Performance Evaluation of the Scheme86 and HP Precision Architectures

Henry Manyan Wu

Submitted to the
Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements
for the Degree of
Master of Science

Abstract

The Scheme86 and the HP Precision Architectures represent very different trends in computer processor design. The former uses wide micro-instructions, parallel hardware, and a low latency memory interface. The latter encourages pipelined implementation, and visible interlocks in both control and memory operations. To compare the merits of the two approaches, algorithms frequently encountered in numerical and symbolic computation, e.g. garbage collection, arbitrary precision integer arithmetic, pattern matching, and FFT, were hand-coded for each architecture. Timings were done in simulators and the results were examined and evaluated to determine the equivalent computing power of each design in the presence of a fast memory system. Based on these measurements conclusions were drawn as to which aspects of each architecture are suitable for a high performance computing engine.

This research, supported in part by the Hewlett Packard Company and the Advanced Research Projects Agency of the Department of Defense under Office of Naval Research contract N00014-86-K-0180, is reproduced with their permission.

1. Introduction

This thesis describes experiments which measured the relative performance of the HP Precision Architecture (internally known as Spectrum) and an experimental processor architecture known as Scheme86. The main objective of the experiments was to compare the different ways in which the two types of processor provide fine grain parallelism to increase the computation rate in the presense of a high speed memory system. In order to perform the comparison without allowing either architecture to suffer from arbitrary implementation and configuration constraints, we used simulation of hypothetical systems, rather than measurement of real machines, to keep external parameters as similar between the two as possible. The results were used to analyze the various approaches in designing high-performance processing units for scientific and engineering applications.

1.1 Background

Most von-Neumann architectures can be viewed as having two subsystems, the processor and the memory system. Operands, results, and sometimes instructions are stored in memory. The processor fetches data from memory, operates on them, and then stores the results back. Frequently the processor requires multiple cycles after fetching a complete set of operands before it is ready to update the result in the memory and initiate another set of fetches. On the other hand the memory system may need more than one cycle to complete a read or a write request. An efficient architecture carefully balances the bandwidth of the two subsystems and overlaps their operation so that the computer may execute faster than their combined delay allows.

Current VLSI technology allows very fast memory systems to be built. It is no longer unreasonable to expect that memory systems, several tens of megabytes in size, can be built with an access time of less than 100 nanoseconds. To efficiently utilize the ever increasing bandwidth of memory, the processor must be designed to be able to keep the memory system busy so that the computer can run at an increased rate. Otherwise the processor will be the bottleneck and the cost associated with having high speed memory will be wasted.

The processor can efficiently utilize memory bandwidth if it runs fast enough so that whenever the memory system is free, it has prepared a memory transaction pending service. Given a particular hardware implementation technology, the use of parallelism can increase the speed of the processor. Parallelism can come in two forms. Processors can incorporate multiple execution sites so that several operations can take place simultaneously. This exploits parallelism by using more hardware or *space*. On the other hand, pipelining allows consecutive operations to overlap their execution by dividing the hardware into chunks that operate on different parts of each operation. This approach can optimally multiply the speed of the machine by the depth of the pipeline. This creates parallelism by efficiently using *time*. Each of these approaches has its pros and cons, and is not fully effective for all applications. In the restricted case of a single pipelined interface to memory, it can be proved theoretically that pipelined machines are slightly more flexible to program and utilize than their multiple execution unit counterparts.

The idea of providing multiple execution sites is not new and in fact has been used since the early days of computer architecture, for example the CDC 6600. Many computers have separate arithmetic units that handle effective address calculations. The more recent Very Large Instruction Word (VLIW) computer proposed by researchers at the Yale University is a good example of machines using a large number of execution units to increase their performance [Ellis 86]. Scheme86, an experimental processor designed at the MIT Artificial Intelligence Laboratory, also has multiple executon units and a long instruction word [Wu 86].

The current vogue in computer architecture is what are frequently called Reduced Instruction Set Computers (RISC) which are essentially machines with only one execution unit but heavily utilizes pipelining to increase its throughput. What made this approach attractive is the need to keep the entire processor on chip in

VLSI implementations, avoiding the problem of critical signals propagating through the relatively low bandwidth channel imposed by the physical package. Some people also believe that freeing up space for more on-chip storage, in the form instruction and data caches or registers, is a better tradeoff than incorporating extra processor hardware. Many academic RISC machines have been designed at the University of California at Berkeley. Hewlett Packard, a major manufacturer of commericial data processing equipment, recently released the instruction set and architecture of their next generation computer. This processor design, called the HP Precision Architecture (or Spectrum), was designed based on extensions of the RISC philosophy [HP 86].

Making a distinction between machines with parallel execution units and those with pipelining can be deceptive. In fact, most practical machines provide, to some extent, parallelism in both forms. In the interest of analyzing architectures, however, this distinction is helpful to understanding the issues involved.

In this experiment we tried to figure out the merits of the two approaches by comparing the runtime of these architectures on hypothetical implementations. Whenever necessary we assumed that the technology was high-end TTL. By comparing how competitive one processor was against another, we looked at which architecture was more attractive from the standpoint of implementation. We also studied our claim that the pipelined approach is fundamentally more flexible, but found that multiple execution units can be more appropriate in special cases.

We needed concrete examples of both kinds of architecture in order to carry out our measurements. For this purpose we picked Scheme86 , a primarily *space-parallel* machine, and Spectrum, an architecture that encourages highly pipelined implementations. Both of these architectures have been designed or implemented in stock TTL technology, giving us a reference with which to pick appropriate operating parameters in our simulation.

2. Theory

This experiment centered around the measurement of processor architectures. In particular we compared two particular processor designs based on two different ways of using parallelism to better utilize memory bandwidth, namely parallelism in time and parallelism in space. Here we discuss why keeping memory busy is important, how the two kinds of parallelism differ, and their pros and cons.

2.1 The Memory Bottleneck

Every computational problem is solved by the processor fetching operands from the storage system, operating on them, and then writing the results out to storage again. Given finite delay involved with fetching and writing to memory, the computational task can complete no faster than the time it takes to bring all the operands into the processor. This situation happens *if* the processor issues read requests as quickly as the memory can accept them, and the processor is able to either overlap its operation with that of memory or take negligible time to work. If the processor can manage to make this happen, by having enough computational resources, it needs to be no faster, since the bottleneck is completely in the memory system.

This is of course not to claim that the scenario is at all accomplishable. High performance processors try to approximate the situation by using several means. One is to cache operands in fast, multi-ported local registers to reduce the need to access memory. This approach pays the price during context switches, when registers have to be saved and restored. Another approach is to use high speed hardware technology. The price is paid in component cost, the amount of heat generated, power consumption, interconnect technology, and design complexity. The processor can also make use of parallelism. As we shall discuss in future sections, parallelism can come both in the time domain or the space domain.

The use of parallelism, coupled with the need to interact with memory over some channel with finite bandwidth, gives rise to a scheduling problem when mapping a computational problem onto the processor's hardware resources.

2.2 Parallelism in Space

Processors can increase their capacity by providing multiple execution sites. Since this approach makes use of more hardware to provide extra speed, we refer to it as parallelism in space. In this scheme, several threads of computation, optimally with few interdependencies, can take place on separate execution units simultaneously. Unlike perhaps the more familiar, although more recent, idea of multiprocessor systems, this kind of parallelism works within *each* processor at a finer grain. Multiple execution units make it possible to have more than one register transfer operation take place at once. Like all kinds of parallelism, however, it introduces problems having to do with dependencies and synchronization.

