Performance Comparison of Load/Store and Symmetric Instruction Set Architectures

D. Alpert † * A. Averbuch † O. Danieli †
† National Semiconductor Israel, P.O. Box 3007
Herzelia B. 46104, Israel
‡ Dept. of Computer Science, School of Mathematical Sciences
Tel-Aviv University
Ramat Aviv, Tel Aviv 69978, Israel
E-Mail: amir@taurus.BITNET or amir@MATH.TAU.AC.IL

Abstract
Is it true that a Load/Store architecture is both simpler and faster than a Symmetric architecture, or does the Symmetric architecture offer a potential performance advantage that can be realized by the use of additional hardware?
In order to answer it quantitatively, we simulated two models that were equal in all aspects except the factor that we measure. We found that the Load/Store model executes 12% more instructions but only 4% more cycles.

1 Introduction
Where do we go after RISC?
Now, that RISC processors performance is close to 1 cycle/instruction, how can a computer architect achieve an even better performance? For a given hardware technology, one may try to execute more than one instruction per cycle (the parallel alternative) or one may try to perform less instructions by executing more powerful instructions.
One way to make the instructions more powerful is to go one step back from the RISC’s Load/Store architecture and use a Symmetric instruction-set which can perform direct operation on memory. In a Load/Store instruction-set architecture the only instructions that access memory are LOAD and STORE. Computational instructions can only use registers as operands. A Symmetric instruction-set architecture, on the other hand, treats register-operands and memory-operands symmetrically, and has the additional capability to perform direct computation on memory. Example:

Symmetric

ADD r1,MEM[r2+disp2]

Load/Store

LOAD MEM[r2+disp1],r3
ADD r1,r3
STORE r3,MEM[r2+disp2]

Both architectures have to perform almost the same basic operations. In the above example these are: two to three address calculations, three memory references and one addition. The Symmetric architecture uses a few instructions, but the instructions of the Load/Store architecture are more streamlined.

The question addressed by this study is: Is it true that the Load/Store architecture is both simpler and faster, or does the Symmetric architecture offer a potential performance advantage that can be realized by the use of additional hardware?
In the past this was a theoretical question. Design trade-offs in the early RISCs were based on silicon technology that allows only 50-100 thousand transistors per chip. This silicon area constraint combined with the RISC general trend to push simplicity to the limit, made the Load/Store architecture the natural choice. Today, however, this question is of practical value. Due to the progress in technology, CMOS VLSI chips are ten times denser and it is feasible to implement on silicon a pipelined processor with a capability to perform computations directly on memory.
In order to answer it quantitatively, we simulated two models that were equal in all aspects except the factor that we wanted to measure.

2 Problem Definition
The main tradeoff between Load/Store and Symmetric architectures is a tradeoff between path-length, the number of instructions executed (for a given program) and CPI, the average number of cycles per instruction. The following identity represents this tradeoff:

Program execution-time = path-length \times CPI / CPU-Frequency

The CPU-frequency is a function of the technology and the micro-architecture: the speed of a gate is a function of technology and the number of gates that have to act sequentially in one cycle is a function of the micro-architecture. Design tradeoffs in this work are based on current CMOS technology. Nevertheless, many of the results reported here can be applied as well to pipelined designs in other technologies.

*This author is currently in Intel Corporation SC4-59, P.O. Box 56122, Santa Clara, CA 95052, USA
The remaining two terms: path-length and CPI, are both affected by the selection between Load/Store or Symmetric architecture. The path-length of the Symmetric architecture is of course no longer than that of the Load/Store architecture. Measuring the path-length difference between the two, however, is not a trivial task, since there are several other factors that affect the path-length. These are: the operations in the instruction-set, the addressing-modes and the compiler quality.

In order to compute the performance difference between the two models we have to measure the difference in the CPI term too. Assuming a pipeline implementation, the CPI is the sum of three terms:

\[ \text{CPI} = 1 + \text{pipe-delay} + \text{storage-delay} \]

The "1" term represents the assumption that each instruction stays exactly one cycle at each pipe-stage and therefore the peak throughput is one cycle per instruction when there are no delays.

