An experimental Investigation of Single and Multiple Issue ILP speedup for stack-based code

Chris Bailey
Dept. of Computer Science,
University of York **,**
chrisb@cs.york.ac.uk

Mike Weeks [*]
Dept. of Computer Science,
University of York,
w.weeks@c.york.ac.uk

[*] *Visiting RA, Part-funded by University of Teesside*

1 Abstract

Stack-processors, which abandon register files and instead work directly on stack-resident operands, have recently enjoyed a resurgence of interest in conjunction with developments such as Java, and have always retained an interest for FORTH users, especially in real-time systems arenas. However, the opportunity to enhance throughput of stack-based architectures has received insignificant attention in recent years. Java, and the recent interest in low power architectures has something to owe to this.

In this paper we provide an insight into an ongoing preliminary study of ILP parallelism in stack architectures, our approach to the problem, and some preliminary results from our work. Whilst this is very much a case of work in progress, we feel that there are clear opportunities for further work in this area, and substantial gains for the stack-processor paradigm.

Our results show that even in the absence of the simplest approaches to code optimisation for parallelism, speed-ups can reach into the order of 2-fold speed-up. This should be applicable to FORTH as well as C and Java, since it is generic to the underlying machine architecture.

1. Introduction

Whilst stack architectures have enjoyed a recent increase in interest, largely due to the rise of Java and proclamations a of set-top-box revolution, they have failed to adopt any of the significant advances in mainstream register-based architectures that have become viable for microprocessors in the past 10 to 20 years.

Whilst there will always be an interest in stack-architectures from the real-time-systems and FORTH community, the application of advanced stack processor architectures to more demanding applications has yet to be realised.

There have been some interesting approaches to parallelism which attempt to execute multiple instructions per cycle. The SC32 processor employs a long-instruction-word format which allows multiple internal operations to be executed simultaneously. It is claimed that this allows ‘optimisation of up to seven Forth words into one [memory] word is possible’, implying up-to 7 instructions per cycle at peak execution. However typical code will average far less, perhaps only one or two instructions per cycle.

The ill-fated T9000 experiment with parallelism took a different approach, based upon a dedicated multi-function pipeline. Here instructions were pre-ordered by the compiler to comprise groups of instructions which would (ideally) match the pipeline configuration of the architecture, and then achieve a speedup.

However, both of these approaches assume a single ALU model, and rely upon contriving to get instruction sequences in the right order, where often the basic-blocks involved are too short to fully exploit available parallelism.

Research being conducted at the University of York, in the Department of Computer Science, is focussed upon the improvement of stack-processor architectures in novel ways. Our most recent activity has centred upon a short-term project to investigate the feasibility of applying super-scalar techniques to an implicitly addressed stack architecture.

The investigation has been achieved by development of a simulator platform, written in C++, which emulates a super-scalar stack execution model. Our simulator is capable of simulating both single and multiple issue models of ILP parallelism, and has yielded important results which not only confirm the viability of the concept of super-scalar stack execution, but suggest areas where code optimisation techniques may well deliver greater gains.

2. Superscalar issue and execution

An approach to parallelism typically employed in register-based processors is to employ multiple and independent functional units, and to co-ordinate their operation through the use of reservation stations [Tomasulo 67] and score-boarding [Thornton 64].

The concept is simple enough : an instruction requires operands, which are drawn from registers, computed upon, and written back to the register file. However, it is often the case that an operand is not yet available in the source register, hence the requirement for that register's content is reserved by the functional unit and acquired when available. Consequently, the result of the operation cannot be written immediately back to the destination register, and this must be recorded in the score-board, so that other functional units are aware that access to that register must be delayed (and acquired through reservation).

So far this is a description of a register-based architecture. However we can read stack-cell for register in the above description quite happily, and thus conceive of a stack-based embodiment of the Thornton/Tomasulo approach to parallel computation.

This must be refined by the addition of a special functional unit, the SMU, or stack-manipulation Unit. Which is responsible for performing operations such as swap, rot, dup, and drop. An interesting observation in this respect is that stack manipulations do not alter content, rather they move operands from place to place, and as such they may operate upon stack contents which are not yet known due to an earlier delay in computation.

