Evaluation of the WM Architecture

Wm. A. Wulf
Computer Science Department
University of Virginia

Abstract

This report describes the results of studies of the WM architecture—its performance, the values of some of its key architectural parameters, the difficulty of compiling for it, and hardware implementation complexity. The studies confirm that, with comparable chip area and without heroic compiler technology, WM is capable of outperforming traditional scalar architectures by factors of 2-9. They also underscore the need to devise higher bandwidth memory systems.

Description of the WM Architecture

A more complete description of WM may be found in [Wul88, Wul90, Wul90b]; here we provide only enough to understand its evaluation.

The quest for greater instruction-level concurrency can be divided into two broad categories: those in which the concurrency is starkly explicit in the programmer's model, and those in which it is not. VLIW machines are an example from the first category [Fis84], the second category includes the superscalar, superpipelined, and some decoupled access-execute architectures [cf, Jou89]. While potentially capable of less concurrency, the second category requires simpler compiler technology and thus it is interesting to investigate the degree of concurrency they can achieve.

WM falls more into this second category; the semantics of its instruction set are strictly sequential, but have been designed to allow concurrency in their implementation. Specifically, in WM eliminates a number of data and control dependencies common in conventional architectures — dependencies that appear purely as artifacts of the architecture. Elimination of these dependencies materially enhances the achievable concurrency.

The resulting design resembles a collection of cooperating software processes synchronized by first-in-first-out queues. Each of some 12 components operates independently and asynchronously — thus it faintly resembles data flow machines [Nik89] in that operations are performed as data becomes available. Figure 1 illustrates this model of the architecture.

The "instr fetch" unit is responsible for fetching instructions and enqueuing them for execution by the integer, floating-point, or control execution units; it proceeds as rapidly as possible, and blocks only when the various instruction queues are full or an unresolved conditional branch is encountered.

The integer and floating-point execution units (IEU, FEU) execute instructions in order from their respective instruction FIFOs. They proceed independently and block only when there is no instruction to execute or they need data that is not yet in their data FIFOs.

The "data fetch" and "data store" units, called "stream control units", asynchronously enqueue/dequeue a "stream" of data — a sequence of data values where the number and "stride" of the items are known in advance. They block only when an input FIFO is full, an output FIFO is empty, or no memory request is outstanding.

Figure 1: A Process Model of WM
The integer and floating-point execution units execute RISC-like, register-to-register operations of the form

\[ r_d := (r_1 \oplus r_2) \odot r_3 \]

That is, three source operands and two operators are specified in each instruction. This leads to an explicitly pipelined model of these execution units as shown in Figure 2, in which the inner operation (\(q_1\)) of one instruction is performed concurrently with the outer operation (\(q_2\)) of its predecessor. Thus, in principle, four arithmetic operations can be executing concurrently — two in the integer unit and two in the floating-point unit.

Load and store instructions are executed by the integer unit; they specify an address, a data size, and an execution unit. A Load causes the data at the specified address to be enqueued on the "input FIFO" of the specified execution unit. A Store causes data to be dequeued from the output FIFO of the specified execution unit and stored at the specified address. Both Load and Store specify the address as a two operator, three operand expression syntactically and semantically identical to an integer instruction.

The operands of an integer or floating instruction may be specified in each instruction. This leads to an explicitly pipelined model of these execution units as shown in Figure 2. In which the inner operation (\(q_1\)) of one instruction is performed concurrently with the outer operation (\(q_2\)) of its predecessor. Thus, in principle, four arithmetic operations can be executing concurrently — two in the integer unit and two in the floating-point unit.

Load and store instructions are executed by the integer unit; they specify an address, a data size, and an execution unit. A Load causes the data at the specified address to be enqueued on the "input FIFO" of the specified execution unit. A Store causes data to be dequeued from the output FIFO of the specified execution unit and stored at the specified address. Both Load and Store specify the address as a two operator, three operand expression syntactically and semantically identical to an integer instruction.

Relationals are explicit operators in both arithmetic units; they produce two results — a condition code and their left operand. Thus, for example,

\[ r_5 := (r_5 < 10) + 1 \]

Note that an implementation may use more (or less) pipelining; what we describe here is the degree of pipelining evident to the programmer in the architectural definition.