Pipe-delay is a function of the pipeline structure. It is composed from three components:

\[ \text{pipe-delay} = \text{resource-delay} + \text{data-delay} + \text{control-delay} \]

A pipe delay occurs when a pipeline-stage cannot perform its task because it needs a resource that is currently occupied by another stage (resource-delay), or because it needs data that has not been computed yet (data-delay) or because it does not know from where to fetch the next instruction (control-delay).

Storage-delay occurs when the CPU must wait for a memory reference. Modern computers reduce this delay by the use of a memory-hierarchy scheme: registers and caches. The slow-storage is referenced only when the data is missing in the fast-storage. The effect of storage-delay is out of the scope of this study. (See [FMM87] for a study on the effect of Load/Store architecture vs. Symmetric architecture on cache hit ratio). We use the same memory-hierarchy for both architectures and assume 100% hit-rate (no access to slow storage).

3 The Evaluation Environment

When trying to evaluate the contribution of a single architectural feature to the overall performance it is very important to keep all other parameters constant. Our approach is to compare two computers which differ from each other by only one aspect: the capability to perform computations directly on memory. We define two architectures: a Load/Store and a Symmetric architecture. For each of the two we define an implementation model, simulate a common workload and measure its performance. The performance difference between the models represents the net effect of the additional direct computations on memory. The evaluation environment we used was developed by the architecture group of National Semiconductor, Israel. Its overall structure looks like this:

![Figure 1: Simulation environment](image)

3.1 The Workload

The workload consists of five programs, written in C. All inputs to all programs were fixed across all runs.

DC - The Unix desk calculator (1938 lines), computing \( \sqrt{e^x} \) with 10-digit precision.

GREP - Search a 300-line file for a complex regular expression (323 lines).

PTC - Pascal To C translator (9718 lines). This program, representing a compiler like application, translating a 70-line pascal program.

SED - A non-interactive editor (1432 lines). Performing simple text substitutions in a 140 lines file.

SORT - The Unix sorting utility (1384 lines). Sorting 160 decimal numbers.

3.2 The Compiler

The compiler's contribution to the overall performance is an important parameter. The difference between two different compilers can be bigger than the effect we are trying to measure. In this work we use for both models the same compiler, the CTP C compiler [Sem86],[BeE88]. This optimizing compiler implements similar optimization techniques to those described by F. Chow in [Cho83]. Its four main parts are: front-end, optimizer, back-end and code reorder. The description below focuses on the parts that are not identical in both models.
Register Allocation Criteria - The benefits from using a register instead of memory results mainly from the fact that a memory-access might cause a data-cache miss or a pipeline contention. However, the use of a register might involve an overhead that results from the need to save/restore this register before/after it is used or before/after a procedure call.

In the Symmetric model, the optimizer has the freedom to choose between a register and memory. It allocates a variable into a register only if the overhead involved in using the register is less than the benefit gained by using the registers.

In the Load/Store model, the compiler must load a memory operand into a register before it uses it. This fact makes the optimizer criteria simpler: every variable that is used more than once is a candidate for a register. The LOAD/STORE overhead is spread over all the uses of this variable.

Code Reordering - In order to eliminate as many pipeline delays as possible we added a code-reordering phase at the very end of the compilation process. Both models use the same algorithms (similar to [GiM86]) but the section that describes the pipeline delays is, of course, different for each model. Note that the code-reordering changes only the order of the instructions and not the path-length.

3.3 Tracer and Simulator

The tracer generates a dynamic trace of the instructions that the program executes. It is implemented as a side-effect-free subroutine that the program calls prior to the execution of each instruction. The fact that the instruction set that we used in this study is a sub-set of the Series 32000 instruction set eliminates the need to simulate the program’s functionality. Instead we actually run the program on a real computer. This greatly simplifies the tracer, making it fast, accurate and reliable.