Consider for example, the case of ADD followed by SWAP. In a normal architecture the swap cannot proceed until ADD completes. However in our embodiment, the swap acts only upon the score-board and not the actual stack-cells, and so it achieves a transposition of the stack content without the need to have the actual values to hand.

There are of course a number of other cases where execution can proceed in this manner. An addition which is independent of previous operations can be issued and executed even if the preceding operations are stalled due to a memory bottleneck or data dependency.

3. A model for Stack-Based ILP

We envisage a stack processor paradigm in which a multiplicity of ALU's exist, along with a load-store unit, and an SMU (as discussed earlier). This can be augmented by branch unit and so-on, to create more parallelism in a future expanded design. A conceptual model of the design is given below in Fig.1.

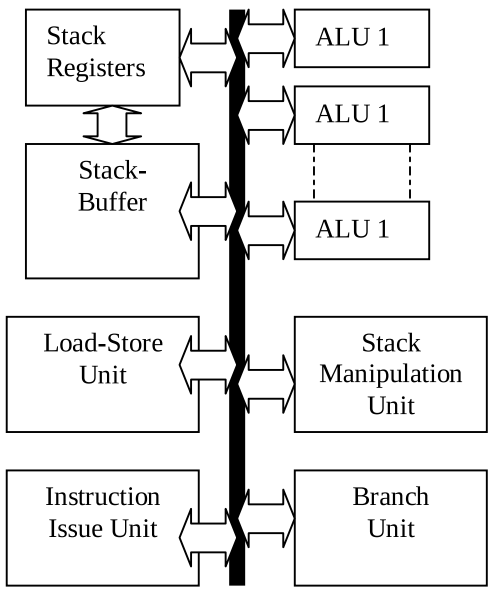

Figure 1, A model for ILP stack architectures

This model has been utilised to prepare some preliminary data in order to investigate the opportunities for ILP speed-up for stack-based architectures.

The basic concept of ILP parallelism in a stack context can be illustrated in the example shown in Figure 2, where the computation of an expression "A+B * C+D" takes place. Here it is assumed that at least 2 ALUs (4 cycles), and 2 multipliers (6 cyles) are available.

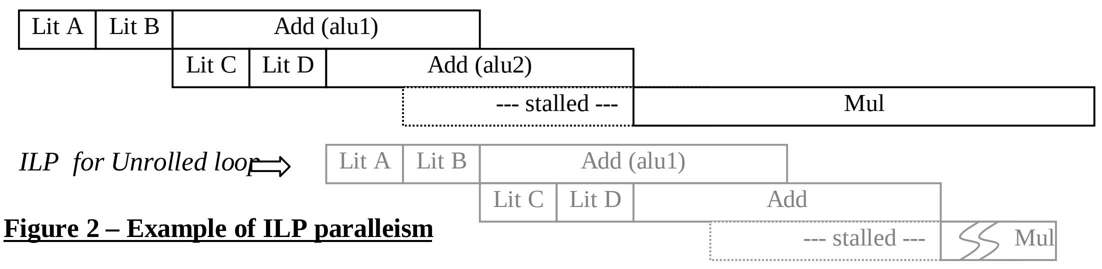

Figure 2 – Example of ILP paralleism

Figure 2 illustrates the example of a computation which contains inherent parallelism, the addition of A plus B is independent of the addition of C plus D, and can therefore be executed in parallel if resources and mechanisms permit. Here it has been assumed that a limited multiple instruction-issue per-cycle mechanism is available, a point we will return to later.

It can be seen that for a single iteration of the calculation the total number of cycles, if executed serially, would be 18 cycles, whereas the case illustrated assuming an ILP model, results in an execution time of only 14 cycles.

If the expression is to be repetitively applied to a data set, then the opportunity to unroll the iterative loop is one which should be exploited. The greyed out section illustrates a second strand of the loop being executed in parallel with the first, yielding 2 iterations in only 18 cycles, or an average of 9 cycles per iteration.

Translating these figures into execution speed-up, a familiar measure of ILP performance, suggests that the single iteration case yields a speed-up of 1.29, and the case for a one-level unrolling of the imagined loop yields a speed-up of 2.00.

Remember that here there has been no attempt to do anything special to the code to enhance its parallelism. It is also assumed that the operands (all literals) are available in one cycle, which may not be the case if memory resident local variables were used instead.

