STATIC DATA DEPENDENCE COLLAPSING IN A HIGH-PERFORMANCE SUPERSCALAR PROCESSOR

Fleur L Steven, Richard D Potter, Gordon B Steven
University of Hertfordshire
Hatfield, Hertfordshire, UK
email: comqgbs@herts.ac.uk.

Lucian Vintan
University of Sibiu
Sibiu, Romania
email: vintan@cs.sibiu.ro

Abstract

We present a technique for ameliorating the detrimental impact of the true data dependencies that ultimately limit the performance of a superscalar processor. This technique, termed instruction combining or static data dependence collapsing, assembles instruction pairs into a single entity that can be executed at run time as a single operation on a complex functional unit. An ideal opportunity for assembling instructions in this manner occurs during instruction scheduling when code is reordered into instruction groups that can be executed in parallel. Results obtained using a trace driven simulator show that instruction combining can increase the available speedup by 40% over a Baseline Model that is limited to single instruction issue. Further results suggest that integrating combining with other aggressive instruction scheduling optimisations can unlock as much as two orders of magnitude of potential speedup. We therefore conclude that instruction combining is a powerful mechanism for realising additional parallelism in a high-performance superscalar architecture.

Key Words

instruction combining, instruction scheduling, superscalar

1. Introduction

The performance of a superscalar processor is ultimately limited by three major elements: memory bandwidth, control hazards and data dependencies. Memory bandwidth limitations can be addressed through the development of high-performance cache hierarchies [1], while the impact of control hazards can be reduced through a combination of branch prediction [2] and global instruction scheduling [3]. This paper examines the third major limitation on performance, data dependencies. In particular, we examine the use of instruction combining to reduce the impact of true data dependencies on instruction-level parallelism.

In essence, instruction combining involves collapsing a pair of instructions when the second instruction is directly dependent on the first. The two instructions are then treated as a single entity and are executed together in a single functional unit. Performance can be improved as long as the time taken to execute the two instructions in a single functional unit is less than twice the time taken in two separate functional units. An examination of two ADD instructions shows just how this improvement can be achieved:

```
ADD  R4, R5, R6   /* R4 = R5 + R6 */
ADD  R8, R4, R7   /* R8 = R4 + R7 */
```

becomes

```
ADD R8, (R5 + R6), R7
```

As long as the two ADDs remain separate, the first ADD will be executed in one cycle and the result, R4, bypassed to the second ADD that will be executed in a second cycle. If the two additions are combined, the three operands can be added using a single functional unit that has been optimised to add three rather than two numbers. A Carry Save Adder, widely used in multiplier designs, followed by a normal Carry Propagate Adder will perform the required additions. The Carry Save Adder adds one complex gate delay ahead of the ALU. Instead of doubling the delay through the functional unit, the total execution time is therefore marginally increased. In many superscalar designs where timing is dominated by cache access times and bus delays, we feel that the impact on the overall processor cycle time will be negligible.

The primary motivation for introducing combining is to reduce the execution time. However, in the above example, it can be seen that the register R4 may no longer

be required. Combining instructions can therefore also reduce the pressure on register utilisation.

At the University of Hertfordshire, we have developed the Hatfield Superscalar Architecture (HSA) [4], a high-performance superscalar processor model, to allow us to investigate the performance limits of compile-time instruction scheduling. Our long term objective is to achieve an order of magnitude performance improvement over traditional RISC implementations that issue no more than one instruction in each processor cycle. Unlike many other superscalar projects, the processor architecture minimises hardware complexity by specifying in-order instruction issue. Instead heavy reliance is placed on the instruction scheduler to reorder the original sequential code into instruction groups that can be issued in parallel at run-time without requiring out-of-order instruction issue. The instruction scheduler will therefore be the primary vehicle for achieving the high performance we desire. Other important features of HSA include the guarded execution of all instructions [5], hardware support for speculative instruction execution and a generalised delayed branch mechanism [6,7]. A suite of software, including an HSA gnu 'C' compiler, an HSA instruction-level simulator, an HSA instruction scheduler and a trace driven simulator, has been written to support this research.