The simulator inputs are a table-description of the microarchitecture and a dynamic trace of the program. Using a scoreboard it simulates the flow of instructions in the pipeline and provides, as its output, a cycle-by-cycle detailed description of the program execution.

4 Common to Both Models

The following sections describes the common architecture which we use for the Load/Store and the Symmetric architectures.

4.1 Instruction Set

In order to avoid significant part of the extra work involved in defining a brand new instruction set (writing a compiler and a functional-simulator), we use an existing instruction set, the Series 32000, as a base line for our instruction set. The Series 32000 is a CISC architecture. It contains many instructions and addressing modes that are infrequently used. In this study we use only the following reduced subset that is appropriate for efficient pipeline implementation:

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADD</td>
<td>Addition</td>
</tr>
<tr>
<td>ADDR</td>
<td>Compute the effective address of an operand</td>
</tr>
<tr>
<td>AND</td>
<td>Logical and</td>
</tr>
<tr>
<td>ASH</td>
<td>Arithmetic shift</td>
</tr>
<tr>
<td>BRCOND</td>
<td>Conditional branch</td>
</tr>
<tr>
<td>BR</td>
<td>Unconditional branch to PC+displacement</td>
</tr>
<tr>
<td>BSR</td>
<td>Branch to subroutine</td>
</tr>
<tr>
<td>CMP</td>
<td>Compare</td>
</tr>
<tr>
<td>JUMP</td>
<td>Like BR but the destination is a register</td>
</tr>
<tr>
<td>LSH</td>
<td>Logical shift</td>
</tr>
<tr>
<td>MOV</td>
<td>Move</td>
</tr>
<tr>
<td>MOVXI</td>
<td>Sign extended move</td>
</tr>
<tr>
<td>MOVZI</td>
<td>Zero extended move</td>
</tr>
<tr>
<td>MUL</td>
<td>Multiply</td>
</tr>
<tr>
<td>OR</td>
<td>Logical or</td>
</tr>
<tr>
<td>QUO</td>
<td>Divide</td>
</tr>
<tr>
<td>REM</td>
<td>Compute the reminder</td>
</tr>
<tr>
<td>SUB</td>
<td>Subtract</td>
</tr>
<tr>
<td>XOR</td>
<td>Logical xor</td>
</tr>
</tbody>
</table>

Table 1: The instruction set

<table>
<thead>
<tr>
<th>Addressing Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Register</td>
<td>The operand is in register</td>
</tr>
<tr>
<td>Register-relative</td>
<td>The operand address is calculated by adding a register and a displacement</td>
</tr>
<tr>
<td>Absolute</td>
<td>The operand address is a 32-bit number</td>
</tr>
<tr>
<td>Immediate</td>
<td>The operand is a 32-bit immediate</td>
</tr>
</tbody>
</table>

Table 2: The addressing-modes

4.2 Branch Mechanism

The branch issue is related to our comparison because the selection between Symmetric architecture and Load/Store architecture implies different pipeline implementations, which in turn affect the branch mechanism and the branch delay. Since the branch-mechanism depends on the pipeline structure and we are interested only in the difference in branch-delay between the two models we evaluate its effect at the end of this study.

---

1 See next sections for a detailed description of the pipeline delays in each model.
3 Scoreboard is a hardware-mechanism, used by the CDC 6600, to control the parallel execution of ten functional units [Tho64].
4.3 Registers

Both models use a flat register-file (without a window mechanism) that contains 16 general purpose registers, a stack-pointer (SP) and a program-counter (PC). The major point is that we use the same number of registers for both models. While it is true that a Load/Store architecture needs more registers than a Symmetric architecture, by examining the code of our workload we find out that only when the number of registers is less than 8 does this need of the Load/Store architectures for more register shows up. (The compiler we use does not perform register allocation across procedures, procedure inlining or loop unrolling).

