Implementation Independent Architectural Comparison

Andrew S. Grimshaw
Dave Chisholm
Brad Segal
Ricky Benitez
Max Salinas
Peter Kester
Stepehn T. McCalla

Computer Science Report No. TR-90-12
June 20, 1990

This work was supported in part by the Defense Advanced Research Projects Agency (DARPA) under contract number N00014-89-J1699.

Implementation Independent Architectural Comparison

Andrew S. Grimshaw
Dave Chisholm
Brad Segal
Ricky Benitez
Max Salinas
Peter Kester
Stephen T. McCalla

Department of Computer Science
University of Virginia, Charlottesville, VA

Abstract:

Users and designers of CPU architectures lack a solid means of comparing architectures, both extant and proposed. The problem is that running benchmarks, the primary method used, measures *both* the architecture and its implementation (CPU and enclosing system). This report describes a method we have developed to compare architectures in an implementation independent manner. Preliminary results suggest that *if the same implementation techniques* are used architectures fall into performance classes with little difference between members of a class. We identify those features that distinguish the classes. The benefit of adding features to an architecture can now be analyzed: features that do not move the architecture to a better class are a waste of resources and can be avoided.

1. Introduction

Comparing CPU architectures is a tricky business. The most common approach is to run a series of benchmark programs on existing instances or simulators of the architectures and compare the raw performance. Thus we see the number of dhrystones, whetstones, and Livermore loops bandied about as measures of architecture performance [Dongarra87,Dongarra88]. The problem is that benchmark performance depends on many factors, only one of which is the architecture itself. These factors include (but are not limited to), 1) the implementation of the CPU itself: the number of ALU's, the number and width of the busses, whether pipelining is used, and the use of register scoreboarding, 2) the system in which the CPU is enclosed: memory

speed, bus width, cache size, and clock speed, and 3) the quality of the compiler. [1]

What is needed is a method to compare different architectures based on their architecturally specified features, not based on their implementation and compilers. For the purposes of this paper an architecture is defined by its instruction set, the number and type of registers specified, and any explicitly specified (and exposed via the instruction set) separate functional units. This is similar to the notion of an ISP [Bell71]. Implementation independent comparison has not often been tried, and when it has been attempted, it has met limited success[Fuller77]. This is due both to the complexity of the task, and to the lack of a method that permits an unbiased comparison.

Our initial motivation for this research was the desire to meaningfully compare a proposed architecture, the WM [Wulf90a-b], with several extant architectures. The lack of well-established techniques to compare architectures quickly became apparent. We set out to develop a technique that would permit us to compare architectures in an implementation independent fashion, and to study the sensitivity of the architecture to system parameters such as memory latency. We then used the technique to compare five different architectures: the MIL-STD-1750A[DoD], the Intel i860[Intel88], the Sun SPARC[Sun87], the MIPS R-3000[Kane89], and the WM[Wulf90a-b]. This paper describes the method developed, and uses the five architectures as examples of how the method is used.

The technique consists of measuring the *number of virtual clock cycles (ticks)* required to execute the inner loops of a sample set of applications. The method used to calculate the number of virtual clock cycles is the heart of the technique. The basic idea is to determine the number of ticks each instruction would take assuming the best reasonable implementation, i.e.,

[1] Benchmarks can be useful to compare systems though.

the time required if we throw hardware at the problem. This time includes (if applicable) address calculation, memory references, operation execution, and pipeline and branch delays. Reasonable implementations may have multiple ALU's, have non-architecturally specified pipelines, and perform register scoreboarding. Calculating the number of virtual clock cycles for the inner loops proceeds in two steps once the best implementation has been defined. First, we determine the *instruction equation* for each instruction. This is the number of virtual clock cycles required to execute the instruction. The instruction equation has several parameters: memory latency, *load distance* (number of ticks that separate the generation of the address from the consumption of the data), *generation distance* (number of ticks that separate the instruction that generates data in pipeline mode from the instruction that consumes the data), branch latency, whether the memory system will support multiple outstanding loads, and basic operation times, e.g., time for integer add, or floating point multiply.

Once the instruction equations have been determined, they are used to determine the number of virtual clock cycles for the application inner loops. The cycle counts can be calculated using different values for the equation parameters, such as memory latency, to determine sensitivity to that parameter. Sensitivity to memory latency is particularly important. Consider an application running on a 40 mHZ CPU. A clock cycle is 25 nano-seconds. While it is possible to build caches that can operate at that speed, main memories currently cannot. When the application misses cache the observed memory latency will increase to the range of two to six clock cycles, drastically impacting performance. This has already been observed on the Intel i860; applications that exhibit high data locality perform very well, quite close to peak performance, while those that operate largely outside of cache fall far short of peak performance.

After performing the above steps for each of the five architectures two points became clear. First, in the limit, with the same implementation tricks being applied to each architecture, the architectures were partitioned into three classes. Within each class the "performance" of each architecture was very close. Two features separated the three classes, the separation of address generation from data consumption/generation in the execution stream, and the ability to asynchronously generate addresses. Second, the main components of the performance difference between RISC and CISC architectures are RISC pipelines and the RISC load/store, not the "instruction complexity".

The first result is particularly intriguing: architectures can be partitioned into equivalence classes. This tells the architecture designer that only certain features will improve the performance of the machine, mainly those features that move the architecture from one equivalence class to another. Any other feature only adds to implementation complexity, potentially increasing cost. The important questions are then, what features distinguish classes, and what new classes are still to be discovered?

The remainder of this paper is divided into five sections and an appendix. Section 2 compares our approach to other architectural comparisons. Section 3 describes the use of the method in detail, how we determine the best reasonable implementation, how different architectural features and system parameters are mapped into the tick equations, and how the code used is generated. Section 4 presents preliminary results of our comparison on the five target architectures. Section 5 presents our plans for a tool based on our technique. Section 6 summarizes the results. The Appendix contains the source code for the inner loops, the annotated assembly language for each of the target architectures, and the raw performance numbers.

2. Related Work

Three different approaches to architectural comparison have been used in the past, system quantitative (benchmarking), qualitative, and implementation independent. The first of these, benchmarking, is the most widely used. The problems with this approach have already been discussed. Benchmark comparisons are primarily useful when one must select an *existing system* to use for a particular application. For that purpose benchmarks are quite useful. Benchmark studies include [Dongarra88, Funk89], and to some extant the CFA study [Fuller77].

The second approach is to qualitatively examine the features of an architecture [Piepho89]. Architectural features are examined to determine their impact on performance, ease of compilation, protection and security, operating systems support, and efficiency of procedure call. Issues such as the orthogonality of the instruction set, and the number and type of addressing modes, dominate the discussion. These are important issues, but they provide little information on the architectural features that effect performance.

The third approach is to compare architectures in an implementation neutral manner. There has been little success with this approach. There have been attempts where two or three specific architectures have been compared, and specific methods have been developed to compare those architectures [Pleszkun88]. However, the techniques developed assumed implementation features that were common to the architectures being studied. In [Colwell85] the authors show that many of the differences observed between CISC's and RISC's were caused more by the different implementation techniques than by inherent differences between RISC's and CISC.

3. Comparison Method

How can one separate the architecture itself from the implementation techniques used? The basic problem is to find a way to put the architectures on equal footing with one another with

respect to implementation. Several issues present themselves when this is attempted.

