Investigating The Limits of Instruction Level Parallelism

Richard D. Potter
Division of Computer Science,
University of Hertfordshire,
College Lane,
Hatfield,
Herts. AL10 9AB

email : comrrdp@herts.ac.uk

Technical Report No. 245
February 1996

Abstract

High performance computer architectures increasingly use compile-time instruction scheduling to reorder code to expose parallelism that can be exploited at run-time. Although respectable performance increases have been reported, there is still a significant performance gap between what has been achieved and what has theoretically been shown to be possible. All scheduling algorithms used to reorder code, either explicitly or implicitly introduce barriers to code motion, which in turn limit the performance realised. Trace driven simulation is used to quantify the amount of instruction level parallelism available in general purpose code and the impact of various artificial barriers to code motion. This work is based on the Hatfield Superscalar Architecture, a progressive multiple instruction issue processor. The results of this study will be used to direct future developments in instruction scheduling technology.

Keywords : Superscalar, Instruction-Level Parallelism, Instruction Scheduling, HSA, Trace Driven Simulation.

Table of Contents

List Of Figures

List Of Tables

1. Introduction

In recent years there has been increasing interest in the use of compile-time instruction scheduling to improve the performance of superscalar processors [Bernstein92] [Smother93]. Instruction schedulers speed up program execution by assembling groups of independent instructions which can be executed in parallel at run time. Scheduling therefore involves moving or percolating each instruction up through the code structure to form these instruction groups.

Our research in this area is based on the Hatfield Superscalar Architecture (HSA), a generic superscalar architecture. HSA prefetches multiple instructions into an instruction buffer from the instruction cache. In each cycle, the instruction decode logic attempts to issue as many instructions as possible from the instruction buffer for parallel execution in multiple functional units. Ideally the instructions issued will correspond to the parallel groups assembled earlier by the instruction scheduler. Since code has already been reordered at compile time, the HSA instruction issue logic always issues instructions in order.

Numerous empirical studies [Lam92], [Smith89] and [Wall91] suggest that large numbers of instructions in a sequential program can theoretically be executed in parallel. For example, Lam and Wilson's study which aims to identify the upper limits of fine-grained parallelism, reported speedups of up to 158 using the SPEC benchmarks. In contrast existing instruction schedulers [Collins95] [Ebcioglu94] [Hank93] and [Wang93], typically achieve speedups in the range of 2 and 7. The speedup available in theory therefore exceeds the speedups achieved by existing instruction schedulers by over an order of magnitude. The objective of our research is to narrow this very significant performance gap.

Although notable performance improvements have been achieved, all instruction scheduling algorithms erect artificial barriers or as Wang [Wang93] called them, ceilings to code motion. For example, a particular algorithm may preclude the execution of any code within a loop body until all the code originally preceding the loop has first been executed. As a result of these ceilings, we have developed a Trace Driven Simulator (TDS), to quantify the instruction level parallelism in typical benchmarks and to relate the available parallelism to alternative instruction scheduling strategies. The results will be used to direct enhancements to the two HSA instruction schedulers which have been developed at the University of Hertfordshire. In particular, future work will be directed to removing those barriers to code motion which significantly limit performance, identified by the TDS tool.

This document is split into 7 main sections. The next section presents a summary of previous work in this field and discusses the major results presented so far. Section 3 introduces the Hatfield Superscalar Architecture and outlines its major features. Section 4 discuses some of the limits to instruction parallelism and those limitations introduced by various scheduling strategies. Section 5 outlines our approach to trace driven simulation. Section 6 presents a selection of the results so far generated. Finally, Section 7 presents the conclusions based on this study and further work suggested by it.

2. This Study and Previous Work

A number of researchers have attempted to quantify the limits of instruction-level parallelism (ILP) and their work has produced a large variety of results. This research has been approached in two distinct ways, either by proposing a feasible system design or by investigating the upper limits based on theoretical models, where many of the constraints have been relaxed other than those features under evaluation. Presented below is a discussion of some of these studies in chronologically order.

Early studies [Tjaden70],[Riseman72] showed that average parallelism within basic blocks rarely exceeded 3 on average. The study by Riseman was one of the first to show the effects of control flow on parallelism. By modelling a machine with foreknowledge of conditional branch direction, Riseman showed that a speedup of 51 was possible. Nicolau and Fisher [Nicolau84] referred to this as an Oracle machine. Nicolau's study was based on evaluating trace scheduling for a VLIW architecture. This work demonstrated that using a restricted processor model with perfect branch prediction and alias analysis, speedups of 90 were possible. Though these speedups were achieved using small scientific benchmarks where there is a large amount potential of ILP.

Smith et al [Smith89] used trace-driven simulation on a model of a realisable superscalar processor with multiple functional units. For a model with perfect renaming, branch prediction and an ideal fetch unit, speedups of between 2.3 and 4.1 were reported. However, several features of this study impeded the performance of the processor. Firstly, the issue rate was limited to 4 instructions per cycle, even though the processor had 9 functional units allowing a peak execution rate of 9 instructions per cycle (IPC). Secondly, the configuration of the functional units poorly matched the instruction class execution frequency. In the study, 38% of the instructions executed were integer operations but the processor was resourced with only a single integer unit. The study noted that the biggest increase in performance occurred when a further integer unit was added. Finally, the processor had a very small look-ahead window of 32 instructions, which severely limits the ability of the processor to take advantage of the ILP present in the benchmarks. The study concluded that non-scientific applications contain enough instruction parallelism to sustain an instruction rate of two. This conclusion is only true based on the assumptions made within this study, for this one restricted design. This highlights the problems when attempting to quantify limits, that each reported limit is subject to all the limits and assumptions made within the study.

The work by Butler et al [Butler91] is a direct progression from the work of Riseman and attempts to refute the claims by Smith et al that the ILP within non-scientific code can only support an issue rate of two. Butler et al attempts to identify ILP for a range of unconstrained as well as realistic processor models. Butler's model used perfect branch prediction as well as unbounded functional units, issue rate and instruction window size. For an Unrestricted Dataflow model, parallelism in the range of 17 to 1165 IPC was shown for nine of the SPEC benchmarks. This study also placed limits on the extraction of ILP by assuming only one branch per cycle can be executed and no instruction following the branch can be executed in the same cycle. This limits the available parallelism to the average size of the basic block. Results for more realistic machine models suggest that, typically ILP inherent within non-scientific code ranges from 2.0 to 5.8 IPC.

A further study in this year by Wall [Wall91], concluded that even with impossibly good techniques parallelism rarely exceeds 7, with 5 more common. The most powerful model in this study had perfect branch prediction, jump prediction, register renaming and alias analysis. The extracted parallelism for this work was in the range 6.5 - 61. Wall's study had several features that limited the parallelism exposed to his models. Firstly, instruction execution is simulated by packing instructions from an instruction trace into a sequence of pending cycles. Each cycle could contain up to 64 instructions, limiting the maximum ILP to this figure. Secondly, in order to model a superscalars instruction window, the study's model only considers a constrained number of pending instructions. By default, this is 2k. Therefore, independent instructions more than 2048 instructions apart in the instruction trace will be prevented from executing concurrently. While 2k is an unrealistically large look-ahead window for a superscalar processor, it still confines the processor to realising parallelism from a very limited pre-fetch window of instructions.

Austin [Austin92] argued that 1st order metrics such as operation frequencies and branch prediction accuracies, were not sufficient to understand the characteristics of dynamic program execution for MII processors. To allow a thorough understanding of the interaction between operation dependencies, how these are impacted by the processor model and their effect on performance, the use of dynamic execution graphs were proposed. These graphs demonstrate how parallelism is distributed throughout the execution of the program. This work is modelled around the latencies of a MIPS processor[1], to generate these graphs from an existing instruction trace. Using an Oracle dataflow model, parallelism was measured for the SPEC benchmarks and was found to range from 13 - 23,303. Optimistic assumptions about the usage of registers after system calls, increased the range to 33,748 IPC. More significantly, the distributions of parallelism were shown to be bursty, with areas of high ILP followed by areas of low ILP. The study concluded that a very extensive look-ahead window is needed, in excess of 100,000 instructions, to extract significant quantities of the available parallelism.

Lam and Wilson [Lam92] attempted to relax the constraints on control flow by using control dependence analysis, executing multiple flows of control simultaneously and using a speculative execution model. As in earlier studies, instruction execution was only constrained by true data dependencies and because an upper bound rather than a limit for a particular processor was being evaluated, unit latencies were used. However, the model used attempted to enhance the benchmark code by application of a perfect procedure inlining technique and loop unrolling. These mechanisms relaxed several limitations inherent in the code and exposed significant parallelism. The results ranged from 47 to 188,470 with a harmonic mean of 158.26.

As many of the benchmarks used in the above studies used the SPEC benchmark suite there is considerable overlap between the studies. Table 2 presents these results. It can be seen from this table that these 'limits' are directly affected by the assumptions used and by the architectures modelled. It can be concluded that there is adequate parallelism inherent in most programs to support an order of magnitude speedup through static instruction scheduling.

[1] (Fp/Int Mult 6, Fp/Int Div 12, Fp Add/Sub 6, all other latencies 1)

Benchmark	Butler[2]	Wall[3]	Austin[4]	Lam[5]
Eqntott	300	-	942.4	3282.9
Espresso	179	40.6	176.3	742.3
Gcc (cc1)	38	26.5	53.0	174.5
Doduc	55	56.8	107.2	-
fpppp	378	60.4	2032.8	-
Matrix 300	1165	-	33748.6	188470
Spice 2g6	17	-	138.4	843.6
Tomcatv	930	59.7	6800.3	3918
xlisp	162	-	13.3	-

Table 2.1 - Limits of Instruction Level Parallelism

This study is an attempt to find, for the benchmarks used within HSA, a reasonable upper bound on the available ILP. As the initial work on the HSA project has concentrate on a smaller benchmark suite, which are designed to be computationally intensive with unpredictable control flow, it will be extremely beneficial to place the results in context. Secondly, the majority of the processor models in the preceding work use superscalar out-of-order techniques, varying instruction latencies and branch prediction. The HSA project, as will be seen in a later section, is an attempt to minimise the hardware needed to issue multiple instructions in parallel by using static scheduling to re-order the code. The difference in techniques, mean that while the above studies provide useful guides to possible performance, they are not directly relatable. However, the principal aim throughout this study is to look at limitations to ILP, placed implicitly or explicitly on the execution model by scheduling algorithms. The effects of these limitations have not been studied so far.

3. Hatfield Superscalar Architecture

The Hatfield Superscalar Architecture (HSA) is a family of abstract Multiple Instruction Issue (MII) processors. The HSA is highly parameterised to allow investigation of hardware and software mechanisms to exploit instruction-level parallelism. The long term objective of the HSA project is to achieve an order of magnitude speedup over a scalar RISC architecture.

The HSA is often referred to as a minimal superscalar, as it is a progressive attempt to remove many of the mechanisms used within superscalar processors. The HSA does not support out-of-order issue, dynamic renaming or score-boarding and avoids branch prediction. Instead it by combines the best features of previous MII architectures, most notably Very Long Instruction Word (VLIW) processors and superscalar. Recent designs have shown that neither of these two methodologies have been sufficient to bridge the performance gap between what is theoretically possible and what has been realised. Both of these two distinct architectures have desirable features.

VLIW

VLIW architectures have the advantage that no dynamic analysis is needed - the compiler performs any necessary analysis and schedules independent instructions into fixed length multi-operation instructions (VLIW's). The compiler can take a global view of the program and utilise sophisticated analysis techniques and code transformations that would be too expensive to perform at run-time. The use of instruction scheduling greatly simplifies the hardware required, leading to faster processor designs. However, schedulers for VLIW architectures [Ellis86] [Wang93] require the processor hardware to conform exactly to the assumptions built into the scheduler with respect to the number if functional units and operation latencies. Therefore the scheduled code is processor specific due to these rigid assumptions. A further problem is introduced by the fixed size of the VLIWs, the density of the schedule is dependent on the ILP present, if the scheduler cannot find sufficient independent instructions to fill the VLIWs, NOPS have to be inserted leading to code expansion problems.

[2] Non Uniform Latencies
[3] Unit Latencies
[4] Mips Latencies
[5] Unit Latencies

Superscalar
Superscalar processors are typically processors that analyse the instruction stream dynamically. In each cycle, every instruction within a given instruction window is analysed for dependencies. The processor takes a local view of program and reorders these instructions dynamically to exploit ILP. The hardware must therefore ensure all dependencies are detected and enforced. This dynamic analysis ensures code compatibility between processors and allows instructions to be considered across branch boundaries, for concurrent issue. The principal drawback with superscalar designs, is the hardware requirement to perform this dynamic instruction stream analysis and to ensure correctness of execution. The complexity of this hardware becomes prohibitively large at a moderate degree of parallelism. Existing designs have had difficulty dynamically analysing 2 to 4 operations, thus restricting the processors ability to exploit high ILP.

The HSA model attempts to combine these distinct designs by using compile time scheduling to reduce the hardware complexity, while maintaining code compatibility. The HSA removes the compatibility barrier for a static instruction scheduler by providing a generalised delayed branch mechanism. The use of this mechanism, allows the parallelism exposed by scheduling to be maintained when the code is converted back to sequential form, suitable for executing on a superscalar.

The HSA has the following major features ; a generalised delayed branch, in-order instruction issue, out-of-order completion, conditional execution, a common pool of functional units, floating point capabilities, speculative execution and guarded instruction execution. Some of these features are discussed in the following sections :

3.1 A Generalised Delayed Branch

HSA provides a delayed branch mechanism, that allows instruction execution to continue while a new instruction stream is fetched from the branch target. Branches have an explicitly encoded count value which indicates the number of instructions that must be dispatched after the branch. Instructions are then promoted to fill this branch delay region, including further branches. This flexibility allows the branch mechanism to adapt to a wide range of cache latencies and instruction issue rates, yet maintains instruction set compatibility over a range of implementations. This may have an impact on the overall speedup, if the processor the code has been scheduled for and the processor executing the code, are far apart in their ability to exploit the exposed parallelism. Results reported so far [Collins95], show that the reduction in performance is much smaller than would be expected. If code scheduled for a machine with infinite resources, is then executed on a machine whose degree of parallelism is only two, there is only an 18% decrease in performance. This result is based on a comparison with the figures for executing code scheduled for this 2-pipe processor. For a machine with a degree of parallelism of eight, this is reduced to only 6%. These results assume that the processor has the ability to squash instructions prior to dispatch .

3.2 In-order Instruction Issue

In the HSA model, instructions are always dispatched in program order to the functional units, as sophisticated static code scheduling is used, making hardware re-ordering of the code redundant. By moving the responsibility for instruction analysis to the instruction scheduler, this has the advantage of avoiding a significant amount of hardware complexity in the instruction issue stage.

3.3 A Common Pool of Functional Units.

The HSA issues instructions to the appropriate functional unit from a common pool of resources from the instruction buffer. The number of functional units within the HSA model is not fixed, Arithmetic, Multiply, Memory Loads, Memory Stores, Branch, Relational, Shift and Floating Point functional units are postulated.

There is a trade off between the complexity of design for a large number of functional units and the usage of these resources. The greater the machine parallelism available, the greater the processor's ability to deal with areas of high ILP. Performance should not be greatly limited by instructions being 'blocked' from execution if there are not sufficient resources to handle it. Conversely, the number of functional units should not be unduly high, as the number of interconnections needed would require a very sizeable section of the chip area and there would be little to gain from a large number of lightly used functional units. This trade-off is not studied here, but further work on the configuration of functional units can be found in [Jourdan95] and [Collins95].

Three input operand functional units are provide in the HSA specification to allow combination of instructions. This allows combining of infrequently executed logic instructions with other operations in an attempt to avoid degrading ALU performance. A three input ALU unit may allow dependency chain to be compressed. The results of combining can be seen in this study in section 6.7.

3.4 Full Floating-Point Capability.

The HSA model fully supports floating-point instructions and has floating point versions of the relevant functional units. Floating point instructions would seem to further hamper performance as they have long instruction latencies associated with them. In HSA, as the scheduler has the ability to perform large scale code motion, it is thought that the floating point instructions with their long latencies will provide an extra boost to the exploitation of instruction-level parallelism. By allowing interleaving of the integer and floating point dependency threads, significant performance increases should be achievable over a scalar implementation of floating point HSA. Floating point capabilities also extend the usefulness of the processor itself, floating point resources can be used by both integer and floating point programs alike, allowing further improvement in performance on a range of applications.

3.5 Guarded Instruction Execution.

All HSA instructions can be guarded by multiple guard Booleans allowing conditional execution. For example consider:

```
        FB1 ADD R1, R2, R3,
```

The ADD instruction will only execute to completion if its Boolean guard B1, evaluates to False at run-time. HSA attempts to 'squash' or remove instructions from the instruction buffer every cycle whose Boolean condition fails. The use of boolean guards helps improve resource utilisation by removing instructions prior to dispatch, therefore saving on functional unit, result bus and write back resource usage. Instructions can also be aborted within functional units allowing further savings.

An instruction scheduler can also use guarded execution to remove branch instructions by converting *if-then-else* constructs to a sequence of guarded instructions. These*if-then-else* constructs typically use conditional branches to select one of two short code segments. Booleans can be used to guard the subsequent *if* and *else* code elements, removing the need for a branch. Recent work has shown that this can reduce the number of dynamic branches dramatically; with the IMPACT group [Mahlke94] reporting a 27% reduction and Collins [Collins95] reporting a reduction of 35%.

3.6 Speculative instruction execution.

To gain maximum performance, all instructions in the HSA instruction set, apart from Stores and Branches can exist in a non-speculative and speculative form. This allows instructions to be executed much earlier than their control dependencies would otherwise allow. Speculative execution allows instruction execution to proceed without waiting on the completion of previous instructions. This can have significant benefits, for example, a load instruction executed speculatively could allow other instructions dependent on its destination register, to begin execution much earlier. This could be especially critical on a slow cache machine or when loading from memory hierarchies. HSA handles the problems of exceptions in the presence of speculative execution by marking all registers with a flag indicating whether or not the register contents are polluted. Only when a non-speculative instruction attempts to access these polluted results will the exception finally be taken, thus allowing precise interrupts.

For a more detailed evaluation of the features of HSA see [Collins93] and [Steven95]

4. Limits To Instruction-Level Parallelism

Instruction Level Parallelism (ILP) is a measure of the interaction between program parallelism and machine parallelism. Program parallelism is defined as the number of instructions a processor may be able to execute concurrently. Its is determined by the true data dependencies and control dependencies within the code. The latencies of the instructions determine how severely true data dependencies limit this program parallelism. Machine parallelism is a measure of the ability of the processor to exploit this program parallelism. Its is determined by the number of instructions the processor can fetch and execute at the same time. If the machine parallelism is smaller than the program parallelism, resource conflicts will limit the processors performance. As these resource conflicts are not true limitations but are removable by duplicating contested resources, it is assumed in this study that machine parallelism is always sufficient. Therefore in this study, the available instruction level parallelism is determined entirely by program parallelism.[6]

The following sections describe the remaining limiting factors, data dependencies and control dependencies. They also explain how they each in turn limit instruction-level parallelism and ultimately performance. Also covered in this section, is the area of memory dependencies.

4.1 Data Dependencies

There are five possible data dependencies between a pair of, three operand RISC instructions as used by HSA[7]. These are comprised of the three types of data dependencies that can limit the execution of the instruction pair.

Read-After-Write (RAW) or *True* dependencies occur when an instruction uses a value produced by a previous instruction. The second instruction is said to have a true data dependency on the first instruction and must be delayed until the preceding instruction produces the required result. Figure 4.1, shows two example HSA instructions, here the ADD requires the result of the preceding SUB instruction (R4) and is data dependent upon the SUB's destination register.

```
SUB R4, R9, R7          /*    R4 = R9 - R7   */
ADD R5, R4, #28         /*    R5 = R4 + 28   */
```

Figure 4.1 - Read After Write Dependency.

It is this prevention of execution of subsequent instructions that can have a critical impact on the execution of the program and severely limit the processors ability to execute more than one instruction per cycle. True data dependencies on long latency instructions can often prevent further instructions being executed leading to zero-issue cycles where no further instructions are executed. When the long latency instruction does finally complete, there is often only one new instruction that can be executed, thus creating a single-issue cycle due to the chaining of data dependencies through an instruction sequence. The number of zero and single issue cycles has a detrimental impact on the processors performance when executing a program. These true dependencies cannot be removed by instruction scheduling and ultimately limit the performance of all multiple instruction issue (MII) processors.