4.4 Two-Operand Instructions

In both architectures, all instructions are two-operand instructions. In computational instructions the second operand is both a source and a destination. For example: the semantics of \texttt{ADD r1,r2} is "add r1 to r2".

5. The Load/Store Architecture Model

5.1 The Pipeline Structure

The pipeline for the Load/Store architecture is based on the MIPS-X pipeline [HoC87] and is similar to that of many other RISC processors. The basic pipeline structure is:

- **IP**: Fetches the next instruction to execute from the Instruction Cache.
- **ID**: Decodes the instruction and reads registers from Register File.
- **EX**: For computational instructions: Executes the instruction. For LOAD & STORE: Calculates the memory address.
- **ME**: This is a no-op stage for computational instructions. LOAD & STORE use this stage to access the Data Cache.
- **M**: Store result. Writes result to the Register File or to Memory.

![Pipeline Structure Diagram](image)

The execution latency of an instruction is five cycles for most of the instructions and the pipeline's maximal throughput is one instruction per cycle.

5.2 Pipeline Delays

**Execution delay** - Three instructions: MUL, QUO and REM execute in more than a single cycle and cause an execution-delay. This delay, however, does not affect our comparison because both models execute exactly the same number of MUL, QUO and REM in the same number of cycles. Execution of a MUL takes 10 cycles and QUO and REM are executed in 25 cycles. Table 3 presents the execution delay in our workload:

<table>
<thead>
<tr>
<th>Instruction</th>
<th>DC</th>
<th>GREP</th>
<th>PTC</th>
<th>SED</th>
<th>SORT</th>
<th>MEAN</th>
</tr>
</thead>
<tbody>
<tr>
<td>Delay</td>
<td>0.158</td>
<td>0.0</td>
<td>0.109</td>
<td>0.0</td>
<td>0.014</td>
<td>0.056</td>
</tr>
</tbody>
</table>

Table 3: MUL, QUO and REM execution-delay in CPI

**Resource dependency** - In this model, a STORE immediately followed by LOAD (Read After Write or RAW) creates a 1-cycle delay. The reason is that the LOAD and the STORE access the DC simultaneously because the STORE writes the DC at the ST-stage and the LOAD reads the DC at the ME-stage. The same type of delay can be found in other implementations ([GiM86], [HoC87],[BeE88]).

**Data Dependency** - In this pipeline there are two types of interlock: an interlock between two computational instructions (interlock delay) and an interlock between a LOAD instruction and a computational instruction (LOAD delay). Examples:

- **Interlock Delay**
  - Instruction 1 (I1): \texttt{ADD r1,r2}
  - Instruction 2 (I2): \texttt{CMP r1,r2}

In order to reduce the delay caused by interlocks we add 3 bypasses to the pipeline. Figure 3 presents the pipe with the new bypasses.

- **Interlock Delay** - Two computational instructions interlock when instruction I2 reaches the ID-stage and one of its source operands, opd, is not ready because it is the destination of a previous instruction I1 that has not reached the ST-stage yet. The point is, that if I2 is about to enter EX then I1 is already done with EX and the result to be stored in opd can be used as soon as it becomes available. To do so, the output of the EX-stage is written into Bypass registers. The bypass registers, BP1 and BP2, are two registers organized as a queue (FIFO). They allow the destination register of a computational instruction to be used as the source of the next two instructions. Logically, it is fed back.
Figure 3: Load/Store pipeline: final structure

to the EX's input via the dotted lines BP1 or BP2. Physically, special hardware is used to detect that this register is to be read from the bypass registers and not from the RF. BP2, reduces the interlock delay to a single cycle if the instructions are consecutive and eliminates it if they are not. BP1 together with BP2 eliminate all interlocks between computational instructions.

LOAD Delay - This interlock occurs when a LOAD instruction loads a register from memory and one of the next two instructions reads this register. The delay is 2 or 1 cycles respectively. In order to reduce the LOAD delay we need a bus from ME to EX (BP2 is not a bus) to bypass data that arrives from the data-cache to the EX-stage. It reduces the 2-cycle LOAD delay to 1 cycle and eliminates all LOAD delays between a LOAD and a non-consecutive instruction.