- Pipelines: how does one handle architecturally specified pipelines? What if they are of different depth? What if there is no pipeline specified in the architecture?
- Separate functional units: if they are not specified in the architecture, should they be included in the best reasonable implementation?
- Multiple functional units: how much hardware is on chip? Should multiple ALU's that perform micro-operations in parallel whenever possible be permitted in the best reasonable implementation? What about extant architectures that do not have multiple functional units?
- Register scoreboarding: if pipelines are "added" to an architecture how should data dependence be controlled?
- Memory: how is memory treated? Should the CPU wait for memory references? Should register scoreboarding be used even if not architecturally specified? Should multiple outstanding memory references be supported?
- Code quality: how can we keep compiler/programmer quality constant? If we don't, then performance numbers could easily reflect code quality rather than the architecture.
- Superscaler architectures: should we permit or perform multiple instruction issue whenever possible? What exactly is meant by "multiple instruction issue" in the presence of multiple, asynchronous, functional units?
- Non-von-Neuman architectures: How should we treat non-von-Neuman machines? Is it possible, or reasonable, to compare them with more traditional machines? Where does one draw the line?

- Should we make implementation independent comparisons? Designers of RISC architectures would say "no" because they have already made the trade-off in the design of their architectures. They have sacrificed features for speed.
- Unit of measure: What should the unit of measurement be? Should it be "time", clock ticks, architectural complexity, or some cross product of these?
- Bottom line: The bottom line is performance on applications programs. But what applications?

In order to do an architectural comparison one must answer each of the above questions. Often, the answers take the form of assumptions that are made about the system environment. Others can be dealt with by fiat, e.g., we will not attempt to compare non-von-Neuman machines.

We answered the above questions by first defining how we wanted to compare architectures, using "time" as determined by scaling the number of *virtual clock cycles* . The process of obtaining "timing" information for applications consists of four steps. First, the applications must be chosen. We show three: string compare, vector dot product, and an infinite impulse response filter (IIR). A larger number of applications, and more than just the inner loops, must be used before definitive conclusions about the architectures can be made. Second, for each architecture, the instruction tick equations must be derived. The instruction equations tell us how many virtual clock cycles each instruction takes. Third, code must be generated for each application for the target architecture. Fourth, the time cost for the inner loops must be determined for the application. This process can be likened to hand simulation. The result is a parameterized equation for the number of cycles required to execute an iteration of the loop. Below each of these steps is described in detail. Along the way we describe the decisions we made regarding the appropriate bases for comparison.

3.1. Instruction Equations

In this section, we show how instruction equations are derived. We first define exactly what we mean by an instruction equation, and by a *virtual clock cycle*. Then, we describe how we handle pipelines, separate and multiple functional units, memory, and instruction issue (superscaler).

A virtual clock cycle is our basic measure. It is the shortest amount of time that an operation can take. An instruction or memory reference may take one or more virtual clock cycles to complete.

The instruction equation for an instruction tells us how many virtual clock cycles the instruction takes to execute. The instruction equation is parameterized by memory latency, branch latency, and primitive operation times (integer add, floating point add, etc.). We have found that for a particular architecture there are a small number of *instruction classes*. Instructions within an instruction class share the same instruction equation. Thus, a separate instruction equation for each instruction is not necessary.

Unfortunately a single instruction equation is not adequate for all instructions. Some instruction equations also depend on the mode the processor is in (dual instruction mode, or pipelined mode), and on where the instruction is used. These are handled on a case by case basis.

An instruction equation is of the form

$$T(\text{instruction}) = \text{const} + T(\text{arguments}) + (T(\text{operation}))^*$$

Where *T(arguments)* represents the delay waiting for arguments, and *T(operation)* is the time to execute, or begin execution, of operations. Because instructions frequently do not use all of the architectural features, some of the terms in the equation may be zero. The constant term reflects the fact that an instruction takes a minimum of one clock cycle. If all of the other terms are zero,

then the constant time is one. Not all terms are closed form equations. *T(arguments)*, for example, usually includes functions such as *MAX* and *MIN*. In order to model the fact that different terms of the equation may represent the critical path during instruction execution, the entire equation may use functions such as *MAX*. Further, when determining *T(operations)* one must consider the internal data dependencies of instruction execution, e.g., the memory operation of an indexed indirect address mode instruction cannot be performed until the address calculation has completed.

When determining the instruction equations the question of what on-chip hardware to assume arises. We have chosen to assume the best reasonable implementation. For us this means that if the implementation would benefit from having two (or more) on-chip ALU's, then we assume that there *are* two on-chip ALU's. This assumption extends to all non-architecturally defined CPU features. It does not extend to features such as the number and type of registers, or features such as floating point support.

The assumption of a "best reasonable" implementation also includes *full/empty* bits for each register, and pipelined execution if it is not architecturally specified. The *full/empty* bit is used to mask (as much as possible) memory latency and to synchronize with the pipe. Thus, when a *load Rn* instruction is executed, the CPU does not block until the register is actually referenced. If the load instruction is executed several instructions before the value is used, much of the memory latency is overlapped. This extends to non-load/store architectures whenever possible.

The speed and capability of the memory sub-system have a major influence on application performance in real systems. This is particularly true now that CPU speeds are increasing much faster than memory speeds. One of our goals is to study the sensitivity of the target architectures to memory system performance. There are two main variables that describe memory system

performance, bandwidth and latency. The first tells us the amount of data that can be moved between the CPU and the memory sub-system, and the second tells us how many clock cycles it takes to complete a memory transaction.[2] Latency is a more serious problem than raw bandwidth, particularly in shared memory multiprocessors. One can add additional bus lines to increase bandwidth, but the memory and interconnect are only so fast.

We have chosen to assume that the memory sub-system can support as many memory operations as the CPU can provide addresses, i.e., the memory sub-system can support multiple outstanding requests. We have chosen to model latency directly as a variable available in the instruction equations. In our examples below we have used two different values, latencies of two and four virtual clock cycles. Other values can be used if desired.

Example 1

Consider the instruction equation for the 1750A *FMBX* (register indexed floating point multiply) instruction in non-pipeline mode. The instruction equation consists of four terms, the instruction fetch/decode term, the address calculation term, the memory latency term, and the term for the multiply itself. Since the 1750A does not have a *load/store* instruction, we must always pay the full memory latency[3]. Further, since we are not using pipeline mode, the cost of the floating point multiply must be paid immediately. The instruction equation is shown below.

$$T(\text{FMBX R13,R8}) = 1 + (T(\text{Iadd}) + T(\text{MemLat})) + T(\text{Fmul}).$$

Pipelines introduce additional complexity to the instruction equations. If the instruction may be pipelined, then the CPU need not necessarily wait for the results to become available. Instead computation may proceed until either the operation completes (exits the pipeline), or

[2]The existence/non-existence of a cache, and the cache size will influence the average latency. We consider the average case in order to simplify analysis.

[3]This is not strictly true. One could use *move* instructions to emulate *load/store*.

until another instruction that is data dependent on the result executes. If an instruction that is data dependent on the result executes before the result is available, the CPU must wait.

We model this in the instruction equations by not "paying" the cost of the operation until the result is used by another instruction. The cost that is paid at that point is a function of the number of virtual clock cycles required to perform the primitive operation, (e.g., floating point multiply), and the *distance* in virtual clock cycles between the instruction that produces the value and the instruction that consumes the value. If the consumption is sufficiently separated, there is no cost penalty. The general form of this subequation is

MAX((T(operation)-*distance*),0).

Example 2

Consider the following SPARC code fragment:

	Assembly Instruction		*Instruction Equation*
1	LY1:		
2	ldf	[%o1],%f2	1
3	ldf	[%o2],%f4	1
4	inc	%o5	1
5	cmp	%o5,%o0	1
6	fmuls	%f2,%f4,%f6	1 + MAX[T(MemLat)-2,T(Fadd)-5,0]
7	inc	4,%o1	1
8	inc	4,%o2	1
9	bl	LY1	1
10	fadds	%f30,%f6,%f30	1 + MAX[T(Fmul) - 4, 0]

This is the inner loop for dot product. Most of the instructions require a single virtual clock cycle. There are two things to note though. First, the load instructions on lines 2 and 3 *do not* incur a delay waiting for memory. Instead, the memory latency is factored into the cost of instruction 6, i.e., *Memlat-2*. The *2* is the *load distance* between the use of register *f4* and the load into *f4*. Second, the instruction equations of lines 6 and 10 refer to floating point operation times minus a constant, e.g., *T(Fmul) - 4*. The constants are the number of instructions that

separate the start of the computation which generates a value in a parameter register, and the instruction which requires the value. If *T(Fmul)* is less than four virtual clock cycles, then there will be no delay.