One version of the inner loop code for the dot product of two vectors is shown in Figure 3. In this code "r0-f31" denote registers in the integer unit and "i0-f31" denote registers in the floating-point unit; the assembler uses the register names to distinguish between integer and floating-point instructions.

Walking through this loop,

- the first two instructions are "load 32-bit floating-point value"; they request that \(a[i]\) and \(b[i]\) be enqueued in the floating-point input FIFO (\(r4\) holds \(i\), \(r6\) and \(r7\) are the base addresses of \(a\) and \(b\) respectively; "<<" is a left shift used to scale \(i\)),

- the third instruction increments and tests \(i\) (the condition code is enqueued for the control unit),

- the fourth instruction forms the partial sum (note the references to \(r0\) access and dequeue the values of \(a[i]\) and \(b[i]\)), and finally

- the last instruction tests whether the condition code generated above is true, and if so loops back.

This sequential explanation is correct; but the execution of the code involves a good deal of concurrency. For example, the first three instructions will be enqueued for execution by the IEU, the fourth for the FEU and the last for the CEU. Since the data will likely not be available the first time that the floating-point operation is enqueued, that instruction will block but the IEU and CEU will continue. A second or third floating operation may be enqueued before the data value arrives in the FEU data FIFO. [Smi84] and [Soh84] have called this phenomenon slip. In this case slip may allow the integer unit to get several loop iterations ahead of the floating-point unit.

---

1 Note that an implementation may use more (or less) pipelining; what we describe here is the degree of pipelining evident to the programmer in the architectural definition.
A better coding of the dot product example would use WM's streaming capability. A single instruction, stream-in, informs a stream control unit that a sequence of operands is to be read from memory — starting at a specified location, with a specified stride and count. The count of the number of operands not yet committed to be processed is maintained, and can be tested by a special conditional jump, JNI (Jump of stream count Not zero in the Integer unit). Thus, dot product becomes:

\[
\text{Sin32F } f_0, r_6, r_5, 4 \\
\text{Sin32F } f_1, r_7, r_5, 4 \\
\text{Loop: } f_4 := (f_0*f_1)+f_4 \\
\text{JNI } f_0, \text{ Loop}
\]

The JNI can be overlapped with the execution of the computation, so this version of the loop issues two floating-point operation per floating-point multiply time. Equivalently, each iteration completes in one floating multiply time.

**Relation of WM to Other Architectures**

In either of the above examples, the exact degree of concurrency, and hence performance, will depend on several factors — including the relative performance of the floating and integer units, the nature of the memory and cache, etc. This is quantified later. However, the fact that the semantics are those of a sequential machine is very important. It is this essential property that avoids the need for heroic compilation techniques; specifically the detection of streaming is a relatively simple extension of familiar scalar optimizations [Dav90]. This too is discussed later.

In summary, the performance of WM arises from

- the ability to issue 2 arithmetic operations per instruction,
- the asynchronous and concurrent issue and execution of integer and floating-point instructions,
- masking of memory latency by the data FIFOs,
- the asynchronous and overlapped handling of control (jumps), and
- the asynchronous generation of load/store operations by the streaming units.

Architecture is, we believe, about the composition of components — deciding what to include, what to exclude, how to integrate the chosen components, what interactions can and cannot be exploited by compilers and applications, etc. We hope to have stood on the shoulders of our predecessors in selecting and integrating components. We specifically note

- the notion of a "decoupled access/execute architecture" (DAE), separating hardware that generates memory requests from that which operates on the requested values, was discovered independently by several researchers in the early 80's [Ple83, Smi84, Goo85, Smi87]. The technique is used in the ZS-1 [Smi87], and in restricted form in the Intel i860 [Int89].

- [Smi84] introduces the use of FIFOs to interface the processor to memory and to synchronize condition codes between the decoupled units. A very restricted form of the idea was used in the Stretch in the early 60's [Buc62].

- the issue of multiple scalar operations in a single instruction is an old trick in digital signal processors, is the essence of "chaining" in vector processors, is behind VLIW machines(Fis84), and in a restricted form is in the "multiply-accumulate" instructions of recent scalar machines [IBM90, Int89]. Architectures that use this approach have been called "superscalar"[Jou89].