2. Related Work

All instruction schedulers already use a simplified form of instruction combining. For example, consider the following instruction fragment. (Note that assignment order is used in all the examples in this paper.)

```
ADD  R6, R7, R8       /* R6 = R7 + R8 */
MULT R10, R6, #10
```

Suppose that, in order to percolate or move the ADD instruction into an earlier instruction group, we have to remove an anti-dependence on the register R6. This is achieved by returning the result of the ADD to a new register R6', and by adding a new MOV instruction to restore the value in R6' to R6:

```
ADD R6', R7, R8
  :
MOV R6, R6'
MULT R10, R6, #10
```

This renaming operation will only reduce the program execution time if we can also later combine the new MOV instruction with the following MULT instruction.

```
ADD R6', R7, R8
  :
MULT R10, R6', #10
```

In general, if two instructions are combined, the resultant instruction will require a total of three source operands. In the above simple case, however, only two source operands were required. The IBM VLIW team led by Kemal Ebcioglu [8] identified all those cases, like the one above, where two instructions can be collapsed to form a single instruction that still requires only two operands. In these cases traditional functional units can still be used, and no additional hardware functionality is required to support instruction combining. Apart from the obvious cases where either the first or second instruction is a MOV, many additional pairs of instructions can be successfully collapsed as long as both instructions include an immediate operand. For example, consider the following two add immediate instructions:

```
ADD R1, R2, #4
ADD R3, R1, #4
```

They can be collapsed to give:

```
ADD R3, R2, #8
```

The first instruction must also be retained whenever the intermediate result is still live after the second instruction. Nevertheless, the total execution time through the code is potentially reduced by the latency of an ADD instruction.

The IBM VLIW group found that combining was particularly useful when a loop count was adjusted and then tested. In this case combining is used to reorder two operations as in the following example:

```
ADD R1, R1, #1        /* R1 = R1 + 1 */
EQ B1, R1, #COUNT
        /* set flag B1 if R1 = COUNT else reset */
```

Using combining, the fragment can be transformed as follows:

```
EQ B1, R1, #COUNT - 1
ADD R1, R1, #1
```

Although the number of instructions is unchanged, a true data dependence has been removed, reducing the delay through the code and allowing any subsequent conditional branch instruction that tests B1 to execute earlier.

The IBM team implemented combining on the Enhanced Percolation Scheduler developed for their VLIW architecture [8] and achieved an additional 10% speedup with the Stanford integer benchmarks and other 'C' programs.

A second IBM group, that included Stamatis Vassiliadis, further developed the idea of combining as part of the SCISM project [9]. The SCISM team investigated the idea of collapsing dependent instructions dynamically at run time. They also removed the restriction that combined instructions must collapse to form a single instruction with no more than two operands.

In the SCISM approach, adjacent instructions are collapsed dynamically during the instruction issue stage.

This hardware process is called Interlock Collapsing. The two instructions are then issued in parallel to a single three-input functional unit, called an Interlock Collapsing ALU (ICALU). At the same time the first instruction in the pair must also be issued to a conventional ALU to ensure that its result is still available to subsequent instructions. A major contribution of the SCISM team was to demonstrate that an ICALU could be designed to handle a comprehensive range of combined arithmetic and logical instructions without significantly increasing the processor cycle time [10].

To evaluate the benefits of ICALU's, the SCISM team carried out a series of simulations using the SPEC benchmarks [11]. Although interlock collapsing was restricted to adjacent integer arithmetic and logical instructions, parallelism was increased by between 3% and 19%. We feel that these results are extremely encouraging, particularly since no attempt was made to schedule instructions at compile time to enable more instructions to combine.

More recently, Stamatis Vassiliadis, now based at Delft University and collaborating with Yiannahis Sazeidas and James Smith at the University of Wisconsin-Madison, has extended the SCISM work in a number of important respects [12]. First, Interlock Collapsing is no longer restricted to instructions that are adjacent in the dynamic instruction stream. Non-adjacent instructions within the processor's instruction window can now be collapsed and forwarded to a single ICALU for execution. Second, Interlock Collapsing is extended to include chains of three dependent instructions. Finally, shift instructions are also combined, since the commonest shift functions can be easily added to an ICALU.

With these relaxed restrictions, simulation figures suggest that between 29% and 47% of all instructions can be collapsed. Significantly, the number of instructions collapsed increases steadily as a function of the size of the instruction window. These results support the belief that there is considerable scope for combining non-adjacent instructions. Unfortunately, however, it is difficult to see how non-adjacent instructions can be collapsed at run time without placing significant pressure on the cycle time of the instruction issue logic. Since this part of the pipeline is already under considerable pressure from increasing instruction issue rates and window sizes, this additional pressure on timing constitutes a major challenge for those wishing to implement dynamic out-of-order instruction collapsing.