Write-After-Read (WAR) or *Anti* Dependencies occur when a register defined as a source operand for one instruction is also defined as the destination register for a succeeding instruction. The following instruction must be delayed to allow its preceding instruction to read the register before its is overwritten. In Figure 4.2, the register R6 cause the WAR dependency between the two ADDS.

```
ADD R4, R6, R8          /*    R4 = R6 + R8   */
ADD R6, SP, #12         /*    R6 = SP + 12   */
```

Figure 4.2 - Write After Read Dependency.

Write-After-Write (WAW) or *Output* Dependencies occur when two instructions specify a common destination register. This is show in Figure 4.3 where the two instruction both use R4 as their destination register.

```
SUB R4, R5, #2          /*    R4 = R5 - 2    */
ADD R4, R6, R7          /*    R4 = R6 - R7   */
```

Figure 4.3 - Write After Write Dependency.

Both WAR and WAW dependencies are a side-effect of the re-use of register and memory locations in the program; they cause a dependency between two otherwise independent instructions. These two dependencies are sometimes called Storage Dependencies to reflect this. Many of these false dependencies arise because compilers attempt to use as few registers as possible. In the case of

[6] In the TDS tool, the processor is modelled as having infinite resources.

[7] HSA instructions are typically of the form :-

<opcode><destination><source_register1> <source_register2>

registers, storage dependencies can be removed by using a different register for the second result. In Figures 4.2 and 4.3 renaming R6 and R4 respectively, will remove the dependencies.

It is important to distinguish True dependencies from these storage dependencies, as this type of dependency represents the flow of data and information through the program and therefore cannot be eliminated readily.

4.2 Control Dependencies

Control Dependencies or procedural dependencies, are introduced as a result of control flow within the dynamic instruction stream. If a processor has foreknowledge of all the instructions that will be executed (this type of machine is often referred to as an *Oracle*) it has been shown that it can achieve dramatic speedups [Lam92]. Unfortunately because of branches in the dynamic instruction stream, the processor only discovers which instructions are executed as the program is executed. For every branch taken, the branch target must be resolved, instructions must be fetched from the branch target stream and any unwanted instructions removed from the instruction buffer. Therefore every execution of an instruction, has a control dependency (sometimes called a procedural dependency) on the immediately preceding branch. The average number of instructions between branches is usually small; Johnson [Johnson91] reported an average of 6. If substantial ILP is to be realised, the effects of these control dependencies can be reduced by speculatively executing instructions along one or more paths before all the preceding conditional branch instructions have been resolved. Lam's study showed a harmonic mean for the speedup of an oracle model of around 158. For a similar processor model using good branch prediction the results were less encouraging, Wall's study reported speedups in the range of 4.1 to 7.4. This suggests that control flow is indeed a major inhibitor to performance.

4.3 Artificial Limitations To Code Motion

As well as the limitations inherent within the code, instruction scheduling algorithms also introduce restrictions on instruction execution and code motion, which in turn limit the speedups realised. These barriers may be introduced as side-effects of an algorithm or may be explicitly dictated by the algorithm. In either case they may significantly limit the amount of ILP realised. Three major areas are considered in this study where these barriers have a significant impact: loops, procedure calls and memory disambiguation. Further work is also carried out to study the effects on instruction combining and the impact of allowing speculative writes.

4.3.1 Loops

In may programs a significant proportion of the execution time is spent in loops. Therefore instruction schedulers attempt to optimise these loops as way of optimising a major part of the execution time, concentrating on generating efficient schedules for inner loops and on minimising the initiation interval between successive iterations. The minimum loop iteration rate cannot be lower than the longest latency instruction within the loop.

As long as a loop body is not duplicated or unrolled, the iteration interval can also never be less than the minimum number of cycles required by the processor to execute a tight loop. In a machine with dynamic branch prediction hardware, this minimum can be a single cycle. However, in a machine like HSA with a delayed branch mechanism, the minimum iteration interval may be two or more cycles depending on the minimum number of branch delay slots. An alternative way of viewing this lower limit is to observe that there must always be a minimum delay before an instruction within a loop can be re-executed.

Loop iteration intervals are also dictated by loop carried dependencies, where an instruction in one iteration uses a result generated in a previous iteration. However, many of these dependencies are introduce by loop indices. These are spurious dependencies and are not an intrinsic part of the loops processing. For example, the final result of incrementing index *i* ten times in ten successive loop iterations can be collapsed into a single addition of ten. The TDS tool examines the effect of combining successive calculations involving literals, effectively collapsing all dependencies involving loop indices.

Another side effect introduced during loop scheduling is that code within the body of the loop is only executed after all the instructions preceding the loop have been executed. This delays the effective starting time of the loop's execution and prevents parallel execution with other code. In particular, two independent loops can never execute in parallel. The impact of this is modelled along with studying the effects of code motion across loops and the effects of code motion into loops.

The final limitations studied are the barriers introduced by the compiler as it generates code for the loop. As most compilers generate code for scalar machines, they produce unnecessary serialising components. These components are not an inherent part of the loop semantics and the effects of their removal are also studied here. These limiting factors include the incrementing of the loop index, as discussed above, comparisons based on the loop indices and branches based on the result of these comparisons.

4.3.2 Procedure Call and Returns

As the concept of modularity increases within applications (especially due to the prevalence of object orientated techniques) the impact on performance of procedure calls and returns will be significant.

Instructions within a procedure are typically never issued until all the instructions in the calling procedure are issued. The same is true for the return from a procedure, where the instructions after the return have to wait for the instructions in the procedure to complete. This imposes a further barrier to ILP. In the Stanford benchmark suite used in this study, procedure entry and exit calls typically accounts for 26% of all branches. The impact of inlining of recursive calls and motion across the procedure are investigated to asses their impact on reducing the impact of these ceilings. Recursive procedure calls are treated as loops, by selectively removing the ceiling associated with procedure entry for directly recursive loops and by partial collapsing of the associated stack frame.

The procedure construct also introduces unnecessary serialising constructs which are added during compilation. A new stack frame is allocated and deallocated for each procedure introducing dependencies on the stack pointer (SP), and a significant amount of memory traffic is generated saving and restoring registers across procedure calls. For a program with many small procedures this will have a major impact on the ability of a scheduler to take advantage of the available parallelism. To reduce these barriers, the use of selective and perfect procedure inlining is studied.

4.3.3 Memory Dependencies

HSA, like other RISC architectures limits memory access to load and store instructions only. To take advantage of any parallelism present in the code, a compiler/scheduler for HSA needs the ability to perform large scale code motion. This will often involve the reordering of memory references. This problem is compounded further, due to the large difference in processor speed and memory speed for many architectures, resulting often in long instruction latencies. This implies that to avoid problems associated with these long latency instructions, we have to execute multiple memory accesses in parallel and or out of order. The memory address therefore require aliasing or disambiguation to find those access which are independent and that can therefore be executed safely in parallel.

Because the addresses of the memory locations being accessed are not always available at compile time, the address are calculated from values held in the registers at run time,. The scheduler has to take a conservative approach and execute memory access in the sequential order of the code. This is the only safe assumption it can make, for example in Figure 4.4, unless the scheduler has the ability to ascertain that the references in I1 and I2 cannot refer to the same memory location. In Figure 4.4 this would mean writing to address 1 and then loading from address 2. It may be more beneficial to execute the load from address 2 and then write to address 1 as the load will free up further dependent instructions. However, this would result in the wrong value being read if it address 1 = address 2.

```
I1          ST (R9, R10 /* address 1 */), R11
                              ....
I2          LD R4, (R7, R8 /* address 2 */)
            ADD R6, R5, R4
```

Figure 4.4 - HSA Memory Reference

The HSA scheduler would have to perform its disambiguation statically. Address 1 for example, is calculated by adding R9 to R10 and address 2 by adding R7 to R8. As these addresses are calculated from values held in the register file they cannot be distinguished at compile/schedule time and would have to executed in order. Static disambiguation techniques are only effective when the memory access patterns are linear and predictable, which is often not the case in general purpose code. Targeting loops is one way to combat the problem, as most memory operations often involve array accesses. Even so this form of static disambiguation involves solving complicated linear diophantine equations, may require the loop bounds which are often not available and it cannot cope very well with pointer de-referencing.

A central point of this research is to examine the effect of memory disambiguation with respect to the scheduling algorithms used in the HSA scheduler.

The impact of speculative writes is also assessed within this study. Unguarded writes can only be fetched as soon as the preceding control-flow has been resolved. Even if the write is guarded it can only be executed once the branch condition has been resolved. These restrictions could limit performance if the store blocks the motion of subsequent loads. This will occur when memory disambiguation fails.

5. Trace Driven Simulation

5.1 Overview of Simulator Model

The HSA TDS tool was developed to quantify the available parallelism in typical benchmarks and to study the effects of the code ceilings introduced. All simulations are performed as shown in Figure 5.1.

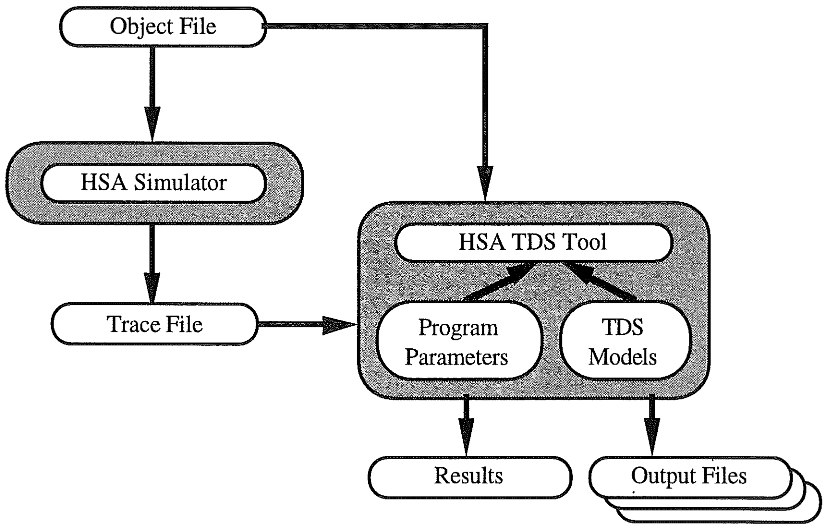

Figure 5.1 Flowchart of Trace Driven Simulation

The optimised object code for the benchmark under evaluation, is first executed on the HSA simulator [Collins93], which generates a compact trace of taken branches and load/store addresses. This trace provides all the relevant information to perform trace driven simulation and considerably smaller than a full instruction trace. Even so, the size of these trace files can reach 2Mb for the full-length versions of the benchmarks used in this study. This dynamic trace stream and the original object code are then loaded into the HSA TDS tool. The TDS tool simulates the execution of the object code based on a model of the execution being tested. Various parts of the model can be activated or deactivated to allow studying of individual and combined effects.

Each instruction as it is examined, is allocated a Parallel Instruction Time-slot (PIT), which indicates the earliest time at which the instruction could be executed. The TDS tool determines for each instruction, when the source operands including memory locations, will be available. The PIT time of the source operand that was computed last is then added to the latency of the instruction to create a new PIT, which is then associated with the destination register.

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. The largest PIT created by the program is then taken as the total execution time for the benchmark.

No instruction can be executed before its input operands are available.

```
1. Instruction To Execute :
16            1        L6:     ASL      R4       R6

Checking Source Operands For Instruction :
R6, 1

Pit That Instruction Can Be Executed In = 1


2. Instruction To Execute :
17      1              ADD      R5       R4       R7

Checking Source Operands For Instruction :
R4, 2          R7, 1

Pit That Instruction Can Be Executed In = 2
```

Figure 5.2 - Sample Breakpoint File From *permcd*

Figure 5.2 is a sample trace of a TDS breakpoint trace file. The first instruction, ASL (arithmetic shift left), has its source operands checked, in this case R6 which is shown to be available in PIT number 1. The shift instruction is allowed to 'execute' in this time-slot and the destination register R4 is associated with the latency of this instruction and this PIT. This can be seen in the ADD instruction which has R4 as a source operand. R4 is checked and is shown to be available next in PIT 2. As this is the latest operand for the ADD instruction, this is the also the time-slot that the instruction will be allocated.

The TDS tool strictly enforces all true data dependencies, as shown above with R4. In contrast anti-dependencies and output dependencies are effectively ignored. Since both these dependencies can be removed by renaming, the TDS tool effectively assumes perfect renaming of both register and memory locations.

The TDS tool has perfect knowledge of all memory addresses accessed by the program under simulation and is therefore able to simulate perfect memory disambiguation. This simple execution model with many of the restrictions on execution relaxed is used to compute the upper bounds of the available instruction-level parallelism and is equivalent to the Oracle model used in other studies. Additional restraints are then introduced to the model allowing the TDS tool to quantify the impact of restrictions introduced by instruction scheduling algorithms.

5.2 Representing Programs Within TDS

5.2.1 Principle Parameters

The TDS tool has a high level of parameterisation, allowing individual components of the tool to be switched on or off, to allow the study of a single effect or the combination of several models and parameters. The TDS tool parameters are grouped around 6 main groups : viewing of data structures and results, simulation, instruction latencies, code alterations, memory and breakpoints.

The TDS tool has facilities for viewing the structures used within the TDS to model caches and the program under test, allowing the user to study the information held in the tool; viewing of the instruction cache, procedures in the current program, the basic blocks that make up the procedures, the current symbol table, distributions of parallel instruction time slots are a few of these.

The simulation group of parameters control the 'execution' of the program and allow the user to specify what output is produced by the TDS tool. Ceilings to code motion can also be set here, these include ; entrance and exits to loops, procedure calls and returns and allowing specific code motion across the ceilings.

The TDS tool has a number of predefined latency groups, the HSA latency set, a Unit latency set and set of long latencies based on the Cydra 5 machine. The user can also set individual latencies for all the principle instruction types within given limits. For Branch and Stores these limits are between zero and ten. Loads can be set with latencies up to twenty six to allow the simulation of very slow memory accesses or from cache hierarchies. Division latencies can be set as high as forty cycles to emphasise the effects of very long instruction latencies. The other main instruction types are Multiplication and Arithmetic, which are allowed latencies up to and including twenty five and twenty, respectively.

The TDS tool also allows manipulation and alteration of the original program code. Under the code alterations menu the tool allows inlining of specific procedures, loop un-rolling, simulation of perfect inlining and combination of instructions. Some of these operations alter the code and place new copies of the code back into the instruction cache. These options cannot restore the code back to its original state without re-loading and should be used with caution.

The final set of parameters allows various methods of memory disambiguation to be selected for modelling. These range from perfect memory disambiguation through more restrictive models, to no disambiguation and allow the impact of the memory disambiguation techniques to be studied. This group also allows ceilings to placed on the memory references preventing them from executing out-of-order, or speculatively before the preceding branches have been resolved. Further restraints can also be placed preventing memory references from moving across procedure calls or out of their associated basic blocks.

The facilities offered by breakpoints will be covered in Section 5.6, Tracing facilities.

5.3 Loading the TDS Tool

5.3.1 Loading The Instructions

The TDS tool allows object files to be selected and loaded into the TDS tool. The traces used for simulation require that the instructions be held in an instruction cache that matches that of the HSA simulator to ensure that the cache addresses given in the trace file will be identical. The instructions are read in a line at a time from the object file, each line is identified, decoded and placed in an instruction record. Each record contains fields to hold the primary components of each instruction including attached labels, types, op-codes, destination operands, source operands and branching information. Any labels in the code are collected as the instructions are loaded and placed in a general symbol table.

Once the main instruction file is loaded, the general symbol table is scanned for known library calls. Any libraries identified are then also loaded. Currently the only standard C libraries supported by the TDS tool are printf and malloc, which contain instructions to make the appropriate system calls. These are again included to maintain a consistent model with the HSA simulator. As the object code is designed to be re locatable, all the branch target labels are then placed within the symbol table and the branch targets of every branch instruction can be then altered to map the branch target label to the correct instruction.

5.3.2 Data Items

As the TDS tool is not required to model and simulate a data cache, all data items, both initialised and uninitialised are collected and placed in a temporary data structure. These labels are then placed into the general symbol table to allow the TDS tool to display and identify them easily. As well as being placed into the symbol table, the data labels are placed directly into the memory map which is used to associate an availability time with each memory location so that memory references can be handled as operands.

5.4 Main Simulation Loop

The TDS tool performs most of its work within a main simulation loop. This loop controls and checks the processing of each instruction and the collection of information about the program under simulation. Although fairly complex, the loop consists of a number of distinct stages ; loading from the trace file, using the trace file to step through the instruction cache, checking operand availability for each instruction and allocating an execution time, checking for ceilings and breakpoints.

5.4.1 Trace Files

The TDS tool requires for each simulation a trace file of each benchmark executing on the HSA simulator, these can be generated on request. An Example of such a trace is given in figure 5.3.

```
B 2    127      S 128 3968     B 129 89       S 90   3840     S 91   3848
S 92   3852     S 93   3856    B 94  4        S 6    5568     B 8    95
B 96   18       S 19   3712    S 20  3720     S 21   3724     S 22   3728
S 23   3732     B 33   9       L 10  5568     S 14   5568     B 17   34
S 43   4164     B 46   33      B 33  9        L 10   5568     S 14   5568
```

Figure 5.3 - An example section of a trace file

The trace file consists of all the branches taken and memory references used by the program. The characters in the trace indicate the type of the instruction, 'B' for a branch, 'L' for a load and 'S' for a store. For a branch, the first number is the address of this branch in the instruction cache and the second number is the branch destination address within the instruction cache. This information is recorded into the trace when the program counter is physically altered in the branch unit of the HSA simulator. For a memory reference the first figure is again the current instruction cache address and the second is the effective address for the memory access. This information is recorded into the trace when the instructions are issued. All the traces files were generated on the single pipe model allowed by the simulator.

The TDS tool reads a line at a time and separates the line into a sequence of tokens, these tokens are then used to control the simulation.

5.4.2 Simulating the 'Execution' of a Benchmark

The program is assumed to begin executing at instruction cache address zero. The instruction at the current program counter is identified and a specific handling routine for each instruction type is called. Each source operand is checked, the latest operand time is associated with the instruction and a destination time is associated with any destination operands. For each instruction type that does not require information from the trace file, this will be as shown in Figure 5.2. At this stage it is assumed that any ceilings being modelled, have effected the operand availability times or the execution time of the instruction. A simplistic model of this is presented in the following section (5.4.3) and further information is given for each simulation model when its results are presented.

For branches and memory references a slightly different sequence of events is initiated. Branches are not guaranteed to appear in the trace file, they are only placed in the trace file by the HSA simulator, if they are taken. For the Stanford benchmarks used, the figure for branches not taken is around 25%. The TDS tool firstly checks if a branch is the next instruction it expects to see in the trace file. If so the current program counter is checked against the branch instruction address read in. If these match, the branch is taken and the program counter is adjusted to the branch target address, this new program counter is checked against the read in branch target cache address, to ensure correctness. The TDS tool has to also handle branches not appearing in the trace file and branches not matching in the trace file. In both of these cases it has to assume that the branch under examination is not taken and execution should continue at the branch's sequential successor.

Memory Loads and Stores, as well as providing effective addresses for memory references, are used to check the simulation is executing as expected. These instructions will be examined by the TDS tool in the same order as they were executed in the HSA simulator, the information they convey is used to maintain simulation correctness. The effective address read in are used to perform memory disambiguation within TDS. Each effective address is placed within a memory map and has the time of the memory reference associated with it. All memory references are checked against this memory map.

The TDS tool will continue stepping through the instruction cache until all the trace file has been processed and further instructions require trace file information, in which case an error has occurred, or until a TRAP instruction is executed, indicating the successful completion of the program.

5.4.3 Ceilings and Breakpoints

In addition to the instructions held in the instruction cache, the TDS tool also has a further data structure which models the program, its procedures and their associated blocks. The program is modelled as a complex data structure, but at the top level it can be reduced to a program name and a list of procedures. The procedures consist of pointers to further procedures, pointers to the instructions that make up the procedure via basic blocks and a back edge list for all the loops in the procedure. This higher level information allows the TDS tool to keep track of specific events, such as loop entrance and loop exits. Which, with the use of global ceiling variables, can be used to set ceilings for code motion that effect all the other procedures that implement the simulation. These ceilings can be for all instructions or for a selected specific type of instruction, most commonly memory references.