To restrict the memory subsystem to a single outstanding memory transaction we must change the instruction equations, adding a memory delay term to load and store operations. For example, line 3 would be changed to:

```
3       ldf     [%o2],%f4               MAX[1,SMR*(MAX[MemLat-0,0])].
```

SMR is a constant defined to be one if the memory system does **not** support multiple outstanding memory requests, and zero if it does. Thus, the instruction equation models the fact that the previous instruction was also a memory operation, and that this instruction must wait until the previous memory operation has completed before it may issue another request. For the remainder of this paper we will assume that *SMR=0* in order to simplify the algebra in the examples.

Once the instruction equations have been derived for each instruction used in an inner loop, the *loop equation* may be determined. The loop equation is an expression that reflects the number of virtual clock cycles required to execute the inner loop in steady state, i.e., once loop startup costs have been paid and the loop is running. The loop equation is parameterized by the same variables as the instruction equations, and is calculated by summing the instruction equations of the loop. For Example 2 above, the loop equation is:

	Instruction Equation
	+ 1
	+ 1
	+ 1
	+ 1
	+ 1 + MAX[T(MemLat)-2,T(Fadd)-5,0]
	+ 1
	+ 1
	+ 1
	+ 1 + MAX[T(Fmul) - 4, 0]

Loop Equation	9 + MAX[T(MemLat)-2,T(Fadd)-5,0] + MAX[T(Fmul)-4,0]

Thus, for the SPARC, using a memory latency of two, a floating point addition time of two, and a floating point multiply time of four, the inner loop takes nine virtual clock cycles to execute.

If the architecture specifies separate functional units we must calculate the instruction equation and loop equation differently. In particular, consider the case where there are separate floating point and integer units. The instruction and loop equations must have two separate components, the time required in the integer unit, and the time required of the floating point unit. An additional consideration is the synchronization required between the units. This is included in the equations as well, in a manner similar to the way memory latency and pipelines are handled.

When there are separate functional units the loop equation has multiple components as well. For each functional unit we compute a separate *functional unit loop equation.* The loop equation is the *MAX* of the functional unit loop equations.

Superscalar architectures present a challenge as well. We only consider an architecture superscalar if it is specified in the architectural definition. A superscalar architecture makes sense only if there are multiple functional units. We feel that we have accommodated this above. The only remaining issue is the instruction fetch and decode time associated with multiple instruction issue. Since we have assumed that instruction fetch/decode has been pipelined away, this is not a problem.

3.2. Code Generation

Code generation is a critical step. The quality of the code generated for each target architecture must be the same or the results may reflect not the differences in the architectures, but differences in the compilers. This is called *compiler bias*. We avoid compiler bias by using the same compiler technology for each target architecture.

We use the *vpo* compiler[Benitez88]. With *vpo* the same optimizations are performed on each target architecture. Further, the instruction selection phase is automated, using a high level description of the target architecture. The generated code is more uniform, and of higher quality, than that generated by hand-written compilers.

Code generation proceeds in four phases. First, *vpo* is ported to the target architecture[4]. Second, each of the three applications is compiled using *vpo*. If loop unrolling is called for it is performed before compiling. Third, we examine the compiler-generated code to determine if there are any optimizations that can be performed to exploit particular architectural features. (*vpo* currently has no notion of memory latency, thus some code motion may be called for to overlap memory access.) Finally, the (potentially) modified code is executed on the target architecture to ensure that the code is correct.

4. Analysis of Results

Using the above techniques we calculated the loop equations for each of the three applications on each of the target architectures. We then determined the cycle counts for each of the loops while varying two parameters, the memory latency, and the operation times. We used two different memory latencies, two virtual clock cycles, and four virtual clock cycles. Two dif-

[4]This has already been done [Whalley90].

ferent basic operation times where used, B.I.T. times and Weitek times[5]. The B.I.T. (Bipolar Integrated Technology) times are always faster than the Weitek times, and reflect the use of a more expensive technology. The two different sets of times are used to illustrate the effect of changing technology, i.e., what is or is not gained by using faster and more expensive parts. The virtual clock cycle times for each implementation technology are shown in Table 1 below.

Table 1		
Operation	Weitek	B.I.T.
Float divide	12	2
Integer divide	10	2
Float multiply	4	1
Other float	2	1
Loads, other	1	1

The results of "running" the three applications using the Weitek times are shown in Figure 1, and for the B.I.T. times in Figure 2. We have provided timing information for both a pipelined and non-pipelined 1750A to illustrate the impact of pipelining. The lower shaded portion of each bar represents the time using a memory latency of two virtual clock cycles. The upper, un-shaded portion represents the time using a memory latency of four virtual clock cycles.

A few comments are in order. First, the WM execution times **do not** reflect the use of register scoreboarding. Since the architectural specification explicitly deals with the management of data dependencies, we felt that it would be inappropriate to perform scoreboarding. In any event, scoreboarding has no effect on WM execution times, *except* for the IIR.

Second, none of the available *C* compilers for the Intel i860 would generate pipelined code using the floating point unit's pipeline. Thus we had to hand code the use of the pipeline. This took quite a bit of effort! In particular coding the IIR was very difficult due to its non-regular

[5]These times were provided by Steller Computer as examples of relative operation times.

nature. We feel that the difficulty of generating pipelined code will spill over into the compiler, complicating the generation of efficient code. Although we have not used ease of compiling as a comparison metric, it is an important factor to consider.