In contrast, the same results encourage our belief that significant additional parallelism can be extracted by implementing instruction combining as an integral part of a static instruction scheduler. The main task of such a scheduler is to assemble groups of instructions that can be issued and then executed in parallel at run time. An instruction scheduler is therefore ideally equipped to move combinable instructions into adjacent instruction slots before combining them within a single parallel instruction group.

While most of the Interlock Collapsing (65-82%) detected in Vassiliadis' simulations was contributed by collapsing instruction pairs, dependent groups of three instructions accounted for 13-30% and zero value operand detection for 5-10%. Detecting operands that are zero enables certain chains of four instructions to be collapsed and executed in functional units with only four source operands. Ebcioglu's group demonstrated that many pairs of instructions could be collapsed to yield a single instruction with no more than two operands. The work of Vassiliadis therefore encourages us to believe that many dependent instruction groups involving three instructions can be collapsed in a similar fashion to yield operations requiring no more than three source operands. Again, an instruction scheduler is ideally placed to undertake these instruction collapsing operations at compile time. As a result, a functional unit with only three inputs can be provided, as long as the instruction scheduler successfully generates instruction groups that require no more than three source operands.

3. Implementation of Static Data Dependence Collapsing

This section considers in detail how instruction combining might be incorporated into a high-performance superscalar architecture. We assume that aggressive instruction scheduling techniques will be deployed to support instruction combining.

We consider three approaches. First, combined instructions could be implemented as a set of complex instructions, where each pair of combined operations is compacted into a new three-operand instruction. Second, the implementation could take the form of separate instructions that are grouped dynamically at run time using the Interlock Collapsing Technique developed by the SCISM group. Third, combined instructions could retain their individual identities, but could be marked or tagged as combined instructions. In this way information compiled by the instruction scheduler can be directly communicated to the instruction issue logic.

The first approach was initially implemented in our HSA architecture. Essentially, if a true data dependency occurs between two instructions and they can be executed in a single functional unit, they are combined into a single CISC-like instruction as shown below:

```
MULT R7, R9, #14
ADD R6, R7, R5
```

becomes

```
ADD R6, (R9 * #14), R5
```

This is an attractive solution when only a small number of new instructions is involved and has been widely adopted on a number of designs to implement limited combining, such as the addition of a MULTIPLY-

ADD instruction (13, 14, 15). However, as a general solution, this approach has a number of drawbacks. First, the large number of new instructions required, all with three source operands, can not be encoded within the standard 32-bit RISC instruction word. Second, long complex instructions are more difficult to decode. Finally all versions of an architecture would have to execute the full range of complex instructions, potentially leading to compatibility problems.

The SCISM approach of collapsing instructions at run time avoids all the problems of complex instructions outlined above. Instead the major concern becomes the increased demands being placed on the instruction decode and issue logic, particularly if non-adjacent instructions are collapsed. However, if instruction scheduling is deployed to ensure that combined instructions are always placed in adjacent instruction slots, the original SCISM form of Interlock Collapsing can usefully be employed. A final disadvantage remains. The first instruction of the collapsed pair must also be executed in a separate functional unit to preserve the program semantics, even though preliminary observations suggest that this intermediate result is usually not required.

The third approach is our preferred solution. Here the combined instructions remain separate, but the instruction scheduler ensures that they are always adjacent and sets a one bit tag on the second instruction of each instruction pair. The instruction issue logic then uses the tags to detect combined pairs of instructions and forwards them to a single three-input functional unit for execution. In contrast to the SCISM approach, no attempt is made to forward the leading instruction to a second functional unit. Instead, in those cases where the intermediate result is required, it must be explicitly generated by duplicating the first instruction in the instruction stream. This approach conserves functional units and simplifies instruction issue, yet is likely to lead to only a tiny expansion in code size. The major disadvantage of this tagging scheme is that, unlike the SCISM approach, it can only be implemented as part of a new architecture.

4. The HSA Trace Driven Simulator

The HSA Trace Driven Simulation (TDS) tool [16] was developed to quantify the availability of parallelism in typical benchmarks. In this paper we use the TDS tool to quantify the potential benefits of instruction combining.

To obtain traces for the TDS simulator, the object code for each benchmark is executed on the HSA instruction-level simulator, which generates a compact trace of taken branches and memory addresses. This trace provides all the relevant information to drive a variety of trace driven simulations and yet is considerably smaller than a full instruction trace. The trace instruction stream and the original object code are then presented to the HSA TDS tool.