The contribution of WM, we believe, comes from the synergistic interplay of these features, tempered by a healthy dose of our experience with optimizing compilers. The way in which the decoupling was done, the role played by FIFOs, and the form of the multiple operations per instruction all strongly affected each other. The result is descended from, but not really like any of the other machines mentioned above. For example,

- the decoupled architectures of both Smith and Davidson [Smi84, Ple83] contain two processors — one to manage data access and the other to manage execution of data manipulation. The two processors are controlled by distinct programs. WM has two data manipulation processors, one of which happens to be able to issue memory access commands. It has eight additional simple processors, the stream control units, that manage data access of a special form (vectors). Our model actually corresponds most closely to a the-
Some Preliminary Results

It is difficult to compare an unimplemented architecture to existing implementations of extant architectures; it is even more difficult to compare it to other unimplemented ones. Thus, although we have attempted some implementation-independent comparisons [Gri90, Wu190], our main focus was on

- how well the unusual features of the WM architecture are utilized by real programs,
- the value of various architectural parameters, such as FIFO size, needed to achieve good performance,
- testing the hypothesis that the single, semantically sequential program does not lose performance compared to the separate programs of Smith-style DAEs,
- consideration of various memory system designs to support the memory bandwidth required,
- exploration of the hardware design space to ascertain whether there are any "gotchas",
- exploration of the compilation issues posed by WM, and verification of the hypothesis that scalar optimization techniques are adequate,

Some of these are discussed in more detail below.

An Implementation-Independent Comparison

[Gri90] undertook an implementation-independent study and compared five architectures on three small benchmarks. The method involved uncovering the inherent data and control dependencies in the benchmarks and using these with two real arithmetic unit times, to derive a set of equations for the virtual clock cycles for each benchmark. In effect, the methodology assumed that each implementation "threw hardware at the problem", and so was constrained only by the inherent dependencies in the program. While unrealistic from a cost/performance perspective, the methodology does fairly represent "the best that can be done" by each of the architectures.

As is indicated by the preliminary performance data described in Section 3, it appears as though WM's synergy has had a significant impact on performance. Whereas studies such as [Smi84] and [Jou89] predict concurrency of a factor of two or a bit better, we regularly see factors of four or more. On some highly structured tight loops, such as SAXPY, dot product or string copy, we see factors of six to nine.

Some Preliminary Results

As is indicated by the preliminary performance data described in Section 3, it appears as though WM's synergy has had a significant impact on performance. Whereas studies such as [Smi84] and [Jou89] predict concurrency of a factor of two or a bit better, we regularly see factors of four or more. On some highly structured tight loops, such as SAXPY, dot product or string copy, we see factors of six to nine.

An Implementation-Independent Comparison

[Gri90] undertook an implementation-independent study and compared five architectures on three small benchmarks. The method involved uncovering the inherent data and control dependencies in the benchmarks and using these with two real arithmetic unit times, to derive a set of equations for the virtual clock cycles for each benchmark. In effect, the methodology assumed that each implementation "threw hardware at the problem", and so was constrained only by the inherent dependencies in the program. While unrealistic from a cost/performance perspective, the methodology does fairly represent "the best that can be done" by each of the architectures.

As is indicated by the preliminary performance data described in Section 3, it appears as though WM's synergy has had a significant impact on performance. Whereas studies such as [Smi84] and [Jou89] predict concurrency of a factor of two or a bit better, we regularly see factors of four or more. On some highly structured tight loops, such as SAXPY, dot product or string copy, we see factors of six to nine.

An Implementation-Independent Comparison

[Gri90] undertook an implementation-independent study and compared five architectures on three small benchmarks. The method involved uncovering the inherent data and control dependencies in the benchmarks and using these with two real arithmetic unit times, to derive a set of equations for the virtual clock cycles for each benchmark. In effect, the methodology assumed that each implementation "threw hardware at the problem", and so was constrained only by the inherent dependencies in the program. While unrealistic from a cost/performance perspective, the methodology does fairly represent "the best that can be done" by each of the architectures.

As is indicated by the preliminary performance data described in Section 3, it appears as though WM's synergy has had a significant impact on performance. Whereas studies such as [Smi84] and [Jou89] predict concurrency of a factor of two or a bit better, we regularly see factors of four or more. On some highly structured tight loops, such as SAXPY, dot product or string copy, we see factors of six to nine.

