as a back-end optimisation tool would be highly beneficial.

This paper presents the results of a newly devised algorithm, proposed by the author, which allows locals to be optimised even across block boundaries, and handles the problems associated with multiple execution paths. The results show that further significant gains can be made in the elimination of local variable references, and highlights many new avenues which should lead to a far more in the future.

## 2. Existing Scheduling algorithms

The Intra-Block Scheduling technique, as proposed by Phil Koopman, exploits the fact that repeated references to local variables within a single block of code can generally be eliminated if a copy is kept on the computation stack when the first reference is made. In the following example “@loc 2” indicates fetches to local variable 2, and highlights the option to eliminate repeated fetches.



We can see in the example that the first fetch is followed by an extra instruction ‘dup’ which duplicates the local content. Later it is brought to the top of stack by the ‘swap’ instruction, as it has by then been buried under the result of the preceding addition. The Intra-Block technique attempts to identify such pairs and, by ranking them, optimises the innermost pairs first wherever nesting of these pairs occurs. Often the extra instructions needed are eliminated later by peep-hole optimisation, particularly when several locals are optimised in the same block.

After repeating the implementation of Intra-Block Scheduling, the method was applied to a range of test benchmarks, revealing that reductions ranging from 10% to 40% of all local references in a program file could be expected. This is illustrated in Figure 1.

however, the real objective is to eliminate dynamic instances of local references. Hence a simulation tool was developed to execute benchmark object code and measure dynamic effects. This indicated that the impact upon code execution was even more significant,

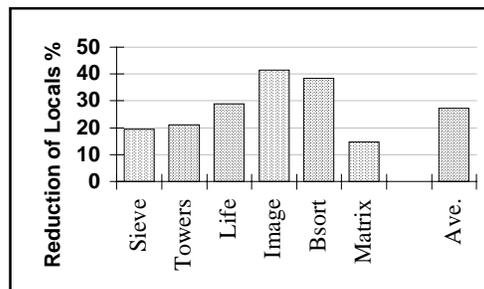


Fig.1 Reduction in locals - static code.

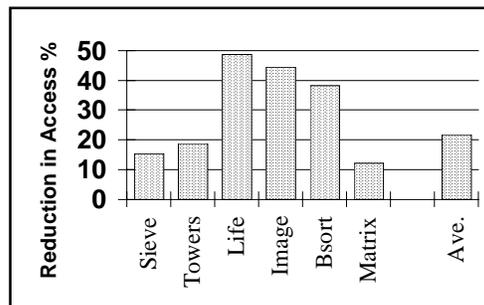


Fig.2 Reduction in locals - dynamic code.

with between 10% and 50% of local references being eliminated from running programs, as shown in Figure 2.

However, whilst Intra-block scheduling is clearly an effective optimisation, there is still significant remaining local-variable traffic to be eliminated. Recent work has shown that Koopman’s algorithm is near-optimal in its ability to eliminate locals [Maierhofer 97], thus further work in this area would yield minimal advantage. It is therefore necessary to extend beyond optimisation within basic-blocks, as will be detailed in the next section.

## 3. New Block-Boundary algorithm

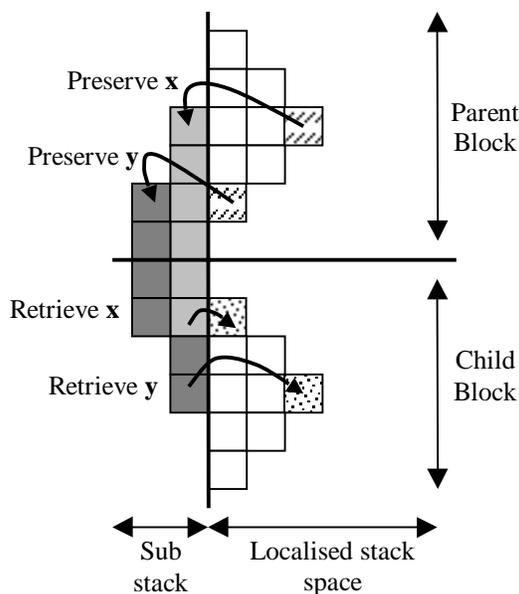
In order to overcome the limitations of intra-block scheduling, a new technique, ‘Inter-Boundary Scheduling’, was developed and tested. A full description is given in section 7.