The TDS tool simulates the execution of the object code based on the machine model being tested, and allocates a parallel instruction time-slot (PIT) to each instruction that corresponds to the earliest time at which the instruction could be executed. For each instruction the TDS tool determines when the source operands, including memory operands, are available. The latency of the instruction is then added to the PIT time of the source operand that was computed last to create a new PIT that is associated with the destination operand. Instructions arbitrarily far apart in the original code can therefore be allocated the same PIT number, indicating that they could have theoretically been executed in parallel. Finally, the largest PIT created by the program is taken as the total execution time for the benchmark.

No instruction can ever be executed before its source operands are available. The TDS tool therefore strictly enforces all true data dependencies. In contrast both anti-dependencies and output dependencies are ignored. Since both these dependencies can be removed by renaming, the TDS tool simulates perfect renaming of both register and memory operands. The TDS tool has a perfect knowledge of all memory addresses accessed by the program and is therefore able to simulate perfect memory disambiguation. This execution model is used to compute an upper bound on the available ILP and is equivalent to the Oracle model used in other studies [17]. By allowing certain pairs of dependent instructions to execute in a single time slot, we can use the TDS model to quantify the additional speedup available as a result of instruction combining. Additional restraints can then be added to the model, to quantify the impact of restrictions introduced by instruction scheduling algorithms.

5. Trace Driven Simulation Results

This study uses the Stanford integer benchmark suite, a collection of eight ‘C’ programs that is designed to be representative of non-numeric code. All the benchmarks were compiled by the HSA gnu ‘C’ compiler that targets the HSA instruction set. The benchmarks are computationally intensive with high dynamic instruction counts.

Four functional units are involved in combining: the Arithmetic Unit, the Relational Unit, the Multiply Unit and the Shift Unit. The Arithmetic Unit can execute any combination of dependent add and subtract instructions. The two-bit shift operation, frequently used in array address calculations, can also be combined with a following addition or subtraction. Similarly, the Relational Unit, that generates one-bit Boolean values, is allowed to combine an add or a subtract instruction with a dependent compare instruction. The Boolean values generated are tested by conditional branch instructions and therefore replace the traditional condition codes. The Multiply Unit is allowed to combine MULTIPLY-ADD and MULTIPLY-SUBTRACT instruction pairs. Finally, the Shift Unit can combine shift instructions with dependent logical ANDs and ORs. These unlikely combinations are a consequence of a decision to relegate

all logical operations in an HSA processor to the Shift Unit. This transfer simplifies the design of an arithmetic unit with three source operands and reduces the already minimal impact of combining on the processor cycle time even further.

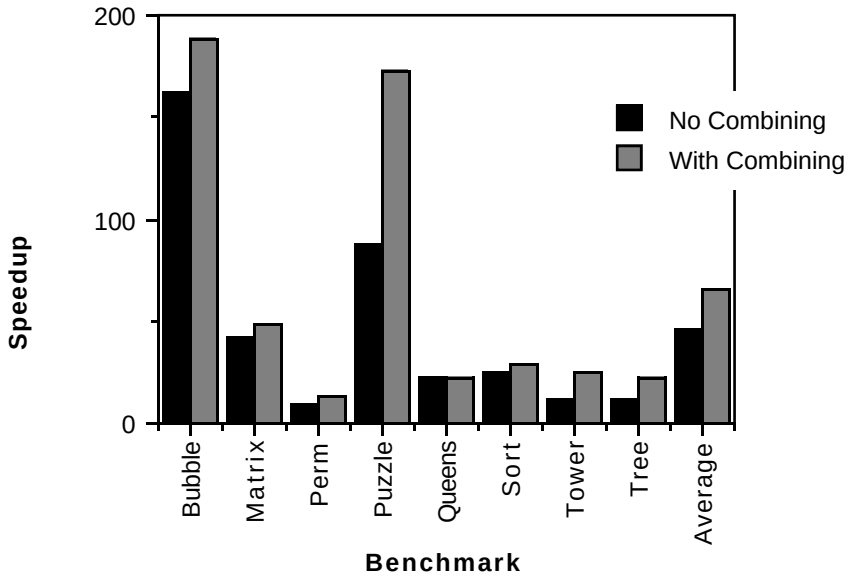
Since trace driven simulation is used, the instruction pairs that are combined can originate arbitrarily far apart in the dynamic instruction stream. The TDS tool therefore measures the potential parallelism available through combining. To realise this potential, the instruction scheduler must succeed in moving the instruction pairs concerned into adjacent instruction slots. In HSA schedulers this involves percolating the second instruction up through the code until it reaches the first instruction.