An Implementation-Independent Comparison

[Gri90] undertook an implementation-independent study and compared five architectures on three small benchmarks. The method involved uncovering the inherent data and control dependencies in the benchmarks and using these with two real arithmetic unit times, to derive a set of equations for the virtual clock cycles for each benchmark. In effect, the methodology assumed that each implementation "threw hardware at the problem", and so was constrained only by the inherent dependencies in the program. While unrealistic from a cost/performance perspective, the methodology does fairly represent "the best that can be done" by each of the architectures.

As is indicated by the preliminary performance data described in Section 3, it appears as though WM's synergy has had a significant impact on performance. Whereas studies such as [Smi84] and [Jou89] predict concurrency of a factor of two or a bit better, we regularly see factors of four or more. On some highly structured tight loops, such as SAXPY, dot product or string copy, we see factors of six to nine.

An Implementation-Independent Comparison

[Gri90] undertook an implementation-independent study and compared five architectures on three small benchmarks. The method involved uncovering the inherent data and control dependencies in the benchmarks and using these with two real arithmetic unit times, to derive a set of equations for the virtual clock cycles for each benchmark. In effect, the methodology assumed that each implementation "threw hardware at the problem", and so was constrained only by the inherent dependencies in the program. While unrealistic from a cost/performance perspective, the methodology does fairly represent "the best that can be done" by each of the architectures.

As is indicated by the preliminary performance data described in Section 3, it appears as though WM's synergy has had a significant impact on performance. Whereas studies such as [Smi84] and [Jou89] predict concurrency of a factor of two or a bit better, we regularly see factors of four or more. On some highly structured tight loops, such as SAXPY, dot product or string copy, we see factors of six to nine.

An Implementation-Independent Comparison

[Gri90] undertook an implementation-independent study and compared five architectures on three small benchmarks. The method involved uncovering the inherent data and control dependencies in the benchmarks and using these with two real arithmetic unit times, to derive a set of equations for the virtual clock cycles for each benchmark. In effect, the methodology assumed that each implementation "threw hardware at the problem", and so was constrained only by the inherent dependencies in the program. While unrealistic from a cost/performance perspective, the methodology does fairly represent "the best that can be done" by each of the architectures.

As is indicated by the preliminary performance data described in Section 3, it appears as though WM's synergy has had a significant impact on performance. Whereas studies such as [Smi84] and [Jou89] predict concurrency of a factor of two or a bit better, we regularly see factors of four or more. On some highly structured tight loops, such as SAXPY, dot product or string copy, we see factors of six to nine.

An Implementation-Independent Comparison

[Gri90] undertook an implementation-independent study and compared five architectures on three small benchmarks. The method involved uncovering the inherent data and control dependencies in the benchmarks and using these with two real arithmetic unit times, to derive a set of equations for the virtual clock cycles for each benchmark. In effect, the methodology assumed that each implementation "threw hardware at the problem", and so was constrained only by the inherent dependencies in the program. While unrealistic from a cost/performance perspective, the methodology does fairly represent "the best that can be done" by each of the architectures.

As is indicated by the preliminary performance data described in Section 3, it appears as though WM's synergy has had a significant impact on performance. Whereas studies such as [Smi84] and [Jou89] predict concurrency of a factor of two or a bit better, we regularly see factors of four or more. On some highly structured tight loops, such as SAXPY, dot product or string copy, we see factors of six to nine.

An Implementation-Independent Comparison

[Gri90] undertook an implementation-independent study and compared five architectures on three small benchmarks. The method involved uncovering the inherent data and control dependencies in the benchmarks and using these with two real arithmetic unit times, to derive a set of equations for the virtual clock cycles for each benchmark. In effect, the methodology assumed that each implementation "threw hardware at the problem", and so was constrained only by the inherent dependencies in the program. While unrealistic from a cost/performance perspective, the methodology does fairly represent "the best that can be done" by each of the architectures.

As is indicated by the preliminary performance data described in Section 3, it appears as though WM's synergy has had a significant impact on performance. Whereas studies such as [Smi84] and [Jou89] predict concurrency of a factor of two or a bit better, we regularly see factors of four or more. On some highly structured tight loops, such as SAXPY, dot product or string copy, we see factors of six to nine.