The new technique is still limited in some respects, and does not provide a complete solution to redundant local variable referencing. However, the new technique is capable of optimising local variables across any parent-child block boundary, even when there are multiple parents and children involved in the relationship of the basic blocks. It is intended to be applied as an additional optimisation, not a replacement for intra-block scheduling.

The algorithm operates by identifying an 'inheritance context', a subset of local variables common to all the related blocks at a given boundary, which can be successfully placed or retrieved by all those related blocks.

The inheritance-context is used as a model to form a portion of stack space residing below the space utilised locally by the blocks themselves. This is suggested as the 'sub-stack inheritance-context' or 'SSIC'. The locally utilised stack-space is assumed to have the same stack-depth at the beginning and end of a block in the current algorithm implementation.

As a result, the SSIC can be used as a channel between the parent and child blocks, through which information can be passed. This is illustrated in Figure 3.



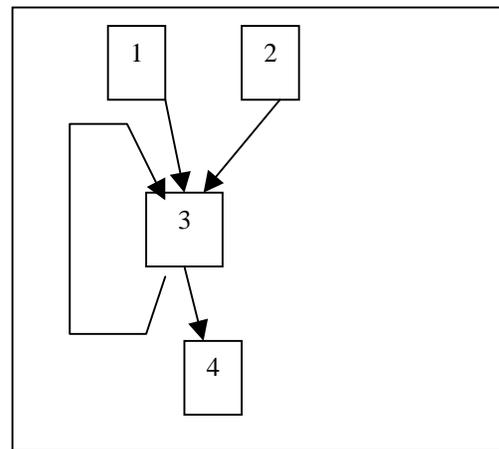
**Fig. 3, Illustration of sub-stack inheritance**

Once the SSIC has been formulated and verified, it may be used to validate the possibility of extracting and using its contents within each subject block, assuming tuck and dup type of operators to build the SSIC at the parent node, and rot and swap variants to extract its content in the child.

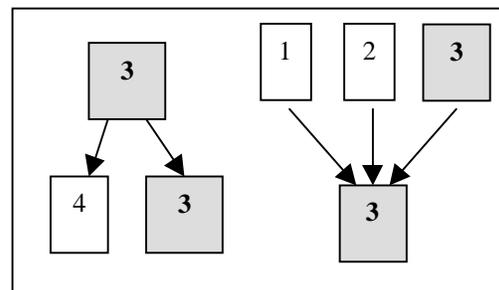
It is actually the case that the SSIC is somewhat similar to the approach taken for optimisation within basic blocks (intra-block scheduling). However, the novelty of the new approach is the fact that it can maintain this sub-stack across block boundaries even when multiple branch targets exist.

The availability of appropriate stack-management operators is an important factor in successful optimisation, as has already been demonstrated for intra-block-Scheduling [Bail97].

An orthogonal and scalable stack management scheme has been developed by the author, and provides an efficient model for support of this mechanism [Bailey 97]. This model is assumed for the remainder of this paper. One of the complications which the algorithm must consider, is the possibility that any block may have more than one parent and each parent may have more than one child. In some cases a child may loop back upon itself to give it the status of a parent of itself. Consider the example shown in Figure 4:-



**Fig. 4, Parent & child block relationships**



**Fig. 5, Boundary relationships**

We can see that block 3 has three possible parents (one of which is actually itself). It has two possible children (again, one of which is itself). In order for a variable to be inheritable, from block 1, by block 3, it must also be inheritable from block 2, and from block 3 itself. Because block-3 participates in this donor role, its child-block (block-4) must also be able to inherit the local, otherwise block 4 will receive a local it does not expect.

One can see that the issue of optimisation is rather complex. However, we can visualise two possible boundary relationships for block 3, (see Fig. 5) one as a parent, and one as a child.

Block 3 can be viewed in a child context, where it inherits variables from blocks 1,2, and 3, or as a parent context, where blocks 3 and 4 inherit variables from it. The sub-stack inheritance-context will only include variables that all parents can donate, and all child/siblings can successfully inherit.