To illustrate we look at a machine with n identical execution units. In every quantum of time t, n operations can take place. Hence, in an amount of time equal to T, the number of operations carried out would be $(\frac{Tn}{t})$, counting even the ones which cannot be usefully employed. The problem is that since these operations are fired at the same time, they cannot make use of each others results. Furthermore, we may perform branching only at n operation intervals. Because these operations generate their results at the same time, only one of these results may be an address to memory unless it is organized in multiple banks and has separate paths to them. Since all these operations require their operands simultaneously, they must all use the same result returning from memory.

2.3 Parallelism in Time

Pipelining is a way of providing parallelism by splitting up computation inside the processor into chunks. As computation flows within the machine to utilize different hardware resources, any subsystem in the processor that is freed up can be used to execute parts of subsequent instructions.

In this scheme, operations are started one by one. Assuming that n pipe stages make up the complete path for one operation, and each stage has a delay of $(\frac{t}{n})$, then the latency of one full operation is still t. However, as soon as one operation passes through the first stage, i.e., after $\frac{t}{n}$, a second one can start. The number of operations that can be initiated in time T is equal to the total time divided by the interval between the start of successive operations. Therefore the number is $(T/\frac{t}{n})$ or $(\frac{Tn}{t}$, exactly the same as that of a machine with n execution units. This number can be reached only if the entire stream of operations within T can be used. We have also ignored the overhead in logic delay necessary to implement the pipeline registers and control logic.

Here the dependency restrictions are less strict than in the case of parallelism in space. Usually the result of an instruction is generated before n stages have passed, with the remaining stages being used for updating. This means that before n operations have taken place, this value can be used, provided that we *forward* it to the appropriate place. Since results are generated sequentially, they can be sent one by one to a single banked memory system, with the restriction that it either has to be able to process a request in less than time $(\frac{t}{n})$ or correctly queue the requests in its own pipeline. Fortunately this kind of memory is not uncommon. In terms of control flow, the worst case is that branches take effect in time t or n stages after they are issued, but they do not have to fall on multiple of n operation boundaries. All things taken ino account, it is a little easier to schedule instructions on a pipelined machine than one with multiple execution units.

2.4 Which Is Better?

It would appear from the above discussion that pipelining is the kind of parallelism we want. First of all it imposes less restrictions in terms of data and control dependencies. It interfaces better with single channel, pipelined memory that is commonly in use. Perhaps most importantly, it requires little more hardware to implement. This is particularly important in VLSI implementations. It may plain not be possible to fit multiple execution units on a single chip. Because of the bandwidth bottleneck across the pins on a chip, it may be advantages to fit memory, in the form of caches, within the chip rather than sacrifice the space for extra processor hardware. This is in fact one of the chief premise of RISC architecture believers, who advocate the use of minimal hardware but heavy pipelining to increase performance.

If this is the case, why would one ever want to employ parallelism in space? The speed of a pipelined processor increases with the number of stages and decreases with the latency of the slowest stage. The number of pipe stages cannot be increased beyond the point where the overhead of implementing an extra stage compared to it's latency is prohibitive, or where the resulting load/use and branch delays become impossible to utilize. The latency of each stage is determined by the hardware technology used, and generally speaking the faster the technology is the more expensive and complicated it becomes to design and fabricate. In light of these real life considerations, at some point it will be less profitable, if not entirely infeasible, to improve performance using pipelining techniques compared to providing multiple execution sites.

Some of the features that make pipelined machines flexible and easy to program involve more hardware and complexity than a bare-bones RISC machines. The technique of forwarding increases the amount of connectivity between pipe stages, even those not adjacent to each other. In the interest of object code compatibility and density, some processors provide automatic interlocking, using even more hardware and time.

The goal of this comparison is to experimentally measure the effectiveness of the two types of parallelism as used in two particular architectures hypothetically implemented in a popular technology, namely TTL. We wish to show whether by using parallelism in space we can implement processors with performance comparable with or exceeding that of pipelined processors.

Since both types of parallelism clearly have their own advantages, we expect the optimal solution is one that combines the use of both. Accordingly, we do not claim that either pipelining or multiple execution units is superior or is the preferable way of designing computers. What we wish to accomplish is to debunk theories which unfortunately do try to claim just that. The results we find are useful in the future design of high-performance processors for scientific and engineering operations.

3. The Scheme86 Architecture

Scheme86 is an architecture designed and being implemented at the Artificial Intelligence Laboratory at MIT [Wu 86]. It was originally designed to be a microcoded architecture running a Scheme interpreter as its control program, executing binary Scheme instructions as opcodes. However, its data-paths and control structures are general enough that it will execute user microcode programs of any flavor downloaded into its writable control store. As a matter of fact, for the purposes of this evaluation, Scheme86 is not being seen as a Scheme engine, but rather as a general purpose, long instruction word machine at the micro-architecture level [Ellis 86].

This viewpoint is particularly interesting because the Scheme86 architecture features a distinctive data-path design. The processor incorporates a total of three independent execution units (`EU`) which operate simultaneously. Two of these are general arithmetic units, and differ only in how they are interfaced to the memory system. The first of the two is dedicated to performing address computation, while the other is used solely for data transactions. The third execution unit performs both a register transfer and a full-word equality test while its operand paths are used to read out register contents as addresses and data to the memory system if desired.

Feeding operands to these execution units is a register array with six logical read ports and three logical write ports. [1] A total of 60 general registers are available, in addition to several special purpose interface registers. The data-paths in Scheme86 are tagged, meaning that some portion of a data word is reserved for dynamically identifying the type and size of the target of pointers. In addition to the three full-word execution units mentioned above, Scheme86 has a dedicated type code unit capable of performing two simultaneous type comparisons. In proposed implementations, Scheme86 tag fields are 8 bits wide, while the width of the datum fields are 24 bits. The architecture uses word addressing, giving an address space of

---

[1] The hardware part is actually a dual-ported register file. The additional read ports are implemented using copies of the array, while two write cycles give the effect of three write ports.

64 megabytes.

Scheme86 is often referred to as a "long instruction word" machine because of its wide instructions, which are over 160 bits in length. An instruction has fields for controlling the register sources and results, arithmetic operations, and various hardware functions. Every instruction explicitly specifies both the next instruction and an alternate location for the purpose of conditional branching. The flexibility in the instruction format allows powerful computation to be performed during a single cycle.