An Implementation-Independent Comparison

[Gri90] undertook an implementation-independent study and compared five architectures on three small benchmarks. The method involved uncovering the inherent data and control dependencies in the benchmarks and using these with two real arithmetic unit times, to derive a set of equations for the virtual clock cycles for each benchmark. In effect, the methodology assumed that each implementation "threw hardware at the problem", and so was constrained only by the inherent dependencies in the program. While unrealistic from a cost/performance perspective, the methodology does fairly represent "the best that can be done" by each of the architectures.

As is indicated by the preliminary performance data described in Section 3, it appears as though WM's synergy has had a significant impact on performance. Whereas studies such as [Smi84] and [Jou89] predict concurrency of a factor of two or a bit better, we regularly see factors of four or more. On some highly structured tight loops, such as SAXPY, dot product or string copy, we see factors of six to nine.
because it has strong data dependencies that generally interfere with the use of a pipelined arithmetic unit. It is the one case in which explicit nop's had to be installed to accommodate WM's data dependency rule (the fact that the result of an instruction is not available as the inner operand of the immediate succeeding instruction).

Two sets of operation times were considered: times proportional to a commercial BIT and Weitek components, respectively. The BIT times were used because they represent a very fast, non-pipelined part. The Weitek times were chosen because the represent something like a worst-case; the part is heavily pipelined, but we used it as though it were not. The relative times were:

BIT
Weitek

all operations require unit time except integer and floating divide (2 each)
the Weitek part is pipelined; the times used were those to complete a full operation; all operations require unit time except floating divide (12), integer divide (10), floating multiply (4), other floating operations (2), and integer multiply (2).

### Table: Benchmarks

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>BIT Times</th>
<th>Weitek Times</th>
</tr>
</thead>
<tbody>
<tr>
<td>FFT</td>
<td>2.40</td>
<td>1.69</td>
</tr>
<tr>
<td>IIR</td>
<td>1.97</td>
<td>1.37</td>
</tr>
<tr>
<td>DOT</td>
<td>6.80</td>
<td>2.72</td>
</tr>
<tr>
<td>LLL5</td>
<td>4.55</td>
<td>2.20</td>
</tr>
<tr>
<td>Whet-p3</td>
<td>1.33</td>
<td>1.13</td>
</tr>
<tr>
<td>STRCPY</td>
<td>5.94</td>
<td>5.94</td>
</tr>
<tr>
<td>STRCMP</td>
<td>3.94</td>
<td>3.94</td>
</tr>
<tr>
<td>SAXPY</td>
<td>8.82</td>
<td>3.23</td>
</tr>
<tr>
<td>LINKLIST</td>
<td>1.54</td>
<td>1.54</td>
</tr>
</tbody>
</table>

**Figure 4:** WM Concurrency in Various Benchmarks

The results are shown in Figure 4. As is obvious, one shouldn't use a pipelined ALU in non-pipelined mode — we observed more than a factor of two reduction in concurrency using times proportional to the Weitek part.

On a more serious level, it is interesting that each of the examples achieves some degree of concurrency, and that it often arises from different sources. In the examples with vector or string data, streaming is a large contributor, but they all also benefit from multiple operations per instruction — especially SAXPY. IIR, which uses only integer operations, does not benefit from the decoupling of execution units, but does benefit from streaming and some multiple operation instructions; it is hurt by Weitek's slow integer-multiply time. LINKLIST, which we had anticipated would get no concurrency at all, is able to fully overlap the execution of its conditional jumps.

### Instruction FIFO Sizes

The instruction FIFOs are what permit sequential program semantics while simultaneously allowing slip, as in decoupled access architectures. Thus, one suspects that the depth of these FIFOs may significantly affect performance. This was studied by Kester [Kes90].

The FIFOs need to be deep enough so that neither of the execution units is starved (idle) simply because the instruction FIFO of the other unit is full, and hence the next instruction for this unit is hidden. The compiler can assist by interleaving instructions for the two units (program semantics are invariant so long as the order of instructions for each unit is maintained separately). This optimization is trivial, at least within a basic block. Thus, if $N_i$ and $N_f$ are the number of integer and floating-point instructions in a basic block respectively, the longest string of integer and floating-point instructions in a basic block need be no more than $[N_i/N_f]$ and $[N_f/N_i]$ respectively. These are also the maximum number of FIFO entries needed to prevent starvation.