As a result of this analysis, each block encountered when parsing the code on the second pass can be optimised for its parent and child status in turn, and written to the output stream.

Hence, wherever the same variables are utilised by all branch targets, then the repeated references to those local variables is eliminated. The technique should be particularly useful in optimising loop indices and so-on, and will thus be expected to yield performance speed-ups whenever memory-latency, or referencing, is a bottleneck.

When a call is encountered in a basic block, this presents an obstacle over which locals cannot be scheduled. However, the tail end of a block which is preceded by a call can still act as a parent if the stack depth after the call is established by back-tracking from the end of the block. Similarly, a child can inherit locals up to the point where a call occurs within its body.

## **4. Limitations and Variants.**

The algorithm presented here has limitations, in that it may only optimise a variable for re-use across a basic-block boundary at one level at a time. As a result of applying the algorithm to the whole program, block-by-block, a variable used in several sequentially entered basic-blocks will eventually be carried forward across multiple blocks. However when an intervening block does not use the variable then such 'incremental inheritance' of a local cannot occur.

### **4.2 Forced Inheritance**

There are also many cases where a parent block could donate a variable beneficially to one child, but not to another of its siblings. This may happen when a block references itself as a child, as in the case of loop constructs. Here the inheritance of a variable is

of high priority since its impact is amplified by the loop.

Forced inheritance could be used selectively in some cases, for instance loops may be identified and then all siblings which do not use the key locals in question will be forced to inherit them, but will discard them when reached by simply dropping appropriate stack cell contents. The penalty this leads to will be outweighed if the loop itself is repeated more than a few times. This was one of the observations made by Koopman in his original study using manual optimisation [Koop92]. In this paper we do not present results for this approach however.

### **4.3 Optimal Sub-stack Ordering**

The order of elements in the sub-stack inheritance context may well have some impact upon the successfulness of local variable elimination. This is because each block involved will preserve or retrieve sub-stack content in differing order to the next, so that one ordering might offer higher success rates than another in scheduling all of the subject variables.

An attempt has already been made to investigate this issue, by applying all possible permutations of the order of variables in the inheritance context in turn, and identifying the best case. This is not ideal, since the number of iterations increases rapidly for larger numbers of variables. With the limited benchmarks utilised in this preliminary study, no advantages were observed as a result of this method. However, advantages will only reasonably be observed if numbers of variables spill into the deep-stack areas (in this case assumed to be anything deeper than 4 cells), since it is then that ordering becomes critical. In this case of 'scheduling congestion' there may be a case for a more directed approach, selecting variables used heavily in loops for example, and ignoring other less critical ones.

## **5. Preliminary Results**

The effectiveness of the proposed algorithm was tested with a set of relatively simple benchmarks, partly based on the Stanford suite used by Koopman. At the time of the initially study, the work was being conducted with a rather limited compiler, but we now have access to a more robust platform, which will result in a more detailed study in the next stage of this research. The benchmarks chosen were as follows:-

**BSORT** – Bubble sort of 200 integers  
**ACKERMAN** – deeply nested call behaviour  
**LIFE** – Conway's Life simulation  
**IMAGE** – smooths a 20x20 pixel image  
**FACT** – calculates factorial of a number  
**MATRIX** – multiply two 2D matrices

Benchmarks were compiled from C-source code into object code targeted to a stack-processor model used in previous studies [Bailey 96 & 93].

Object code was first optimised with the existing intra-block scheduling algorithm, and simulated, then additionally optimised by the new method to eliminate further local variable references, before repeating simulations.

In each case peephole optimisation was applied to eliminate redundant stack manipulations before simulation. This is an important step, since scheduling techniques can create a lot of stack manipulations, most of which can be eliminated by peephole optimisation to reduce static and dynamic instruction counts.

Results for the two algorithms are presented as follows; Figure 6, shows reductions achieved for each algorithm, and the combined effect; Figure 7 shows the resulting execution time of benchmarks relative to the unoptimised case; and Figure 8 shows dynamic instruction count.