To obtain a Baseline Model for our simulation results, all the benchmarks were first executed on the HSA instruction-level simulator with an instruction issue rate of one and perfect branch prediction. Apart from MULTIPLY, with an instruction latency of three, and DIVIDE, with a latency of sixteen, unit latencies were assumed. All the speedups obtained using the TDS tool are relative to this Baseline Model.

5.1 The Available Parallelism

We then used the TDS simulator to obtain speedups relative to the Baseline Model, both with and without instruction combining (Figure 1). Without combining, speedups ranged from 9.5 to 162.2, with an average of 46.35. With combining, speedups ranged from 13.7 to 188.5, and the average speedup increased to 65.14, an increase of 40.5%. These figures suggest that combining has the potential to increase superscalar processor performance dramatically.

Figure 1 Impact of Combining on Available Parallelism



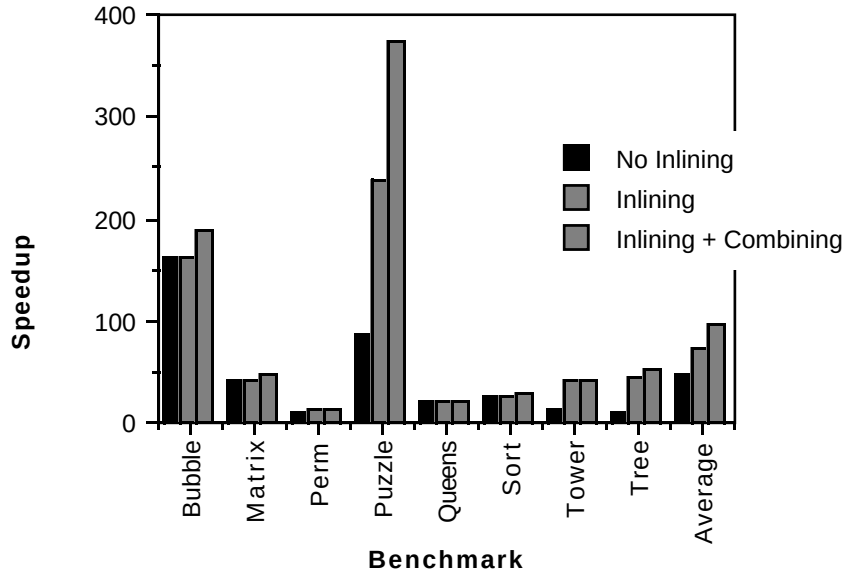
5.2 Function Inlining and Combining

Since Trace Drive Simulation removes all control and data dependencies apart from true data dependencies, it might be reasonable to assume that the Oracle Model used in the previous section would yield figures that represent the ultimate limits of parallel instruction execution. This, however, is far from the case. Inlining is used in this section to illustrate this point.

Inlining is a powerful technique that is used by most instruction schedulers to uncover additional instruction scheduling opportunities. Even the most straightforward of implementations allows the function call and return instructions to be removed. Under more favourable circumstances, all related stack manipulation instruction can also be removed, along with the instructions to store and reload registers on function entry and exit. We used the TDS tool to simulate these ideal circumstances to see how much additional parallelism was available.

The impact of perfect inlining is shown in Figure 2. Without combining, perfect inlining increased the average available parallelism from 46.35 to 73.3, an increase of 58%. The addition of combining increased this figure to 96.0, a further increase of 31%. Procedure entry and exit code is an artefact of both programming languages and of their traditional implementation using a succession of stack frames. The TDS simulations suggest that the performance cost of the traditional mechanism is potentially very significant in a multiple-instruction-issue environment.

Figure 2 Function Inlining



While speedups of 73 are implausible in a commercial superscalar processor, at least in the immediate future, many schedulers attempt to tap into this rich vein of potential parallelism. The first stage is to inline function calls, particularly those within loops, followed by the judicious removal of stores, loads and stack pointer manipulations. These instructions can either be removed during the actual inlining process by using data dependence analysis or by collapsing pairs of operations during subsequent instruction scheduling. Consider the following two examples:

i) ST offset(SP), Ri
:
LD Ri, offset(SP)

ii) SUB SP, SP, #stack-frame-size
:
ADD SP, SP, #stack-frame-size

In both cases, the operations can be removed whenever the second instruction percolates successfully up to the first instruction.