For example, the assembly language instruction shown in Figure 3.1, which performs two arithmetic operations, a register transfer, a memory operation, two type checks, and a conditional jump, can be executed on Scheme86 in one cycle. In this instruction, `EU1` generates a memory address, `EU2` computes the data, `EU3` performs a register transfer, while the type code unit checks if the type of two operands are integers. A memory write is initiated and execution continues either at an error handler or the integer add routine.

```
(state
 (EU1 (assign ((reg memory-address) (reg stack)) (+ (reg stack) 1)))
 (EU2 (assign ((reg memory-write-buffer) (reg C)) (+ (reg A) (reg B))))
 (EU3 (assign (reg A) (fetch (reg memory-read-buffer))))
 (TCU (test A (reg A) (type fixnum))
      (test B (reg B) (type fixnum)))
 (store (reg memory-address) (reg memory-write-buffer))
 (if (or (type-incorrect? a)
         (type-incorrect? b))
     (goto TYPE-ERROR-HANDLER)
     (goto FIXNUM-ADD)))
```

Figure 3.1 Sample Scheme86 Instruction

The Scheme86 architecture is optimized for low latency memory accesses. The memory read results are received not by the general register file, but instead are fed into specialized interface registers (`MEMORY-READ-BUFFER`). The timing of the processor cycle is arranged so that register reads are completed just as the content of the buffer becomes stable, and the arithmetic units may proceed to operate immediately. The result of the operation is then fed directly into the memory system again as the next address to use. While memory is being accessed the processor performs internal book-keeping chores, such as instruction fetching (from a separate control store) and register updates. This strategy allows a memory transaction to complete and the result be made available for use in the cycle immediately following the one issuing the transaction. Because of the overlap, it is arguable whether Scheme86 makes use of any pipelining techniques. Even if we were to consider this approach pipelining, the pipe length is no more than two deep, no possibility of delay slots or interlocks exists, and the latency of each pipe stage is not the same, something very different from classical pipelined machines such as Spectrum.

The main advantage of this approach is that the memory-to-memory latency is kept to an absolute minimum. For example, a pointer fetched from memory can be incremented and immediately used as a read address after one `ALU delay`. Usage similar to this occur frequently in modern, structured and object oriented programming languages, especially in scientific and engineering applications. Many of LISP programs spend much of the time doing `CAR-CDR` chaining, [2] and C programs often make extensive use of references to data in structures organized as linked lists. In general, all indirect data accesses, including basic system operations like variable references from stack frames, benefit from this approach.

The drawback of this approach is that the processor cycle time is now linked to memory speed. In fact, assuming that the processor can be designed faster than memory, then the minimum cycle length of Scheme86 is the latency of memory plus the delay through its arithmetic unit along with some register delays. This was not a

[2] An informative study performed recently at MIT showed that during the compilation of a Scheme program, `CAR` and `CDR` were the most frequently called primitives, and each alone out-numbered the frequency of arithmetic instructions by an order of magnitude.

concern in the case of Scheme86 's, because the architecture was designed with high speed memory in mind. In proposed implementations, fast static CMOS memory will be used for the entire memory system to keep the latency to less than 100 nanoseconds. As it turned out, the processor's internal operation will be the bottleneck. All in all, Scheme86 machines are expected to run at cycle times around 150 nanoseconds, given the current plans for implementation in Advanced Schottky TTL technology.

Scheme86 represented a collection of ideas but was never a very precisely defined architecture. For the purposes of simulation in this experiment, we closely modeled the description of the machine as specified in [Wu 86], which detailed the architecture and a proposed TTL design. Since then implementation plans have evolved and many optimizations were suggested. These were not accounted for in this comparison. As it turned out, the optimizations would not have greatly affected the figures collected in our particular set of experiments.

4. Spectrum

The Spectrum architecture is an extension of the Reduced Instruction Set Computer (RISC) design principles. It shares with RISC's the ideas of direct implementation of the instruction set, a small number of fixed size instructions, a limited number of addressing modes, reduced memory access through the use of load/store instructions, and visible pipeline delays in the form of branch delay slots and memory access interlocks. The Spectrum architecture is well documented in literature [HP 86]. Here we will focus on some points that are particularly interesting with respect to this experiment.

One of the first things of interest is the notion of pipelined implementation. The Spectrum instruction set is designed to enable and encourage hardware implementations which divide the work of each instruction into multiple pipe stages. The exact number and function of each stage depends largely on the implementation, but roughly speaking three distinct phases can be identified. They are the instruction fetch/register read phase, the execution phase, and the write phase. In a pipelined implementation using this organization, the machine would simultaneously be operating on three instructions, delegating to each pipe stage the appropriate part of each. This approach has the problem that an instruction starts executing before the previous one can produce a result or cause an effect. This is problematic when the first instruction is a branch. Spectrum solves the problem by using delayed branches, specifying that the instruction following a branch is always executed. Any side effects caused by the second instruction can be suppressed. The time spent in its execution, however, cannot be reclaimed. Because of the existence of these branches the instructions in the pipeline might not always be doing useful work. When this is the case the effective throughput is less than the theoretical maximum, which is three times the reciprocal of each instruction's latency.

A second consequence of the pipelining approach has to do with the timing of memory references. Referring to the above pipe structure and also to figure 4.1,

```
    -------------
t | I | E | W |          I: Instruction/Register Fetch phase
    -----------------
t+1   | I | E | W |      E: Execution phase
        -----------------
t+2       | I | E | W |  W: Write (Commit) phase
            -------------

    -------------
    | t |t+1|t+2|        Effective Throughput
    -------------
```

Figure 4.1 A Hypothetical Spectrum Pipeline

a memory transaction start at the end of the execution phase when the effective address is computed. At that point, the next instruction's execution phase is just starting. It is thus very difficult to finish the transaction in time for this instruction to make use of the result. Depending on the ratio of the latency of each pipe stage to the latency of the memory system, the memory result may not be used until cycles $(t+2)$, $(t+3)$, or even later. The cycles in between are called *load/use interlocks*. For example, a 125 nanosecond Spectrum processor has one load/use interlock when interfaced with a 125 nanosecond memory system, while a 62.5 nanosecond version will have two such interlocks when wired to the same memory. [1] Interlock cycles may be filled by instructions not depending on the completion of pending memory transactions. However, the Spectrum architecture specifies that null cycles are inserted automatically by the hardware when an interlock is detected and the programmer do not explicitly make use of them.

The effectiveness of this approach depends largely on the memory reference pattern of the program. If many independent references need to be made, such as when registers are unconditionally restored from the stack when exiting a procedure invoked with a callee-saves convention, the processor makes good use of the memory system by being able to start requests as quickly as the memory system can receive

[1] Here we assume that caching and virtual address translation, if needed, are performed as part of the memory system.

them. [2] However, when dealing with memory references that are chronologically dependent, this mechanism slows things down because of the extra cycles involved. The results of our experiments seem to indicate that the effect of this problem is minimal when the number of interlocks per read is less than two.

Another noteworthy aspect of the Spectrum architecture is the sole use of load/store instructions for accessing memory. Memory results can only be loaded into the general registers, which also serve as the sole source of data to memory. One implication of this feature is that effective address calculations are never implicit in arithmetic instructions. To add two numbers in memory, for example, we need two load instructions to fetch the operands, one to perform the add, and yet another to write the result out. This strategy helps to minimize the cycle time of simple instructions. In the optimal case where the coder or compiler is effective in keeping operands in registers rather than in memory, faster cycles contribute to shortening the computation time. In cases where memory operations are required, Spectrum provides an adequate mix of addressing modes, including those that modify registers, so that most effective address computation can be done entirely within load/store instructions.

The Spectrum implementation commercially available now is known as the HP9000/840 computer. Similar to Scheme86's proposed implementation, the 840 (internally known as the Indigo) is implemented using standard TTL technology. It has a three stage pipe. Each stage has a latency of 125 nanoseconds. It is debatable whether this number is suitable as a guideline to the speed of this architecture in TTL. It is often claimed that issues like virtual memory, reliability, and commercial requirements slowed this processor down. A Spectrum processor built to academic standards and requirements and does not have to wait for cache and virtual memory operation will probably run faster.