The TDS tool allows various trace methods to be used during the simulation. By far the most useful of these traces is the one controlled by breakpoint information. This allows breakpoints to be created for specific instruction counts or instruction cache addresses. When a breakpoint is encountered, the options to remove the breakpoint, trace a specific number of instructions, trace a specific number of branches and memory references, examine the memory map and examine register usage at this point are presented to the user. This allows valuable information to displayed in a controllable way. Tracing instructions shows which ceilings are being implemented, the availability of operands and the allocation of PITs to instructions as well as further information regarding register usage. This feature was used to debug the TDS tool and is left in as it provides an in-depth and useful tool to examine exactly what is happening and in what order at any stage. As yet there is no facility to enter the trace mode when an error occurs. This feature has been omitted because any errors that occur within the TDS tool are likely to be fatal errors. These errors are likely either to be a problem with the implementation or user error. Simple user error is the most common form, typically using object files and trace files that do not match.

Further trace facilities are available to the user. Outputting the trace file to a given file can be selected, presenting a more condensed form of the trace information. A tuple output file can be generated, this is contains highly condensed output, consisting of a collection of instruction cache addresses and their associated PITs. A branch error file can also be selected, storing information for all the branches not taken,. These outputs are a representative sample of the TDS tool outputs.

6 . Results

This section presents the results of removing many of the constraints placed on executing code sequences, to allow us to quantify the amount of ILP in non-numeric code. The effects of placing barriers to code execution and the impact they have on the available ILP are also presented. The results are displayed in terms of the speedup of a suite of benchmark programs running on the TDS tool compared to the same programs executing on a single pipe HSA model. Speedup is defined as the number of cycles taken by a scalar processor to execute the program divided by the number of cycles taken by a superscalar processor to execute the same program. The results obtained are then analysed to assess the cost, in terms of performance and impact on the available ILP, of these constraints, both as singular effects and as combinations of constraints. The full set of results are available in Appendix A.

6.1 Benchmark Programs

For this implementation of TDS, the small integer Stanford benchmark suite was selected. This collection of eight 'C" programs was designed by John Hennessey, to be representative of non-numeric code while at the same time being compact. The benchmarks are computationally intensive with high dynamic instruction counts. Under test, they perform very much like larger benchmark suites. All the benchmarks were compiled by the HSA gnu 'C' Compiler which targets the HSA instruction set. For this initial study, 'cut-down' versions of the benchmarks where used. The primary reason for using these cut-down versions was the ease of testing and debugging with the smaller dynamic instruction counts and also to avoid generating large trace files until TDS was more stabilised. These cut-down versions (indicated by a "cd" suffix on the benchmark names), still implement exactly the same algorithms as the full length versions but work on reduced problem sets, thus keeping most of the desired properties.

As shown in Table 6.1 each of the benchmarks consist of a small number of procedures (the average size is approximately 8). Each procedure has an average basic block size of around four instructions. Three of the benchmarks *permcd*, *towcd* and *treecd*, are highly recursive with a large number of procedure calls being made throughout the execution of the program. The average percentage of total instructions that are branches for these three benchmarks is about 17%, of which 54% are procedure call and returns. In contrast *bublcd*, *matxcd*, *puzlcd* and *queencd* spend over 80% of their time executing within loops. An average of 50% of the total number of lines in these benchmarks accounts for 88.4% of the total execution.

Program	**Num. Procs**	**Lines**	**Basic Blocks**	**Avg Size Of Block**	**Description**
bublcd	7	106	28	4	Bubble Sort an array of 50 Integers
matxcd	7	133	24	6	Multiplication of two 10 by 10 integer matrices.
permcd	7	97	25	4	Recursive computation of all the permutations for 5 elements.
puzlcd	8	589	145	4	Recursively solves a cube packing problem.
queencd	6	164	33	5	Recursive solution of the eight queens chess problem.
sortcd	8	145	35	4	Recursive quicksort of 100 elements.
towcd	12	222	47	5	Solves the tower of Hanoi problem for 7 discs.
treecd	11	219	54	4	Performs a binary tree sort of 100 elements.

Table 6.1 - The Stanford "Cut Down" Benchmark Suite

Four of the benchmark suite, *bublcd*, *matxcd*, *sortcd* and *treecd*, contain a mixture of Multiply and Divide instructions. Although these represent a small proportion of the dynamic instruction count, (the percentage of the dynamic composition for these instructions is 2% and 0.5%, respectively) the long latency of these instructions may seriously limit the potential for parallelism, by delaying subsequent instructions in the long dependency chains that exist in the benchmarks.

All the cut down benchmarks execute under 40,000 instructions, which means that while the individual speedups achieved are individually meaningless, useful interpretations can still be drawn by comparison across a range of results.

6.1.1 Composition of the Benchmarks

Program	**St**	%	**Ld**	%	**Arith**	%	**Mult**	%
bublcd	1377	(8.8)	2801	(18.0)	4334	(27.9)	100	(0.6)
matxcd	1516	(5.9)	3215	(12.6)	10760	(42.1)	1200	(4.7)
permcd	1723	(20.1)	1716	(20.0)	3192	(37.2)	0	(0.0)
puzlcd	2304	(6.5)	2876	(8.1)	12180	(34:5)	0	(0.0)
queencd	3157	(14.7)	3796	(17.6)	5631	(26.1)	0	(0.0)
sortcd	939	(7.1)	2044	(15.5)	4093	(30.9)	200	(1.5)
towcd	2892	(18.0)	3235	(20.1)	5808	(36.1)	0	(0.0)
treecd	2193	(9.2)	5103	(21.4)	7985	(33.6)	200	(0.8)
Avg.	2013	11.3	3098	16.7	6748	33.6	213	0.95
	Div	%	**Bool**	%	**Branch**	%	**Shift**	
bublcd	50	(0.3)	2698	(17.4)	2808	(18.1)	1373	(8.8)
matxcd	200	(0.8)	1330	(5.2)	1940	(7.6)	5390	(21.1)
permcd	0	(0.0)	420	(4.9)	1314	(15.3)	212	(2.5)
puzlcd	0	(0.0)	6084	(17.2)	6559	(18.6)	5347	(15.1)
queencd	0	(0.0)	3512	(16.3)	3849	(17.9)	1604	(7.4)
sortcd	100	(0.8)	1701	(12.9)	2392	(18.1)	1760	(13.3)
towcd	0	(0.0)	794	(4.9)	2429	(15.1)	922	(5.7)
treecd	100	(0.4)	2437	(10.3)	5123	(21.6)	595	(2.5)
Avg.	56	(0.3)	2372	(1.1)	3302	(16.5)	2150	(9.6)

Table 6.2 - Composition of Benchmarks by Instruction Type

Table 6.2 gives a distribution for each instruction type for the benchmark programs based on the dynamic instruction count. If shift, multiplication, divide and arithmetic instructions are grouped together to form a super-group of ALU operations, these comprise 44.5% of all the operations executed. This figure compares favourably with those reported by Gross [Gross88] for the MIPS instructions set usage, where ALU operations typically consisted of between 40-50% of all operations.

This is also true of loads and stores within the benchmarks which comprise 28% of the total, Gross reported that loads and stores instructions are between 25-30% of the total. The MIPS work was performed on the full length versions of the Stanford benchmark set and several much larger benchmarks. Typically these benchmarks running for millions of instructions and this shows that the "cut-down" Stanford set of benchmarks compare favourably. Although, some studies have shown that small integer benchmarks do not contain many pairs of ambiguous memory references, this was not found in this initial study.

6.1.2 Branch Instruction Profile

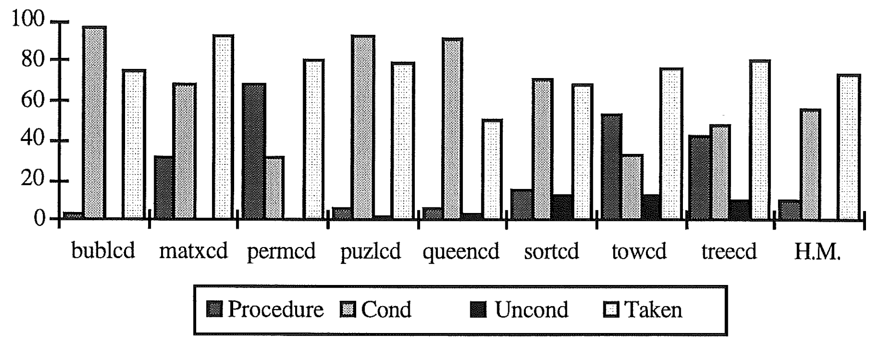

Figure 6.1 - Distribution of Branch Types and Branch Taken Figures.

Figure 6.1 gives the distribution of branch types for the Stanford cut-down benchmarks, in terms of the percentage of branches that where procedure calls and returns (Procedure), conditional (Cond) and unconditional (Uncond). The high proportion of conditional branches in the benchmark suite is a good representation of the complex dynamic control flow which is typical of non-numeric code. A further good indication is that of branches taken. Although in some benchmarks, notably matxcd, a high proportion of all branches are taken, the harmonic mean is around 73%, leaving 27% of branches untaken. This figure correlates to the figures reported by Gross [Gross88] in which taken branches are less than 20% of the dynamic instruction count. Based on the harmonic means, taken branches are around 12% of this count for the cut-down benchmarks.

6.1.3 Basic Block Statistics

Benchmark	**Static Average**	**Dynamic Average**	**Avg. Branch Distance**
bublcd	3.8	4.5	5.5
matxcd	5.5	12.3	13.1
permcd	3.9	6.1	6.5
puzlcd	4.1	5.0	5.4
queencd	5.0	5.4	5.6
sortcd	4.1	4.7	5.5
towcd	4.7	6.3	6.6
treecd	4.1	4.5	4.6

Table 6.3 - Average Basic Block Sizes, Branch Distances

Table 6.3 presents the average basic block size for each program. The dynamic average is calculated by dividing the total number of instructions executed by the total number of basic blocks. The table also presents the average distance between branches of each program. This distance is defined as the total number of instructions between branch instructions, including the branch instruction itself. This is related to the average basic block sizes but not identical because basic blocks need not be separated by a branch instruction.

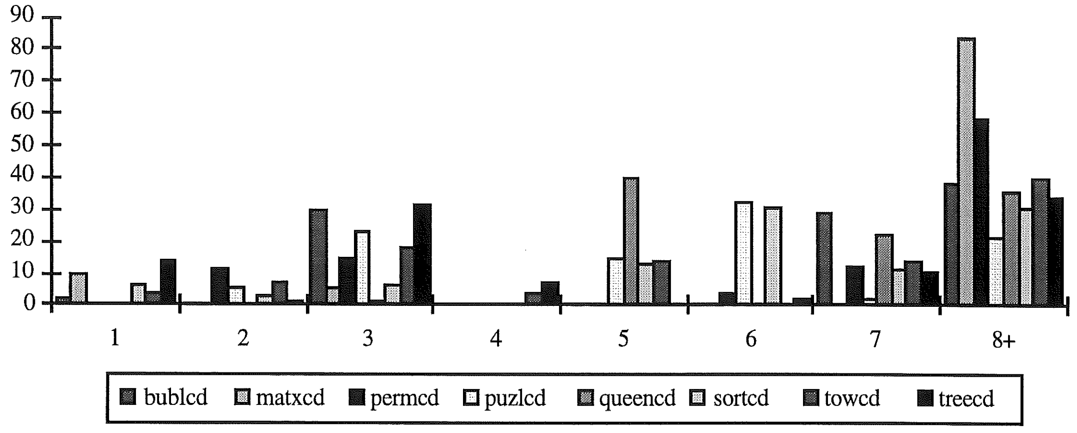

Figure 6.2 - Dynamic Run Length Distribution of Taken Branches

For general purpose code, the instruction runs are quite short. Figure 6.2 shows the distribution of run lengths for the Stanford benchmarks. The run-length is determined by the number of instructions between taken branches. Branches not taken are counted as part of the run and every time a branch is taken a new run length count is started. This is measured as the program execution is simulated by the TDS tool and the output is optional. The average run length for the Stanford suite is around 6, but this is aided by the large number of run lengths of eight instructions or more. The size of these run-lengths has a critical impact on instruction fetching.

Harmonic means (H.M.) are used throughout this study as well as arithmetic means to avoid distortion in the reported speedups by applying larger weightings to the programs reporting the smaller speedups.

For a set of N programs, each with a speedup S_n the harmonic mean of all speedups is: Harmonic Mean = $N \,/\, (\Sigma_N \; 1/S_n)$.

The effects of this distortion can be seen in a number of the results reported. For example figure 6.3, *puzlcd* has a considerably greater speedup than any other of the benchmarks, the arithmetic mean reported here is 35.1 but the harmonic mean is only 21.5. The arithmetic mean does not accurately reflect the speedups for the whole benchmark suite and is presented to allow comparisons only.

6.2 Oracle Base Model

As a base model for comparison, the TDS tool initially ran the benchmark programs with no constraints being modelled. This Oracle model represents a processor with perfect memory disambiguation, infinite resources and perfect branch prediction. Therefore the execution time of each program is dependent on the true data dependencies (which represent the flow of data and information) present in the programs only.

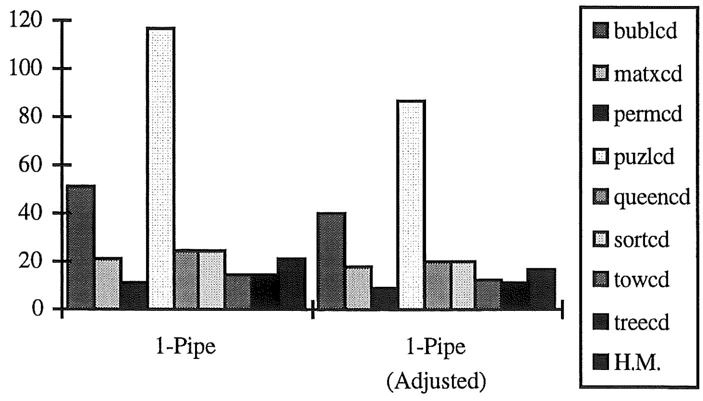

Figure 6.3 - No Ceilings Modelled

Figure 6.3 shows these execution times as speedups over a single pipe HSA model. This single pipe model has a fetch rate of one instruction per cycle, only one functional unit of each type and imposes the same set of latencies as the HSA parallel models. The actual latencies used are presented later in Table 6.4. The total number of machine cycles that each benchmark requires to execute on this single pipe machine is collected from the HSA simulator and used to calculate this speedup. The set of figures from Figure 6.3, identified by the '1-Pipe' legend, show the speedup of the TDS tool over these base figures. The second set of figures, identified by the '1-pipe(adjusted)' legend, are the speedups based on the set of base figures, with all the zero issue cycles due to branches removed. The HSA architecture uses branch delayed regions in favour of branch prediction to minimise the impact of control flow. As the base figures are taken from the HSA Simulator executing unscheduled code, each branch instruction has an associated empty branch delay region. This impacts unfairly on the overall machine cycle counts, as there is an additional penalty associated with each branch. These zero issue cycles have been removed in the 1-Pipe(Adjusted) field, for all the branches taken. It is this second set of base figures that will be used throughout the rest of the results section.

For these set of figures the H.M. of the speedups is 21.5 and 17.4 respectively, with a range between 9.5 for *permcd* and 87.2 for *puzlcd.* The available instruction level parallelism ranged from 9.6 for *permcd* to 99.3 for *puzlcd* with an average around 29 instructions being executed per cycle. These Oracle figures represent an upper bound for the ILP and will be used to quantify the impact of the limitations introduced within this study.

6.3 Procedure Call and Returns

Typically, instructions within a procedure are never issued until all the preceding instructions have completed and the code after procedure call is not executed until all the instructions in the procedure have completed. The TDS tool simulates these limitations by placing ceilings on the entry and exit from procedure calls preventing code from executing any earlier than these barriers. The impact of recursive procedure calls have been included in this model.

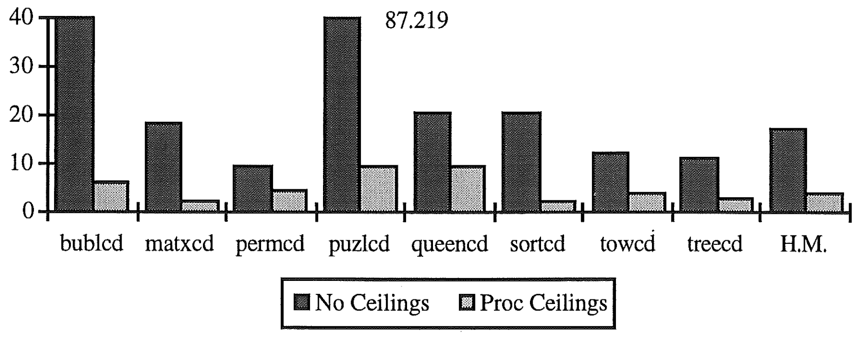

Figure 6.4 - Effect of Ceilings on Procedure Call & Return.

Figure 6.4 shows the effect of ceilings on the procedure call and return. The H.M. is 4.072, with a range of 2.238 for *matxcd* to 9.986 for *queencd.* This is a reduction of 13.266 in the overall speedup compared to the base model. The three highly recursive benchmarks *permcd*, *towcd* and *treecd* all suffer an appreciable decreases in performance. All the benchmarks suffered a significant reduction in performance and this can be attributed to several reasons. Firstly, the available parallelism is limited to the size of the average procedure. For the Stanford benchmarks the average procedure size is around 25 instructions (*puzlcd* has the largest static average and *matxcd* the smallest). This limitation is compounded by the large number of small procedures within some of the benchmarks. Data dependencies introduced by stack frames further reduce the ILP within the procedures. Secondly, several of the benchmarks are highly recursive. The barriers on procedure entry and exit prevent parallel execution of successive recursive procedure calls. Finally, many of the loops in the benchmarks have procedure calls embedded within them, preventing loop iterations from executing in parallel. The overhead of procedure calls to ILP has is therefore very significant.

6.3.1 Procedure Inlining

Procedure entry overhead can be reduced by procedure inlining. To study this further, two further models were developed. The first model, selective procedure inlining attempts to inline procedures based on a simple set of heuristics ; the size of the procedure, how complex the control flow is within the procedure and if the procedure contains further procedure calls. Selective procedure inlining was configured to inline procedures which contained less than 20 instructions and which made no further procedure calls. SP dependencies were removed, along with the associated saving and restoring of preserved registers. For most of the benchmarks, this model was able to find and inline at least two procedures. The second model is that of perfect inlining. This is based on the work of Lam [Lam92]. All manipulations of the SP are removed along with the BSR calls themselves. Loads and Stores associated with the procedure's stack frame are also removed, to simulate the effects of instruction scheduling collapsing the stack frame save and restore instructions, once the preceding dependencies had been removed.

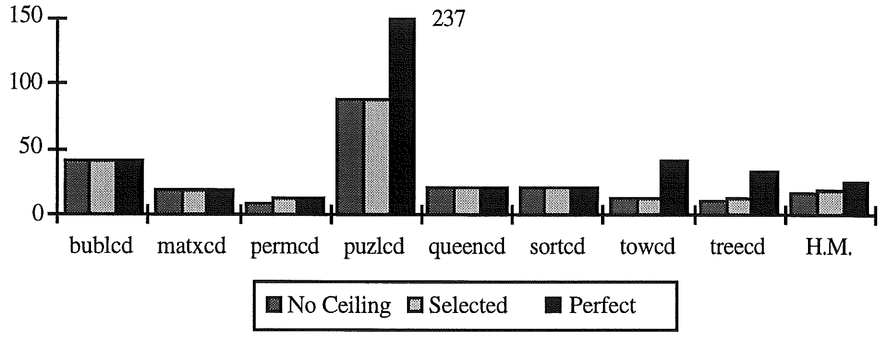

Figure 6.5 - Effect of Procedure Inlining

Figure 6.5 presents these results. For the selected procedure inlining model, the average number of dynamic instructions removed is approximately 400 per benchmark. The removal of the procedure overhead gives a performance increase of around 9% across the benchmarks. The effects of removing BSR calls from within loops is most notable in *permcd* where inlining the two procedure calls within its dominant loop, gave a performance increase of approximately 44%.

The figures for perfect inlining represent an upper bound to what is theoretically possible, using procedure inlining. Because all the stack frames have been removed, the chains of dependent loads and stores which may limit subsequent code motions, have been eliminated. This simulates the condition where the procedure has been inlined and the scheduler succeeds in removing all the values previously saved across the procedure call and return in registers. The improvement is most dramatically shown in *puzlcd* where a massive speedup of 237.0 is observed. The H.M. for this set of figures is more conservative, at 26.2 compared to the no ceilings base model of 17.3. Here the flow of information in the procedures, is almost entirely within the registers themselves.