The WM simulator was instrumented to collect these data dynamically, so that the most frequently executed basic blocks are weighted most heavily. As might be expected, the results are not very interesting for programs that heavily exploit streaming, such as many scientific computations. For these the critical concurrency is between the stream units and one of the execution units, and slip between the execution units is irrelevant. For programs such as the Whetstone benchmark that do not stream heavily, however, we get results like those shown below. Thus, for example, 6.5% of the basic blocks require an instruction depth of 2 for the IEU and 1 for the FEU in order to prevent a loss of performance from instruction starvation. It is encouraging that quite modest depths — three or four — are enough to prevent performance degradation due to starvation in all cases.
Data FIFO Size

The data FIFOs allow a program to issue memory requests early, and hence at least partially mask the effects of memory latency. Thus, an obvious question is how large these FIFOs should be — or, alternatively, how is performance affected by their sizes? This was studied by both Kester [Kes90] and Wad [Wul91].

The answer to this question is, in part, dependent on the compiler’s ability to move load operations to earlier in the instruction stream. However, we can get an upper bound on the size required by considering only streaming code. A stream control unit will request the next read as soon as there is room in the FIFO; thus, it behaves like a program with explicit loads.

The more interesting cases with respect to FIFO depth are those in which the average bandwidth required by the processor is less than that provided by the memory system — but the peak bandwidth is greater. To illustrate we’ll consider the inner loop of the Stanford FFT benchmark and assume all floating operations require a single cycle. The loop requires 12 cycles and exhibits the following pattern of memory requests:

```
read read read read write write write write
```

Remembering that this "wraps around", it’s easy to see that the peak bandwidth required is 7 references in 7 cycles (4 reads and 3 writes), but the average bandwidth required is the 8 references in 12 cycles. The need for the maximum number of FIFO entries happens when the average bandwidth required by the CPU and provided by the memory system are the same — which gives rise to an actual memory reference pattern such as

```
read read read read write write write write write read read
```

This particular example requires two entries in both the input and output FIFOs to smooth the bursty CPU requests. Detailed analyses by Wad [Wul91] have confirmed that four entries are adequate in all but artificially contrived cases; the real problem will be providing enough sustained bandwidth to routines such as SAXPY.

Hardware Feasibility Explorations

Our hardware implementation efforts focused on two primary topics: (1) the implementation of the Stream Control Unit (SCU), and (2) the modeling of the WM architecture using the VHSIC Hardware Description Language (VHDL). An independent study, albeit for somewhat different purposes was done by Seitz [Sei90].

Implementation of the Stream Control Unit: The goals of designing the SCU were: (1) to develop a set of...
reusable elements that could be employed to implement other features of the WM computer, (2) to obtain realistic estimates of speed, power, area, and pin counts for the SCU and other modules, and (3) to evaluate multiple SCU designs and determine an optimal implementation.

Based on the IC designs which were developed, a complete 32-bit SCU is estimated to require approximately 50,000 transistors. Tradeoffs between speed and size can substantially reduce the transistor count, if necessary. A prototype SCU with an 8-bit datapath and four address generation units was implemented [Pam90]. The prototype was thoroughly tested and functions correctly.

VHDL Model of the WM Architecture: A functional model of the WM architecture has been developed using the VHDL Hardware Description Language (VHDL) [Sal89]. The model is implementation independent in that the WM’s architecture is described without including features which would imply or preclude any legal WM implementations. The model serves as a workbench in which the performance effects of varying system buffer sizes, memory delays, operator delays, and so forth, can be observed. In addition, the modular nature of the model will facilitate the gradual transition from implementation-independent to implementation-dependent descriptions as behavioral model elements are replaced with structural descriptions in the model.

Seitz Study: As an educational experiment, Professor Chuck Seitz, at CalTech, created a project for his VLSI class during the Spring of 1990 based on the WM specifications. The class was broken into teams with each team assuming the responsibility of the architecture, modeling, and simulation of a portion of the WM machine. A complete switch-level model was developed with supporting simulation (COSMOS) for a machine that included an IFU, an EU, and FIFOs (no FEU was included in the design but hooks for later addition of an FEU were). Hardware to support streaming was included but not address translation hardware. A 16K byte instruction cache was also included in the design. It was estimated that the full design would consume approximately 100 mm² of silicon.