[2] If the memory latency to processor pipe speed ratio is over 2 (more than 1 load/use interlock per read), the memory system must be internally pipelined to allow the processor to issue references in every cycle.

5. Workload Selection

For the results of this experiment to be meaningful, we were very concerned about isolating all extraneous factors of the system we were measuring, that is, the processor architecture. To this end we chose to perform the simulation using small kernel programs, hand-coded in each architecture's native instruction set.

5.1 Rationale and Ramifications

The reasons for this choice is easy to explain. We could not simply calculate the performance of these architectures using standard instruction mixes. The instruction set of these machines are so different from each other and also from a generic processor that no choice of an instruction mix would have been fair and intuitive. The performance of these machines are also highly dependent on the *dynamic* behavior of the programs, in particular the way memory is accessed. Using trace data would also be problematic because of the difference in the two instruction sets.

We wanted to be careful not to include factors imposed by high-level language implementations. Hence it was impossible to use large software applications as our benchmark, since they are typically written in these languages and the task of rewriting them in assembly code would be prohibitive. By keeping the size and number of the kernels small, we were able to optimize the implementation as it was fit on each machine. This was all done by hand-coding, under the assumption that humans would approximate and most probably exceed the performance of the best compilers. The small size of the code also made their behavior very tractable, and any results obtained could easily be explained by reviewing the structure of the code. It was also possible to have multiple experts review some of the kernels to make sure that they were indeed reasonably optimized implementations. [1] This would not have been feasible with real application code.

This particular choice led to some concern that our experiments did not

[1] The benchmarks for Spectrum were reviewed by Hewlett Packard and certified to be efficient implementations of the algorithms.

account for the reality that most of the actual programs that machines built with these architectures will eventually run are going to be in the form of compiled code. Our results would be meaningless if it were substantially more difficult to write a compiler for one architecture than the other. Fortunately, there are existence proofs of excellent compilers for both types of architecture [Coutant 86, Johnson 86, Ellis 86]. In fact, the problem of optimizing for parallelism in time (Spectrum) versus parallelism in space (Scheme86) are very similar in nature, and progress in one area is readily usable in the other [Ellis 86].

Another factor in defense of our choice is that we had a very specific application of these architectures in mind, namely the area of high performance scientific and engineering computation. It was thus possible to select a set of benchmarks that is truly representative of the kind of programs of interest. This was also the reason why we decided not to use the industry's standard set of kernels in favor of a hand picked collection of what we considered common and important algorithms. We were particularly concerned with processor performance, so the fact that kernels do not typically make use of I/O did not bother us.

Still, the choice of the appropriate benchmark kernels is a an ill-understood "black art". The set we came up with included a two-space LISP garbage collector, an s-expression pattern matcher, an arbitrary precision arithmetic package (Bignums), and an integer-FFT routine. Our choices were influenced by many criteria. We wanted both symbolically *and* numerically oriented code to be represented. Each kernel had to be the generalization of a larger class of algorithms. The programs should be the vital parts of actual applications. Finally, these programs ought to make extensive use of low level mechanisms, such as procedure calling, dispatching, and dynamic typing, to name a few. We also tried to ensure that the data structures used contained a balanced mix of both dynamically and statically allocated data with and without manifest types; we included kernels that operate on both linked lists of union structures and simple arrays of integers.

Before hand-coding each of these benchmarks, we implemented each algorithm in either C, Scheme or sometimes both to make sure that they were correct. In addition they provided a control so that we were able to verify that the kernels produced the correct answers. The high-level language implementations also served nicely as documentation to the more obscure, highly "bummed" machine code.

On Spectrum, these kernels were carefully coded to avoid causing load/use interlocks or branch delays. Because the code has to run efficiently on hypothetical Spectrum's with a variable number of interlock slots, we optimized the code for as many load/use delay cycles as we could without sacrificing performance in other ways. The code for Scheme86 followed the Spectrum implementation as closely as possible, although that sometimes meant we had to do a little more work. For example, although doing post-incrementing in one cycle required more computer resources (execution units) than doing pre-incrementing on Scheme86 , we stuck to the former because the code was more natural that way on Spectrum , which executed both with the same speed.

5.2 The Garbage Collector

The garbage collector tested was an assembly language implementation of the one in the C version of the MIT Scheme system. It was based on the Minsky-Fenichel-Yochelson algorithm [FenYoc 69], with extensions to handle statically determined stable regions known as the *constant* and *pure* spaces. This program made heavy use of such low level processor functions as pointer manipulation, memory block move, n-way dispatching, and equality tests and magnitude comparisons. The algorithm is basically a breath first tree traversal useful not only for storage reclamation and compaction, but also for summarizing the state of complicated networks represented as tree structures. As a test case we garbage collected the Scheme runtime system, which contained about half a megabyte of reclamable data. [2]

[2] In the absence of caching and virtual memory, the runtime of this algorithm is independent of the size of the address space or the amount of garbage, and depends

Since Scheme86 was designed with LISP implementations in mind, it was to no one's surprise that given the extensive tag-support in Scheme86 's architecture, garbage collection was the benchmark it fared best on. However, we did some calculations to factor this advantage out and the results turned out to be consistent with those obtained in the other cases.

5.3 The Pattern Matcher

The algorithm for the pattern matcher was taken from [Abelson 85]. Originally written in Scheme, it formed part of a query language system. Pattern matching is a simplified case of unification which in turn is the basis of logic programming system such as Prolog. The program was part recursive and part iterative [3] , and in our hand-coded implementation a caller-saves convention was used to implement procedure calls. Besides exercising the procedure call mechanism of the architectures, it performed a recursive tree-walk down its input parameters and made heavy use of `CAR-CDR` chaining. Part of the algorithm also involved allocating and searching through association lists. This resembled the *deep search* approach for variable lookup in high-level language compilers and interpreters. This benchmark also involved doing tagging operations. The input list and pattern we used contained about 100 elements, with parts up to 10 levels deep.

5.4 Arbitrary Precision Arithmetic (Bignum)

The arbitrary precision integer arithmetic package, often called the Bignum package, was based on the "classical" algorithms from Knuth's work [Knuth 69]. We chose to test two of the operators, namely addition (which is essentially the same as subtraction) and multiplication, knowing that the code for division would be too hard to code, analyze or debug. To do Bignum addition we used loops that iterate through the two operand arrays of digits, adding corresponding elements, keeping

purely on the amount of active data.

[3] As expressed in Scheme, it is more accurate to say that it was sometimes *tail-recursive.*

track of the propagating carry, and writing the result to a third array. This is very similar to another popular benchmark for scientific computation: the vector inner product. Bignum multiplication is just two nested loops doing almost the same kind of operation. Bignum arithmetic is frequently found in algebraic simplification systems and data encryption applications.

The radix size chosen for our package was 24 bits, given that it is Scheme86's proposed word size. In an attempt not to handicap Spectrum's performance using an unnatural word length, [4] we added 24 bit arithmetic instructions to the Spectrum simulator. The semantics of these instructions followed those of their 32 bit counterparts as much as possible. This allowed us to ignore a particular implementation detail of the two architectures and compare the results for this benchmark directly. Note that since Scheme86 's cycle time is theoretically limited by memory and address computation delay rather than data calculation latency, increasing the word size to 32 bits (without a corresponding change in address width) would not affect the worst case cycle time.