### **5.1 A Brief Note on the Presentation of Data**

Results presented here relate to the reduction in all local variable references, which is a more direct measure of how benchmark code will improve. It is important to appreciate the difference between this, and Phil Koopmans approach of quoting reductions in 'optimisable' references., which measures the effectiveness of the algorithm, rather than its contribution to performance improvement. Koopman found that his algorithm would remove 90 to 95% of all such 'optimisable' local references [Koop92].

### **5.2 Effectiveness of algorithms**

Studying Figure 6, it is apparent that the original intra-block scheduling algorithm, as proposed by Phil Koopman, was capable of eliminating an average of about 35% of all dynamic local variable references normally present during execution.

However, one can also see from Figure 6, that when Inter-Boundary scheduling is applied after locals within basic blocks have been optimised, the percolation of local variable

operands between parent and child blocks results in a further significant removal of Local variable references.

Benchmark results vary, but the average yield is approximately 25% removal of dynamic variable references left alone by the existing intra-block algorithm. It is rather interesting to note that in some cases, such as the factorial benchmark, the intra-block algorithm could offer no optimisation, since factorial is dominated by a single variable and short conditional blocks. However the inter-boundary scheduler is able to percolate operands across the conditional block boundaries and eliminate repeated references in some cases.

The opposite is the case for MATRIX, which has few blocks, most of which contain lengthy linear sequences. Here some intra-block optimisation is possible, but nothing significant is gained for inter-boundary scheduling. This implies that the two algorithms are complementary in their operation and advantages.

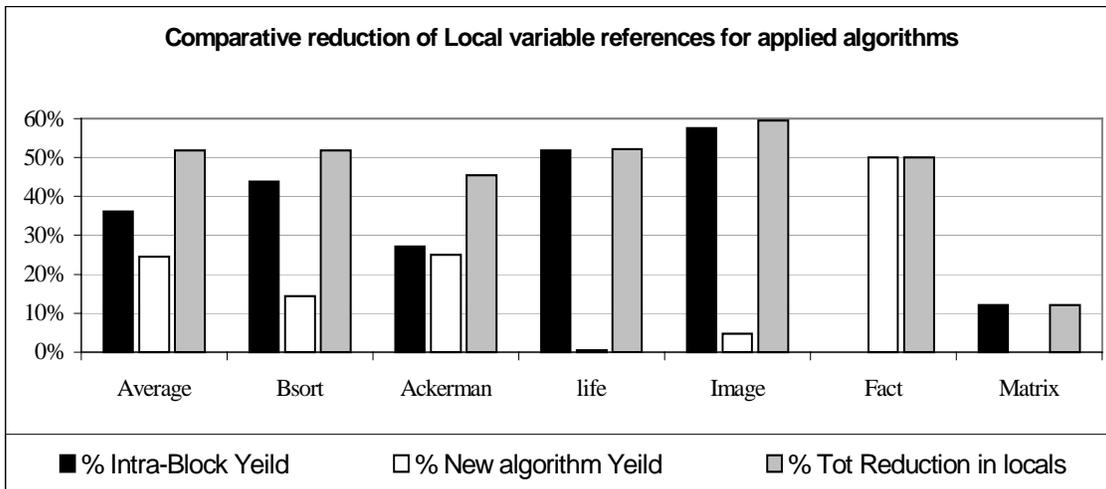
The combined case, where both of the two algorithms are employed, shows that an average of over 50% of local variable references can be eliminated by applying both optimisation algorithms, compared to the 35% for intra-block scheduling alone.