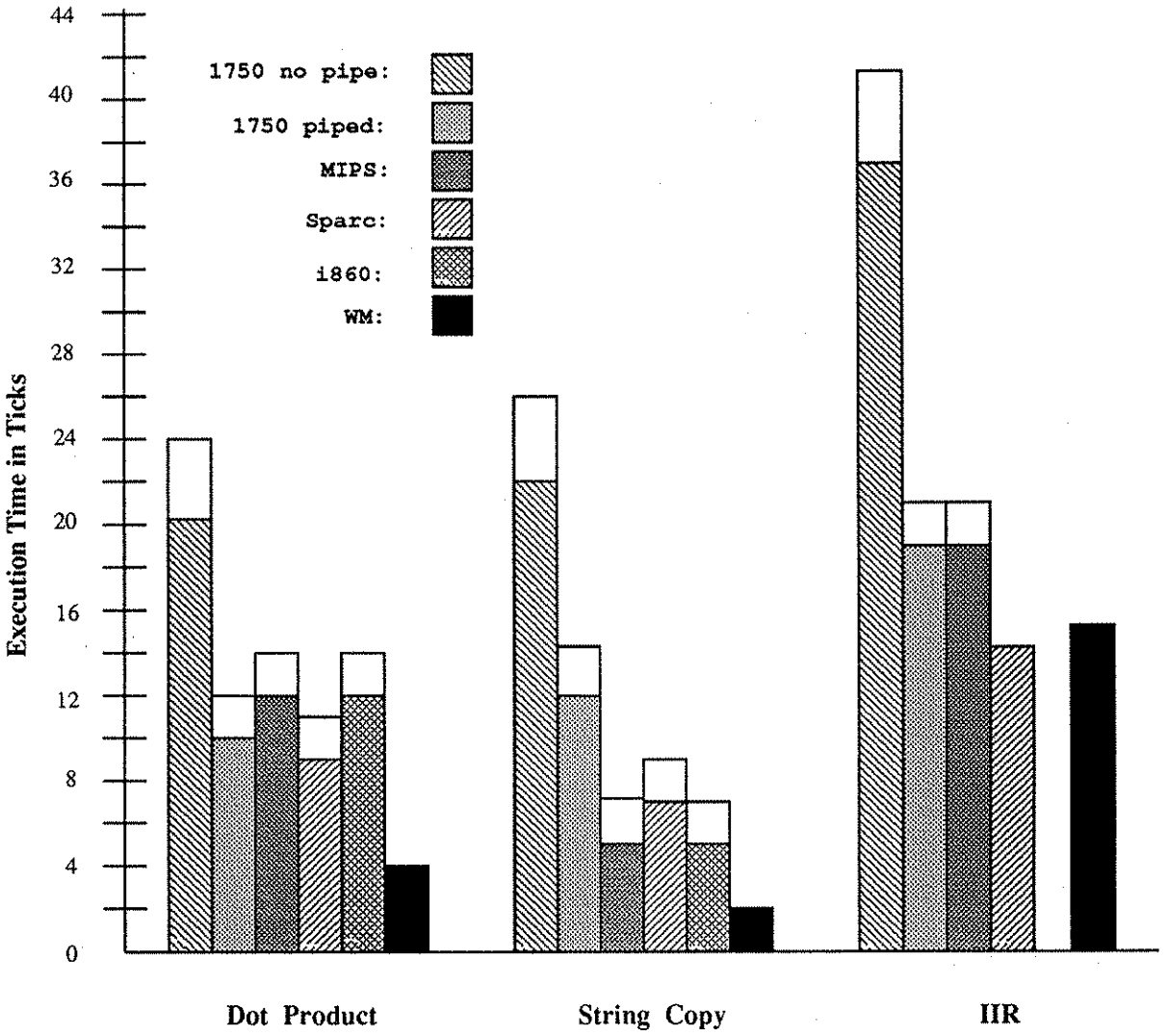

Figure 1. Weitek Times, memory latency of 2 and 4 ticks.

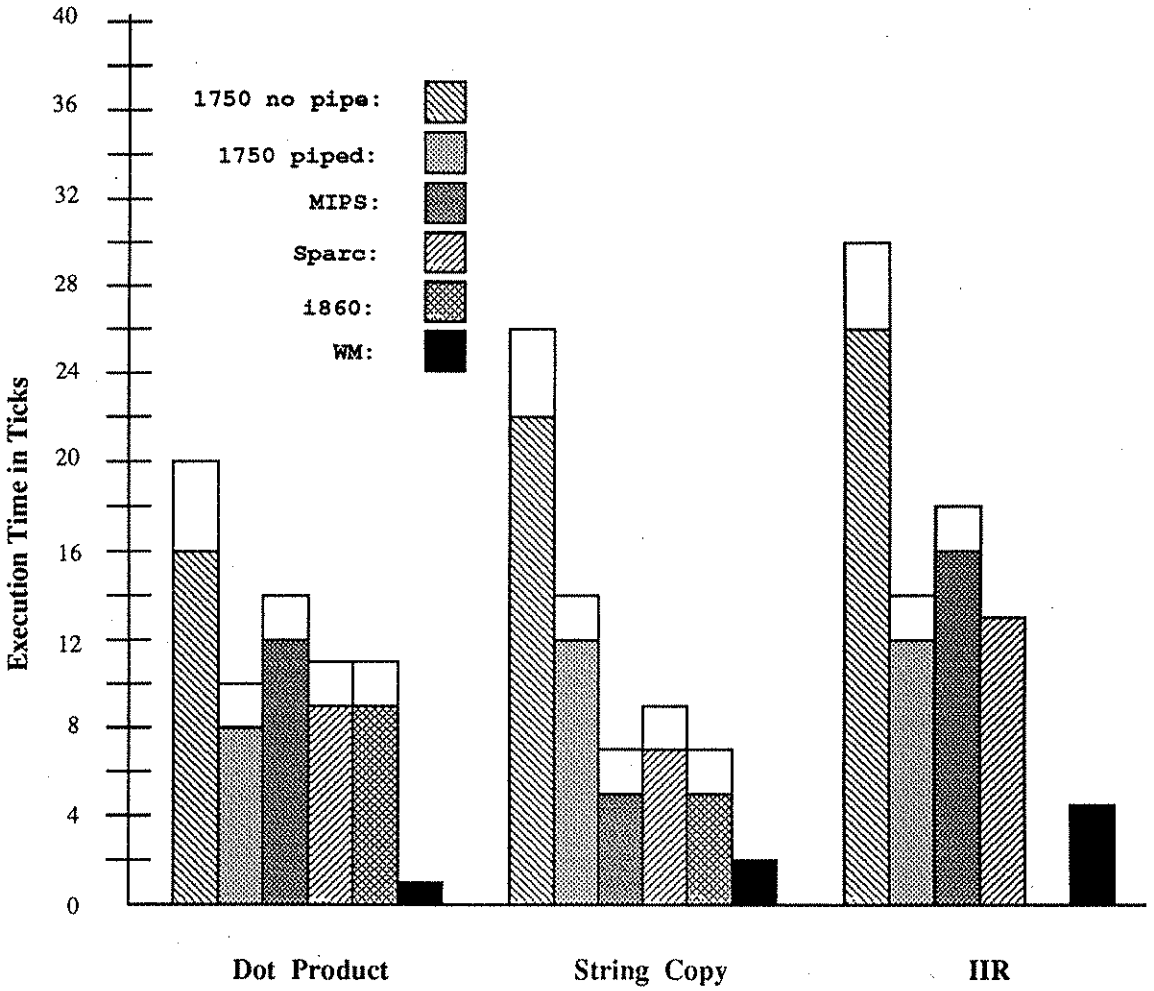

Figure 2. B.I.T. Times, memory latency of 2 and 4 ticks.

Third, note that the differences between the architectures are not as great as one might expect. All of the RISC machines are much the same, the exception being performance on the IIR when using the Weitek times. Fourth, when we pipeline the 1750A it performs much as a RISC machine. The primary difference is that it remains more sensitive to memory latency. Fifth, the WM outperforms the other RISC machines in the address calculation-intensive applications, dot product and string copy. When the inner loop is floating point intensive, and the Weitek times are used, the WM's advantage is eliminated. This is due largely to our decision not to scoreboard the WM.

5. Future Work

To further test our method of comparing architectural performance will require the use of more than three simple inner loop applications. Toward this end we have begun the development of VIRTICAL (VIRtual clock TIck CALculator), a tool to automate the process of generating loop and application equations. We will use the tool to compare architectures over a large implementation space, varying primitive operation times and memory characteristics.

A complete description of VIRTICAL is the subject of a later paper. Briefly, VIRTICAL works as in Figure 3. For each architecture to be examined a YACC grammer defining the assembly language will be written. The production actions of the YACC specification call VIRTICAL library routines that perform the bulk of the work. These library routines build flow graphs, keep track of basic blocks, and build dependence graphs. The arcs of the dependence graphs are labeled with resource dependencies, data dependencies, and architectural feature dependencies.

The C file generated by YACC is compiled and linked with the VIRTICAL libraries to generate an executable called an architecture module (AM). Thus there is a separate AM for each architecture. The AM takes two input files, a parameter definition file (PDF), and a source assembly file. The user must annotate the assembly code branch statements with probabilities to permit accurate calculation of the number of virtual clock cycles. The PDF contains resource descriptions, as well as values for parameters such as the memory latency and the floating point multiply time.