5.3 Final Results

Table 4 and figure 4 present a detailed summary of the performance of the Load/Store model.

All programs are weighed equally and therefore all right bars are scaled to 1. The five bars represent (left to right) the performance of:

1. Initial pipeline,
2. Pipeline with BP2,
3. Pipeline with BP2 and BP1,
4. Final pipeline with BP2, BP1 and ME-to-EX.
5. Ideal pipeline with no delay (path-length).

6 The Symmetric Architecture Model

The Symmetric model attempts to speedup execution by three means:

1. Reduce the path-length (instruction count) by using a more powerful instruction set.
2. Eliminate the LOAD delay (the major delay) by using a different pipeline structure.
3. Avoid the introduction of other delays.

The following sections present the Symmetric architecture, explain how it reduces the path-length and analyze its performance.
6.1 Symmetric Instruction-Set Architecture

The Symmetric architecture is identical to the Load/Store architecture except for the Symmetric model's extra capability to operate directly on memory. Its more powerful instruction set creates a potential to reduce the path-length. The following example demonstrates the difference between the two architectures. It shows part of the strcmp routine (from the C run-time library) which compares two null terminated strings pointed by registers Rq and Rp: The

<table>
<thead>
<tr>
<th>Load/Store Loop</th>
<th>Symmetric Loop</th>
</tr>
</thead>
<tbody>
<tr>
<td>LOAD MEM[Rp],r1</td>
<td>LOAD MEM[Rp],r1</td>
</tr>
<tr>
<td>LOAD MEM[Rq],r2</td>
<td>LOAD MEM[Rq],r2</td>
</tr>
<tr>
<td>CMP 0, r1</td>
<td>CMP 0,r1</td>
</tr>
<tr>
<td>BNE L-out</td>
<td>BNE L-out</td>
</tr>
<tr>
<td>ADD 1, Rp</td>
<td>ADD 1,Rp</td>
</tr>
<tr>
<td>ADD 1, Rq</td>
<td>ADD 1,Rq</td>
</tr>
<tr>
<td>CMP r2, r1</td>
<td>CMP MEM[Rq], r1</td>
</tr>
<tr>
<td>BEQ Loop</td>
<td>BEQ Loop</td>
</tr>
</tbody>
</table>

Table 5: strcmp body

Symmetric code is one instruction shorter, 12.5%. It code could have been made another instruction shorter by eliminating the LOAD-instruction and using a CMP-instruction with two memory operands but in this study we do not allow memory to memory operations. The rationale behind this restriction is its low impact on the overall performance on one hand and the significant complexity that it saves on the other hand.4

6.2 Basic Pipeline Structure

The pipeline we use to implement the Symmetric architecture is similar to the Z80,000 pipeline [Zi184].

![Pipeline Diagram]

The idea behind this pipeline structure is to exploit the potential parallelism contained in instructions that operate directly on memory by performing an address calculation (at AG) in parallel with an ALU operation (at EX).

6.3 Pipeline Delays

Execution-delay - same as in the Load/Store model.

Resource Dependency - Like in the Load/Store model, in this model there is a Read-After-Write resource dependency too. The Data-Cache structure is the same as in the Load/Store model but in this model a write instruction followed by a read instruction causes a 2-cycles delay which is twice as long as in the Load/Store model.

Theoretically, a resource-dependency can be eliminated by incorporating additional hardware resources and performing memory references out-of-order. In this model, as in the Load/Store model, we do not introduce any additional hardware and therefore the RAW delay is not removed.

Data Dependencies - There are 3 types of data dependencies: register interlock, AG interlock and AG-to-AG interlock. In order to reduce the delay caused by these interlocks we add 4 bypasses to the pipeline. The pipeline structure with these new bypasses is presented in fig. 5.