6.3.2 Direct Recursion Inlining

To asses the impact of procedure ceilings on recursive procedure calls, the TDS tool can identify directly recursive calls. These procedures that directly call themselves become candidates for dynamic inlining. This models the effect of not producing a new stack frame for every direct call. This inlining is not as comprehensive as static procedure inlining. For each direct recursive procedure call, ceilings will not be set on procedure entry and exit. Any SP manipulations are identified and ignored. This helps to partially collapse the stack frame construct, but the associated state restoring loads and stores are still executed.

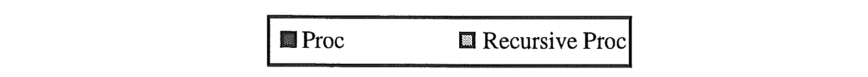

Figure 6.6 - Inlined Recursive Procedure Calls

Figure 6.6 presents the results for inlining direct procedure calls, displaying the figures for ceilings on procedure call and returns to allow comparison. On average 36% of the benchmarks' procedure calls and returns are removed, with *queencd* having 96.6% of all its procedure calls and returns inlined. The change in speedups exhibited across the suite, when compared to the procedure call and return figures, range from, 0.0 for *bublcd* and *matxcd* (which had no direct procedure calls) to +10.54 for queencd. Although *queencd* had a dramatic speedup increase, the other recursive programs did not exhibit such large performance increases. Both *sortcd* and t*reecd* had much smaller performance increase of around 13%, while *permcd* managed an increase of around 20%. By not handling recursive procedure calls as a special case, would cause a performance loss, for this benchmark suite, of 10.5%.

Program	**No. of Proc Calls Inlined**[8]	**% of BSR& MOV's** [9]	**Change in speedup**
bublcd	0	0	0.0
matxcd	0	0	0.0
permcd	205 (410)	57.0	1.0
puzlcd	19 (38)	10.7	0.7
queencd	112 (224)	96.6	10.5
sortcd	88 (176)	45.6	0.3
towcd	126 (252)	19.2	0.2
treecd	684 (1368)	62.9	0.4
Total	1234 (2468)	292	13.1
Average	154.3 (308.5)	36.5	1.6

Table 6.4 - Recursive Procedure Statistics

Table 6.4 presents a summary of the results. *queencd* reported the largest impact as its dominant loop contains a recursive procedure call. When this call is inlined, the barrier to exploitation of ILP caused by this procedure call is removed allowing the code from across the call and the code from within the call to be considered for parallel execution. The *treecd* benchmark reports a much lower speedup than the number of inlined procedure calls would suggest, as its control flow is dominated by recursive procedure calls. These procedures though have complex control flow and call subsequent procedures thus reducing the increase in ILP.

6.4 Effect of Loop Ceilings

In this section, the effects of barriers within loop execution are examined. For the eight benchmark programs, the average number of instructions within a loop is 36% of all instructions but these count for 58% of all instructions executed[10]. These figures are lower than the commonly quoted figure of 90%, as several of the benchmarks notably *permcd*, *towcd* and *treecd* (the recursive benchmarks) have loops which account for under 12% of the total number of instructions executed.

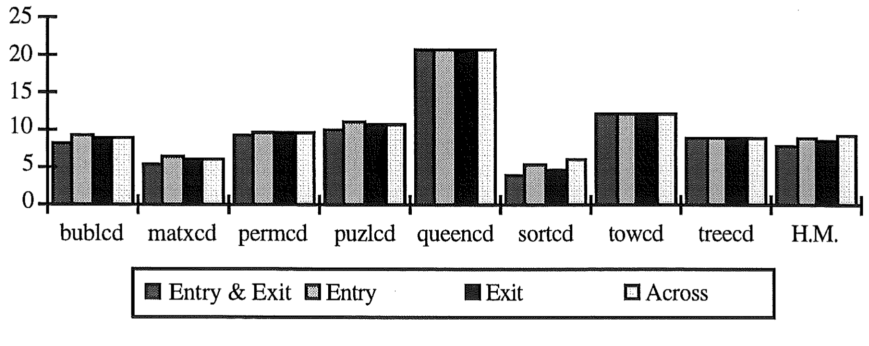

Figure 6.7 - Effect of Loop Ceilings On Code

At the other end of the scale bublcd, *matxcd* and *puzlcd* spend over 91% of their time executing loops. Firstly, the impact of preventing a loop from executing until all the code proceeding the loop has

[8] Shows the number of procedure calls and then the figure including return from procedures removed.
[9] Percentage based on programs execution count.
[10] See Appendix A - Set 47 for full results.

completed, as outlined in the limitations section is studied. The effects of reducing this barrier by allowing subsequent code motion is then considered, these code motions include code motion across loops, code motion into loops and code motion out of loops. Figure 6.7 presents the results for these four barriers with no other code restrictions being modelled and with the normal HSA instruction latencies.

The 'Entry & Exit' category represents the most severe model of the group of four. The benchmarks with the largest number of loops, many of these nested, had the greatest performance degradation. The performance of *bublcd* was reduced by 32.5, *puzlcd* by 77.9 and *sortcd* by 16.9. The main cause of this degradation was the effect of these ceilings on nested loops, where an outer loop enclosed an inner loop. The ceilings on entry and exit, effectively serialised the execution of the outer loops. The *queencd* benchmark was unaffected by these loop ceilings. The speedup was reduced by only 0.3, as its execution is dominated by a single simple loop which accounts for over 77% of its execution. *permcd* reported the smallest reduction in speedup of 0.2, this programs has no nested loops and relies on recursive procedure calls for the bulk of their execution. This shows that the assumption made when scheduling a loop body have a significant degrading factor on performance.

The 'Entry' legend shows the effect of allowing code to percolate into the loop but no higher. Here a single ceiling is set at the loop entrance. Initial expectations of an improved performance, as one of the barriers to code motion had been removed were not proved. The H.M. of improved speedups is 0.1 with a range of 0.02 for *treecd* to 1.2 for *sortcd.* This increase is limited for several reasons, primarily there is only a small pool of additional instructions that can take advantage of this model. The performance benefit occurs due to overlapping of the encapsulating loops execution with the encapsulated loop's body. The execution of code from within the inner loop is unaffected (Inner most loops, account for most of the instruction execution. In *bublcd* 89% of the program's execution occurs within an inner loop.). Below the innermost loop bodies, there are either a small number of instructions that deal with a backward branch, in the case of nested loops or instructions that rely on data set within the loop body. This limits severely the exposure of additional ILP within this model. This is often compounded by dependencies on the SP In ideal circumstances, for an unconstrained superscalar model, the performance benefit is approximately 10%. However, this figure is achieved with perfect memory disambiguation and infinite resources and it may prove to be too difficult to repeat this is in a realisable superscalar model. Any impacts of this code motion into loops lengthening inner loop execution has not been studied here.

The 'Exit' category reports the effect of code motion out of loops. In this model, code from below the loop body is prevented from executing until after the loop has completed. The H.M. of the increases in speedup is 0.1 with a range of 0.0 for *queencd* to 9.2 for *matxcd.* Again only a small number of instructions are candidates for this code motion, loop carried and SP dependencies prevented most code from percolating higher.

Finally the 'Across' set of figures, quantify the effect of allowing code motion across loops. The TDS tool places execution ceilings only on the loop code preventing it from moving higher than the loop entrance. All other instructions are allowed to percolate as high as possible. With this model only *sortcd* shows any increase in performance, the increase in speedup is 2.1. The impact on performance is very similar to code motion into loops. Code within the loop bodies is often serial in nature, preventing substantial code motion across the loop due to dependencies within the loop body. Typically after the exit from an inner loop, there are several register restore operations which are dependent on the SP and a number of iteration control instructions which have limited scope for concurrent execution, without code transformations. It has beneficial effects for a small numbers of instructions in the outer loop, effectively allowing most of them to be executed in parallel.

The impact of these loop-ceilings for ILP is on average a 64% decrease in the performance when compared to the Oracle base model. As expected, these barriers to code motion do have a critical impact on performance.

As a final measure of loop assumptions, the effect of serialising all the loops was studied. This would occur if there was no concurrent execution of loops, if each loop iteration could only be executed if all proceeding iterations of the loop had completed.

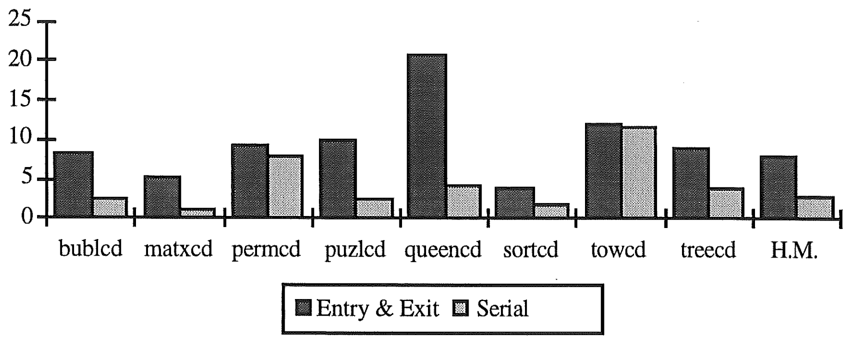

Figure 6.8 - Effect of Serialising All Loops

Figure 6.8 demonstrates the dramatic impact that this extreme case has had. Using the figures for entry & exit as a guide, the H.M. is 2.8, a reduction of 5.0. The H.M. of the reduction suffered by each benchmark is 1.6. As expected the benchmarks with the greatest dependency of loop execution (*bublcd*, *puzlcd* and *matxcd*) have been the most adversely affected. There has been almost an order of magnitude decrease over the base model figures, showing that the loop iteration rate is a critical limitation to parallelism. This is effectively a simulation of loop execution where the loop iteration rate is equal to the critical path through the loop body. For the code presented above, the loop iteration rate is controlled by the data dependencies and their associated instruction latencies within the loop.

6.4.1 Loop Construct Removal

The loop construct removal model studies the effects of collapsing the dependencies introduced by loop indices by simulating loop unrolling. The TDS tool aims to remove the effects of code generation for scalar machine. For each loop, all the registers which are incremented once per iteration by a constant are identified. TDS uses iterative data flow analysis [Aho86] on each of the benchmark programs to achieve this. TDS then identifies and marks all incremental instructions, comparisons of loop indices and branches based on the result of these comparisons. These instructions are then ignored when they are encountered within the trace. The technique implemented identifies the majority of loop indices and simulates near perfect loop unrolling.

Figure 6.9 presents the results for the number of instructions executed before and after loop construct removal. The average number of instructions removed per benchmark is around 2566.8, or about 13% of the total dynamic instruction count.

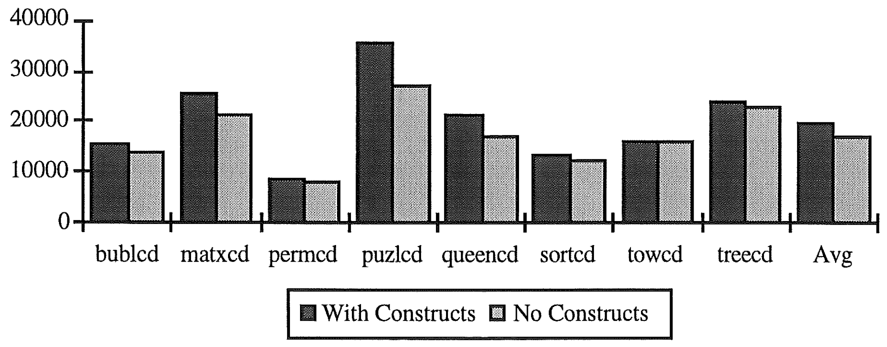

Figure 6.9 - Instruction Exec. Counts For Benchmark and After Loop Construct Removal.

Figure 6.10 presents the speedup for the benchmarks after loop-construct removal. They are presented with reference to procedure ceilings, as without any limitations apart from those inherent in the code, loop construct removal made no visible impact[11].

[11] See Appendix A - Set 7. Although the number of instructions executed by the benchmarks has decreased by around 13%, no change visible in the reported speedups.

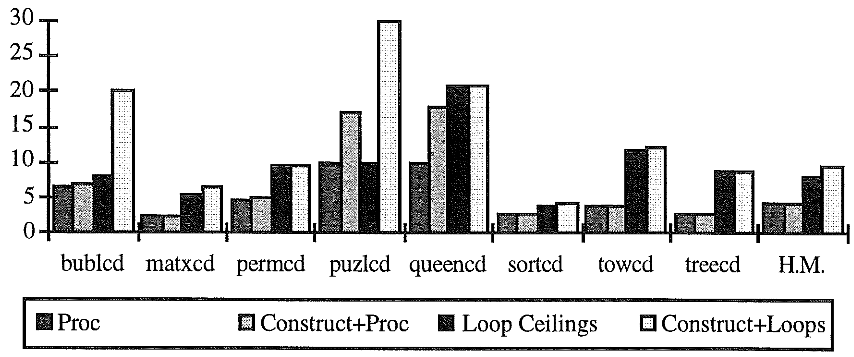

Figure 6.10 - Effect of Loop Construct Removal For Benchmarks.

For Figure 6.10 the impact can be clearly seen, the harmonic mean for loop construct removal with procedure ceilings is 4.4, around 8% faster than when the loop components are not removed. When compared to the results for loop ceilings the HM increases to 9.6 around 22% faster. These figures suggest that loop transformations to remove the effect of these unnecessary serialising constraints should be considered.

6.5 Memory

6.5.1 Memory Disambiguation

Up to this point all results have been presented with a perfect disambiguation model, the effect of ambiguous memory references has been removed. For a statically scheduled architecture such as HSA, this is not a realistic model , the majority of the address components are held within registers and will therefore be unknown at the time scheduling will be performed. The models used so far have been further boosted by the use of results forwarding. If a load follows a store to the same location, it is modelled as being generated at the same time (if no other ceilings are in place and data dependency considerations have been met) as the store. Stores are forced to wait for any preceding conflicting memory references to complete and loads are allowed to percolate as high as they can. This means that a large number of memory address disambiguations would have had to been resolved, which is unlikely in a model based on static dependence analysis.

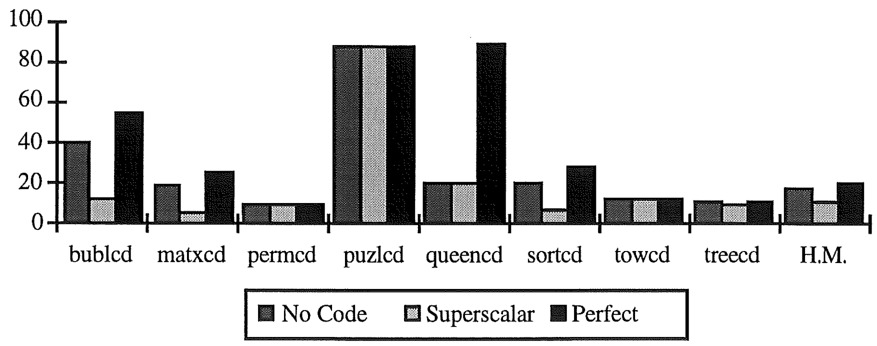

Figure 6.11 - Effect of Memory Model on Performance

Figure 6.11 presents the results of different memory disambiguation models for the benchmark programs. The 'No Code' figures, represent the base model which has a perfect memory disambiguation model, along with the result forwarding described previously. In this model stores can complete out-of-order and can bypass all memory references apart from the references to the same effective address.

The 'Superscalar' model is based on Johnson's model [Johnson91]. Here loads are still perfectly disambiguated, but stores are forced to complete in order and only after all preceding instructions have completed. This allows insight into the effect of allowing loads to by pass stores which is a more realistic model. Here the H.M. for the speedups has been reduced to 10.7, with *bublcd, matxcd* and *sortcd* being effected significantly. A possible cause is that these benchmarks and *treecd* to a lesser extent, contain long latency divide instructions. Data dependency chains involving divide instructions would greatly effect the times at which the stores at the end of a procedure or loop iteration could execute. These stores could then delay subsequent loads from the same location.

The final model in this group is the 'Perfect' memory model, this represents an idealised cache, with memory latencies of zero. Surprisingly, the benchmarks with greatest percentage of memory references in their dynamic instruction count, *permcd*, *treecd* and *towcd* did not exhibit drastic speedups. Only *queencd* reported a sizeable increase in performance of 67.9. The HM for this model was 20.5, an increase of 2.5 over the Oracle base model.

Figure 6.12 presents the results for the most restrictive memory models.

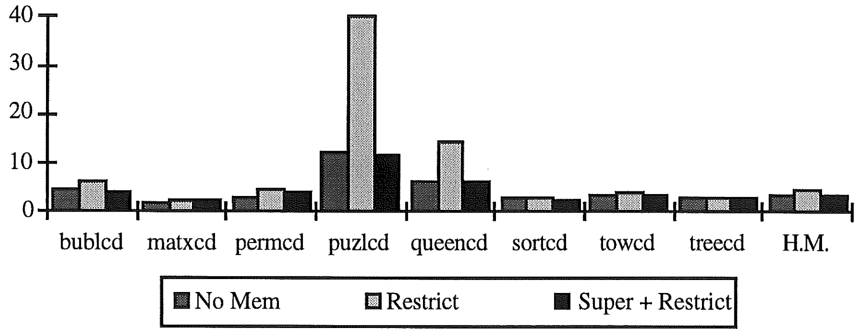

Figure 6.12 - Restrictive Memory Models

The first model, 'No Mem' represents a processor with no memory disambiguation. Here loads and stores are executed in program order although result forwarding is still used. The H.M. is reduced considerably from 17.3 for perfect disambiguation to 3.6. The programs least effected by this are permcd, queencd, towcd and treecd again, suggesting that their execution is not limited by memory references.

The 'Restrict' model places a further memory reference ceiling on the execution model. A barrier is placed to prevent memory reference motion across procedure calls within the TDS tool. In a realistic model, a scheduler would often be unable to move a memory reference across a procedure call, as the call would introduce many new memory references. This additional memory references would be very difficult to statically disambiguate successfully. This is especially problematic, if a procedure stored and restored a large number of items onto its stack frame. The problem of nested procedure calls would also impose significant problems and prevent most memory references being percolated across the procedure call. The impact of this ceiling reduced the H.M. to 4.6. The principle cause of this reduction is the result of constraining the load operations to execute later. In the 'No Mem' model, the loads could percolate up to the top of a procedure, where the preserved register stores onto the stack frame would prevent then floating higher. In this new model, memory references are often limited by further procedure calls within the current procedure, delaying their execution further. This has further repercussions, delaying any instructions dependent on the destination result of the load. These barriers effectively re-introduce the effects of the procedure call and return ceilings for memory references.

The effect of preventing load reference motion is further illustrated by the final set of figures, here the superscalar disambiguation model has this added constraint. The Harmonic mean of the speedups is reduced from 10.7 to 3.8.

6.5.2 Speculative Writes

In a realistic processor, stores cannot be issued until the immediately preceding conditional branch has been executed. If speculative issue was allowed, the store could only be promoted as far as the immediately preceding relational instruction. The TDS tool quantifies the effects of these limitations by placing a memory ceiling at either the last conditional branch or at the last relational instruction controlling a branch to be executed. Although this would only a save a single cycle on an HSA, the cumulative effects of many such speculative writes may produce a reasonable saving in execution time.

Figures 6.13 and 6.14 present the results for these ceilings. The first figure shows the effect speculative writes without any other constraints and then with a restrictive memory model. The impact of allowing

Speculative writes increases the H.M. speedup by only 0.004 for the benchmark suite. The restrictive memory reference model is used to prevent memory references from executing before the immediately executed procedure call. The impact is as expected but the performance benefit of speculative writes is diminished to only an increase of 0.02 for the H.M.