The AM parses the assembly file and generates the flow and dependence graphs. It then uses these graphs and the parameter information to generate instruction, loop, and program equations for the assembly code program. The program execution time in virtual clock cycles is gen-

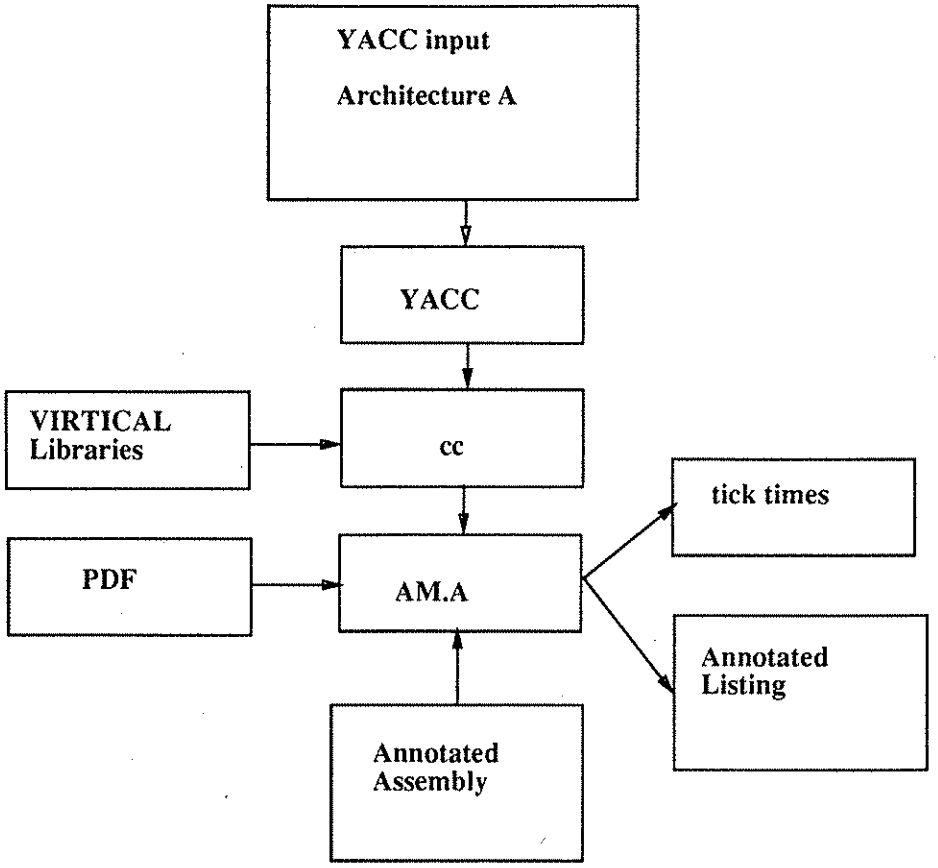

Figure 3. Structure of VIRTICAL.

erated, and, optionally, an annotated listing of the program is also generated. The annotations are the instruction and loop equations.

6. Summary

We set out to develop a method to compare different architectures based on their architecturally specified features, not on their implementation and compilers. We have developed a method based on virtual clock cycles, instruction equations, and best reasonable implementations. The instruction equations are a measure of how long (how many virtual clock cycles) it takes to execute a particular instruction. The instruction equations include several different implementation parameters, such as memory latency and primitive operation times. The instruc-

tion equations are also used to model implementation features such as pipelines and register scoreboarding. The instruction equations can be used to determine the run time, in virtual clock cycles, of applications.

To illustrate the method we compared five architectures, the MIL-STD-1750A, the Intel i860, the SPARC, the MIPS R-3000, and the WM. Three application inner loops were timed using the method: dot product, string copy, and infinite impulse response filter. We found that for the three applications the architectures examined fall into three basic classes, with little difference between members of a class. The features that distinguish these classes are the use of load/store as opposed to complex addressing modes, and the use of separate units to perform address calculation.

This study has opened up the possibility of studying architectural performance in an implementation independent manner. We have begun construction of a tool to automate the generation of instruction equations. We plan on applying our method, using VIRTICAL, to the recently announced IBM RISC System/6000 [IBM90], an architecture in which addresses are generated asynchronously. We predict that the RISC System/6000 will fall into the same class as the WM.

References

Bell71

Bell, C. G., and A. Newell, *Computer Structures: Readings and Examples*, McGraw-Hill, 1971.

Benitez88

Benitez, M. E., and J. W. Davidson, "A Portable Global Optimizer and Linker", *Proceedings of the SIGPLAN '88 Symposium on Programming Language Design and Implementation*, Atlanta, GA, June, 1988, 329-338.

Colwell85

Colwell, R. P., Hitchcock, C. Y., et. al., "Computers, Complexity, and Controversy", *IEEE Computer*, September, 1985, pp. 8-19.

DoD

MIL-STD 1750A Programmer's Manual.

Dongarra88

Dongarra J,"Performance of various Computers Using Standard Linear Equations Software", *Computer Architecture News*, ACM Press, March, 1990, pp. 17-31.

Dongarra87

Dongarra J., Martin J. L., and J. Worlton, "Computer Benchmarking: paths and pitfalls", *IEEE Spectrum*, July, 1987, pp.38-43.

Fuller87

Fuller, S. H., and W. E. Burr, "Measurement and Evaluation of Alternative Computer Architectures", *IEEE Computer*, October, 1977, pp. 24-35.

IBM90

Special Issue on the RISC System/6000, *IBM Journal of Research and Development*, volume 34, number 1, January, 1990.

Intel88

i860 Programmer's Reference Manual, Intel Corp., Santa Clara, CA., Feb. 1988.

Kane89

Kane, G.,*MIPS RISC Architecture*, Prentice Hall, Englewood Cliffs, NJ., 1989.

Piepho89

Piepho, R. S., and W. S. Wu, "A Comparison of RISC Architectures", *IEEE Micro*, August, 1989, pp. 51-62.

Pleszkun88

Pleszkun, A. R., and G. S. Sohi, "The Performance Potential of Multiple Functional Unit Processors", *Proceedings of the 15th Annual Symposium on Computer Architecture*, 1988.

Sun87

The SPARC Architecture Manual, SUN Microsystems, Inc., Mountain View, CA., 1987.

Whalley90

Whalley, David B., "Ease: An Environment for Architecture Study and Experimentation", PhD. Dissertation, Department of Computer Science, University of Virginia, May, 1990.

Wulf90a

Wulf, Wm. A., "The WM Computer Architectures Principles of Operation", Computer Science Technical Report No. TR-90-02, University of Virginia, January, 1990.

Wulf90b

Wulf, Wm. A., and Charles Hitchcock, "The WM Family of Computer Architectures", Computer Science Technical Report No. TR-90-05, University of Virginia, March, 1990.

7. Appendix A

MIL-STD 1750A

```
;; Dot product for the 1750

-- No Pipe --

LOOP: DLBX      R12,R8        1 + T(Iadd) + T(MemLat)
      FMBX      R13,R8        1 + T(Iadd) + T(MemLat) + T(Fmul)
      FAR       R2,R0         1 + T(Fadd)
      AISP      R8,2          1 + T(Iadd)
      SOJ       R5,LOOP       1 + T(Iadd) + 1


-- Instruction Pipe / Multiple ALU's --

LOOP: DLBX      R12,R8        1 + T(Iadd)
      FMBX      R13,R8        T(Iadd) + T(MemLat)
      AISP      R8,2          T(Iadd)
      SOJ       R5,LOOP       T(Iadd)
      FAR       R2,R0         1 + MAX[T(Fmul) - X, 0] + MAX[T(Fadd) - Y, 0]

;; where
;;   X = T(Iadd) + 1
;;   Y = 2 + 4*T(Iadd) + T(MemLat) + MAX[T(Fmul)-X,0]
```