Exact speed-up figures also depend on the relative speed of data references such as LIT and locals, compare with the ALU latency. However, with moves toward 32 and 64 bit computing seeming increasingly likely for future stack architectures it seems reasonable to assume that ALU latencies will be relatively large compared to on-chip data cache, for instance.

4.The Simulator

This section describes the methodology used to simulate the super-scalar stack processor architecture. The simulator evaluates both the sequential and parallel instruction timings. The sequential model consists of an instruction fetch engine that feeds a pending instruction queue and an issue/execute unit that pulls opcodes from the queue before executing them. The instruction must complete before the next pending instruction may be operated upon.

The assembler program code passed to the simulator is in the form of a text file which is converted to an internal symbolic representation for simulation purposes.

The following sections describe the model and simulator implementation used for our studies.

4.1 Instruction Fetch Engine

Retrieves instruction words from memory, breaks them down into individual instructions (if packed instructions are used) and places them onto a ‘‘pending instruction queue”. Since a single memory fetch can contain several instructions, the fetch queue is kept well ahead of the issue unit. When an issued instruction causes a change in the program counter, the following instructions in the pending instruction queue become invalid and therefore must be flushed. The simulation enables the queue depth to be altered, so that we can study its effect on performance and instruction fetch traffic.

4.2 Issue Engine

The issue unit perfoms two major tasks. At the beginning of each clock cycle, the issue unit interrogates the tag scoreboards of the data stack and other registers, and those of the functional units. Any modifications to the scoreboard tags are then updated. The issuing unit then pulls instructions from the pending instruction queue and distributes them to their respective functional units. If a functional unit is unavailable the issuing unit must stall the instruction. The issue unit is also responsible for the execution of branch and call operations.

4.3 Arithmetic & Logic Unit (ALU)

An ALU management unit allows the issuing unit to utilise multiple ALU’s in a scalable (up to 8 ALU’s maximum) and seamless manner. The issuing engine passes all ALU opcodes and operands to this intermediary unit, which in turn passes it to any available idle ALU, otherwise the issuing engine is stalled.

Each ALU utilises register score-boarding and reservation tables. Upon receiving an instruction, the unit places the instructions into a reservation table and updates the scoreboard stack. When the required input data is valid, the ALU places the result of the operation in its output register and sets its status. On the next clock cycle this will be interrogated by the Issue unit and its destination updated with the result.

4.4 Load/Store Unit (LSU)

The load/store unit must allow instructions to be issued on consecutive cycles irrespective of whether it is ready to execute them. Inclusion of a reservation station queue allows instructions to be queued until they are ready to be executed. The size of the queue is configurable for simulation purposes.

4.5 Stack Manipulation Unit (SMU)

The function of the SMU is to manipulate the position of the cells in the data stack, for such operations as dup, swap, rot, drop and so-on. These operations do not act directly upon stack data, but act upon the score-board, in order to signal the new required content of each stack register without needing to have access to, or to wait for, the data itself.

5. Simulation Models

In our work to date we have assumed two models of instruction issue. Single issue allows a new instruction to be issued on every new clock cycle, whilst the multiple-issue model allows a number of instructions to be issued per cycle.

In each model we have assumed that memory accesses and instruction fetches all achieve a 100% cache hit ratio, with a split cache (Program/Data) since the aim here is to measure efficiency of ILP execution, rather than interactions with non-ideal memory configurations.

A third and final consideration is the timing and latency of functional units. The table below, Table-1, gives the assumed timings.

Event	Cycles
Int ALU	4
Mult/Div	8
Load/Store	1
Stck manip.	1
Logicial op.	1

Table-1, Latency model used for study

Data was produced for a small set of test programs, including the familiar *Bsort* benchmark and a recursive n-factorial computation. To further test the capabilities of an ILP mechanism, we developed a simple benchmark '*Pcalc1*' which performs a calculation on array contents in such a way as to create some parallelism within each loop. The *Pcalc* program computes as follows :-

A[n] = B[n] . (C[n]+D[n]) (n=0 to 10)

This was expanded upon by the use of *Pcalc2* which implemented a single-level unrolling of the critical loop of *Pcalc1*, in an attempt to extract more parallelism.