Another unspecified instruction added to both architectures is a 24 bit multiply. We felt that actual implementations of these architectures, if used for high-end scientific and engineering applications, would probably incorporate a multiplier in hardware. Such devices which run at high speed are readily available in VLSI implementation nowadays. The overhead incurred by software emulation of its function would be so high that the result obtained in this experiment would be skewed. This instruction was also useful for the FFT benchmark.

The 24 bit multiply, as one would expect, generated 48 bits of result. Since both machines are in principle three-bus architectures, [5] the destination of the top 24 bits of the product cannot be independently specified. We decided that on both architectures, it would go into some predefined register. This is consistent with the way similar problems are handled in these architectures. [6]

---

[4] Spectrum instructions use 32 bit arithmetic.
[5] This is only true of a single execution unit in Scheme86
[6] For example, the `ADDIL` instruction on Spectrum always puts its result in

For addition the operands contain more than 35 digits of 24 bits each. We tested multiplication with a 20 digit Bignum.

5.5 Fast Fourier Transform (FFT)

The FFT algorithm needs no introduction. It is a classic example of a computation intensive scientific program. Here we chose to do integer FFT's, because we did not want to allow floating point performance to affect our conclusions about the processor's architecture. The particular implementation we used came originally from [Gabriel 86], but was modified to work with integers using techniques outlined in [Burrus 85]. Again 24 bit integers were chosen, and the 24 bit multiply instruction was used. A 1024 element FFT was found to cost an amount of time in the same order of magnitude as the other benchmarks, and was thus chosen to be included in our main set of results. Other array sizes provided handy examples for analyzing the effect of different data working set sizes when doing cache effectiveness studies.

register one.

6. System Parameters

For our results to be meaningful, a proper choice of system parameters must be made. Our decision to use simulation greatly increased the freedom in making our choice. However, the problem remained difficult because of our desire to measure a relatively abstract notion, namely processor architecture. Great care was taken to minimize the introduction of extraneous factors. In particular, implementation dependent parameters must not enter into the final results. The problem was further complicated by the fact that the architectures under test were designed with very different purposes in mind.

6.1 Cycle Length (Cycle Time)

The first problem we faced was the assignment of cycle *lengths* to both machines. Although the simulators faithfully and accurately produced cycle *counts* for each benchmark, the results would be meaningless if we did not have a notion of how long each cycle would take, at least relatively, on real machines.

Reasonable estimates for the cycle time can be made because we restricted our experiment to that of measuring the architectures assuming that that they were implemented in standard TTL. Proposed or actual implementations of both machines in this technology were available as a guideline.

Yet the problem was not that simple. The two architectures were originally designed for very different reasons, and they greatly affected implementation strategy. Scheme86 is an academic processor designed to run a very special program: the Scheme interpreter. In many ways reliability and flexibility were sacrificed in favor of a simplistic, speedy design. Spectrum, on the other hand, is a general purpose commercial processor designed to work with virtual storage and time sharing operating systems.

The solution we chose was quite unique. First, we used cycle time estimates to decide which was the faster processor. Then, with that as our control, we speeded up the other processor until it ran as fast as the first, as indicated by the runtime of the kernels. We were then able to examine the resulting cycle time of the second processor and see whether it was possible to implement it using the control processor's technology. This approach was ideal because we were interested not solely in deciding which processor was empirically faster, but also their relative characteristics and their suitability for future implementation in various technologies.

As it turned out, Scheme86 's expected cycle time of 150 nanoseconds made it the faster of the two machines. It was thus chosen as the control processor. Because of the pipelined approach Spectrum uses to access memory, given a fixed memory speed (see section below) it was not possible to continuously vary its cycle time in the interest of keeping the pipe stages balanced. In addition, the processor cycle to memory cycle length ratio determined the number of load/use delays one must insert. With the memory speed we chose (around 100 nanoseconds) it was convenient to test Spectrum at cycle times of 125 and 62.5 nanoseconds. The former number was actually close to the actual cycle time of an available Spectrum implementation, making the choice additionally interesting.

This method of analysis makes sense of course only because our interest was in measuring hypothetical implementations of architectures. It would be extremely faulty when used to infer the merits of the actual machines.

6.2 Memory Speed

Scheme86 and Spectrum were originally designed to be interfaced to two very different kinds of memory systems. The former specified the use of high speed, uniformly organized memory, while the latter expected a hierarchical structure of composed caches and layers of virtual address translation. Since memory speed is a crucial factor in determining runtime, but was not a parameter of our system under test (processor), it was important that we used a uniform memory speed for both processors. We also did not want the effective memory speed to in any way be probabilistic, since that would make our result depend even more on our choice of workload. A flat memory system was thus chosen.

Since our concern was the area of high-performance applications, it was reasonable to assume that the memory speed would be high. Given the memory chip technology at the time of our experiments, we expected such a memory, consisting of tens of megabytes of *real* storage, would have a latency of around 100 nanoseconds. [1] This was in fact the kind of memory system Scheme86 expected to have. On the other hand, this also corresponded nicely to the latency Spectrum 's memory would have if its cache were 100% effective, for example if its size was larger than the working set of data of the problem it was solving.

As discussed earlier, the particular choice of memory speed affected our assignment of cycle times for the machines. As a result of the number we picked, Scheme86's cycle time was fixed at 150 nanoseconds, while we decided to test Spectrum at 125 and 62.5 nanoseconds.

6.3 Instruction Fetch Technology

Scheme86 uses wide instructions that are fetched from a writable control store separate from main memory, and Spectrum fetches relative narrow instructions through an I-cache sourced by the same memory used to hold data. These two approaches makes the available instruction fetch bandwidth of the two architectures potentially different. Considering that Scheme86 was designed to run only the Scheme interpreter, and Spectrum was from the outset a general purpose machine, these differences reflect more the requirements of the architectures' proposed application, rather than the architects' intention if the sole purpose was performance on large scientific applications. It is conceivable to outfit Scheme86 with a Spectrum like I-cache to make it feasible for large programs, or vice versa to equip Spectrum with a separate control store to optimize it for a special piece of code. For the comparison

[1] This number included degradation due to fanout, buffering, reliability margins, etc. The chips we had in mind were 64K static RAM chips with an access time of 45 nanoseconds. At the time of writing a 25 nanosecond version of this chip is available.

of the two architecture to be meaningful, this difference must be factored out.

Because of the fact that Scheme86 instructions are so much wider in width, it would on the surface seem like it instruction bandwidth requirement is much more than that of Spectrum . If this were the case, it would clearly be unreasonable to factor instruction fetching out of our timing, for it would represent a major deficiency of Scheme86 's architecture, and it wouldn't even be clear whether it were possible to construct an adequately fast and large instruction unit for this architecture.

Fortunately, this is in fact not the case. Since both machines in principle use hard-wired instruction, or, in other words, direct implementation of the instruction set, the bit rate of instructions flowing into the processor is governed only by how much work gets done per unit time. The more number of bits flow into the machine, the more work can be specified and performed. A wider instruction simply means that more *semantic information* gets transferred per chunk. In short, since the number of bits needed to specify and control the execution of an operation is constant, the instruction bandwidth is dependent only on the speed at which the operation is performed, and has little to do with instruction width. The faster the machine, the higher the required instruction bandwidth must be.