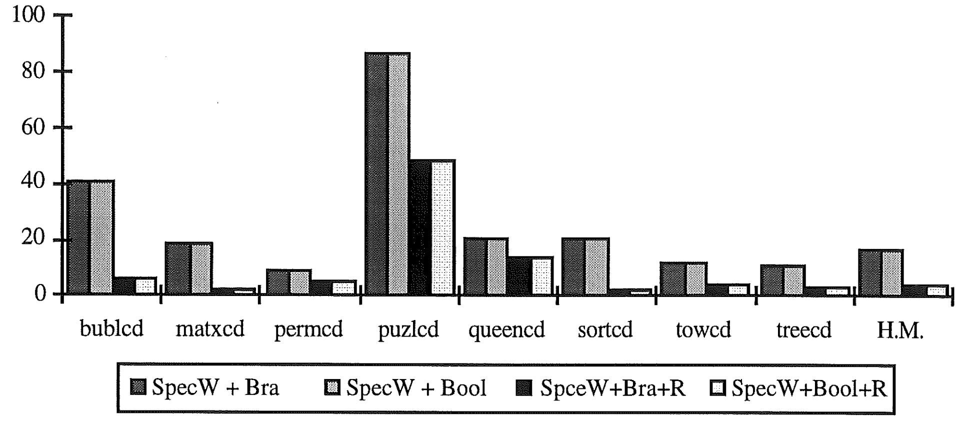

Figure 6.13 - Speculative Writes

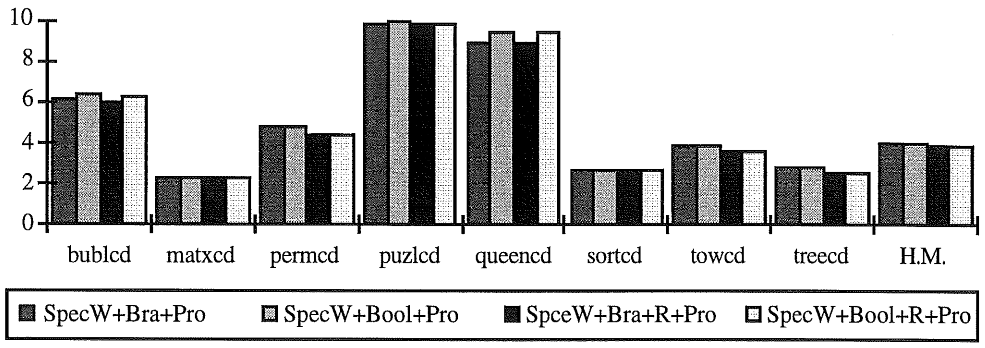

Figure 6.14 - Speculative Writes - With Procedure Ceilings

Figure 6.14 presents two further sets of results for progressively more restrictive models. The first set of figures are for speculative writes under a procedure ceiling model. An increase in the H.M. of the speedups of 0.03 is recorded under this model. The second figures are for a restrictive model using procedure ceilings as well. Again a very small performance increase of allowing speculative writes of 0.03 is noted. These figures are as expected. In the Stanford benchmarks, the small number of stores within loops and that these stores do not significantly block the motion of subsequent memory references means that any impact on stores is not passed on to the performance of the benchmarks.

Further work with memory models[12] show that even with limited or no memory disambiguation, where stores will block the motion of subsequent memory references, the largest reported change of allowing speculative writes was 0.6. These figures suggest that allowing speculative writes will not gain significant performance advantages for a more constrained model such as HSA.

6.6 Instruction Latencies

6.6.1 Adjusting Instruction Set Latencies

To allow comparison of the figures reported for this TDS tool to other limits of ILP studies, the instruction latencies used can be adjusted. Unit latencies were modelled, this avoids the constraints placed by using real instruction latencies and allows the upper limit for ILP to be approximated. The effects of long latencies are also studied by taking the worst-case figures for the Cydra-5 machine [Beck93], the most noticeable instruction latencies here are the load instruction which can take up to 20

[12] See Appendix A - Sets 37-42.

cycles and the Arith. latency, the most frequent instruction type executed, which is 5 cycles. The complete latency sets are given in Table 6.5.

Latency	Unit	HSA	Long
Load	1	1	20
Store	1	1	1
Branch	1	1	3
Arithmetic	1	1	5
Division	1	32	22
Multiply	1	3	5
Boolean	1	1	2
Shift	1	1	1

Table 6.5 - Latencies Used

Figure 6.15 presents the results for these two instruction latency sets at opposite ends of the scale, from theoretical to a realised, though long latency implementation. The Unit latency set's H.M. is increased to 19.0, suggesting that the longer latency instructions do inhibit the exploitation of ILP (although the increase in Speedup is only 1.6). Unsurprisingly the much larger set of latencies, has a detrimental speedup. Apart from *queencd*, which suggest that the majority of load and arithmetic instructions are not part of its critical path. The H.M. for the long latency figures is 4.1, ranging from 1.6 for *permcd* to 17.3 for *puzlcd.*

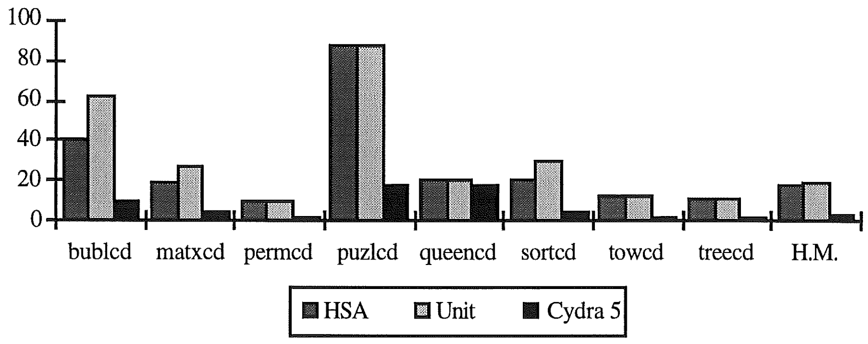

Figure 6.15 - Varying Latencies For Model

6.6.2 Minimum Division

To investigate the impact of the long latency instructions, as suggested by the results in the previous section, the impact of the very long latency HSA divide instruction is evaluated. It has been suggested that these long latency instructions delay further dependent instructions and therefore lengthen the critical path.

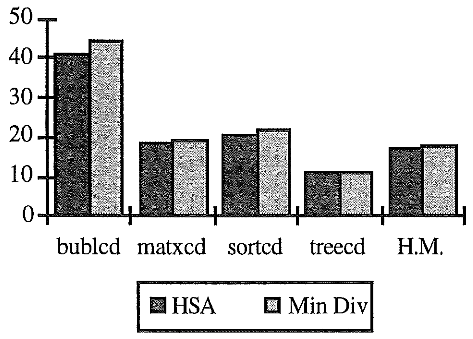

Figure 6.16 - Division Latency = 1.

The large latency also may cause problems for scheduling as the latency region associated with this instruction, will need to be filled with code to minimise the subsequent performance loss. The HSA divide instruction has a latency of 32 cycles and therefore should have a measurable impact on the model's performance. The four benchmarks with divide instructions (bublcd, matxcd, sortcd and treecd) were run with a divide latency set to a single cycle.

The H.M. of the speedups reported in Figure 6.15 is 17.6, an increase of 0.2. Surprisingly, *bublcd,* which has the smallest number of divides reported the largest increase of 3.5. These figures suggest that the long latency of the divide instruction does not critically limit the performance for these benchmarks. The benchmarks have though only a very small number of divides, under 1% of the dynamic instruction count, so this still may be an important factor when a more significant proportion of divides is present. As reported earlier, the long dependency chains created by divides instructions still have an impact on memory references, which will be magnified by more restrictive memory models.

6.7 Instruction Combining

HSA uses three input functional units and its instruction set takes advantage of this by providing 'Combined' instructions. To evaluate the use of combined instructions, the TDS tool attempts to aggressively combine as many instructions as possible. The combinations available are defined within instruction classes allowing assessment of the combinations between single instruction types, as well as more aggressive multiple combinations. The main combinable groups within HSA are Arithmetic, Multiplication, Shift, Relational and Branch. The combinations are simulated within TDS at run time. This dynamic combining removes the control dependency that would hinder static evaluation and exposes considerably more instructions to combining. This dynamic combination within TDS is modelled to generate an upper bound on the performance increase.

Each instruction is examined as its executed by the TDS tool. If it is part of the currently selected Combinable Instruction Group (CIG) and it has not been combined with a predecessor instruction, its is selected for combining. The instruction's destination register is marker as a COMBINE type and has the time of the availability of the instruction's source operands associated with it. If a subsequent instruction is processed that uses this destination register and the instruction is also a member of the CIG combination is simulated. Instead of using the destination register, the availability time associated with the destination register is used.

```
I1:    MULT   R4,R4,1309           I3:    ADD R5,(R4*1309),13849
........
I2:    ADD    R5,R4,13849
```

Figure 6.17 - Combining HSA code.

In Figure 6.17, there is a true data dependency between I1 and I2, on R4 . This would normally prevent I2 from executing until at least three cycles after I1 (ignoring any result forwarding). With combination selected, as I1 is being examined, the combine processes marks (subject to the constraints outlined above) R4 as a COMBINE type. R4 also has associated with it the latest source operand time (in I1, it would be the time R4 was previously available). When I2 is subsequently processed, as R4 is examined it will be identified as a combinable result and the two instructions will be combined to form a new instruction I3. I3 can then be executed in parallel with I1. R5 is now generated much earlier and may allow early execution of dependent instructions. I2 can then be removed. If live variable analysis is then used and if R4 has no further uses, I1 can be removed also. This final stage is omitted in the current implementation of TDS.

A limited number of instructions are selected for combination within this implementation of TDS; these include Arith, Shift and Mult instructions. The full list of currently combined instructions is available in Appendix B.

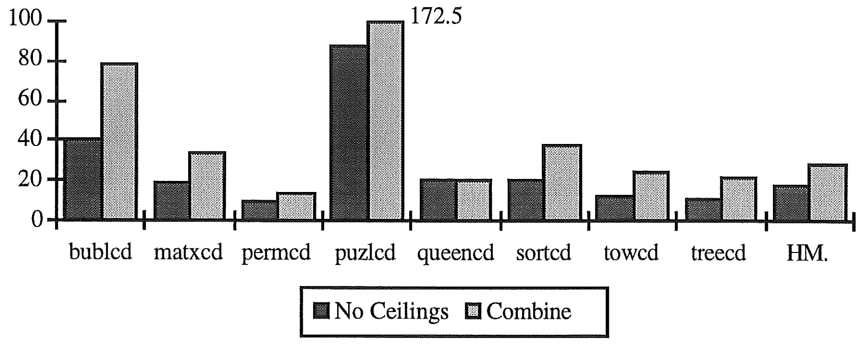

Figure 6.18 - Combining With No Restrictions

Figure 6.18 presents the results for combining when no restrictions are being modelled. 16.8% of all instructions executed in this model were successfully combined, with matxcd having the largest percentage of combined instructions, 33.4%. All but one of the benchmarks reported a significant speedup over the base model figures[13]. The average speedup increase across the benchmarks, was 22.9 and the change in Harmonic Speedups was 11.2. The most surprising observation was that many of the combinations were a direct result of SP manipulations within the procedure stack frames. The instructions incrementing and decrementing the SP were a prime candidates for combination, being simple integer Add or Sub instructions. As combining is performed dynamically, these SP manipulations are also combined between procedures.

Figure 6.19 presents the figures for combining with procedure entry and exit ceilings. The impact of combining is still very notable on this restricted model. Every benchmark reported a speedup over the Procedure Entry and Exit Ceilings figures, with an average speedup increase of 2.1. The ceilings on procedure entry and exit prevented substantial collapsing of the stack frame as seen in Figure 6.18 and consequently combining has a reduced effect. Further procedure calls within procedures also prevented much combination across procedure calls. The harmonic mean for the performance increase, for all the benchmarks, is 1.0.

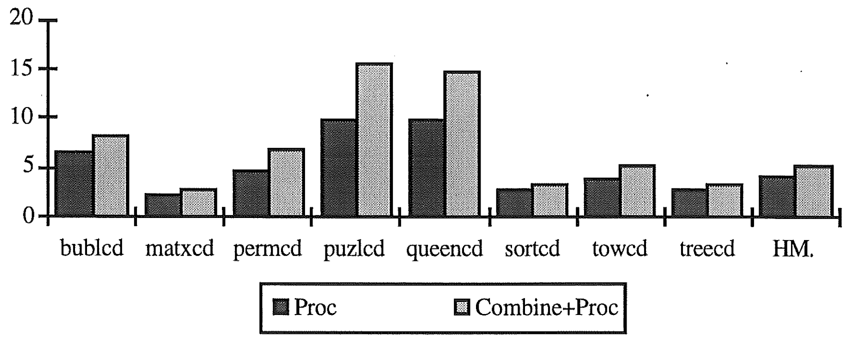

Figure 6.19 - Combining With Procedure Ceilings

These figures show that there is an observable performance increase with instruction combining. If combining can be implemented without substantial hardware costs or any impact on the cycle time, it justifies its inclusion in the HSA specification. The results are based on a limited implementation of a subset of the possible combinations, increasing the set of combinations and the number of operands combined simultaneously, much greater performance increases should be possible.

6.8 Distributions of Parallelism

All the distributions presented in this section were extracted from the execution traces as they were processed by the TDS tool. By default, only the distributions over the first 2,000 cycles are recorded, but this can be redefined by the user. The TDS currently records an occupancy count for each PIT. Further information regarding the instruction mix and register usage is not stored due to limitations in

[13] See 1 and 43 from Appendix A.

the support environment. All the benchmarks were run to completion, although only the first few hundred cycles are shown below in figures 6.20a-d[14]. These distributions were generated with the HSA latency set on the oracle machine model. The y-axis represents the instruction count for the PIT groupings and the x-axis represent the PIT numbers in increasing order (the PIT numbers are grouped into tens (0-9, 10-19 etc.).

The *sortcd* benchmark distribution represents the bursty distributions reported by Austin for the SPEC benchmarks [Austin92], here large areas of parallelism are followed by areas of low parallelism. The rest of the benchmarks typically have am ILP distribution which shows that the majority of their instruction execution occurs within the first few hundred cycles.

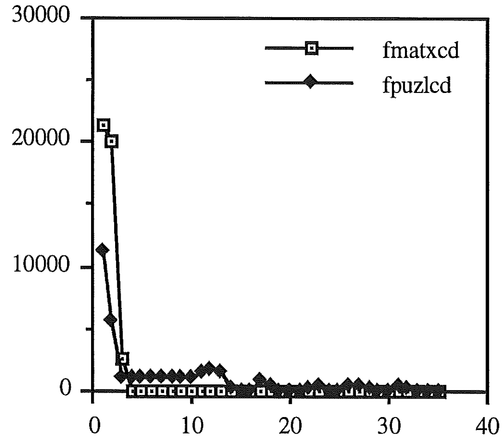

Figure 6.20a - ***matxcd*** **and** ***puzlcd***

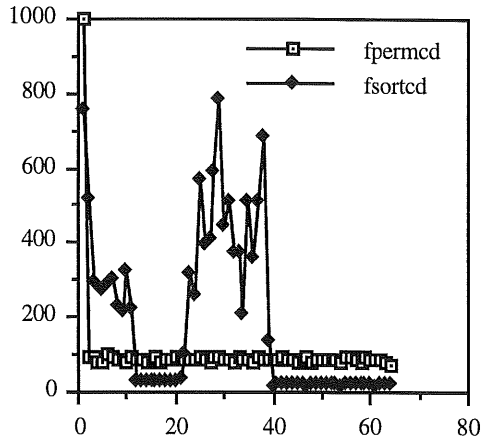

Figure 6.20b - ***permcd*** **and** ***sortcd***

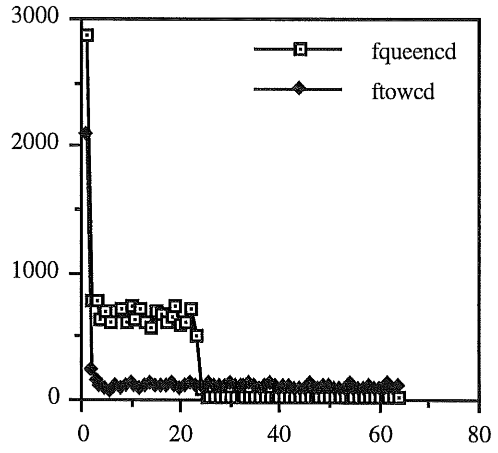

Figure 6.20c - ***queencd*** **and** ***towcd***

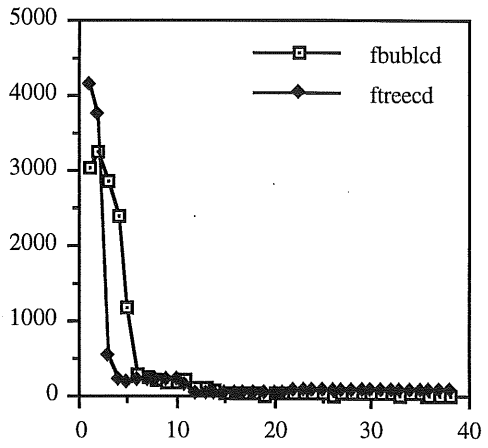

Figure 6.20d - ***bublcd*** **and** ***treecd***

From the distributions it can be seen, that the critical path through the data is limited by a small number of instructions. More work is required to identify the cause of these distributions, but they are a useful indication of the parallelism inherent in these small benchmarks These distributions are idealistic, as the removal of control flow barriers allows significant concurrent execution of instructions arbitrarily far apart in the instruction trace.

[14] Please note the scale differences in the x and y-axis.

7. Conclusions and Future Work

This project has shown that using trace-driven simulation, to quantify the upper limits of instruction-level parallelism, ILP ranged from 1.0 for a highly restricted model, based on serialising loops, to 264.4 for an Oracle processor with perfect procedure inlining. The average ILP across the models used, is around 20 instructions per cycle, which is encouraging. This may increase further for scheduled code, where code motion and code transformation could expose significantly more parallelism. Compared to the average ILP for results reported so far for the Conditional Group Scheduler (CGS), of 4 instructions per cycle fully executed and around 5 instructions per cycle issued, it shows that even for the small benchmarks there is still much room for improvement. The range of parallelism correlates well both the Wall's and Lam's studies. The base model is directly relatable to Wall's perfect model, with perfect branch prediction, jump prediction, register renaming and alias analysis. The TDS tool offers a further useful analysis of the benchmarks, by recording the distributions of parallelism. Initial work with these distributions showed that most of the programs execution could be performed within a small number of cycles, with only a few instructions being executed within the later cycles. The ideal situation is to have the ILP match the machine parallelism, so the functional units are used as greatly as possible. Theobald [Theobald92] defined this as smoothability. The ability to asses the distribution of parallelism may prove beneficial in assessing the quality of schedules produced by further HSA instruction schedulers.

The most surprising result so far reported is that for Procedure Ceilings, speedups over the scalar RISC machine, were degraded by an average of 76%. This ceiling proved even more detrimental than serialising loop execution. Several benchmarks reporting a speedup of little over 2. These figures are aided by the other perfect methods still in place, perfect memory disambiguation and perfect renaming. The impact on a realistic processor would be even more significant. This suggests that reducing the procedure overhead is critical and techniques such as directed procedure inlining should be implemented. A study by Serrano [Serrano95] reported that for a simple compile inlining decision algorithm, for every program tested, inlining did not hamper code execution and on average execution time exhibited a speedup of 10%. Although the impact of procedure inlining is dependent on the cost of function calls. These figures are further strengthened by the results for selected inlining of procedures within the Stanford set, where even with a very simplistic inlining decision algorithm, speedup increases of up to 43% where reported. There is concern that aggressive inlining will cause unreasonable code explosion. Increased object code size for compilers between 10-24% have been reported by previous studies. However, in the study by Serrano, object code size was actually reduced by up 10%. Although this is based on work on functional languages, it can only be taken as an promising result. Further work is needed in this area to select an algorithm that balances the performance gain from inlining and the overall code expansion.

The figures reported for Memory Disambiguation show that this is another critical area and should be carefully considered by any static scheduler. If no memory disambiguation is implemented, there is almost 80% depreciation in performance. There are unfortunately no figures for a reasonable implementation of static disambiguation for the HSA model. However, based on the work trying to implement such a model, few memory references within the benchmarks would be have been statically disambiguated successfully. Static disambiguation is hampered by the problems of non-linear array access, memory locations being read out of another memory location (i.e. pointer de-referencing) and the need for information not always available (i.e. loop bounds). As the Instruction-Issue rate increases for MII processors, parallel and out-of-order execution of loads and stores will become a critical factor, suggesting that techniques that expose load and store parallelism will become essential. The use of a hybrid disambiguation methods (which uses both static and run-time information to control disambiguation) should be carefully considered. Huang et al. [Haung94] introduced a technique called Speculative Disambiguation (SpD). SpD creates for each ambiguous memory reference, two copies of the code. In one copy, the addresses are assumed to be the same, in the other they are assumed to be different. Code without side effects is executed speculatively else it is conditionally guarded, based on the result of comparing the addresses. The study reported a significant speedup of around 10% at the sot of large scale code expansion. Further more on processors with insufficient resources, SpD can actually slow down the machine. But the benefits of applying a similar technique such as this to critical areas of the code could be significant based on the provisional results presented by the TDS tool.