No optimisation of the code to maximise opportunities for parallelism was attempted. It is worth noting that overwhelming effort in the generation of 'optimal' stack code has focussed upon minimising stack depth [Bruno 75], which inherently serialises the code structure. Therefore our results are more representative of worst-cases rather than best, or typical expectations.

6. Preliminary Results

Figure 3 shows the speedup obtained when our raw compiler-generated code was simulated on the ILP simulator , assuming ALU latency of 4 cycles, (8 for multiply), and 1 cycle for other operations.

It is clear that some speedup is obtainable through the use of a general single-issue per cycle ILP policy, and although fairly modest gains are seen, they are fairly significant given the lack of code optimisation before execution.

Individual results show some not unexpected effects. Factorial is dominated by short recursive instruction sequences, with minimal computation. It is not surprising that there seems to be little parallelism achieved here. Conversely, *Pcalc1* and *Pcalc2* show higher speed-up, which the unrolled loop of *Pcalc2* delivering a small improvement.

In Register-based machines it is well know that memory and data dependencies are a major cause of poor ILP performance, and in stack architectures this is equally applicable. We believe we have uncovered some useful findings relating to this issue.

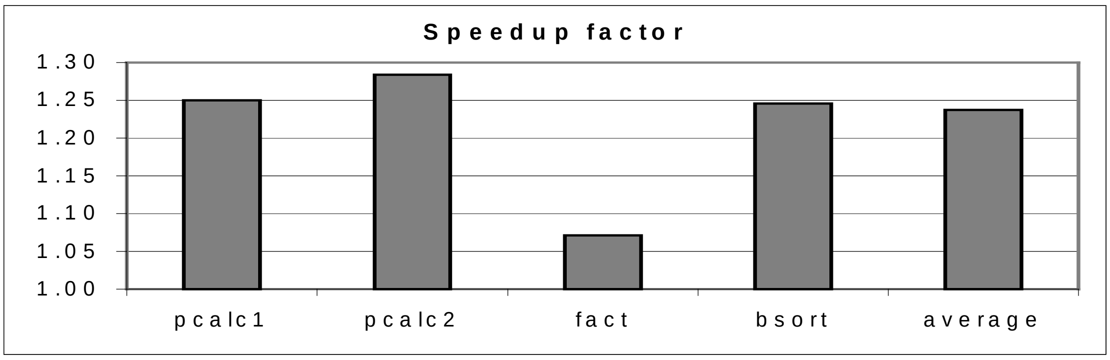

Figure 3, Speedup for Raw Code

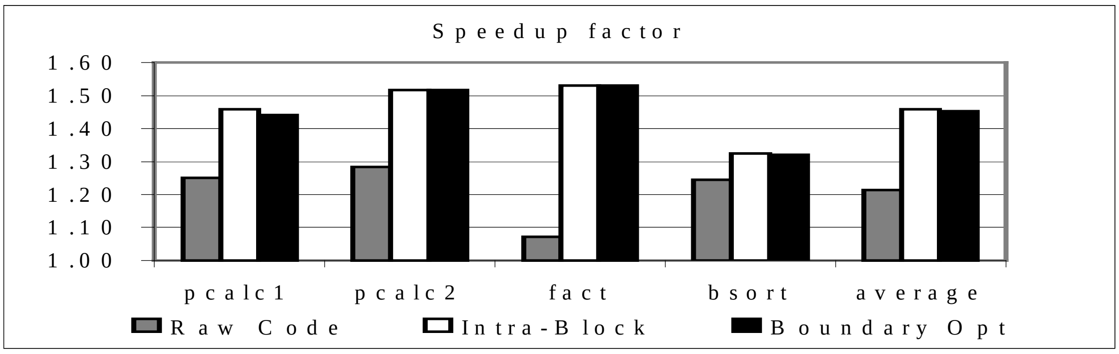

Figure 4, Speedup after Optimisations

6.1 Local Variable Optimisation vs. ILP

By applying techniques such as Koopman's Intra-Block Scheduling, and the authors newer Inter-boundary scheduling algorithm, it has proven possible to significantly reduce these dependencies as far as memory referencing is concerned. This impact of this existing optimisation is quite clear when one examines the second set of results (figure 4), which show that the application of Intra-block scheduling actually increases the ILP speed-up obtainable, by virtue of removing load-store blockages.