This optimistic and theoretical view of the situation needs to be qualified. Because the size of instructions are fixed, and the flow of bits is discrete in time, parts of an instruction may be wasted if the amount of work to be perform at the instance requires less than the full word to specify. Because Scheme86 has to work with larger chunks of bits per unit time, there is more of a chance that bits will be wasted. Furthermore, the Spectrum instruction format relies much more on decoding than Scheme86's, so it requires less number of bits to encode the same instruction. All in all, we expect Scheme86 's instruction bandwidth to be somewhat higher than Spectrum's. However, the general feeling with people we consulted seem to indicate that with currently available instruction fetching technology, this difference is insignificant, and may be safely ignored.

Given the above assumption, we proceeded to unify the time in the two simulators we attribute to instruction fetching. To simplify our analysis, this was set to zero. In other words, we assumed that the two machines were equipped large enough instruction caches.

7. Simulation

Simulation, rather than analysis or actual timings, was chosen as the basic method for these experiments. This choice was made not only to make the experiments much easier to conduct, but was actually the only alternative given the requirements of this comparison.

7.1 Rationale

Our goal was to compare architectures, not implementations. As a result actual machines cannot be used for timing because that would certainly introduce implementation dependent factors. For example, on the only Spectrum currently available [1] the processor always interlocks and hence costs two cycles when the instruction following a memory read request involves a register write. Not only is this deficiency not specified in the architecture, it is not expected to be present on future Spectrum machines. This problem was obviously not modeled when designing the simulator.

At least one of the architectures we compared has not been implemented. A proposed design and an implementation plan of Scheme86 were completed in the summer of 1986, but construction never commenced. Working machines using the Scheme86 architecture are not expected to exist until late 1987. Spectrum machines with cycle times fast enough to match Scheme86's performance are also not expected to appear until that time. This ruled out the possibility of measuring real systems.

Some of the system parameters we chose to use were not realizable, at least at the time of the experiments. We wanted the processors to run on a fast, flat memory system with no caching. Although Scheme86 was designed with such a memory in mind, all proposed Spectrum implementations at the performance level of interest has caching and virtual memory built in. The only way to model the memory we want would have been to use a Spectrum with an infinitely sized data cache. This is clearly not feasible except in simulation. We wanted instruction fetching to cost no

[1] Hewlett Packard Series 9000 Model 840

time. This required that we use an instruction cache of an extremely large size. This was very easily accomplished in the simulators.

We made some necessary alterations to the published architectures. To implement the FFT benchmark it was necessary to include a 24 bit multiplier on both machines. To be comparable with Scheme86, Spectrum had to use 24 bit arithmetic when running the Bignum package. These changes were very straightforward to implement in the simulators.

Running the kernels in simulators, we avoided any overhead that may be incurred by the operating system. Furthermore no artifacts associated with hardware or software monitors were introduced. Every instruction executed in the simulator directly contributed to performing the workload. Some of our kernels finished in quite a short time, and the introduction of these external factors would have skewed the results in unacceptable ways.

There was no need to introduce random behavior into the simulators. Each kernel program was run as the single process, with no operating system intervention. The programs were chosen to involve no I/O, and consisted only of processor/memory interactions. The memory system modeled used no caching or virtual storage. The net result of all these was that the behavior of the simulators was completely deterministic. Accordingly only one set of results need to be collected. The need for repetition was eliminated. This would not have been possible if we tried to measure real systems.

The simulators provided additional data on useful metrics such as cache hit rates (had there been caches) and instruction mix statistics, trace data, and hardware utilization. Some of these numbers would have been extremely expensive to collect in real systems. Furthermore the simulators were instrumented with features that made debugging the kernels much easier than it would have been on a real machine.

7.2 Approach

Except perhaps for the FFT kernel, the benchmark programs we chose were not meant to be stand-alone programs. For example, the garbage collector and Bignum package most frequently occur as components of a LISP system. The pattern matcher was taken from a query system. These programs require a non-trivial amount of computation to set up their input.

It is chiefly because of this reason that we chose to embed the simulator as a primitive function or a subroutine other programs. The host programs, all written in C, ran natively on the computer performing the simulation. Instead of calling the kernel directly, the host program invoked the simulator, which then took control and started executing instructions for the machine simulated. The simulated code shared the hosts' data structures in their data space.

In the case of Spectrum, the garbage collector and pattern matcher used the C version of the MIT Scheme system as the host. The simulator itself was also written in C. Two tiny C programs which were nothing more than number readers and printers served as the host program for the rest of the benchmarks. The Spectrum machine code that makes up the benchmark suite were compiled into the host programs' executable images. As a consequent of this decision, the simulator ran only on real Spectrum hardware. This restriction can probably be lifted given some work.

The Spectrum simulator modeled the architecture at the instruction level. Instead of directly modeling the behavior of hardware parts in a hypothetical implementation of the architecture, procedures and C operators abstractly performed the register transfer operations specified by each instruction. It was ironic to discover that writing such a simulator felt very similar to writing *microcode* for Spectrum, except of course that Spectrum was never intended to have microcode.

To be compatible and comparable with Scheme86, 24 bit arithmetic instructions not specified in the original architecture had to be implemented. The semantics of these instructions closely followed their 32 bit counterparts on a real Spectrum machine. The shorter word length made the instructions a lot easier to implement in the simulator in C. In addition, we added multiply instructions to Spectrum. These delivered 48 bit signed and unsigned products, the top 24 bits of which always resides in register one.

The Scheme86 simulator ran under the Scheme system. Scheme procedures and data structures were used to set up input. A Scheme86 assembler was written in Scheme. Scheme86 microcode were stored in a C array, and thus were dynamically reloadable. The Scheme86 simulator was simply a Scheme primitive, which when invoked would fetch instructions from this array as control signals.

Also written in C, the simulator simulated Scheme86 at the hardware level. Though not a logic or gate level simulator, each of its modules modeled a hardware subsystem in the proposed implementation, and each procedure performed the function of some hardware part. In every cycle, the procedures representing each subsystem were always invoked. Their behavior was specified by the encoding of the current micro-instruction. This was not true in the Spectrum simulator, in which, through the instruction dispatch mechanism, only the relevant handlers were used each cycle.

Scheme86 and the machines Scheme ran on (MC68020, Spectrum) use different addressing conventions. The former was designed to use word-addresses, while the latter are byte-addressed machines. Because of the need for Scheme86 to share Scheme's heap, all the memory transaction functions in the simulator must do address translations. This turned out to be hard to do efficiently because of the existence of untyped non-pointer object in Scheme. The actual conversion mechanism chosen was adequate for our set of benchmarks, but would not work in general. Consistency checks were put in to make sure that no odd cases arose. Since this problem does not exist in real life, the cost of doing the conversion was of course not accounted for in the simulation.

7.3 Correctness

Great care was taken to ensure the correctness of the simulators. Both simulators passed the ultimate test: they flawlessly executed all the benchmarks and gave correct results. Small, tractable examples were also used to both validate the simulators' operation and verify that any data collected were accurate.

The simulators used discrete time models. The smallest unit of time was a cycle, in which one instruction completed. [2] The simulators counted cycles. It was the users' job to assign the corresponding amount of time. Because of this simplistic model, accuracy was reasonably assured.