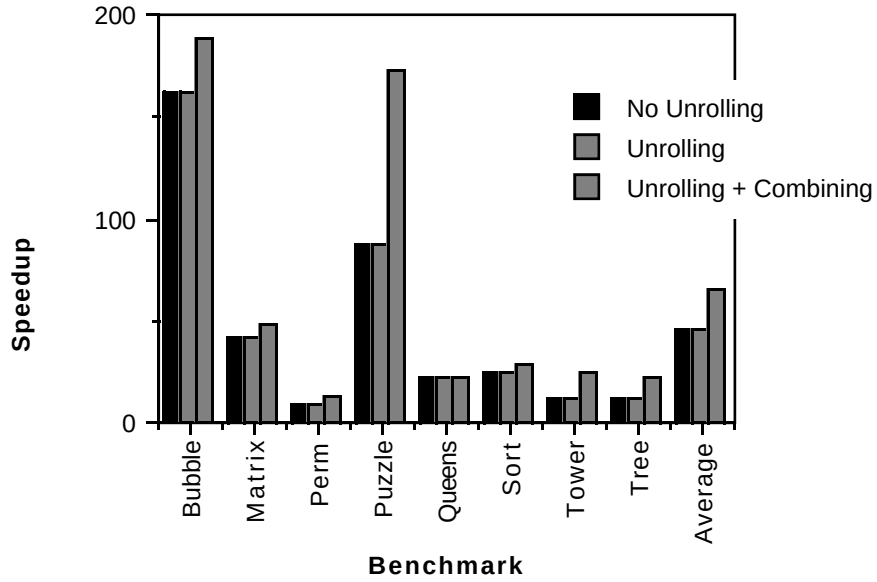
Combining still yields a very significant performance advantage with the perfect inlining model. Nonetheless, the additional speedup with inlining is now only 31%, which is much less than the 40.5% achieved without inlining. This fall suggests that a disproportionate number of the data dependencies collapsed without inlining involved stack pointer manipulations and other operations associated with function entry and exit.

5.3 Perfect Loop Unrolling and Combining

Loop constructs are a further artefact of programming languages and their implementations. Typical loop implementations introduce a succession of loop carried dependencies between instructions that manipulate loop indices and other induction variables. It is possible that the parallelism available within loops is limited by these artificial constructs rather than by the data manipulations performed by the loop. As a simple example, a loop might increment all the entries in a thousand-word array by one. At an abstract level the thousand additions required can be executed in parallel. In practice, however, a loop count is likely to be incremented one thousand times, and at run time these additions will appear as a chain of a thousand dependent operations.

To investigate the impact of loop constructs on the availability of parallelism, the TDS tool was used to simulate perfect loop unrolling. In this model all the instructions that manipulate loop induction variables were removed. The results presented in Figure 3 show that perfect unrolling has remarkably little impact on the available parallelism. Similarly, combining now produces a speedup of 65.16, virtually unchanged from the figure of 65.14 in Figure 1. We therefore conclude that in these benchmarks the introduction of artificial loop constructs has little impact on the availability of instruction-level parallelism.

Figure 3 Perfect Unrolling



5.4 Perfect Memory Model and Combining

Parallel instruction execution is also artificially limited by the provision of a finite number of registers. Ideally, all variables would be held permanently in registers, with no intermediate stores and loads between successive uses of variables. Instruction scheduling involves grouping instructions that are initially arbitrarily far apart in the instruction stream. At an abstract level the perfect memory model can be seen as a recognition of the need to reallocate registers during instruction scheduling to reflect changing instruction localities.

At a lower level, the model recognises that an aggressive instruction scheduler can eliminate pairs of stores and loads that reference the same memory address. Consider the following instruction fragment:

```
ADD R1, R2, R3
ST 8(SP),  R1
  :
LD R5, 8(SP)
SUB R4, R5, R6
```

An aggressive instruction scheduler will collapse the store and load instruction, providing of course that the load can be successfully percolated up to the slot occupied by the store instruction. Finally, if the SUB also percolates up to the ADD position, these two instructions can be combined.

The TDS tool was used to simulate such a perfect memory model with zero load and store latencies (Figure 4). Without combining the potential speedup increases to 66.5, and with combining to a remarkable 102.48, an increase of 54.1%. The very high figure achieved with combining suggests that the simulator had some success in collapsing instructions that originated on either side of store/load pairs.