The impact of speculative writes was not found to be significant. In the later models, the use of speculative writes saved under 0.2%. Based on these figures, the decision made for the HSA model, of not supporting speculative writes seems justified. In the benchmark programs used, stores did not significantly block the execution of loads and so their impact on the overall execution time was minimal. The impact of result forwarding which is imbedded within the current memory models, according to Johnson gained very little over allowing loads to execute out-of-order. It gained 1% for a two instruction decoder and 4% with four-instruction decoder. The impact of these techniques for a resource constrained processor is beyond the boundaries of this study, but may show whether the implementation costs associated with these methods are justified

Instruction combining within TDS produced some very favourable results, even with a restrictive model it had a H.M. increase in performance of around 10%. When limited combining was first tried on the Harp project [Steven93], it was dismissed as not being a critical factor in increasing performance and this was accounted to combining too few instructions. In the TDS tool, combining is performed on a much larger and aggressive scale and appears from these results, to add noticeably to the performance. The other argument against combining in HARP, was that it is effects where negated by a concurrent execution. For scalar processor, combining a Shift and an Add, will save a cycle but when this performed in parallel, the Shift and the Add will be executed concurrently, saving nothing. In the TDS tool's models, it was shown that often the instructions being combined had a true data dependency, thus preventing this parallel execution. While the results for a restrictive, but still very much ideal processor model may be disappointing, it sill suggest that combining may be beneficial to many programs and should be carefully considered.

The final area of study, the impact of constraints on loop, was as proposed at the beginning of this report, a highly important factor. Any ceilings placed on loop execution severely effected the overall execution time. The impact on nested loops was particularly noticeable and the heuristics used to schedule such loops should be chosen with care. Scheduling nested loops as a single unit is one possibility. The performance bottleneck introduced by loop induction variables was also shown to be significant. Removal of induction variable dependencies boosted ILP by over a factor of 2. These figures suggest that scheduler which target MII processors should strive to remove these spurious dependencies.

The work carried out on this TDS tool has shown that the areas previously targeted by the instruction schedulers developed so far, both for the HARP and the HSA projects have been justified. The results reported within this study establish that there is room for significant improvement and that the key objective of the HSA project, namely to achieve an order of magnitude speedup over a scalar RISC processor, is theoretically achievable. Further work in the area of scheduling techniques is needed, especially to improve procedure inlining and also to develop enhanced hybrid memory disambiguation techniques for scheduling. These two techniques both highlight the need to asses the quality of the schedules produced to gain the maximum cost/benefit ratio.

8. References

[Aho86] A.V. Aho, R. Sethi and J.D. Ullman. *Compilers : Principles, Techniques and Tools*, Addison-Wessley, Reading, Mass. 1986.

[Beck93] G.R. Beck, D. Yen and T.L. Anderson. *The Cydra 5 Minisupercomputer: Architecture and Implementation*, The Journal of Supercomputing, Vol. 7, No. 1/2, pages 143-180, 1993.

[Bernstein92] D. Bernstein, D. Cohen, Y. Lavon and V. Rainish. *Evaluation of Instruction Scheduling on the IBM Risc System/6000*, In Proceedings of the 275h Annual International Symposium on Microarchitecture, pages 226-235, Portland, Oregon, December 1992.

[Butler91] M. Butler, T-Y Yeh, Y. Patt , M. Alsup, H. Scales and M. Shebanow. *Single Instruction Stream Parallelism is Greater Than Two*, Proceedings of the 18th Annual International Symposium on Computer Architecture, pages 276-286, Toronto, Canada, May 1991.

[Collins93] R. Collins. *Developing a Simulator for the Hatfield Superscalar Processor*, Division of Computer Science Technical Report No. 172, University of Hertfordshire, December 1993.

[Collins95] R. Collins. *Exploiting Instruction-Level Parallelism In A Superscalar Architecture*, Ph.D. Thesis, University of Hertfordshire, October 1995.

[Ellis85] J.R. Ellis. *Bulldog : A Compiler for VLIW Architectures*, Ph.D. Thesis, Yale, U/DCS/RR-364, Yale University, Feb 1985.

[Gross88] T. Gross, J. Hennessy, S. Przybyski and C. Rowen. *Measurement and Evaluation of the MIPS Architecture and Processor*, ACM Transactions on Computer Systems, Vol 6 no.3, pages 229-257, August 1988.

[Huang94] A.S. Huang, G. Slavenburg and J.P. Shen. *Speculative Disambiguation : A Compilation technique for Dynamic Memory Disambiguation*, In Proceedings of the 21st Annual International Symposium on Computer Architecture, pages 200-210, Chicago, April 1994.

[Johnson91] M. Johnson. *Superscalar Microprocessor Design.*, Prentice-Hall, Englewood Cliffs, New Jersey, 1991.

[Jourdan95] S. Jourdan, P. Sainrat and D. Litaize. *Exploring Configurations of Functional Units in an Out-of-Order Superscalar Processor,* The 22nd International Symposium on Computer Architecture, June 1995.

[Lam92] M. Lam and P.R. Wilson. *Limits of Control Flow on Parallelism,* Proceedings of the 19th Annual Symposium on Computer Architecture", pages 46-57, Gold Coast, Australia, May 1992.

[Melvin91] S. Melvin and Y. Patt. *Exploiting Fine-Grained Parallelism Through a Combination of Hardware and Software Techniques,* In Proceedings of the 18th International Symposium on Computer Architecture, pages 287-296, May 1991.

[Nicolau84] A. Nicolau and J.A. Fisher. *Measuring the Parallelism Available for Very Long Instruction Word Architectures,* IEEE Transactions on Computers, Vol. C-33, pages 968-976, November 1984.

[Noonbrg94] D. Noonburg and J. Shen. *Theoretical Modelling of Superscalar Processor Performance,* In Proceedings of the 27th Annual International Symposium on Microarchitecture, pages 52-62, November 1994.

[Riseman72] E.M. Riseman and C.C. Foster, *The Inhibition of Potential Parallelism by Conditional Jumps*, IEEE Transactions on Computers, Vol. C-22, pages 1404-1411, December 1972.

[Serrano95] M.Serrano. *A Fresh Look To Inlining Decision*, In Proceedings of ICIS 1995.

[Smith90] M.D. Smith, M. Johnson and M.A. Horowitz. *Limits on Multiple Instruction Issue*, Proceedings of the 3rd International Symposium on Architectural Support for Programming Languages and Operating Systems, pages 290-302, April 1989.

[Smother93] M. Smotherman, S. Chawal, S. Cox and Brian Malloy. *Instruction Scheduling for the Motorola 88110, In Proceedings of the 26th Annual International Symposium on Microarchitecture*, pages 257-262, Austin, Texas, December 1993.

[Steven93] F.L. Steven, G.B. Steven and L. Wang. *An Evaluation of the Architectural Features of the iHarp Processor*, Division of Computer Science, Technical Report No.170, University of Hertfordshire, December 1993.

[Steven94] G. Steven. *The Hatfield Superscalar Architecture,* Division of Computer Science Technical Report, University of Hertfordshire, 1994.

[Steven95] G. Steven, B. Christianson, R. Collins, R. Potter and F. Steven. *A Superscalar Architecture to Exploit Instruction-Level Parallelism*, To be Published in IEEE Microprogramming and Microprocessors, 1996.

[Theobald92] K.B. Theobald, G.R. Gao and L.J. Hendren. *On the Limits of Program Parallelism and its Smoothability,* Proceedings. of the 25th Annual International Symposium On MicroArchitecture, pages 10-19, Portland, Orgeon, 1992.

[Tjaden70] G.S. Tjaden and M.J. Flynn. *Detection and Parallel Execution of Independent Instructions*, IEEE Transaction on Computers, Vol. C-19, pages 889-895, October 1970.

[Wall91] D. Wall. *Limits of Instruction-Level Parallelism*, Proceedings of the 4th International Symposium on Architectural Support for Programming Languages and Operating Systems, pages 176-188, July 1991.

[Wang93] L. Wang. *Instruction Scheduling For a Family of Multiple Instruction Issue Architectures*, PhD. Thesis, University of Hertfordshire, December 1993.

Appendix A - Results for the HSA TDS Tool.

1. No Code Restrictions[15]

Program	1-pipe Exec Times[16]		Instrs	Max Pit	ILP	Speedups	
fbublcd.ins	20119	15911	15542	391	39.747	51.455	40.693
fmatxcd.ins	30510	26882	25552	1441	17.732	21.173	18.655
fpermcd.ins	10674	8546	8578	896	9.574	11.913	9.538
fpuzlcd.ins	41426	31050	35351	356	99.301	116.365	87.219
fqueencd.ins	24568	20690	21550	992	21.724	24.766	20.857
fsortcd.ins	18262	15464	13230	741	17.854	24.645	20.869
ftowcd.ins	19864	16124	16081	1312	12.257	15.140	12.290
ftreecd.ins	32841	24291	23737	2179	10.894	15.072	11.148
Totals :	198264	158958	159621	8308	229.083	280.529	221.269
Average :	24783	19870	19952.6	1038.5	28.635	35.066	27.659
Harmonic :						21.474	17.338

2. Ceilings on Loop Entrance and Exit

Program	1-pipe Exec Times		Instrs	Max Pit	ILP	Speedups	
fbublcd.ins	20119	15911	15542	1952	7.962	10.307	8.151
fmatxcd.ins	30510	26882	25552	5152	4.960	5.922	5.218
fpermcd.ins	10674	8546	8578	913	9.395	11.691	9.360
fpuzlcd.ins	41426	31050	35351	3085	11.459	13.428	10.065
fqueencd.ins	24568	20690	21550	995	21.658	24.691	20.794
fsortcd.ins	18262	15464	13230	3901	3.391	4.681	3.964
ftowcd.ins	19864	16124	16081	1347	11.938	14.747	11.970
ftreecd.ins	32841	24291	23737	2718	8.733	12.083	8.937
Totals :	198264	158958	159621	20063	79.496	97.550	78.459
Average :	24783	19870	19952.6	2507.9	9.937	12.194	9.807
Harmonic :						9.632	7.872

3. Ceiling On Loop Entrance (Code Motion Into Loop)

Program	1-pipe Exec Times		Instrs	Max Pit	ILP	Speedups	
fbublcd.ins	20119	15911	15541	1754	8.860	11.470	9.071
fmatxcd.ins	30510	26882	25551	4351	5.872	7.012	6.178
fpermcd.ins	10674	8546	8577	903	9.498	11.821	9.464
fpuzlcd.ins	41426	31050	35350	2855	12.382	14.510	10.876
fqueencd.ins	24568	20690	21549	992	21.723	24.766	20.857
fsortcd.ins	18262	15464	13229	2980	4.439	6.128	5.189
ftowcd.ins	19864	16124	16080	1329	12.099	14.947	12.132
ftreecd.ins	32841	24291	23736	2713	8.749	12.105	8.954
Totals :	198264	158958	159613	17877	83.622	102.759	82.721
Average :	24783	19870	19951.6	2234.6	10.453	12.845	10.340
Harmonic :						10.864	8.844

4. Ceiling On Loop Exit (Code Motion Out Of Loop)

Program	1-pipe Exec Times		Instrs	Max Pit	ILP	Speedups	
fbublcd.ins	20119	15911	15541	1803	8.620	11.159	8.825
fmatxcd.ins	30510	26882	25551	4378	5.836	6.969	6.140
fpermcd.ins	10674	8546	8577	901	9.519	11.847	9.485
fpuzlcd.ins	41426	31050	35350	2909	12.152	14.241	10.674
fqueencd.ins	24568	20690	21549	995	21.657	24.691	20.794
fsortcd.ins	18262	15464	13229	3450	3.834	5.293	4.482
ftowcd.ins	19864	16124	16080	1344	11.964	14.780	11.997
ftreecd.ins	32841	24291	23736	2713	8.749	12.105	8.954
Totals :	198264	158958	159613	18493	82.331	101.085	81.351
Average :	24783	19870	19951.6	2311.6	10.291	12.636	10.169
Harmonic :						10.424	8.497

[15] Using Stanford cut down benchmarks and HSA latencies where not otherwise stated.
[16] Figures for benchmarks on HSP Simulator for single pipe model. First figure is the complete execution time, second is discounting branch delay slots.

5. Ceiling For Loop Code Only

Program	1-pipe Exec Times		Instrs	Max Pit	ILP	Speedups	
fbublcd.ins	20119	15911	15541	1803	8.620	11.159	8.825
fmatxcd.ins	30510	26882	25551	4378	5.836	6.969	6.140
fpermcd.ins	10674	8546	8577	901	9.519	11.847	9.485
fpuzlcd.ins	41426	31050	35350	2909	12.152	14.241	10.674
fqueencd.ins	24568	20690	21549	995	21.657	24.691	20.794
fsortcd.ins	18262	15464	13229	2533	5.223	7.210	6.105
ftowcd.ins	19864	16124	16080	1344	11.964	14.780	11.997
ftreecd.ins	32841	24291	23736	2713	8.749	12.105	8.954
Totals :	198264	158958	159613	17576	83.720	103.002	82.974
Average :	24783	19870	19951.6	2197.0	10.465	12.875	10.372
Harmonic :						11.155	9.068

6. Serialising The Loops[17]

Program	1-pipe Exec Times		Instrs	Max Pit	ILP	Speedups	
fbublcd.ins	20119	15911	15542	6595	2.357	3.051	2.413
fmatxcd.ins	30510	26882	25552	21847	1.170	1.397	1.230
fpermcd.ins	10674	8546	8578	1080	7.943	9.883	7.913
fpuzlcd.ins	41426	31050	35351	12029	2.939	3.444	2.581
fqueencd.ins	24568	20690	21550	4917	4.383	4.997	4.208
fsortcd.ins	18262	15464	13230	7991	1.656	2.285	1.935
ftowcd.ins	19864	16124	16081	1391	11.561	14.280	11.592
ftreecd.ins	32841	24291	23737	6475	3.666	5.072	3.752
Total :	198264	158958	159621	62325	35.675	44.409	35.624
Average :	24783	19870	19952.6	7790.6	4.459	5.551	4.453
Harmonic :						3.418	2.809

7. Removal Of Loop Constructs (Perfect Loop Unrolling)

Program	1-pipe Exec Times		Instrs	Max Pit	ILP	Speedups	
fbublcd.ins	20119	15911	14069	390	36.074	51.587	40.797
fmatxcd.ins	30510	26882	21562	1441	14.963	21.173	18.655
fpermcd.ins	10674	8546	8190	896	9.141	11.913	9.538
fpuzlcd.ins	41426	31050	26960	356	75.730	116.365	87.219
fqueencd.ins	24568	20690	17025	991	17.180	24.791	20.878
fsortcd.ins	18262	15464	12119	740	16.377	24.678	20.897
ftowcd.ins	19864	16124	15970	1312	12.172	15.140	12.290
ftreecd.ins	32841	24291	22942	2179	10.529	15.072	11.148
Total :	198264	158958	138837	8305	192.166	280.719	221.422
Average :	24783	19870	17354	1038.1	24.021	35.090	27.678
Hamronic :						21.482	17.344

8. Removal of Loop Constructs and Loop Ceilings

Program	1-pipe Exec Times		Instrs	Max Pit	ILP	Speedups	
fbublcd.ins	20119	15911	14069	788	17.854	25.532	20.192
fmatxcd.ins	30510	26882	21562	4212	5.119	7.244	6.382
fpermcd.ins	10674	8546	8190	906	9.040	11.781	9.433
fpuzlcd.ins	41426	31050	26960	981	27.482	42.228	31.651
fqueencd.ins	24568	20690	17025	993	17.145	24.741	20.836
fsortcd.ins	18262	15464	12119	3767	3.217	4.848	4.105
ftowcd.ins	19864	16124	15970	1329	12.017	14.947	12.132
ftreecd.ins	32841	24291	22942	2717	8.444	12.087	8.940
Totals :	198264	158958	138837	15693	100.318	143.408	113.671
Average :	24783	19870	17354.6	1961.6	12.540	17.926	14.209
Harmonic :						11.729	9.642

[17] Involves having a run time ceiling for every occurrence of a loop header, therefore preventing loops from overlapping their iterations.

9. Removal of Loop Constructs and Procedure Ceilings

Program	1-pipe	Exec Times	Instrs	Max Pit	ILP	Speedups	
fbublcd.ins	20119	15911	14069	2356	5.972	8.539	6.753
fmatxcd.ins	30510	26882	21562	10550	2.044	2.892	2.548
fpermcd.ins	10674	8546	8190	1752	4.675	6.092	4.878
fpuzlcd.ins	41426	31050	26960	1818	14.829	22.787	17.079
fqueencd.ins	24568	20690	17025	1156	14.728	21.253	17.898
fsortcd.ins	18262	15464	12119	5430	2.232	3.363	2.848
ftowcd.ins	19864	16124	15970	4075	3.919	4.875	3.957
ftreecd.ins	32841	24291	22942	8349	2.748	3.934	2.909
Totals :	198264	158958	138837	35486	51.147	73.735	58.870
Average :	24783	19870	17354.6	4435.8	6.393	9.217	7.359
Harmonic :						5.425	4.426

10. Removal of Loop Constructs and Unit Latencies

Program	1-pipe	Exec Times	Instrs	Max Pit	ILP	Speedups	
fbublcd.ins	20119	15911	14069	257	54.743	78.284	61.911
fmatxcd.ins	30510	26882	21562	1010	21.349	30.208	26.616
fpermcd.ins	10674	8546	8190	896	9.141	11.913	9.538
fpuzlcd.ins	41426	31050	26960	356	75.730	116.365	87.219
fqueencd.ins	24568	20690	17025	991	17.180	24.791	20.878
fsortcd.ins	18262	15464	12119	507	23.903	36.020	30.501
ftowcd.ins	19864	16124	15970	1312	12.172	15.140	12.290
ftreecd.ins	32841	24291	22942	2179	10.529	15.072	11.148
Totals :	198264	158958	138837	7508	224.747	327.793	260.101
Average :	24783	19870	17354.6	938.5	28.093	40.974	32.513
Harmonic :						23.605	18.967

11. Removal of Loop Constructs and Serial Loops

Program	1-pipe	Exec Times	Instrs	Max Pit	ILP	Speedups	
fbublcd.ins	20119	15911	14069	6547	2.149	3.073	2.430
fmatxcd.ins	30510	26882	21562	21847	0.987	1.397	1.230
fpermcd.ins	10674	8546	8190	1080	7.583	9.883	7.913
fpuzlcd.ins	41426	31050	26960	12029	2.241	3.444	2.581
fqueencd.ins	24568	20690	17025	3819	4.458	6.433	5.418
fsortcd.ins	18262	15464	12119	7886	1.537	2.316	1.961
ftowcd.ins	9864	16124	15970	1374	11.623	14.457	11.735
ftreecd.ins	32841	24291	22942	6375	3.599	5.152	3.810
Total :	198264	158958	138837	60957	34.177	46.260	37.078
Average :	24783	19870	17355	7619.6	4.272	5.783	4.635
Harmonic :						3.505	2.877

12. Code Ceiling On Procedure Call and Return

Program	1-pipe	Exec Times	Instrs	Max Pit	ILP	Speedups	
fbublcd.ins	20119	15911	15541	2404	6.465	8.369	6.619
fmatxcd.ins	30510	26882	25551	11549	2.212	2.642	2.328
fpermcd.ins	10674	8546	8577	1791	4.789	5.960	4.772
fpuzlcd.ins	41426	31050	35350	3118	11.337	13.286	9.958
fqueencd.ins	24568	20690	21549	2072	10.400	11.857	9.986
fsortcd.ins	18262	15464	13229	5690	2.325	3.209	2.718
ftowcd.ins	19864	16124	16080	4099	3.923	4.846	3.934
ftreecd.ins	32841	24291	23736	8545	2.778	3.843	2.843
Totals :	198264	158958	159613	39268	44.229	54.012	43.158
Average :	24783	19870	19951.6	4908.5	5.529	6.751	5.395
Harmonic :						4.989	4.072