### **5.3 Code expansion & Instruction Counts**

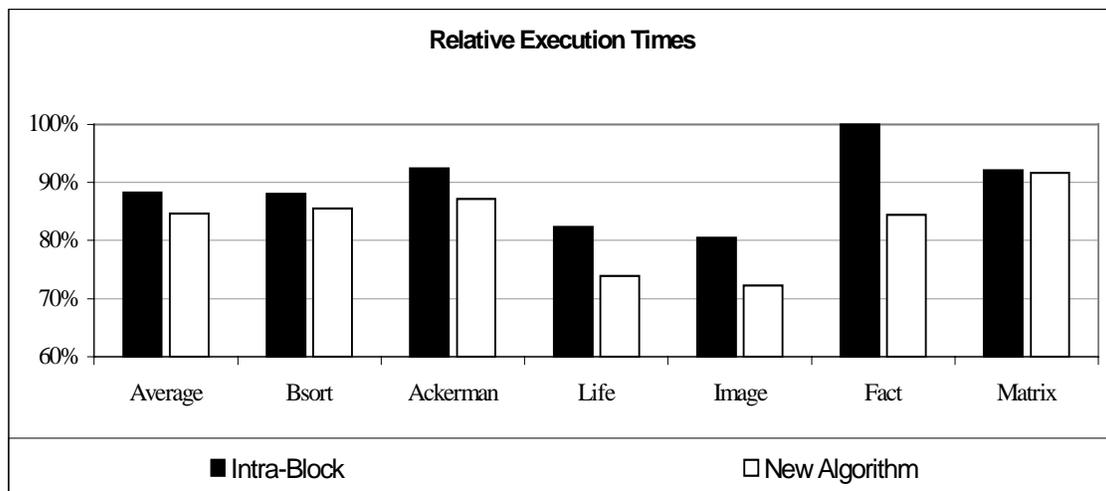
Figure 8, shows relative instruction counts for execution of the benchmarks. Interestingly, the impact upon instruction counts gives a reduction on average. Some programs suffer code expansion as a result of extra instructions needed to manage the stack content. However many programs obtain a reduction, which is explained by the fact that a several adjacent stack management operations can be peephole optimised to fewer, instructions, or even removed completely.

It is also notable that the application of Inter-boundary scheduling never causes code expansion for the benchmark set given here, and typically results in further reductions, again a result of enhanced peep-hole opportunities.

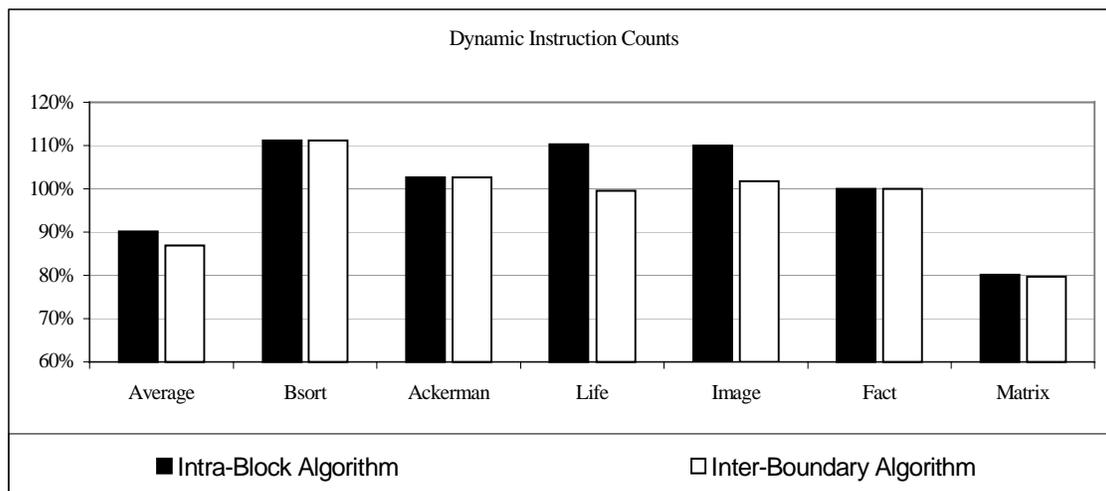
The ability to reduce instruction counts has additional importance. To date it has been assumed that local variable scheduling is only worthwhile where memory reference cost is high compared to that of an instruction.



**Fig. 6. Reduction in local references for each algorithm, and their combined effect**



**Fig 7. Execution times after optimisation with old and new algorithms**



**Fig. 8. Instruction counts, relative to unoptimised benchmarks**

However, even if locals are held on-chip and accessible in a single cycle, as in some recent Java architectures [Sun97, Sun99, Turley96], the ability to reduce overall instruction counts (by as much as 20% in one case) could still yield performance gains.

### **5.4 Execution Times**

Having considered the impact of the algorithms themselves, the more important figure of execution time should be examined.