5.5 Procedure Ceilings and Combining

Finally, we consider a far more restrictive and therefore potentially more realistic model. Here we assume that all functions are scheduled separately and that there is no code motion across function calls. Under these circumstances, all instructions prior to the function call in a calling routine must be executed before any instruction within the called function can be executed Similarly, all instructions within a function body must be executed before any instruction after the function return is executed. We implement these restrictions by placing an artificial barrier or ceiling at all function entries and exits. No instruction is allowed to execute before a prior ceiling in the dynamic instruction stream. The function entry ceiling is set by the instruction before each call that is assigned the latest execution time by the TDS tool. Similarly, the function exit ceiling is set by the instruction within the function that is assigned the latest execution time.

With these major restrictions, potential parallelism is drastically reduced (Fig. 5). Speedups now range from 2.99 to 24.33, with an average of 8.03. If instruction combining is added, the average speedup is raised to 9.77, with speedups ranging from 3.30 to 25.39. Encouragingly, even with these severe restrictions, instruction combining still delivers an increased speedup of 21.7%. Yet in many respects the restrictions imposed by this model are unnecessarily severe. Most instruction schedulers implement some inlining and also attempt

inter-procedural code motion, typically as a final code scheduling pass. Both of these practices breach the function entry and exit ceilings postulated by this model.

Figure 4 Perfect Memory Model

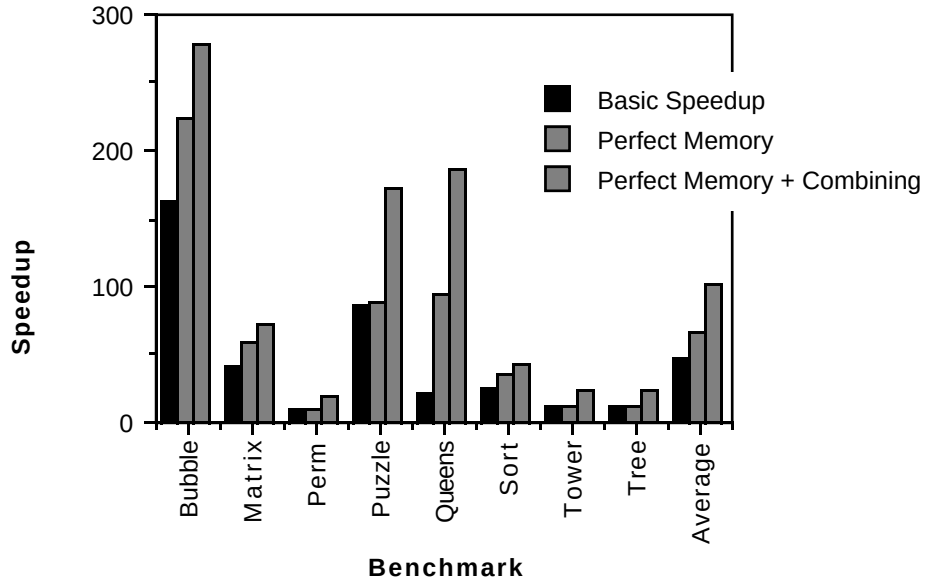
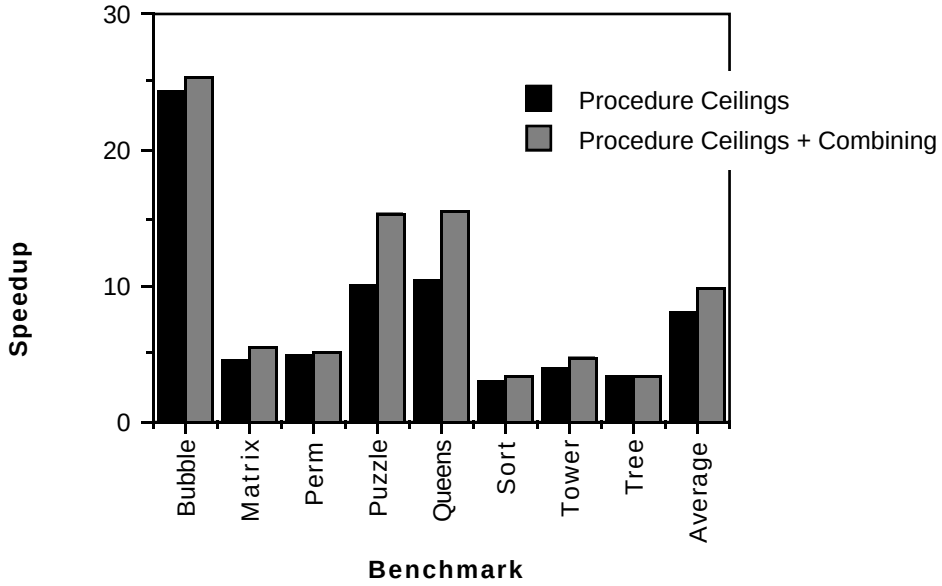


Figure 5 Procedure Ceilings



6. Conclusions

In this paper we have considered a technique for reducing the impact of true data dependencies on the performance of multiple-instruction-issue processors. We term this technique instruction combining or static data dependence collapsing.