13. Selective Procedure Inlining

Program	1-pipe Exec Times		Instrs	Max Pit	ILP	Speedups	
fbublcd.ins[18]	20119	15911	15338	391	39.228	51.455	40.693
fmatxcd.ins[19]	30510	26882	24748	1441	17.174	21.173	18.655
fpermcd.ins[20]	10674	8546	7622	622	12.254	17.161	13.740
fpuzlcd.ins[21]	41426	31050	35351	356	99.301	116.365	87.219
fqueencd.ins[22]	24568	20690	21550	992	21.724	24.766	20.857
fsortcd.ins[23]	18262	15464	12826	741	17.309	24.645	20.869
ftowcd.ins[24]	19864	16124	16069	1306	12.304	15.210	12.346
ftreecd.ins[25]	32841	24291	22633	1777	12.737	18.481	13.670
Totals :	198264	158958	156137	7626	232.031	289.256	228.049
Average :	24783	19870	19517.1	953.2	29.004	36.157	28.506
Harmonic :						23.928	19.397

14. Selective Inlining and Procedure Ceilings

Program	1-pipe Exec Times		Instrs	Max Pit	ILP	Speedups	
fbublcd.ins	20119	15911	15338	543	28.247	37.052	29.302
fmatxcd.ins	30510	26882	24748	4013	6.167	7.603	6.699
fpermcd.ins	10674	8546	7622	1485	5.133	7.188	5.755
fpuzlcd.ins	41426	31050	35350	3118	11.337	13.286	9.958
fqueencd.ins	24568	20690	21549	2072	10.400	11.857	9.986
fsortcd.ins	18262	15464	12826	1929	6.649	9.467	8.017
ftowcd.ins	19864	16124	16069	4093	3.926	4.853	3.939
ftreecd.ins	32841	24291	22633	4685	4.831	7.010	5.185
Totals :	198264	158958	156135	21938	76.690	98.316	78.841
Average :	24783	19870	19516.9	2742.2	9.586	12.290	9.855
Harmonic :						8.776	7.085

15. Perfect Procedure Inlining

Program	1-pipe Exec Times		Instrs	Max Pit	ILP	Speedups	
fbublcd.ins	20119	15911	15325	391	39.194	51.455	40.693
fmatxcd.ins	30510	26882	24331	1441	16.885	21.173	18.655
fpermcd.ins	10674	8546	6789	622	10.915	17.161	13.740
fpuzlcd.ins	41426	31050	34642	131	264.443	316.229	237.023
fqueencd.ins	24568	20690	21550	992	21.724	24.766	20.857
fsortcd.ins	18262	15464	13230	741	17.854	24.645	20.869
ftowcd.ins	19864	16124	13460	385	34.961	51.595	41.881
ftreecd.ins	32841	24291	19388	741	26.165	44.320	32.781
Totals :	198264	158958	148715	5444	432.141	551.344	426.499
Average :	24783	19870	18589.4	680.5	54.018	68.918	53.312
Harmonic :						31.873	26.193

[18] _Rand, _Initrand, _printf
[19] _Rand, _Innerproduct
[20] _Swap x2 , _Initialise, _printf
[21] Control flow too complicated to in-line
[22] Control flow too complicated to in-line
[23] _Initrand, _Rand, _printf
[24] _Makenull x3, _Error x3, _printf x2
[25] _Initrand, _Rand, _printf, _malloc x2

16. Recursive Procedures and Procedure Ceilings

Program	1-pipe Exec Times		Instrs	Max Pit	ILP	Speedups	
fbublcd.ins	20119	15911	0	2404	6.465	8.369	6.619
fmatxcd.ins	30510	26882	0	11549	2.212	2.642	2.328
fpermcd.ins	10674	8546	205	1791	4.790	5.960	4.772
fpuzlcd.ins	41426	31050	19	2987	11.835	13.869	10.395
fqueencd.ins	24568	20690	112	1087	19.825	22.602	19.034
fsortcd.ins	18262	15464	88	5267	2.512	3.467	2.936
ftowcd.ins	19864	16124	126	3923	4.099	5.063	4.110
ftreecd.ins	32841	24291	684	7782	3.050	4.220	3.121
Totals :	198264	158958	1234	36790	54.788	66.192	53.315
Average :	24783	19870	154.3	4599	6.849	8.274	6.664
Harmonic :						5.315	4.340

(Change from (12) 3 Unchanged. Avg of changes is +2.031, H.M. is +0.307. Max is +9.048 fqueencd)

17. Motion Across Procedures

Program	1-pipe Exec Times		Instrs	Max Pit	ILP	Speedups	
fbublcd.ins	20119	15911	15542	2255	6.892	8.922	7.056
fmatxcd.ins	30510	26882	25552	11217	2.278	2.720	2.397
fpermcd.ins	10674	8546	8578	1425	6.020	7.491	5.997
fpuzlcd.ins	41426	31050	35351	2315	15.270	17.895	13.413
fqueencd.ins	24568	20690	21550	1321	16.313	18.598	15.662
fsortcd.ins	18262	15464	13230	5599	2.363	3.262	2.762
ftowcd.ins	19864	16124	16081	3289	4.889	6.040	4.902
ftreecd.ins	32841	24291	23737	7890	3.008	4.162	3.079
Totals :	198264	158958	159621	35311	57.033	69.090	55.268
Average :	24783	19870	19952.6	4413.9	7.129	8.636	6.908
Harmonic :						5.574	4.559

18. Code Motion Across Loops (with Procedure Ceilings)

Program	1-pipe Exec Times		Instrs	Max Pit	ILP	Speedups	
fbublcd.ins	20119	15911	14216	2207	6.441	9.116	7.209
fmatxcd.ins	30510	26882	25551	11217	2.278	2.720	2.397
fpermcd.ins	10674	8546	8577	1425	6.019	7.491	5.997
fpuzlcd.ins	41426	31050	35350	2315	15.270	17.895	13.413
fqueencd.ins	24568	20690	21549	1321	16.313	18.598	15.662
fsortcd.ins	18262	15464	13229	5599	2.363	3.262	2.762
ftowcd.ins	19864	16124	16080	3289	4.889	6.040	4.902
ftreecd.ins	32841	24291	23736	7890	3.008	4.162	3.205
Totals :	198264	158958	158288	35263	56.581	69.284	55.547
Average :	24783	19870	19786.0	4407.9	7.073	8.661	6.943
Harmonic :						5.583	4.600

19. Superscalar Memory Disambiguation[26]

Program	1-pipe Exec Times		Instrs	Max Pit	ILP	Speedups	
fbublcd.ins	20119	15911	15541	1309	11.872	15.370	12.155
fmatxcd.ins	30510	26882	25551	4611	5.541	6.617	5.830
fpermcd.ins	10674	8546	8577	896	9.573	11.913	9.538
fpuzlcd.ins	41426	31050	35350	356	99.298	116.365	87.219
fqueencd.ins	24568	20690	21549	992	21.723	24.766	20.857
fsortcd.ins	18262	15464	13229	2312	5.722	7.899	6.689
ftowcd.ins	19864	16124	16080	1312	12.256	15.140	12.290
ftreecd.ins	32841	24291	23736	2341	10.139	14.029	10.376
Totals :	198264	158958	159613	14129	176.124	212.099	164.954
Average :	24783	19870	19951.6	1766.1	22.016	26.512	20.619
Harmonic :						13.050	10.734

[26] Stores In Enforced Order, Loads still have perfect disambiguation and can float as high as possible.

20. No Memory Disambiguation

Program	1-pipe Exec Times		Instrs	Max Pit	ILP	Speedups	
fbublcd.ins	20119	15911	15542	3541	4.389	5.682	4.493
fmatxcd.ins	30510	26882	25552	14119	1.810	2.161	1.904
fpermcd.ins	10674	8546	8578	2839	3.021	3.760	3.010
fpuzlcd.ins	41426	31050	35351	2477	14.272	16.724	12.535
fqueencd.ins	24568	20690	21550	3309	6.513	7.425	6.253
fsortcd.ins	18262	15464	13230	5242	2.524	3.484	2.950
ftowcd.ins	19864	16124	16081	4477	3.592	4.437	3.602
ftreecd.ins	32841	24291	23737	7668	3.096	4.283	3.168
Totals :	198264	158958	159621	43672	39.217	47.956	37.915
Average :	24783	19870	19952.6	5459.0	4.902	5.995	4.739
Harmonic :						4.336	3.552

21. Restrictive Memory Model

Program	1-pipe Exec Times		Instrs	Max Pit	ILP	Speedups	
fbublcd.ins	20119	15911	15542	2455	6.331	8.195	6.481
fmatxcd.ins	30510	26882	25552	10412	2.454	2.930	2.582
fpermcd.ins	10674	8546	8578	1776	4.830	6.010	4.812
fpuzlcd.ins	41426	31050	35351	641	55.150	64.627	48.440
fqueencd.ins	24568	20690	21550	1451	14.852	16.932	14.259
fsortcd.ins	18262	15464	13230	5362	2.467	3.406	2.884
ftowcd.ins	19864	16124	16081	3682	4.367	5.395	4.379
ftreecd.ins	32841	24291	23737	8221	2.887	3.995	2.955
Total :	198264	158958	159621	4949	93.338	111.490	86.792
Average :	24783	19870	19953	619	11.667	13.936	10.849
Harmonic :						5.581	4.562

22. Perfect Memory and Restrict Memory

Program	1-pipe Exec Times		Instrs	Max Pit	ILP	Speedups	
fbublcd.ins	20119	15911	15542	2305	6.743	8.728	6.903
fmatxcd.ins	30510	26882	25552	8966	2.850	3.403	2.998
fpermcd.ins	10674	8546	8578	896	9.574	11.913	9.538
fpuzlcd.ins	41426	31050	35351	479	73.802	86.484	64.823
fqueencd.ins	24568	20690	21550	705	30.567	34.848	29.348
fsortcd.ins	18262	15464	13230	5121	2.583	3.566	3.020
ftowcd.ins	19864	16124	16081	2409	6.675	8.246	6.693
ftreecd.ins	32841	24291	23737	6874	3.453	4.778	3.534
Totals :	198264	158958	159621	27755	136.247	161.966	126.857
Average :	24783	19870	19952.6	3469.4	17.031	20.246	15.857
Harmonic :						6.995	5.730

23. Superscalar Disambiguation and Restrictive Memory

Program	1-pipe Exec Times		Instrs	Max Pit	ILP	Speedups	
fbublcd.ins	20119	15911	15542	3671	4.234	5.481	4.334
fmatxcd.ins	30510	26882	25552	10479	2.438	2.912	2.565
fpermcd.ins	10674	8546	8578	2215	3.873	4.819	3.858
fpuzlcd.ins	41426	31050	35351	2593	13.633	15.976	11.975
fqueencd.ins	24568	20690	21550	3209	6.715	7.656	6.447
fsortcd.ins	18262	15464	13230	5646	2.343	3.235	2.739
ftowcd.ins	19864	16124	16081	4553	3.532	4.363	3.541
ftreecd.ins	32841	24291	23737	8634	2.749	3.804	2.813
Totals :	198264	158958	159621	41000	39.517	48.246	38.272
Average :	24783	19870	19952.6	5125.0	4.940	6.031	4.784
Harmonic :						4.630	3.771

24. Limited Memory Disambiguation and Restrictive Memory

Program	1-pipe Exec Times		Instrs	Max Pit	ILP	Speedups	
fbublcd.ins	20119	15911	15542	2455	6.331	8.195	6.481
fmatxcd.ins	30510	26882	25552	10412	2.454	2.930	2.582
fpermcd.ins	10674	8546	8578	1776	4.830	6.010	4.812
fpuzlcd.ins	41426	31050	35351	641	55.150	64.627	48.440
fqueencd.ins	24568	20690	21550	1451	14.852	16.932	14.259
fsortcd.ins	18262	15464	13230	5362	2.467	3.406	2.884
ftowcd.ins	19864	16124	16081	3682	4.367	5.395	4.379
ftreecd.ins	32841	24291	23737	8221	2.887	3.995	2.955
Totals :	198264	158958	159621	34000	93.338	111.490	86.792
Average :	24783	19870	19952.6	4250.0	11.667	13.936	10.849
Harmonic :						5.5807	4.562

25. Perfect Memory Model[27]

Program	1-pipe Exec Times		Instrs	Max Pit	ILP	Speedups	
fbublcd.ins	20119	15911	15542	291	53.409	69.137	54.677
fmatxcd.ins	30510	26882	25552	1040	24.569	29.337	25.848
fpermcd.ins	10674	8546	8578	895	9.584	11.926	9.549
fpuzlcd.ins	41426	31050	35351	355	99.580	116.693	87.465
fqueencd.ins	24568	20690	21550	233	92.489	105.442	88.798
fsortcd.ins	18262	15464	13230	541	24.455	33.756	28.584
ftowcd.ins	19864	16124	16081	1311	12.266	15.152	12.299
ftreecd.ins	32841	24291	23737	2178	10.899	15.079	11.153
Totals :	198264	158958	159621	6844	327.251	396.522	318.373
Average :	24783	19870	19952.6	855.5	40.906	49.565	39.797
Harmonic :						25.609	20.495

26. Minimum Division[28]

Program	1-pipe Exec Times		Instrs	Max Pit	ILP	Speedups	
fbublcd.ins	20119	15911	15542	360	43.172	55.886	44.197
fmatxcd.ins	30510	26882	25552	1410	18.122	21.638	19.065
fpermcd.ins	10674	8546	8578	896	9.574	11.913	9.538
fpuzlcd.ins	41426	31050	35351	356	99.301	116.365	87.219
fqueencd.ins	24568	20690	21550	992	21.724	24.766	20.857
fsortcd.ins	18262	15464	13230	710	18.634	25.721	21.780
ftowcd.ins	19864	16124	16081	1312	12.257	15.140	12.290
ftreecd.ins	32841	24291	23737	2179	10.894	15.072	11.148
Totals :	198264	158958	159621	8215	233.678	286.501	226.094
Average :	24783	19870	19952.6	1026.9	29.210	35.813	28.262
Harmonic :						21.722	17.532

27. Wall Latencies[29]

Program	1-pipe Exec Times		Instrs	Max Pit	ILP	Speedups	
fbublcd.ins	20119	15911	15542	258	60.240	77.981	61.670
fmatxcd.ins	30510	26882	25552	1010	25.299	30.208	26.616
fpermcd.ins	10674	8546	8578	896	9.574	11.913	9.538
fpuzlcd.ins	41426	31050	35351	356	99.301	116.365	87.219
fqueencd.ins	24568	20690	21550	992	21.724	24.766	20.857
fsortcd.ins	18262	15464	13230	508	26.043	35.949	30.440
ftowcd.ins	19864	16124	16081	1312	12.257	15.140	12.290
ftreecd.ins	32841	24291	23737	2179	10.894	15.072	11.148
Totals :	198264	158958	159621	7511	265.332	327.394	259.778
Average :	24783	19870	19952.6	938.9	33.167	40.924	32.472
Harmonic :						23.595	18.959

[27] Perfect Memory - MEM_LD_LATENCY =0, MEM_ST_LATENCY =0.
[28] Minimum Division - DIV_LATENCY =1, everything else as HSA.
[29] Wall Latencies - All latencies are unit latency

28. Cydra 5 Style Long Latencies[30]

Program	1-pipe	Exec Times	Instrs	Max Pit	ILP	Speedups	
fbublcd.ins	20119	15911	15542	1652	9.408	12.179	9.631
fmatxcd.ins	30510	26882	25552	6455	3.958	4.727	4.165
fpermcd.ins	10674	8546	8578	5382	1.594	1.983	1.588
fpuzlcd.ins	41426	31050	35351	1793	19.716	23.104	17.317
fqueencd.ins	24568	20690	21550	1199	17.973	20.490	17.256
fsortcd.ins	18262	15464	13230	3252	4.068	5.616	4.755
ftowcd.ins	19864	16124	16081	6573	2.447	3.022	2.453
ftreecd.ins	32841	24291	23737	10930	2.172	3.005	2.222
Totals :	198264	158958	159621	37236	61.336	74.126	59.387
Average :	24783	19870	19952.6	4654.5	7.667	9.266	7.423
Harmonic :						4.620	3.708

29. Speculative Writes Using Immediately Preceding Branches

Program	1-pipe	Exec Times	Instrs	Max Pit	ILP	Speedups	
fbublcd.ins	20119	15911	15542	391	39.749	51.455	40.693
fmatxcd.ins	30510	26882	25552	1441	17.732	21.173	18.655
fpermcd.ins	10674	8546	8578	896	9.574	11.913	9.538
fpuzlcd.ins	41426	31050	35351	356	99.301	116.365	87.219
fqueencd.ins	24568	20690	21550	1019	21.148	24.110	20.304
fsortcd.ins	18262	15464	13230	741	17.854	24.645	20.869
ftowcd.ins	19864	16124	16081	1312	12.257	15.140	12.290
ftreecd.ins	32841	24291	23737	2187	10.854	15.016	11.107
Totals :	198264	158958	159621	8343	228.469	279.817	220.675
Average :	24783	19870	19952.6	1042.9	28.559	34.977	27.584
Harmonic :						21.396	17.277

30. Speculative Writes Using Immediately Preceding Branches and Procedure Ceilings

Program	1-pipe	Exec Times	Instrs	Max Pit	ILP	Speedups	
fbublcd.ins	20119	15911	15542	2577	6.031	7.807	6.174
fmatxcd.ins	30510	26882	25552	11549	2.212	2.642	2.328
fpermcd.ins	10674	8546	8578	1791	4.790	5.960	4.772
fpuzlcd.ins	41426	31050	35351	3150	11.223	13.151	9.857
fqueencd.ins	24568	20690	21550	2313	9.317	10.622	8.945
fsortcd.ins	18262	15464	13230	5694	2.323	3.207	2.716
ftowcd.ins	19864	16124	16081	4131	3.893	4.809	3.903
ftreecd.ins	32841	24291	23737	8549	2.777	3.842	2.841
Totals :	198264	158958	159621	39754	42.566	52.040	41.536
Average :	24783	19870	19952.6	4969.2	5.321	6.505	5.192
Harmonic :						4.924	4.019

31. Speculative Writes Using Immediately Preceding Branches and Restrictive Memory

Program	1-pipe	Exec Times	Instrs	Max Pit	ILP	Speedups	
fbublcd.ins	20119	15911	15542	2615	5.943	7.694	6.085
fmatxcd.ins	30510	26882	25552	10412	2.454	2.930	2.582
fpermcd.ins	10674	8546	8578	1776	4.830	6.010	4.812
fpuzlcd.ins	41426	31050	35351	642	55.064	64.526	48.364
fqueencd.ins	24568	20690	21550	1517	14.206	16.195	13.639
fsortcd.ins	18262	15464	13230	5366	2.466	3.403	2.882
ftowcd.ins	19864	16124	16081	3682	4.367	5.395	4.379
ftreecd.ins	32841	24291	23737	8284	2.865	3.964	2.932
Totals :	198264	158958	159621	34294	92.195	110.117	85.675
Average :	24783	19870	19952.6	4286.8	11.524	13.765	10.709
Harmonic :						5.531	4.520

[30] Taken From Worst case cydra 5 latencies, main effect is the long load instruction latency.
Branch = 3 Store = 1 Load = 20 Arith = 5 Mult = 5 Div = 22 Shift = 1 Boolean = 2

32. Speculative Writes Using Immediately Preceding Branches, Restrictive Memory and Procedure Ceilings

Program	1-pipe	Exec Times	Instrs	Max Pit	ILP	Speedups	
fbublcd.ins	20119	15911	15542	2628	5.914	7.656	6.054
fmatxcd.ins	30510	26882	25552	11751	2.174	2.596	2.288
fpermcd.ins	10674	8546	8578	1879	4.565	5.681	4.548
fpuzlcd.ins	41426	31050	35351	3173	11.141	13.056	9.786
fqueencd.ins	24568	20690	21550	2315	9.309	10.613	8.937
fsortcd.ins	18262	15464	13230	5851	2.261	3.121	2.643
ftowcd.ins	19864	16124	16081	4522	3.556	4.393	3.566
ftreecd.ins	32841	24291	23737	9680	2.452	3.393	2.509
Totals :	198264	158958	159621	41799	41.372	50.509	40.331
Average :	24783	19870	19952.6	5224.9	5.172	6.314	5.041
Harmonic :						4.691	3.821

33. Speculative Writes Using Immediately Preceding Booleans

Program	1-pipe	Exec Times	Instrs	Max Pit	ILP	Speedups	
fbublcd.ins	20119	15911	15542	391	39.749	51.455	40.693
fmatxcd.ins	30510	26882	25552	1441	17.732	21.173	18.655
fpermcd.ins	10674	8546	8578	896	9.574	11.913	9.538
fpuzlcd.ins	41426	31050	35351	356	99.301	116.365	87.219
fqueencd.ins	24568	20690	21550	1018	21.169	24.134	20.324
fsortcd.ins	18262	15464	13230	741	17.854	24.645	20.869
ftowcd.ins	19864	16124	16081	1312	12.257	15.140	12.290
ftreecd.ins	32841	24291	23737	2185	10.864	15.030	11.117
Totals :	198264	158958	159621	8340	228.500	279.855	220.705
Average :	24783	19870	19952.6	1042.5	28.563	34.982	27.588
Harmonic :						21.402	17.281