4By compiling and simulating the workload without this restriction we found that the average frequency of memory-to-memory instructions is 4.4%. In order to support memory-to-memory instructions at a throughput of 1 instruction per cycle, we have to add to the pipeline a second port to the data-cache and one more adder.

Figure 5: Symmetric pipeline: final structure
Register interlock - When the source-operand of an instruction is a register and that register is the destination-operand of the previous instruction, a register interlock arises and the pipe is stalled for 1 cycle. The EX-to-EX bypass eliminates all register interlocks. In particular it eliminates all LOAD delays, which account for 80% of the pipeline delays in the Load/Store model.

AG-to-AG Interlock - This interlock is similar to the interlock between two computational-instructions in the Load/Store model: The adder at the AG-stage computes a result and 1-3 cycles later it needs this result as one of its source-operands. Example: The solution

```
II: ADDR MEM[rl+1],rl  # Increment rl
I2: CMP O,MEM[rl+O]    # Use of rl in address calculation
```

is similar too: "remember" the last three results. "Remembering" means a three register FIFO: BP1, BP2 and BP3. The bypasses are dotted to stress the point that they cannot be used to transfer to the AG-stage results that were computed at the EX or ME stages. Using these bypass-registers, the AG-stage can "read" an operand from the output of AG, ME or EX by reading its own local copy from bypass-register BP1, BP2 or BP3 respectively.

AG Interlock - When an instruction that performs an address-calculates the ID-stage and it cannot read its operand because the register is the destination of one of the three previous instructions, an AG-interlock arises and the pipe is staled for 1-3 cycles. This interlock is inherent to the pipeline structure because the operand that is needed at the AG-stage is calculated at latter stages: EX (if it is a computation) or ME (result of a LOAD). The best we can do is to reduce it. Instead of waiting until the result is written to the RF, we can read it as soon as it is ready. Two bypasses are needed: One from EX to AG to bypass results of computational instructions and one from ME to AG to bypass the result of LOADs.

6.4 Summary of Results

Table 6 summarizes the performance of the final Symmetric model:

<table>
<thead>
<tr>
<th></th>
<th>DC</th>
<th>GREP</th>
<th>PTC</th>
<th>SED</th>
<th>SORT</th>
</tr>
</thead>
<tbody>
<tr>
<td>Instructions</td>
<td>467,256</td>
<td>465,161</td>
<td>640,448</td>
<td>142,793</td>
<td>531,296</td>
</tr>
<tr>
<td>Cycles</td>
<td>640,864</td>
<td>518,130</td>
<td>804,092</td>
<td>164,086</td>
<td>585,492</td>
</tr>
</tbody>
</table>

The average CPI is 1.199. The AG-interlock, the RAW resource dependency and the execution-delay cause an average delay of 0.055 CPI, 0.084 CPI and 0.060 respectively.

Figure 6 summarizes the evaluation of the Symmetric model:

The bars, from left to right, represent the performance results of the following:

1. Plain Pipe.
2. With EX-to-EX Bypass.
3. With AG-to-AG Bypass-Registers.
5. With ME-to-AG Bypass.
6. Ideal pipe - Path-length.

Note how in two applications, DC and PTC, the final average CPI is much higher than the average because of their high execution-delay. (They execute MUL, QUO and REM relatively frequently).

Note also, that in SORT the fourth bar (left to right) is slightly lower than the fifth bar. This performance degradation is caused by an anomaly of the code-reorganizer and not by the bypass form ME to AG.

7 Control-dependency

Up until this point, all the cycle counts we reported did not include the delay caused by control dependency (branch-delay). In order to make a complete comparison we evaluate the difference in the branch-delay too.