Figure 7 shows the relative execution times of the same set of benchmarks where the execution time relative to the unoptimised case, representing 100%, is plotted.

Execution Speedups of as much as 15% are observed in some cases, with the average figure, inclusive of MATRIX, being only 5%. However it has already been noted that MATRIX has nothing significant to exploit for the new algorithm, and excluding this benchmark yields an average closer to 10% speedup.

Whilst the speedups associated with inter-boundary scheduling seem relatively small, these are improvements delivered on top of the existing improvement offered by the old algorithm, and raises the average speedup from 1.2 to 1.33 for the benchmark set excluding MATRIX.

One should also remember the preliminary nature of these results. A more complex implementation of the intra-boundary scheduler might include some of the enhancements discussed earlier, such as forced inheritance, and selectivity of variables optimised. This will undoubtedly result in more improvements, especially with a more suitable and robust test suite of benchmarks.

### **5.3 Possible trade-offs**

In the authors previous studies, which focussed on intra-block scheduling, it was found that optimisation of local variable references will alter the behaviour of the stack, and in turn, the behaviour of stack-buffers present in the system [Bailey 95a]. This indicated that slightly larger stack-buffers would be needed in systems where optimisations is applied.

Whilst it is only speculation at this stage, it seems likely that a further increase in active stack depth will be observed with the new

technique proposed in this paper, and subsequently the impact upon stack buffers will again be felt. A follow up study is intended by the author in the near future, to assess these effects.

## **6. Conclusions and further work**

The removal of local variable references from stack-based object code is an important step in overcoming bottlenecks often cited as disadvantageous for stack architectures.

There are wider implications for this work. There have been few well documented foray's into instruction-level parallelism in stack-organised architectures: The Transputer T9000 and SC32 architectures are both models which employ restricted ILP paradigms, however, [May et al; Hayes 89].

As any superscalar architect knows, data dependencies are a performance-killer. The elimination of local variables may not only be a method of reducing unwanted memory accesses, but may well prove to be an enabling medium for superscalar techniques to be applied in the next generation of stack architectures.

Gains reported so far are modest, but worth the effort, given the right arena. Particular interest might emerge from development of C-to-FORTH systems, where library code could be optimised after compilation in ways which would be inconsistent and laborious if done by hand.

In real-time systems, squeezing more performance out of a system is often a boon to the engineer with hard-pressed timing margins, but there are cautions to be observed [Bailey 2000].

A more in depth study will be the next stage for this research, using better compiler and benchmark suites, and covering aspects such as buffer behaviour and instruction set complexity, as reflected in earlier studies by the author [Bailey 97].

With further refinement, the intra-boundary scheduling algorithm shows significant promise for the elimination of a major performance bottleneck, and one which has to date allows stack-processors to be rated as inferior to its register based relatives.

## 7. Description of the algorithm

### FIRST PASS:

- 1 Parse each block in input stream in turn and record the following :-
  - 1.1 The position of each local reference
  - 1.2 The stack depth at each position
  - 1.3 Whether it is a load or store
  - 1.4 All branch targets (Child-Blocks)

Each block record should have a duplicate, which will not be altered in the second pass.

### SECOND PASS

As each block is parsed from input stream :-

## **2 BOUNDARY MAPPING**

- 2.1 Build a list of all parents
- 2.2 Build a list of all siblings
- 2.3 Build a list of all children
- 2.4 Build a list of all co-parents

*\* See note 1 following the algorithm.*

## **3 CHILD SCHEDULING**

- 3.1 Build a list of locals common to all child blocks, and co-parents.
- 3.2 Build Sub-stack Inheritance context  
A list of common locals in order of appearance in the subject block.
- 3.3 Test each co-parent to see if it can create the same SSIC.
- 3.4 Test each child to see if it can retrieve locals from the SSIC.  
  
*\* see note 2 at the end.*
- 3.5 Any local failing steps 3.3 and 3.4 is removed from the common locals list and analysis restarts at 3.3
- 3.6 When a valid SSIC is available, each reference to a common local in the subject block is followed by an instruction pushing a copy into the sub-stack area at the required depth.
- 3.7 Each inserted instruction causes an update of the modifiable block record, but not the duplicate record. The update modifies instruction position and stack depth preceding an inheritance point.