Not Piped			Piped		
Memory Latency:	2	4	Memory Latency:	2	4
B.I.T.	16	20	B.I.T.	8	10
Weitek	20	24	Weitek	10	12

```
;; The IIR loop for the 1750

-- Not Piped --
      L              R3,N
      DL     R8,ONEHALF
      DL     R6,K1
      DL     R4,K2
      L          R12,A
      L              R13,B
      DLB    R12,2
      DLR    R14,R0
      DLB    R12,4
      DLR    R10,R0
      LISP   R2,6
LOOP:
      FMR    R14,R4              1 + T(Fmul)
      FMR    R0,R6         1 + T(Fmul)
      FAR    R0,R14              1 + T(Fadd)
      FABX   R13,R2              1 + T(Iadd) + T(MemLat) + T(Fadd)
      FMR    R0,R8         1 + T(Fmul)
      DSTX   R12,R6              1 + T(Iadd) + T(MemLat)
      DLR    R14,R10           1 + 1
      DLR    R10,R0              1 + 1
      AISP   R2,2          1 + T(Iadd)
      SOJ    R3,LOOP           1 + T(Iadd) + 1

-- Piped --
      L          R3,N
      DL         R8,ONEHALF
      DL         R6,K1
      DL         R4,K2
      L          R12,A
      L          R13,B
      DLB        R12,2
      DLR        R14,R0
      DLB        R12,4
      DLR        R10,R0
      LISP       R2,6
LOOP:
      FMR    R14,R4              1
      FMR    R0,R6         1
      FAR    R0,R14              T(Fmul)
      FABX   R13,R2              T(Iadd) + MAX[T(Fadd)-1, T(MemLat)]
      FMR    R0,R8         1 + T(Fadd) - 1
      DSTX   R12,R6              T(Iadd) + MAX[T(Fmul), T(Iadd)] - 1
      DLR    R14,R10           1
      AISP   R2,2          T(Iadd)
      SOJ    R3,LOOP           T(Iadd)
      DLR    R10,R0              1
```

Not Piped			Piped		
Memory Latency:	2	4	Memory Latency:	2	4
B.I.T.	26	30	B.I.T.	12	14
Weitek	37	41	Weitek	19	21

```
;; String copy for the 1750

;; R12 is address of S1
;; R13 is address of S2
;; R4 is index
;; R7 = 0xFF00, R8 = 0x00FF

-- No Pipe --

LOOP: LBX     R12,R4        1 + T(Iadd) + T(MemLat)        ;; Loads R2
      LR      R5,R2         1 + 1
      ANDR    R5,R7         1 + 1
      BEZ     UP0           1 + 1
      STBX    R13,R4        1 + T(Iadd) + T(MemLat)        ;; Stores R2
      ANDR    R2,R8         1 + 1
      BEZ     DONE          1 + 1
      AISP    R4,1          1 + T(Iadd)
      BR      LOOP          1 + 1

UP0:  XBR     R2
      SUBI    R2,S2,R4
DONE:


-- Instruction Pipe / Multiple ALU's --

LOOP: LBX     R12,R4        T(Iadd)
      LR      R5,R2         T(MemLat) + 1   ;; Data dependency on R2
      ANDR    R5,R7         1               ;; Data dependency on R5
      BEZ     UP0           1
      NOP                   1               ;; NOP to fill delay slot
      STBX    R13,R4        T(Iadd)         ;; T(MemLat) has no effect on loop
      ANDR    R2,R8         1
      BNZ     LOOP          1
      AISP    R4,1          T(Iadd)
      BR      DONE          1

UP0:  XBR     R2
      SUBI    R2,S2,R4
DONE:
```

Not Piped			Piped		
Memory Latency:	2	4	Memory Latency:	2	4
B.I.T.	22	26	B.I.T.	12	14
Weitek	22	26	Weitek	12	14

```
;;
```

```
;; Dot product for the MIPS
;;

  [dp:9, 0x4001b0]      mtc1    r0,f2
  [dp:9, 0x4001b4]      mtc1    r0,f3
  [dp:10, 0x4001b8]     blez    r4,0x4001f4
  [dp:10, 0x4001bc]     move    r2,r0
  [dp:10, 0x4001c0]     move    r3,r5
  [dp:10, 0x4001c4]     move    r7,r6
                                                  Int              Flt

  [dp:11, 0x4001c8]     lwc1    f4,0(r3)    1 + MAX[T(add)-4,0]    1
  [dp:11, 0x4001cc]     lwc1    f5,4(r3)          1                1
  [dp:11, 0x4001d0]     lwc1    f6,0(r7)          1                1
  [dp:11, 0x4001d4]     lwc1    f7,4(r7)          1                1
  [dp:11, 0x4001d8]     addiu   r2,r2,1           1                1
  [dp:11, 0x4001dc]     mul.d   f8,f4,f6          1      1 + MAX[T(MemLat)-1,0]
  [dp:11, 0x4001e0]     slt     r1,r2,r4    1 + MAX[T(add)-2,0]    1
  [dp:11, 0x4001e4]     addiu   r3,r3,8           1                1
  [dp:11, 0x4001e8]     addiu   r7,r7,8           1                1
  [dp:11, 0x4001ec]     bne     r1,r0,0x4001c8    1                1
  [dp:11, 0x4001f0]     add.d   f2,f8,f2          1      1 + MAX[T(Fmul)-5,0]

  [dp:12, 0x4001f4]     jr      r31
  [dp:12, 0x4001f8]     mov.d   f0,f2
  [dp:5, 0x4001fc]      nop

Memory
Latency:     2        4
-----------------------
B.I.T.      12       14
Weitek      12       14
```

```
;;
;; IIR loop for the MIPS
;;

  [iir:5, 0x400200]     lw      r3,24(sp)
  [iir:5, 0x400204]     mtc1    r6,f14
  [iir:5, 0x400208]     mtc1    r7,f15
  [iir:12, 0x40020c]    slti    r1,r4,3
  [iir:9, 0x400210]     lwc1    f0,0(r3)
  [iir:9, 0x400214]     lwc1    f1,4(r3)
  [iir:10, 0x400218]    lwc1    f2,8(r3)
  [iir:10, 0x40021c]    lwc1    f3,12(r3)
  [iir:12, 0x400220]    bne     r1,r0,0x400284
  [iir:12, 0x400224]    li      r2,2
  [iir:12, 0x400228]    lw      r5,28(sp)
  [iir:12, 0x40022c]    lwc1    f16,16(sp)
  [iir:12, 0x400230]    lwc1    f17,20(sp)
  [iir:12, 0x400234]    lwc1    f18,-32752(gp)
  [iir:12, 0x400238]    lwc1    f19,-32748(gp)
  [iir:12, 0x40023c]    addiu   r6,r3,16
  [iir:12, 0x400240]    addiu   r5,r5,16
                                                  Int                 Flt

  [iir:15, 0x400244]    mul.d   f6,f2,f14          1                   1
  [iir:13, 0x400248]    mov.d   f12,f0             1                   1
  [iir:15, 0x40024c]    lwc1    f4,0(r5)           1                   1
  [iir:15, 0x400250]    lwc1    f5,4(r5)           1                   1
  [iir:16, 0x400254]    addiu   r2,r2,1            1                   1
  [iir:15, 0x400258]    mul.d   f10,f12,f16        1                   1
  [iir:15, 0x40025c]    add.d   f8,f4,f6           1         1 + MAX[T(MemLat)-2,0]
  [iir:16, 0x400260]    slt     r1,r2,r4     1 + MAX[T(add)-3,0]       1
  [iir:14, 0x400264]    mov.d   f0,f2              1                   1
  [iir:16, 0x400268]    addiu   r5,r5,8            1                   1
  [iir:15, 0x40026c]    add.d   f4,f8,f10          1         1 + MAX[T(Fadd)-4,T(Fmul)-5,0]
  [iir:16, 0x400270]    addiu   r6,r6,8            1                   1
  [iir:15, 0x400274]    mul.d   f2,f4,f18          1         1 + MAX[T(Fadd)-2,0]
  [iir:15, 0x400278]    swc1    f2,-8(r6)       1 + MAX[T(Fmul)-1,0]
  [iir:16, 0x40027c]    bne     r1,r0,0x400244     1                   1
  [iir:15, 0x400280]    swc1    f3,-4(r6)          1                   1

  [iir:17, 0x400284]    jr      r31
  [iir:17, 0x400288]    nop
  [iir:1, 0x40028c]     nop
```