Trace driven simulations were used to study the synergistic effects of instruction combining with several different simulation models. Our results suggest that the potential performance benefits from instruction combining are considerable, with increased average speedups ranging from 21.7% to 54.1% depending on the model. Interestingly, even with the severely

limiting procedure ceilings model, improvements of 21.7% are made.

The average speedups simulated range from 8.03 to 102.5. Speedups of 96 with function inlining and 102.5 with the perfect memory model encourage our belief that order-of-magnitude speedups can ultimately be realised through static instruction scheduling. To achieve our goal roughly a tenth of the rich vein of parallelism theoretically available in our more aggressive simulation models must be realised. Our simulation results emphasise that instruction combining should play a major part in achieving our objective.

References

1. Rotenburg, E, Bennett, S and Smith, J E "Trace Cache: A Low Latency Approach to High Bandwidth Instruction Fetching", Micro29, Paris, 1996, pp24-34.
2. Yeh, T and Patt, Y N "Alternative Implementations of Two-Level Adaptive Branch Prediction", 19th Annual International Symposium on Computer Architecture, Gold Coast, Australia, May 1992, pp124-134.
3. Rau, B R and Fisher, J "Instruction-Level Parallel Processing: History, Overview and Perspective", The Journal of Supercomputing, Vol.7, No. 1/2, 1993, pp9-50.
4. Steven G B, Christianson D B, Collins R, Potter R and Steven F L "A Superscalar Architecture to Exploit Instruction Level Parallelism", Microprocessors and Microsystems, Vol.20, No 7, March 1997, pp391-400.
5. Steven G B and Collins R "Instruction Scheduling for a Superscalar Architecture", Proceedings of the 22nd Euromicro Conference, Prague, September 1996, pp643-650.
6. Collins R and Steven G B "An Explicitly Declared Delayed-Branch Mechanism for a Superscalar Architecture", presented at EuroMicro94, Liverpool, September 1994. Published in Microprocessing and Microprogramming, Vol.40, No.10-12, December 1994, pp677-680.
7. Egan, C, Steven, F L and Steven, G B "Delayed Branches versus Dynamic Branch Prediction in a High-Performance Superscalar Architecture", Euromicro97, Budapest, September 1997.
8. Nakatani, T and Ebcioglu, K "Combining as a Compilation Technique for VLIW Architectures", 22nd Annual International Workshop on Microprogramming and Microarchitecture, SIGMICRO Newsletter, Vol. 20, No. 3, September 1989, pp43-55.
9. Vassiliadis S, Phillips, J and Blaner, B "Scism: A Scalable Compound Instruction Set Machine Architecture", IBM Journal of Research and Development, 38(1), pp59-78, January 1993.
10. Vassiliadis S, Phillips, J and Blaner, B "Interlock Collapsing ALUs", IEEE Transactions on Computers, 42(7): pp825-839, July 1993.
11. Malik, N, Eickemeyer, R J, and Vassiliadis, S "Interlock Collapsing ALU for Increased Instruction-Level Parallelism", Micro25, September 1992, pp149-157.
12. Sazeides, Y and Vassiliadis, S "The Performance Potential of Data Dependence Speculation and Collapsing", IEEE Micro 29, 1996, pp238-247.
13. Wulf, W A "The WM Computer Architecture", Computer Architecture News, March 1988, pp70-84.
14. Oehler, R R and Groves, R D "IBM RISC System/6000 Processor Architecture", IBM Journal of Research and Development, 34(1), January 1990, pp23-36.
15. Weiss, S and Smith, J E "Inside IBM Power and PowerPC", Morgan Kaufmann, 1994.
16. Potter R and Steven G B "Investigating the Limits of Fine-Grained Parallelism in a Statically Scheduled Superscalar Architecture", 2nd International Euro-Par Conference Proceedings, Vol.2, Lyon, France, August 1996, pp779-788. (Published as Lecture Notes in Computer Science 1124 by Springer)
17. Lam M S and Wilson R P "Limits of Control Flow on Parallelism." 19th Annual International Symposium on Computer Architecture, Gold Coast, Australia, May 1992, pp46-57.