Unconditional Branches - The two model are equal in terms of unconditional branches. There is a 1-cycle delay when the IF-stage waits until a special adder at the ID-stage computes the non-sequential PC.
Conditional Branches - We use a CMP instruction to perform the comparison and set the condition flags. A BRcond instruction computes the branch-target and selects the next-PC according to the flags. As with the unconditional branches, the BRcond computes the non-sequential-PC at the ID-stage. The CMP instruction evaluates the condition at the EX-stage. If the IF-stage waits until the condition is resolved at the EX-stage before it fetches the next instruction then each BRcond results a delay of one cycles in the Load/Store model and three in the Symmetric model. Instead, we incorporate a branch-prediction mechanism. The two most simple (and common) prediction policies and their delay are:

**Table 7: Conditional branches delay (in cycles).** L/S stands for Load/Store and Symm for Symmetric.

<table>
<thead>
<tr>
<th>Outcome</th>
<th>Prediction</th>
<th>Never-Taken</th>
<th>Always-Taken</th>
</tr>
</thead>
<tbody>
<tr>
<td>Branch Not-Taken</td>
<td></td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Branch Taken</td>
<td></td>
<td>1</td>
<td>3</td>
</tr>
</tbody>
</table>

Under the assumption of 15% wrong prediction ([Le84], [DiM87], [McH86]), the preferable prediction-policy for both models is never-taken. For correct prediction, both models perform equally. For incorrect prediction, the delay in the Symmetric model is two cycles longer than the delay in the Load/Store model because the EX-stage is two stages ahead. Using branch-spreading, we managed to reduce the average delay in the Symmetric model from 3 to 2.81.

In order to compute the overall delay due to conditional branches we have to know their frequency. This information is presented in table 8. Note that both models execute the same number of conditional branches but their frequencies are not the same because the Load/Store path-length is longer. The average delay due to conditional branches (assuming 85% correct prediction) is therefore:

In the Load/Store model: $0.15 \times 1.00 \times 0.1473 = 0.0221$ CPI.

In the Symmetric model: $0.15 \times 2.81 \times 0.1656 = 0.0698$ CPI.

The difference between the two models is 0.048 CPI.

### 8 Comparison Between The Two Models

#### 8.1 The Path-length

In table 9, we see that the use of a more powerful instruction-set reduces the path-length of a Load/Store architecture by an average factor of 1.12 (In this section we use a geometric mean whenever we report an average ratio).

<table>
<thead>
<tr>
<th></th>
<th>DC</th>
<th>GREP</th>
<th>PTC</th>
<th>SED</th>
<th>SORT</th>
</tr>
</thead>
<tbody>
<tr>
<td>L/S</td>
<td>500,152</td>
<td>542,143</td>
<td>685,488</td>
<td>175,644</td>
<td>569,558</td>
</tr>
<tr>
<td>Symm</td>
<td>467,256</td>
<td>465,161</td>
<td>640,448</td>
<td>142,793</td>
<td>531,296</td>
</tr>
<tr>
<td>Ratio</td>
<td>1.07</td>
<td>1.17</td>
<td>1.07</td>
<td>1.23</td>
<td>1.07</td>
</tr>
</tbody>
</table>

**Table 9: Path-length**

The path-length reduction result from the fact that the Symmetric model executes less LOAD and STORE instructions because they are frequently combined with an operator into an operation on memory. A similar result, 16%, was reported in [FMM87].

#### 8.2 Number of Cycles

As explained before, a reduction in the path-length does not necessarily result in a performance gain. More powerful instructions are also more complicated to execute. In order to measure the performance contribution of the Symmetric architecture we implemented two pipelines, one for each architecture. Table 10 presents the performance of these pipelines:

<table>
<thead>
<tr>
<th></th>
<th>DC</th>
<th>GREP</th>
<th>PTC</th>
<th>SED</th>
<th>SORT</th>
</tr>
</thead>
<tbody>
<tr>
<td>L/S</td>
<td>642,427</td>
<td>570,213</td>
<td>796,831</td>
<td>199,753</td>
<td>640,694</td>
</tr>
<tr>
<td>Symm</td>
<td>640,864</td>
<td>518,130</td>
<td>804,092</td>
<td>164,086</td>
<td>585,492</td>
</tr>
<tr>
<td>Ratio</td>
<td>1.06</td>
<td>1.10</td>
<td>0.99</td>
<td>1.22</td>
<td>1.09</td>
</tr>
</tbody>
</table>

**Table 10: Optimized pipeline performance**

The average performance-ratio is to 1.078, smaller than the path-length ratio. This is because the Load/Store pipeline executes an average instruction in 1.151 cycles while the Symmetric pipeline does so in 1.199 cycles. The extra cycles (in both models) results from data-dependency and resource dependency (in both models the execution delay is the same and branch delay is not included).