In the case of Scheme86, simulation was done close to the hardware level. Each module in the simulator corresponded to an actual hardware subsystem in the proposed implementation, and each procedure modeled an actual piece of hardware. Many debugging hooks were put in to ensure that each subsystem independently produced the correct result.

The Spectrum simulator, on the other hand, modeled the functional behavior of the instruction set rather than mimicked the hardware. Again the design was modular, with abstract functions used throughout to implement common operations. Not only did this approach ensure a proper implementation for all instructions, it also made adding instrumentation trivial.

All the important metrics measured by the simulators were collected using multiple means. For example, we kept track of the number of times each instruction was used, along with a total count of how many instructions were executed, and made sure the two numbers matched. Instrumentation in the memory system counted the number of memory references, and this had to match the number of load and store instructions issued.

All in all, we believe that the simulation faithfully modeled the characteristics of the architectures and the data collected were accurate.

[2] Both machines had single cycle instructions

8. Peripheral Experiments

Our choice to use simulation opened up opportunities to do all kinds of experiments and data collection. Here we describe some of the various tests we performed on the architectures. The results for these tests are reported in the next chapter.

8.1 Spectrum Load/Use Effectiveness

We were interested in knowing whether a lot of time was wasted on Spectrum waiting for memory to return. In order to find out, the Spectrum simulator was instrumented to separately tabulate unused interlock slots. We ran the kernels at one, two, and three load/use delays per memory read. By doing so we came up with a dynamic measure of how effective these slots can be scheduled.

8.2 Scheme86 Execution Unit Usage

We suspected that utilization of Scheme86 's execution unit would not be high. To prove our theory, in every instruction the simulator marked the execution units used. The number of active cycles for each execution unit compared to the total number of cycles ran through the machine gave us an idea of the percentage of time Scheme86 was idle.

8.3 The Cost of Tags

Unlike Scheme86, Spectrum did not come with tag extraction and manipulation instructions. To study how much this costed the architecture in running our kernels, we examined the instruction mix figures collected by the simulator. The kernels were written so that all bit extract and deposit operations dealt with tags, so their frequency gave us a good indication of how often we had to deal with tagging. By factoring out these cycles in the runtime calculation, we estimated the performance of Spectrum if it had tag support.

8.4 What If We Had a Cache?

Although our major experiments were performed assuming a flat memory system, we simulated the action of caches to find out how they would perform. A number of set-associative caches were tried, using different total sizes and line sizes. We also measured the effect when instructions were cached as well as data.

8.5 What If Spectrum Had Multiple Execution Units?

We changed the Spectrum simulator so that it could execute two instructions in every cycle, applying the constraint that the second cannot make use of the result of the first. However, we allowed both instructions to simultaneously issue memory references. This test gave us estimates of the speedup possible if Spectrum had multiple execution units.

9. Results and Analysis

Experiments were conducted using the methods and parameters described in the previous chapters. Extensive use of script files made the experiments quick to run and easy to repeat. In the following discussion, we shall refer to Spectrum running at 125 nanosecond as the Spectrum125 and the same architecture running at 62.5 nanoseconds as the Spectrum625 .

9.1 Runtime Figures

The most important result is the figures showing the speed of the processors. This information is summarized in Figure 8.1. As is evident from the results, Spectrum requires a cycle time of 62.5 nanoseconds to compare favorably with a Scheme86 implemented using 150 nanosecond technology.

Unit: $10^{-3}s$	Scheme86	Spectrum	
Test	(150 ns)	(125 ns)	(62.5 ns)
Garbage Collection	39.97	99.06	56.24
Pattern Match	0.958	1.672	1.252
Bignum Addition	0.01910	0.03175	0.01844
Bignum Multiplication	0.256	0.423	0.236
FFT (1024)	10.21	20.28	10.14

Figure 9.1 Raw runtimes of the benchmarks at different cycle times.

The results obtained were consistent with what was expected. We know that Scheme86 has three times as many execution units and five times the instruction width as Spectrum, and consequently we expect that Spectrum needs to run over three times as fast to be as powerful. The fact that empirical data show Scheme86 to be only 2.4 times faster (using Spectrum as base) proves our claim that it is more flexible to make use of parallelism gained by pipelining than by multiple execution units. Our numbers also show that any Scheme86 execution unit is, on the average, idle 33% of the time. This is consistent with the performance measured.

9.2 Tag Handling

Figures on the mix of instruction types dynamically encountered during execution of these benchmarks indicate that the absence of tag handling hardware on Spectrum mildly affected its performance on symbolic code. In the garbage collector tag manipulation accounted for 23% of the elapsed cycles. There the type tag were examined and dispatched on. In the pattern matcher 16% of the time was spent masking tags to prepare valid pointers, with an additional 8% used to assemble the correct segment address on Spectrum. In these benchmarks the tagging scheme followed the one used by the MIT CScheme implementation which puts an 8 bit type in the top byte of a pointer. A better encoding mechanism may reduce the cost of tag extraction.

To estimate the performance of a hypothetical Spectrum extension that featured tag handling hardware, we can subtract cycles attributed to tag manipulation from our runtime calculations. Applying this rule, Spectrum125 required 76.3 ms to complete the GC. Spectrum625 used 43.30 ms, as compared to 39.97 ms on Scheme86. In the case of the pattern matcher, the numbers were 1.27, 0.91 and 0.96. No tag manipulation was required on the other kernels. The reader is reminded that since tag handling instructions were often used to fill delay slots, some of the time originally lost do tagging operations may not be reclamable even with hardware support. In other words, the Spectrum numbers represent a *lower bound* on its runtime.

Even allowing for the lack of tag handling hardware on Spectrum , the results still show that it would require a 62.5 ns Spectrum to provide comparable performance to Scheme86.

9.3 Load/Use Interlocks

On the Spectrum architecture with only one load/use interlock after each load instruction (125 ns), almost all such slots can be scheduled for useful work. Less than 2% of the time was spent in interlocked cycles, except for Bignum addition, in which around 9% of the cycles were wasted. With two slots per load the range of idle cycles resulting from an interlock delay is from 0% to 20%. When the number of slots was at 3, the maximum further increased to 30%. The minimum, however, stayed at 0%. [1]

These number show that making use of load/use interlocks is not a problem when there are less than three per load. The large range in the percentages suggests that these problem is highly application dependent.

9.4 Cache Performance

Our hypothetical cache experiments show that caches of the kind and size currently planned for Spectrum machines are effective. In cases where the working set of memory is larger than the cache, [2] the hit rate is near 80%. When the working set is small the hit rate is over 95%. This justifies our original assumption that processors of this type tend to have high speed and low latency memory. The cache performed well even when instructions were also fetched from the unit, proving that instruction fetching would indeed take little time even in real implementations.

9.5 Concluding Remarks

The remaining question is whether it is easier to build a 62.5 nanosecond Spectrum than a 150 nanosecond Scheme86. The existing implementations do not provide reliable clues to answer this question, mainly because they are not designed solely with speed in mind. Many other factors, such as reliability, backward compatibility, and cost effectiveness influence the final product. In the case of Scheme86, time and unsophisticated technology impaired its expected speed. However, judging from the results, it is fair to say that space-parallelism is quite useful and can be employed

[1] The so-called "one write port" problem on some production Spectrum machines was not present in the simulator. This meant that any instruction can be used to fill interlocks.