## **4 PARENT SCHEDULING**

- 4.1 Build a list of locals common to all parent blocks, and all siblings.
- 4.2 Build Sub-stack Inheritance context  
A list of common locals in order of appearance in the subject block.
- 4.3 Test each parent to see if it can create the same SSIC.
- 4.4 Test each sibling to see if it can retrieve locals from the SSIC.
- 4.5 Any local failing steps 3.3 and 3.4 is removed from the common locals list and analysis restarts at 3.3
- 4.6 Each reference to a common local in the subject block is replaced by an instruction which retrieves (moves) the associated SSIC operand to the top of stack, and following instructions have position & depth records altered.

### NOTES

1. A subject block will have at least one parent. Where there are more than one parent, each parent may have other children, which are classed as the siblings. The donation of a local from the subjects parents must also be inheritable by all of their children (i.e. the siblings). The subject is classed as a sibling.

A subject block will have one or more children. The children may have other parents (the co-parents). When a subject block donates a local to its children, this local must be inheritable by all children of the co-parents, and capable of donation by all co-parents.

2. Validating the ability to donate or inherit a local is achieved by emulating the sub-stack as it is constructed I the parent or deconstructed by the child.

A local is able to be donated when the current stack depth plus the current sub-stack depth is within reach of an operation that can copy the top of stack cell into the sub-stack.

A local is inheritable if its sub-stack depth, plus the depth of the blocks local stack space is within reach of an operation that can move it to the top of stack.

## **8. REFERENCES**

[**Amdahl 64**] Amdahl, G, Blaauw, G., Brooks, F. P., Bailey, C., Sotudeh, R., (1995). Architecture of the IBM System/360, IBM Journal, Apr. 1964, p87-101.

[**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 1995a**] Bailey, C., Sotudeh, R. *The Effects of Intra-Block Scheduling in a Stack Processor Environment*. Proc. Rochester Forth Conference, Rochester, USA, June 1995.

[**Bailey97**] Bailey, C. "Instruction complexity and implicit execution architectures: orthogonality, optimization, and VLSI-Design", Proc. Of the 11<sup>th</sup> ICMCM, Washington DC, USA, 1997.

[**Bailey2000**] Bailey, C., "Achieving minimal and deterministic interrupt execution in stack-based processor architectures" – Euromicro Digital Systems Design 2000, Maastricht, Sep 5th-7<sup>th</sup>, 2000.

[**Bell70**] Bell, G, Cady, R., McFarland, F., Delagi, B., O'Laughlin, J., Noonan, R., and Wulf, W. (1970), A new Architecture for minicomputers – The DEC PDP-11, Proc. Of AFIPS , p657-675.

[**Koop92**] Koopman, P. "A preliminary exploration of optimised stack code generation, Proc. RFC '92

[**Hayes 1989**] Hayes, J., R., and Lee, S., C., *The architecture of the SC32 Forth Engine*. Journal of FORTH applications and research.

[**May et al**] May, D., Shepherd, R., Thompson, P. "The T9000 Transputer", INMOS Ltd (Now SGS Thompson).

[**Maierhofer97**] Maierhofer, M., Ertl, A., "Optimizing Stack Code", Forth-Tagung, 1997, Ludwigshafen.

[**Bailey 1996**] Bailey, C. *Optimisation techniques for stack-based architectures*. Ph.D. Thesis, July 1996, University of Teesside, Middlesbrough, UK. Online <http://www-users.cs.york.ac.uk/~chrisb/>

[**Sun97**] PicoJava Processor Datasheet, Sun Microsystems Inc. Dec, 1997. At internet URL:<http://solutions.sun.com/embedded/databook/pdf/datasheets/805-2990-01.pdf>

[**Sun99**] PicoJava II Processor Datasheet, Sun Microsystems Inc. Dec, 1997. At internet URL <http://solutions.sun.com/embedded/databook/pdf/datasheets/805-4634-02.pdf>

[**Turley96**] Turley, J. "New Embedded CPU Goes Shboom", Microprocessor Report, Vol 10, No. 5, April 15, 1996.