Memory Latency:	2	4
B.I.T.	16	18
Weitek	19	21

```
;;
;; String copy for the MIPS
;;

   [strcpy:7, 0x4002a0]  lb      r2,0(r5)
   [strcpy:6, 0x4002a4]  move    r3,r4
   [strcpy:7, 0x4002a8]  addiu   r4,r4,1
   [strcpy:7, 0x4002ac]  addiu   r5,r5,1
   [strcpy:7, 0x4002b0]  beq     r2,r0,0x4002cc
   [strcpy:7, 0x4002b4]  sb      r2,-1(r4)                    Int

   [strcpy:7, 0x4002b8]  lb      r2,0(r5)                      1
   [strcpy:7, 0x4002bc]  addiu   r4,r4,1                       1
   [strcpy:7, 0x4002c0]  addiu   r5,r5,1                       1
   [strcpy:7, 0x4002c4]  bne     r2,r0,0x4002b8    1 + MAX[T(MemLat)-2,0]
   [strcpy:7, 0x4002c8]  sb      r2,-1(r4)         1 + MAX[T(add)-3,0]

   [strcpy:8, 0x4002cc]  jr      r31
   [strcpy:8, 0x4002d0]  move    r2,r3
   [strcpy:14, 0x4002d4]         nop
   [strcpy, 0x4002d8]    nop
   [strcpy, 0x4002dc]    nop
```

Memory Latency:	2	4
B.I.T.	5	7
Weitek	5	7

SPARC

```
;; Dot product for the SPARC

1            .seg  "text"
2            .proc 7
3            .global     _dp
      _dp:
4            sethi %hi(L2000000),%o3
5            ldf   [%o3+%lo(L2000000)],%f30
6            mov   0,%o5
7            cmp   %o5,%o0
8            bge,a LY2
```

			Int	Float
LY1:				
9	ldf	[%o1],%f2	1	1
10	ldf	[%o2],%f4	1	1
11	inc	%o5	1	1
12	cmp	%o5,%o0	1	1
13	fmuls	%f2,%f4,%f6	1	1 + MAX[T(MemLat)-2,T(Fadd)-5,0]
14	inc	4,%o1	1	1
15	inc	4,%o2	1	1
16	bl	LY1	1	1
17	fadds	%f30,%f6,%f30	1	1 + MAX[T(Fmul) - 4, 0]

```
LY2:
18           retl
19           fmovs %f30,%f0
20           .seg  "data"
21           .align      8
      L2000000:
22           .word 0
23           .word 0
```

Memory Latency:	2	4
B.I.T.	9	11
Weitek	9	11

```
;; IIR for the SPARC

1           .seg  "text"
2           .proc 16
3           .global     _iir
_iir:
4           save  %sp,-112,%sp
5           st    %i4,[%fp+84]
6           st    %i2,[%fp+76]
7           st    %i3,[%fp+80]
8           ldf   [%fp+80],%f20
9           st    %i1,[%fp+72]
10          ldf   [%i5+8],%f12
11          ldf   [%i5],%f28
12          ldf   [%fp+72],%f22
13          sethi %hi(L2000000),%o0
14          ldd   [%o0+%lo(L2000000)],%f0
15          fmovs %f1,%f25
16          fmovs %f0,%f24
17          mov   12,%i2
18          add   %i2,%i5,%i3
19          ld    [%fp+92],%i5
20          mov   2,%i4
21          cmp   %i4,%i0
22          bge   L77005
23          add   %i2,%i5,%i5
                                                  Int              Float
LY1:
24          ldf   [%i5],%f6             1                 1
25          fmuls %f12,%f22,%f4         1                 1
26          fmuls %f26,%f20,%f2         1                 1
27          inc   %i4                   1                 1
28          cmp   %i4,%i0               1                 1
29          fadds %f6,%f4,%f8           1   1 + MAX[T(MemLat)-4,T(Fmul)-4,0]
30          inc   4,%i5                 1                 1
31          fadds %f8,%f2,%f10          1       1 + MAX[T(Fadd) - 2, 0]
32          fmovs %f12,%f26             1                 1
33          fmuls %f10,%f24,%f12        1       1 + MAX[T(Fadd) - 2, 0]
34          inc   4,%i3                 1                 1
35          bl    LY1                   1                 1
36          stf   %f12,[%i3]            1       1 + MAX[T(Fmul) - 3, 0]
L77005:
37          ret
38          restore
39          .seg  "data"
40          .align      8
L2000000:
41          .word 0x3fe00000
42          .word 0
```

Memory Latency:	2	4
B.I.T.	13	13
Weitek	14	14

```
;; String copy for the SPARC

1           .seg  "text"
2           .proc 66
3           .global     _strcpy
      _strcpy:
4           mov   %o0,%o4
5           ldsb  [%o1],%o5
6           dec   %o0

                                                  Int              Float
      LY1:
7           inc   %o0                          1                  1
6           stb   %o5,[%o0]      1 + MAX[T(MemLat)-1, 0]          1
7           tst   %o5                          1                  1
8           inc   %o1                          1                  1
9           bne   LY1                          1                  1
10          ldsb  [%o1],%o5                    1                  1


11          retl
12          add   %g0,%o4,%o0
13          .seg  "data"
```

Memory Latency:	2	4
B.I.T.	7	9
Weitek	7	9

```
// Dot product for the i860.

      .file "dotprod.c"
// ccom  -X22 -X74 -X80 -X83 -X247 -X254 -X266 -X278 -X325 -X350 -X383 -X422
//      -X424 -X501 -X523 -X524 -X525

      .text
      .align      4
_dotprod:
//        .bf
      fmov.dd     f0,f16
      mov   r0,r19
      fld.d       8(r18)++,f26
      fld.d       8(r17)++,f24
      adds        1,r19,r19
      pfmul.dd    f26,f24,f0

      fld.d       8(r18)++,f26
      fld.d       8(r17)++,f24
      adds        1,r19,r19
      pfmul.dd    f26,f24,f0

      fld.d       8(r18)++,f26
      fld.d       8(r17)++,f24
      adds        1,r19,r19
      pfmul.dd    f26,f24,f20
      pfadd.dd    f20,f16,f0

      fld.d       8(r18)++,f26
      fld.d       8(r17)++,f24
      adds        1,r19,r19
      pfmul.dd    f26,f24,f20
      pfadd.dd    f20,f16,f0

      fld.d       8(r18)++,f26
      fld.d       8(r17)++,f24
      adds        1,r19,r19
      pfmul.dd    f26,f24,f20
      pfadd.dd    f20,f16,f0

      fld.d       8(r18)++,f26
      fld.d       8(r17)++,f24
      adds        1,r19,r19
      pfmul.dd    f26,f24,f20
      pfadd.dd    f20,f0,f16
      br    .L7
      nop
      fld.d       8(r18)++,f26      1
      fld.d       8(r17)++,f24    1
      d.fmov.dd   f16,f22           1
      pfmul.dd    f26,f24,f20     1 + MAX[T(MemLat) - 1, 0]
      adds        1,r19,r19   1
      pfadd.dd    f20,f22,f16       1 + MAX[T(Fmul) - 1, 0]
      subs  r19,r16,r0        1
      bc    .L6               T(branch)
```

```
//          .ef
        pfmul.dd     f0,f0,f20
        fmov.dd              f16,f22
        pfadd.dd     f20,f22,f16
        fmov.dd              f16,f22
        pfmul.dd     f0,f0,f20
        pfadd.dd     f20,f22,f16
        fmov.dd              f16,f22
        pfadd.dd     f0,f0,f16
        pfadd.dd     f0,f0,f18
        pfadd.dd     f0,f0,f20
        pfadd.dd     f0,f0,f24
        fadd.dd              f22,f20,f22
        fadd.dd              f22,f24,f22
        fadd.dd              f18,f22,f22
        fadd.dd              f22,f16,f16
        bri    r1
         nop
        .align       4
        .data
//_i   r19    local
//_sum       f16    local

//_vecsize   r16    local
//_a   r17    local
//_b   r18    local

        .text
        .data
        .globl       _dotprod

        .text
```