[2] GC of a 1 Mb heap using a 64Kb cache, and a 4096 element FFT with a 16 Kb cache

to speed up processors without going to the expense of providing fast hardware technology. It can be used effectively when higher speed logic implementations are either unattainable or undesirable.

One noteworthy point is that our results are obtained at some particular memory speed and logic family chosen because they represent what is expected to be commonplace in the kind of technical computing environment we are interested in. Because Spectrum encourages pipelined implementations and memory speed does not govern its cycle time, its performance is more consistent across implementation technologies than Scheme86 . On the other hand Scheme86, as currently designed, will not do very well when interfaced with slow memory. It is also not very suitable for VLSI implementation because of the amount of logic and connectivity it requires. Speculation is that Scheme86 would compare well with Spectrum implemented in ECL and TTL, and Spectrum will fare much better with VLSI technology.

Spectrum was originally designed as a scalable architecture that allows implementations at all performance levels. The primary mechanisms for varying performance are different logic families and memory speeds. The first factor is limited by physical device and signal propagation delays, along with fabrication constraints. The second parameter can be altered by the use of large and effective caches. For very high end applications, even combining these two measures may not be adequate to produce a fast enough machine. As specified now, Spectrum provides few hooks to promote space-parallelism. Since space-parallelism can be an effective means of raising performance, this problem is worth further research.

As a test of this theory we simulated a Spectrum implementation that contained two execution units. In this hypothetical machine two instructions can be executed each cycle when there are no register usage conflicts. [3] Without any recoding to make use of the space-parallelism gained, the speed improvements varied from 35% (garbage collection) to 60% (all others). It is reasonable to expect that with careful code generation, a better speedup can be sustained.

[3] In this test we allowed multiple memory transactions in the same cycle.

10. Conclusions and Future Work

We performed experiments to measure the relative performance of Scheme86 and Spectrum. We found that Scheme86 was very competitive in performance to Spectrum, showing that using parallelism in space by providing multiple execution units can be a feasible method of designing high performance processors that is interfaced to high speed memory. We also showed that pipelined machine are more flexible to program and can be improved when parallelism in space is also exploited.

10.1 What We Did Wrong

In choosing to use hand-coded kernels as our workload, we severely limited the number of test cases we could use. More time and effort would have allowed us to come up with a larger assortment of tests to make our results more statistically conclusive.

We chose to base our experiments on two architectures that actually existed. Although this had the side benefit that our results can also be used to gain *some* insight into Spectrum and Scheme86 , it was very hard to separate the effects of the two kinds of parallelism from the millions of other design decision made in coming with them. Perhaps what's even worse is that since we measured *hypothetical* implementations and configurations of these architectures, our results can be very misleading if used inappropriately to deduce the actual performance of actual Scheme86 or Spectrum machines.

Many implementation details about Spectrum processors were internal to Hewlett Packard. This hurt our ability to draw some more concrete conclusions regarding issues like how fast we can implement one in TTL or other technology, whether it is feasible to put in multiple execution units, and how much the processor is slowed down because of virtual memory and other commercial requirements. The Spectrum architecture and instruction set went public just as this work was about finished, making it extremely hard to consult other experts on the subject. Furthermore Spectrum was not a pure RISC or pipelined machine. In hind sight we may have been better off looking at an academic processor like the Berkeley RISC machines or Standford's MIPS processors, of which much more implementation detail is known.

Although there may be people who claim that the memory speed at which we tested the processor was unrealistically high compared to the current state of the art, we maintain that we probably still tested the architectures at a memory speed which was too *low*. Judging from the improvements in the speed, density, and power consumption of static RAM chips in the last few years, the kind of memory systems we talked about are not only possible, but already available. To make this reasearch be useful to future work we should have assumed even faster memory.

10.2 Future Work

Because of the pace at which the computer architecture field is going, it is probably pointless to spend more effort on these two particular architectures, as they will probably be obsolete in a few years. However, the problem of parallelism at the processor data-path level will remain to be an important one. The answer to how best to improve the speed of computers will change as application and implementation technology evolve. Experiments with the same goals as ours ought to be performed as new theories in computer architecture are proposed in order to provide concrete evidence to performance claims and keep computer architects honest.

Acknowledgments

I would like to thank Prof. Gerald Sussman for supervising my work on this project. He came up with the idea of such a comparison and provided lots of help and encouragement along the way.

This thesis was done in cooperation with the Hewlett Packard Laboratories, where Mike Cannon served as my manager and supervisor. Mike helped me resolve many of the technical issues on a day to day basis and made sure I was always focused in the right direction.

Many thanks is due to Bob Shaw at HP Labs who had many lengthy discussions with me and provided an uncountable number of good ideas to improve my work.

Many thanks is also due to Alan Snyder at HP Labs who had some not so lengthy discussions with me and yet provided an equal number of good ideas.

Other members of our group at HP, including Walt Hill, Roy D'Souza, Jim Kempf, Peter Canning, and Scott Maravich also gave valuable comments and criticisms. Daryl Odnert and Suneel Jain of the Computer Languages Lab, and Vivek Pendharkar of the Architecture Lab at HP Cupertino kindly reviewed and improved the benchmark programs written for Spectrum.

Prof. Jon Allen of MIT served as faculty advisor while I was at HP. He reviewed the proposal on this work and made sure it was on track throughout it's duration. Many good suggestions also came out of meetings from him.

Prof. William Dally took time to carefully review a presentation I made on the subject of this thesis. His comments were very helpful to me in the final preparation of this document.

References

[Abelson 1985]
Abelson, Harold, and Sussman, Gerald Jay, with Sussman, Julie. 1985. *Structures and Interpretation of Computer Programs.* Cambridge, Mass.: MIT Press.

[Burrus 1985]
Burrus, C. S., and Parks, T. W., with Potts, James F. 1985. *DFT/FFT and Convolution Algorithms, Theory and Implementation.* Jon Wiley and Sons.

[Coutant 1986]
Coutant, D. S., Hammond, C. L., Kelly, J. W. 1986. Compilers for the New Generation of Hewlett-Packard Computers. *Hewlett-Packard Journal*, January 1986.

[Ellis 1986]
Ellis, John R. 1986. *Bulldog: A Compiler for VLIW Architectures.* Cambridge Mass.: MIT Press

[FenYoc 1969]
Fenichel, R., and Yochelson, J. 1969. A Lisp garbage collector for virtual memory computer systems. *Communications of the ACM* 12(11):611-612.

[Gabriel 1986]
Gabriel, Richard P. 1985. *Performance and Evaluation of Lisp Systems.* Cambridge, Mass: MIT Press

[HP 1986]
Precision Architecture and Instruction Reference Manual. Cupertino, Ca.: Hewlett Packard Manual Part Number 09740-90014

[Johnson 86]
Johnson, Mark Scott, Miller, Terrence C. 1986. Effectiveness of a Machine-Level, Global Optimizer. *SIGPLAN Notices*, July 1986. The Association for Computing Machinery.

[Knuth 69]
Knuth, Donald E. 1969. *The Art of Computer Programming, Vol. 2.* Reading, Mass.: Addison-Wesley Publications.

[Wu 1986]
Wu, Henry M. 1986. Scheme86 - An Architecture for Microcoding a Scheme Interpreter. S.B. thesis, Department of Electrical Engineering and Computer Science, MIT.