Examination of Figure 4 suggests that the speed-up obtained for single-issue ILP execution is roughly doubled by the application of these optimisation strategies alone. However it is also apparent that the Inter-Boundary algorithm adds little, and even detracts from the obtainable speed-up. Without more substantial studies we cannot be certain that it is not a useful addition however.

This is quite an important finding, which suggests that although architectures such as PicoJava and PSC1000 [Case 96, Turley 96] may not gain anything from direct application of local variable scheduling (since locals are accessed on chip in a single cycle), they may well benefit if future versions of these architectures seek to adopt some form of ILP parallelism.

6.1 Multiple Issue Experiments

At present, our study is somewhat preliminary, and multiple-issue execution was considered beyond the scope of the study. However it appears that single-issue ILP will never achieve substantial speed-ups. It has been possible to derive some hand-calculated speed-up figures for a multiple issue model however.

For example, the *Pcalc1* routine takes 58 cycles for each iteration of the critical loop in a serial execution architecture. With a single issue per clock ILP policy, a speed-up of 1.38 is possible, based on the same assumptions as our earlier whole-benchmark figures.

However, if a model is adopted where multiple instructions can be issued in a single clock cycle, then a speed-up of 1.87 seems obtainable, according to our manual calculations, even without the benefit of compiler technology to exploit this mechanism properly. Our calculation assumed that each clock cycle consists for 4 micro-cycles, with a new instruction issue for each micro-cycle (effectively a super-pipelined issue policy).

Also, we can return to the example of Figure 2, where the assumption is that ALU operations can be issues in parallel with some other operations (such as Lit). As already noted, this example achieves a speed-up of 2.00 by using a limited super-scalar issue policy and loop-unrolling.

With a genuine effort to devise new optimisation strategies for stack-based code, it may well prove possible to generate stack code with higher degrees of available parallelism in the future, and thus seek to design architectures to exploit this.

7. Conclusions

The field of applications for stack based computation encompasses an ever growing area, including FORTH and JAVA. Future systems may demand higher throughput, and potentially with lower power consumption. A super-scalar stack architecture may just be able to deliver the high throughput these systems demand whilst keeping silicon and power budgets under control.

In this study we have presented some very preliminary ‘work in progress’, and as such the results are not wholly convincing in terms of raising the stature of ILP stack architectures against the well-developed alternatives of super-scalar register-file machines.

However, the study has shown that there is clearly a capability within stack architectures to exploit some Instruction-Level Parallelism (ILP), and with appropriate and carefully designed compiler optimisations, there is potential to greatly enhance that capacity. Such techniques have yet to be developed, but counterparts in the register-file domain exist and have been proven valuable in the overall goal of higher degrees of parallelism.

The specifics of an ILP super-scalar or ILP capable stack architecture are also areas where much work can be done, and where little established knowledge has been employed. The choice of single issue, super-pipelined issue, or true super-scalar issue of instructions will have a significant impact upon achievable speed-ups, but not without comprehensive support from code optimisers yet to be developed.

Future work in this area would be both interesting, and potentially important, for the future of silicon based stack architectures. We have already begun work on a new simulator to explore super-scalar schemes and hope to be able to report more significant findings at a future conference.

8. References

[Tomasulo 67] Tomasulo, R., M., 1967, “An efficient Algorithm for exploiting multiple arithmetic units”, IBM Journal of Research & Development, Vol 11, No1, p 343.

[Thornton 64] Thornton, J., E., 1964, “Parallel operation in Control Data 6600”, Proc. of AFIPS Fall Joint Computer Conference, Vol 2, Page 33-40.

[Bruno 75] Bruno, J.,L., Lasagne, T., 1975, “The generation of Optimal Code for Stack Machines”, Journal of ACM, Vol 22, No. 3, P 382-396.

[Case 96] Case, B., 1996, “Implementing the Java Virtual Machine”, Microprocessor Report, March 25, 1996,p 12-17.

[Turley 96] Turley, J., 1996, “New Embedded CPU goes Shboom”, Microprocessor Report, April 17, 1996, pages 1, and 6-10.