Memory Latency:	2	4
B.I.T.	9	11
Weitek	12	14

```
// IIR loop for the i860.

        .file "iir.c"
// ccom  -X22 -X74 -X80 -X83 -X247 -X254 -X266 -X278 -X325 -X350 -X383 -X422
//      -X424 -X501 -X523 -X524 -X525

        .text
        .align      4
_iir:
        shl    2,r16,r18
        fmov.dd      f18,f16
        mov    r19,r17
//          .bf
        br     .L7
        or     12,r0,r16
        adds   -4,r16,r27  1
        adds   r17,r27,r27 1
        fld.l  0(r27),f28  1
        adds   r16,r17,r28        1
        fmov.sd      f28,f22             1 + Max[T(MemLat) - 1, 0]
        fmul.dd      f16,f22,f22 T(fmul)
        adds   r16,r20,r27 1
        fld.l  0(r27),f28  1
        fmov.sd      f28,f18             1 + T(MemLat)
        fadd.dd      f22,f18,f18 T(fadd)
        adds   -8,r16,r27  1
        adds   r17,r27,r27 1
        fld.l  0(r27),f28  1
        fmov.sd      f28,f24             1 + T(MemLat)
        fmul.dd      f20,f24,f24 T(fmul)
        fadd.dd      f18,f24,f24 T(fadd)
        fmov.ds      f24,f27             1
        orh    ha%.L16,r0,r30     1
        fld.l  l%.L16(r30),f26    1
        orh    16384,r0,r27       1
        ixfr   r27,f25            1
        frcp.ss      f26,f24             T(frcp) + Max[T(MemLat) - 2, 0] (floating divide)
        fmul.ss      f26,f24,f18 T(fmul)
        fsub.ss      f25,f18,f18 T(fsub)
        fmul.ss      f24,f18,f24 T(fmul)
        fmul.ss      f26,f24,f18 T(fmul)
        d.fsub.ss    f25,f18,f18 T(fsub)
        d.fmul.ss    f27,f24,f25 T(fmul)
        d.fmul.ss    f18,f25,f27 T(fmul)
        fst.l  f27,0(r28)   0 (overlapped with fsub)
        adds   4,r16,r16    0 (overlapped with fmul)
        subs   r18,r16,r0   0 (overlapped with fmul)
        bnc    .L6          T(branch)
//          .ef
        bri    r1
         nop
        .align      4
        .data
//_i   r16    local

//_n   r18    local
//_k1  f16    local
```

```
//_k2 f20   local
//_a  r17   local
//_b  r20   local

      .text
      .data
      .globl      _iir

      .text
```

Memory Latency:	2	4
B.I.T.	36	44

```
Weitek      71    79// STRING COPY for the i860
// r17 - address of source string
// r16 - address of destination string

      ld.b  0(r17),r26
      bte   0,r26,done
      adds  1,r17,r17
      ld.b  0(r17),r27
      subs  r17,r16,r18
loop:
      st.b  r26,0(r16)          1
      adds  1,r16,r16           1
      or    r0,r27,r26          1 + Max[T(MemLat) - 2, 0]
      bnc.t loop                T(branch)
      ld.b  r18(r16),r27              1

done:
      bri   r1
      st.b  r26,0(r16)
```

Memory Latency:	2	4
B.I.T.	5	7
Weitek	5	7

WM

```
;;
;; Dot product for the WM
;;

_dp:
      r31 := (r20 <= 0)
CVTID f22 := 0
      JumpIT      L1
      SinD  f0,r21,r20,8
      SinD  f1,r22,r20,8
      Jump  L1

                                  Int          Float
L2:
double      f22 := (f0 * f1) + f22          0     MAX[T(Fmul),T(Fadd)]
L1:   JNIf0 L2                   0          0

double      f20 := f22
      JumpI r4


Memory
Latency:   2       4
--------------------
B.I.T.          1   1
Weitek          4   4
```

```
-- INFINITE IMPULSE RESPONSE FILTER     Nb. recurrences
--     FOR i in 3 .. N LOOP
--          a(i) := (b(i) + a(i-1)* k1 + a(i-2)* k2) /2;
--     END LOOP;
--                    or
--   for(i=3; i<=N; i++) a[i] = (b[i] + a[i-1]*k1 + a[i-2]*k2) / 2;

LLH   r20 := N
LUH   r20 := N
LW    r20                       -- r5 == N    loop control
      r5  := r0
LLH   r20 :=   k1
LUH   r20 :=   k1
LD    r20                           -- r6 ==  k1
double        f6 := f0
LLH   r20 :=   k2
LUH   r20 :=   k2
LD    r20
LLH   r12 := A
LUH   r12 := A
LLH   r13 := B
LUH   r13 := B
double        f4 := f0               -- r4 ==  k2
      r12 := (r12 + 12) -- address of A[3]
      r13 := (r13 + 12) -- address of B[3]
                             -- load a(2)
LD    r9 := (r12 - 4)        -- load a(1)
LD    r9 := (r12 - 8)
LLH   r20 := half
LUH   r20 := half
LD    r31 := r20
      SinD  r0, r13, r5, 8     -- Stream in FIFO 0
      SoutD r0, r12, r5, 8     -- Stream out FIFO 0
double       f10 := f0         -- a(i-1) := a(2)
double       f11 := f0         -- a(i-2) := a(1)
double       f20 := f0                       Float
                        --
 lp:                                               T(Fmul)
double       f9 := (f10 * f6) + f0    --     MAX[ T(Fmul), T(Fadd) ]
double       f9 := (f11 * f4) + f9    --     T(Fadd)
double       f11 := f10           --         T(Fmul)
double       f10 := (f9 * f20) --              1
double       f0  := f10           --       0
      jnif0 lp           --
                         -- loop if not done

    .section data        -- Loop control variable
N:   .word 5
k1:  .double 1
k2:  .double 1
A:   .double 2,3,4,5,6,7,8,9,10       -- A vector
B:   .double 2,3,4,5,6,7,8,9,10       -- B vector
half: .double      0.5
```

Memory Latency:	2	4
B.I.T.	5	5
Weitek	15	15

```
;;
;; String copy for the WM
;;
;;
;; Hand generated code.


;; Unix string copy
;;
;; strcpy(s1,s2)
;;   char *s1, *s2;
;;   { char *s = s1;
;;     while (*s1++ = *s2++);
;;     return (s);
;;   }

      r11 := (31 ^ 20)              ;; get large count
      LLH r5 := s1                  ;; base address of s1
      LUH r5 := s1                  ;; base address of s1
      LLH r6 := s2                  ;; base address of s2
      LUH r6 := s2                  ;; base address of s2
      SinB  r0, r5, r11, 1
      SoutB r0, r6, r11, 1

                     Int
loop: r0 := (r0 <> 0)       1          ;; copy and test for null terminator
      JumpIT loop           1          ;; loop if

      r0 := 0                       ;; write the terminator
      StopAll
      SYNCH

      .section data
s1:   .byte 1,2,3,4,5,0             ;; each byte specification will be
                                ;;    word aligned; careful
s2:   .word 0,0,0


Memory
Latency:   2       4
--------------------
B.I.T.         2   2
Weitek         2   2
```