### 8.3 Data-dependency

In the Load/Store model, data dependency delay results from LOAD-delay which causes a 1-cycle delay. In the Symmetric model data dependency results from AG-interlock, which causes a 1 or 2-cycle delay. Table 11 presents the delay which results from data-dependencies. (The numbers are in CPI units):
The average delay caused by data dependency is 0.075 CPI in the Load/Store model and 0.065 CPI in the Symmetric model.

### 8.4 Resource-dependency

The only resource dependency results from contention on the data-cache. Both models use the same data-cache: a two-way set-associative physical cache. A STORE which is immediately followed by a LOAD causes a 1-cycle delay in the Load/Store model and two cycles delay in the Symmetric model. Table 12 presents the delay which results from resource dependencies:

<table>
<thead>
<tr>
<th></th>
<th>DC</th>
<th>GREP</th>
<th>PTC</th>
<th>SED</th>
<th>SORT</th>
</tr>
</thead>
<tbody>
<tr>
<td>Load/Store</td>
<td>0.076</td>
<td>0.035</td>
<td>0.041</td>
<td>0.116</td>
<td>0.107</td>
</tr>
<tr>
<td>Symmetric</td>
<td>0.027</td>
<td>0.063</td>
<td>0.076</td>
<td>0.042</td>
<td>0.066</td>
</tr>
</tbody>
</table>

Table 12: Data dependency delay

The average delay caused by data dependency is 0.075 CPI in the Load/Store model and 0.055 CPI in the Symmetric model.

### 8.5 Overall Performance

Table 13 presents the overall performance including the branches-delay:

<table>
<thead>
<tr>
<th></th>
<th>DC</th>
<th>GREP</th>
<th>PTC</th>
<th>SED</th>
<th>SORT</th>
</tr>
</thead>
<tbody>
<tr>
<td>L/S</td>
<td>652,451</td>
<td>578,806</td>
<td>808,783</td>
<td>204,764</td>
<td>657,209</td>
</tr>
<tr>
<td>Symm</td>
<td>689,035</td>
<td>542,279</td>
<td>837,678</td>
<td>177,998</td>
<td>631,889</td>
</tr>
<tr>
<td>Ratio</td>
<td>0.96</td>
<td>1.07</td>
<td>0.97</td>
<td>1.15</td>
<td>1.94</td>
</tr>
</tbody>
</table>

Table 13: Performance with branches-delay

The total number of cycles with branches-delay is computed by adding the number of cycles without branches-delay (reported in Table 10), to the number of branches (reported in Table 8) multiplied by the average branch-cost (0.15 for the Load/Store model and 0.42 for the Symmetric). Taking into account the branches-delay, the performance-ratio between the two models becomes 1.0375.

### 9 Summary

Two pipeline models, one implementing a Load/Store architecture, the other a Symmetric architecture were compared under identical simulation environments. The Symmetric architecture instructions are more powerful, but also more complex; therefore the pipeline model for the Symmetric architecture contains an additional stage with additional adder, more bypasses and extra port to the register file. Our simulations show that the path-length of the Load/Store architecture is 1.12 longer than that of the Symmetric architecture. Nevertheless, most of this advantage is lost because of various pipeline delays that reduce the speedup factor from 1.12 to 1.0375. The main delaying contribution is due to resource dependency (0.064 CPI) and control dependency (0.048 CPI).

For more details about this study see [Dan88].

### References