34. Speculative Writes Using Immediately Preceding Booleans and Procedure Ceilings

Program	1-pipe	Exec Times	Instrs	Max Pit	ILP	Speedups	
fbublcd.ins	20119	15911	15542	2499	6.219	8.051	6.367
fmatxcd.ins	30510	26882	25552	11549	2.212	2.642	2.328
fpermcd.ins	10674	8546	8578	1791	4.790	5.960	4.772
fpuzlcd.ins	41426	31050	35351	3124	11.316	13.261	9.939
fqueencd.ins	24568	20690	21550	2185	9.863	11.244	9.469
fsortcd.ins	18262	15464	13230	5692	2.324	3.208	2.717
ftowcd.ins	19864	16124	16081	4099	3.923	4.846	3.934
ftreecd.ins	32841	24291	23737	8547	2.777	3.842	2.842
Totals :	198264	158958	159621	39486	43.424	53.054	42.368
Average :	24783	19870	19952.6	4935.8	5.428	6.632	5.296
Harmonic :						4.959	4.048

35. Speculative Writes Using Immediately Preceding Booleans and Restrictive Memory

Program	1-pipe	Exec Times	Instrs	Max Pit	ILP	Speedups	
fbublcd.ins	20119	15911	15542	2538	6.124	7.927	6.269
fmatxcd.ins	30510	26882	25552	10412	2.454	2.930	2.582
fpermcd.ins	10674	8546	8578	1776	4.830	6.010	4.812
fpuzlcd.ins	41426	31050	35351	641	55.150	64.627	48.440
fqueencd.ins	24568	20690	21550	1480	14.561	16.600	13.980
fsortcd.ins	18262	15464	13230	5364	2.466	3.405	2.883
ftowcd.ins	19864	16124	16081	3682	4.367	5.395	4.379
ftreecd.ins	32841	24291	23737	8223	2.887	3.994	2.954
Totals :	198264	158958	159621	34116	92.839	110.888	86.299
Average :	24783	19870	19952.6	4264.5	11.605	13.861	10.787
Harmonic :						5.560	4.544

36. Speculative Writes Using Immediately Preceding Booleans, Restrictive Memory and Procedure Ceilings

Program	1-pipe	Exec Times	Instrs	Max Pit	ILP	Speedups	
fbublcd.ins	20119	15911	15542	2550	6.095	7.890	6.240
fmatxcd.ins	30510	26882	25552	11751	2.174	2.596	2.288
fpermcd.ins	10674	8546	8578	1879	4.565	5.681	4.548
fpuzlcd.ins	41426	31050	35351	3147	11.233	13.164	9.867
fqueencd.ins	24568	20690	21550	2187	9.854	11.234	9.460
fsortcd.ins	18262	15464	13230	5849	2.262	3.122	2.644
ftowcd.ins	19864	16124	16081	4490	3.582	4.424	3.591
ftreecd.ins	32841	24291	23737	9630	2.465	3.410	2.522
Totals :	198264	158958	159621	41483	42.230	51.521	41.160
Average :	24783	19870	19952.6	5185.4	5.279	6.440	5.145
Harmonic :						4.727	3.851

37. Speculative Writes Using Immediately Preceding Branches and No Disambiguation

Program	1-pipe	Exec Times	Instrs	Max Pit	ILP	Speedups	
fbublcd.ins	20119	15911	15542	5399	2.879	3.726	2.947
fmatxcd.ins	30510	26882	25552	14119	1.810	2.161	1.904
fpermcd.ins	10674	8546	8578	2839	3.021	3.760	3.010
fpuzlcd.ins	41426	31050	35351	2979	11.867	13.906	10.423
fqueencd.ins	24568	20690	21550	3921	5.496	6.266	5.277
fsortcd.ins	18262	15464	13230	5290	2.501	3.452	2.923
ftowcd.ins	19864	16124	16081	4499	3.574	4.415	3.584
ftreecd.ins	32841	24291	23737	9813	2.419	3.347	2.475
Totals :	198264	158958	159621	48859	33.567	41.033	32.543
Average :	24783	19870	19952.6	6107.4	4.196	5.129	4.068
Harmonic :						3.915	3.191

38. Speculative Writes Using Immediately Preceding Branches and Limited Disambiguation

Program	1-pipe	Exec Times	Instrs	Max Pit	ILP	Speedups	
fbublcd.ins	20119	15911	15542	391	39.749	51.455	40.693
fmatxcd.ins	30510	26882	25552	1441	17.732	21.173	18.655
fpermcd.ins	10674	8546	8578	896	9.574	11.913	9.538
fpuzlcd.ins	41426	31050	35351	356	99.301	116.365	87.219
fqueencd.ins	24568	20690	21550	1019	21.148	24.110	20.304
fsortcd.ins	18262	15464	13230	741	17.854	24.645	20.869
ftowcd.ins	19864	16124	16081	1312	12.257	15.140	12.290
ftreecd.ins	32841	24291	23737	2187	10.854	15.016	11.107
Totals :	198264	158958	159621	8343	228.469	279.817	220.675
Average :	24783	19870	19952.6	1042.9	28.559	34.977	27.584
Harmonic :						21.396	17.277

39. Speculative Writes Using Immediately Preceding Branches and Superscalar Disambiguation

Program	1-pipe	Exec Times	Instrs	Max Pit	ILP	Speedups	
fbublcd.ins	20119	15911	15542	2538	6.124	7.927	6.269
fmatxcd.ins	30510	26882	25552	5719	4.468	5.335	4.700
fpermcd.ins	10674	8546	8578	1518	5.651	7.032	5.630
fpuzlcd.ins	41426	31050	35351	2262	15.628	18.314	13.727
fqueencd.ins	24568	20690	21550	3159	6.822	7.777	6.550
fsortcd.ins	18262	15464	13230	3043	4.348	6.001	5.082
ftowcd.ins	19864	16124	16081	2507	6.414	7.923	6.432
ftreecd.ins	32841	24291	23737	4196	5.657	7.827	5.789
Totals :	198264	158958	159621	24942	55.112	68.136	54.179
Average :	24783	19870	19952.6	3117.8	6.889	8.517	6.772
Harmonic :						7.550	6.152

40. Speculative Writes Using Immediately Preceding Booleans and No Disambiguation

Program	1-pipe Exec Times		Instrs	Max Pit	ILP	Speedups	
fbublcd.ins	20119	15911	15542	4776	3.254	4.213	3.331
fmatxcd.ins	30510	26882	25552	14119	1.810	2.161	1.904
fpermcd.ins	10674	8546	8578	2839	3.021	3.760	3.010
fpuzlcd.ins	41426	31050	35351	2811	12.576	14.737	11.046
fqueencd.ins	24568	20690	21550	3703	5.820	6.635	5.587
fsortcd.ins	18262	15464	13230	5264	2.513	3.469	2.938
ftowcd.ins	19864	16124	16081	4492	3.580	4.422	3.589
ftreecd.ins	32841	24291	23737	8927	2.659	3.679	2.721
Totals :	198264	158958	159621	46931	35.233	43.076	34.126
Average :	24783	19870	19952.6	5866.4	4.404	5.385	4.266
Harmonic :						4.059	3.314

(Change From (37) - 2 Unchanged - Avg of changes is +0.264 with max of +0.623 fpuzlcd.ins)

41. Speculative Writes Using Immediately Preceding Booleans and Limited Disambiguation

Program	1-pipe Exec Times		Instrs	Max Pit	ILP	Speedups	
fbublcd.ins	20119	15911	15542	391	39.749	51.455	40.693
fmatxcd.ins	30510	26882	25552	1441	17.732	21.173	18.655
fpermcd.ins	10674	8546	8578	896	9.574	11.913	9.538
fpuzlcd.ins	41426	31050	35351	356	99.301	116.365	87.219
fqueencd.ins	24568	20690	21550	1018	21.169	24.134	20.324
fsortcd.ins	18262	15464	13230	741	17.854	24.645	20.869
ftowcd.ins	19864	16124	16081	1312	12.257	15.140	12.290
ftreecd.ins	32841	24291	23737	2185	10.864	15.030	11.117
Totals :	198264	158958	159621	8340	228.500	279.855	220.705
Average :	24783	19870	19952.6	1042.5	28.563	34.982	27.588
Harmonic :						21.402	17.281

(Change From (38) - 6 results unchanged. Avg of changes is +0.015)

42. Speculative Writes Using Immediately Preceding Booleans and Superscalar Disambiguation

Program	1-pipe Exec Times		Instrs	Max Pit	ILP	Speedups	
fbublcd.ins	20119	15911	15542	2525	6.155	7.968	6.301
fmatxcd.ins	30510	26882	25552	5719	4.468	5.335	4.700
fpermcd.ins	10674	8546	8578	1518	5.651	7.032	5.630
fpuzlcd.ins	41426	31050	35351	2262	15.628	18.314	13.727
fqueencd.ins	24568	20690	21550	3159	6.822	7.777	6.550
fsortcd.ins	18262	15464	13230	3041	4.351	6.005	5.085
ftowcd.ins	19864	16124	16081	2507	6.414	7.923	6.432
ftreecd.ins	32841	24291	23737	4194	5.660	7.830	5.792
Totals :	198264	158958	159621	24925	55.149	68.184	54.217
Average :	24783	19870	19952.6	3115.6	6.894	8.523	6.777
Harmonic :						7.556	6.157

(Change From (39) - 5 benchmarks reported no change. Avg. of changes for remainder is +0.013)

43. Combining Instructions[31]

Program	1-pipe Exec Times		Instrs	Max Pit	ILP	Speedups	
fbublcd.ins	20119	15911	15542	202	76.941	99.599	78.767
fmatxcd.ins	30510	26882	25552	802	31.860	38.042	33.519
fpermcd.ins	10674	8546	8578	622	13.791	17.161	13.740
fpuzlcd.ins	41426	31050	35351	180	196.394	230.144	172.500
fqueencd.ins	24568	20690	21550	992	21.724	24.766	20.857
fsortcd.ins	18262	15464	13230	402	32.910	45.428	38.468
ftowcd.ins	19864	16124	16081	658	24.439	30.188	24.505
ftreecd.ins	32841	24291	23737	1091	21.757	30.102	22.265
Totals :	198264	158958	159621	4949	419.816	515.430	404.621
Average :	24783	19870	19952.6	618.6	52.477	64.429	50.578
Harmonic :						35.137	28.493

[31] Instructions 'combined' dynamically at run time.

44. Combining With Procedure Ceilings

Program	1-pipe Exec Times		Instrs	Max Pit	ILP	Speedups	
fbublcd.ins	20119	15911	15542	1953	7.958	10.302	8.147
fmatxcd.ins	30510	26882	25552	9323	2.741	3.273	2.883
fpermcd.ins	10674	8546	8578	1224	7.008	8.721	6.982
fpuzlcd.ins	41426	31050	35351	2013	17.561	20.579	15.425
fqueencd.ins	24568	20690	21550	1400	15.393	17.549	14.779
fsortcd.ins	18262	15464	13230	4445	2.976	4.108	3.479
ftowcd.ins	19864	16124	16081	3029	5.309	6.558	5.323
ftreecd.ins	32841	24291	23737	7424	3.197	4.424	3.272
Totals :	198264	158958	159621	30811	62.143	75.514	60.290
Average :	24783	19870	19952.6	3851.4	7.768	9.439	7.536
Harmonic :						6.427	5.242

Change from 12. H.M. of changes is +1.04, Average is +2.142. Range is +5.467(fpuzlcd) to +0.555 (fmatxcd).

45. Instruction Counts For Benchmarks

	St	%	Ld	%	Alu	%	Mult	%
fbublcd	1377	(8.8)	2801	(18.0)	4334	(27.9)	100	(0.6)
fmatxcd	1516	(5.9)	3215	(12.6)	10760	(42.1)	1200	(4.7)
fpermcd	1723	(20.1)	1716	(20.0)	3192	(37.2)	0	(0.0)
fpuzlcd	2304	(6.5)	2876	(8.1)	12180	(34.5)	0	(0.0)
fqueencd	3157	(14.7)	3796	(17.6)	5631	(26.1)	0	(0.0)
fsortcd	939	(7.1)	2044	(15.5)	4093	(30.9)	200	(1.5)
ftowcd	2892	(18.0)	3235	(20.1)	5808	(36.1)	0	(0.0)
ftreecd	2193	(9.2)	5103	(21.4)	7985	(33.6)	200	(0.8)
Total :	16101	90.3	24786	133.3	53983	268.4	1700	7.6
Average :	2013	11.3	3098	16.7	6748	33.6	213	0.95
Harmonic :		9.3		15.2		32.8		1.0[32]

	Div	%	Bool	%	Bra	%	Shift	
fbublcd	50	(0.3)	2698	(17.4)	2808	(18.1)	1373	(8.8)
fmatxcd	200	(0.8)	1330	(5.2)	1940	(7.6)	5390	(21.1)
fpermcd	0	(0.0)	420	(4.9)	1314	(15.3)	212	(2.5)
fpuzlcd	0	(0.0)	6084	(17.2)	6559	(18.6)	5347	(15.1)
fqueencd	0	(0.0)	3512	(16.3)	3849	(17.9)	1604	(7.4)
fsortcd	100	(0.8)	1701	(12.9)	2392	(18.1)	1760	(13.3)
ftowcd	0	(0.0)	794	(4.9)	2429	(15.1)	922	(5.7)
ftreecd	100	(0.4)	2437	(10.3)	5123	(21.6)	595	(2.5)
Total :	450	(2.3)	18976	(89.1)	26414	(132.3)	17203	(76.4)
Average :	56	(0.3)	2372	(1.1)	3302	(16.5)	2150	(9.6)
Harmonic :		0.5		8.4		15.1		5.7

46. Branch Counts For Benchmarks

	Total	Bsr+Movs	%	Cond	%	UnCo	%	Taken	%
fbublcd.ins	2808	104	(3.7)	2698	(96.1)	2	(0.1)	2095	(74.6)
fmatxcd.ins	1940	610	(31.4)	1330	(68.6)	0	(0.0)	1807	(93.1)
fpermcd.ins	1314	894	(68.0)	420	(32.0)	0	(0.0)	1054	(80.2)
fpuzlcd.ins	6559	354	(5.4)	6084	(92.8)	121	(1.8)	5180	(79.0)
fqueencd.ins	3849	232	(6.0)	3512	(91.2)	105	(2.7)	1930	(50.1)
fsortcd.ins	2392	386	(16.1)	1701	(71.1)	305	(12.8)	1623	(67.9)
ftowcd.ins	2429	1310	(53.9)	794	(32.7)	325	(13.4)	1856	(76.4)
ftreecd.ins	5123	2174	(42.4)	2437	(47.6)	512	(10.0)	4136	(80.7)
Total :	26414	6064	226.9	18976	532.1	1370	40.8	19681	602
Average :	3301.8	758	28.4	2372	66.5	171.3	5.1	2460.1	75.3
Harmonic :		343.0	10.4		55.7		0.5		73.1

32 Based on the four sets of figures that generate a value

47. Loop Measurement

Program	Prog. Lines	Prog. Exec Count	Loop Lines	%	Loop Exec Count	%
fbublcd.ins	106	15542	47	44.3	15048	96.8
fmatxcd.ins	133	25552	63	47.4	22290	87.2
fpermcd.ins	97	8578	27	27.8	1721	20.1
fpuzlcd.ins	589	35351	363	61.6	31514	89.1
fqueencd.ins	164	21550	79	48.2	17327	80.4
fsortcd.ins	145	13230	52	35.9	9576	72.4
ftowcd.ins	222	16081	14	6.3	186	1.2
ftreecd.ins	219	23737	34	15.5	3083	13.0
Total :	1675	156137	679	286.8	100745	460.2
Average :	209.4	19518	84.9	35.9	12593.1 57.5	

Program	Loops	Nesting (num*depth)	Comments
fbublcd.ins	3	1*2	fbublcd dominated by inner loop (22 lines) accounts for 89% of execution.
fmatxcd.ins	5	2*2	fmatxcd dominated by a single loop which accounts for 70% of the overall execution time. Loop contains a BSR.
fpermcd.ins	3	0	fpermcd spends only 20% of its time its three simple loops. 2 of the loops contain 2 BSR calls.
fpuzlcd.ins	50	14*3, 1*2	Loop in _Fit (16 lines) accounts for 34%. Procedure _Puzzle contains the majority of the loops, 45. Their inner loops account for 37% and all the loops in this procedure, account for 38% of the total program execution.
fqueencd.ins3	1*2		Dominated by a single loop in _Try (52 lines) which is responsible for 77% of the computation time. Has a BSR which recursively calls _Try.
fsortcd.ins	4	0	Largest loop in _Quicksort has most complex control of any of the loops (26 lines), program spends 54.6% within this loop.
ftowcd.ins	2	0	Has 2 very small loops (4 & 8 lines) which account for only 1.2% of execution.
ftreecd.ins	2	0	Program has 2 loops, each contain a BSR which are responsible for 13% of the computation time .
Total	72	14*3, 7*2	

Current Combinable Instructions

B.1 Computational (ALU) Instructions

ADD Ri, (Rj + Rk),Rl	Ri:= Rj+ Rk + Rl
ADD Ri, (Rj + Rk),#Imm	Ri:= Rj + Rk + #Imm
ADD Ri, Rj (ASL #2), Rk	Ri:= (Rj * 4) + Rk
ADD Ri, Rj (ASL #2), #Imm	Ri:= (Rj * 4) + Rk
ADDV Ri, (Rj + Rk),Rl	Ri:= Rj+ Rk + Rl
ADDV Ri, (Rj + Rk),#Imm	Ri:= Rj + Rk + #Imm
ADDV Ri, Rj (ASL #2), Rk	Ri:= (Rj * 4) + Rk
ADDV Ri, Rj (ASL #2), #Imm	Ri:= (Rj * 4) + Rk
SUB Ri, (Rj + Rk), Rl	Ri := Rj + Rk - Rl
SUB Ri, (Rj - Rk), Rl	Ri := Rj - Rk - Rl
SUB Ri, (Rj + Rk), #Imm	Ri := Rj + Rk - #Imm
SUB Ri, (Rj - Rk), #Imm	Ri := Rj - Rk - #Imm
SUBV Ri, (Rj + Rk), Rl	Ri := Rj + Rk - Rl
SUBV Ri, (Rj - Rk), Rl	Ri := Rj - Rk - Rl
SUBV Ri, (Rj + Rk), #Imm	Ri := Rj + Rk - #Imm
SUBV Ri, (Rj - Rk), #Imm	Ri := Rj - Rk - #Imm

B.2 Shift Instructions

Where 0<=Imm <32;

AND Ri, Rj (ASL #Imm), Rk	Ri := (Rj << #Imm) AND (Rk)
AND Ri, Rj (ASL #Imm), #Imm2	Ri := (Rj << #Imm) AND (#Imm2)
AND Ri, Rj(ASR #Imm), Rk	Ri := (Rj >> #Imm) AND (Rk)
AND Ri, Rj(ASR #Imm), #Imm2	Ri := (Rj >> #Imm) AND (#Imm2)
OR Ri, Rj (ASL #Imm), Rk	Ri := (Rj << #Imm) OR (Rk)
OR Ri, Rj (ASL #Imm), #Imm2	Ri := (Rj << #Imm) OR (#Imm2)
OR Ri, Rj(ASR #Imm), Rk	Ri := (Rj >> #Imm) OR (Rk)
OR Ri, Rj(ASR #Imm), #Imm2	Ri := (Rj >> #Imm) OR (#Imm2)

B.3 Multiply Instructions

ADD Ri, (Rj * Rk), Rj	Ri := Rj * Rk + Rj
ADD Ri, (Rj * #Imm), Rj	Ri := Rj * #Imm +Rj
ADD Ri, (Rj * Rk), #Imm	Ri := Rj * Rk + #Imm
ADD Ri, (Rj * #Imm), #Imm2	Ri := Rj * #Imm + #Imm2
SUB Ri, (Rj * Rk), Rj	Ri := Rj * Rk - Rj
SUB Ri, (Rj * #Imm), Rj	Ri := Rj * #Imm -Rj