Compiler Explorations

To verify that WM requires only minor extensions of traditional scalar optimizations Davidson produced a high quality compiler for it [Dav90].

The compiler translates Kernighan and Ritchie C source into WM assembly language. The compiler is implemented using a machine independent front-end that generates code for an abstract, stack-based machine. The back-end of the compiler consists of a retargetable optimizer driven by a machine description. The stack-based intermediate language is powerful enough to support additional Algol-like languages such as Pascal and Ada with minor extensions.

The following optimizations are implemented [Ben88]: unreachable code elimination, branch chaining, branch minimumization, peephole optimization, common subexpression elimination, constant folding, copy propagation, global register allocation via graph coloring, dead variable elimination, code motion, strength reduction, induction variable elimination, recurrence elimination via software pipelining and instruction scheduling. The only WM-specific optimizations were to detect streaming opportunities and issuing load and conditional instructions as soon as possible; both of these are extensions of conventional scalar techniques [Dav90].

The quality of the code produced is illustrated by the following coding of the 5th Livermore Loop. This example is interesting because it contains a recurrence (x[i] is a function of x[i-1]); such situations, also called "loop carried dependencies", must be detected by vectorizing and parallelizing compilers to prevent incorrect vectorization and/or concurrent execution of the loop body.

\[
\text{for } (i = 2; i < n; i++)
\]
\[
x[i] = z[i] * (y[i] - x[i-1]);
\]

The compiler produced the code shown in Figure 8; the code has been slightly hand-edited to insert comments and change mnemonics to conform to the most recent machine definition (the compiler is slightly behind the definition). We assume that the integer registers r20, r22, and r23 contain the addresses of x, z, and y respectively; r21 contains n. Register 31 in both the IEU and FEU is "always zero"; stores into it (as in the first 2 lines) do not affect its value.

The key point, of course, is that the actual loop is the three instructions beginning at L20. The JNI, like other jumps, can be overlapped with the floating-point operations, thus this code produces a new x[i] every other floating-point cycle.

Utilization of Two Operations/Instruction

The WM C compiler was instrumented to collect data on the nature of the code generated, including the number of times that two operations per instruction are used. As shown in Figure 9, over a fairly large benchmark sample, about 30% of all instructions use both operations — or, equivalently, almost half of all operations are contained in such instructions.

Instructions that do not fit the 2-op format (e.g., branches) were not counted. Of those that use the 2-op format, we counted those that used zero, one or two operations (zero operations occur in an instruction is used for a simple register-register transfer). The rightmost column shows the percentage of operations appearing in the 2-op format.
r31 := (r21-1)<=0  -- check if loop should be executed
L64f  r31 := (8) + r20  -- generate memory request for x[1]
r24 := (r21) - 1  -- compute number of items to stream
f22 := f0  -- dequeue x[1]
jumpIT  L1  -- jump if loop not to be executed
r19 := (16) + r22  -- compute address of z[2]
Si64f  f1, r19, r24, 8  -- stream in z[2], z[3], ..., z[n]
r19 := (16) + r23  -- compute address of y[2]
Sin64f  f0, r19, r24, 8  -- stream in y[2], y[3], ..., y[n]
Sout64f  f0, r19, r24, 8  -- stream out x[2], x[3], ..., x[n]
L2:  f22 := (f0-f22)*f1  -- compute x[i] (nb, dequeues z[i] & y[i])
f0 := (f31)+f22  -- store x[i] (nb, f22 == next x[i-1])
Jni  L2  -- jump if stream count not zero

Figure 8: Compiled Code for the 5th Livermore Loop

Summary

At this point quite a bit about the potential performance of the WM architecture. We know that modest levels of concurrency in the range of 2-9 are possible over a reasonably broad range of applications. We know that each of the kinds of concurrency contributes to the total performance in some applications. We know that a surprising number of operations fit the 2-operator/instruction format. We know that modest values for the architectural parameters such as FIFO size are adequate to achieve this level of performance. We know that non-heroic compilation techniques are adequate to achieve this performance. Finally, we have indications that implementation will consume chip area comparable to that of a conventional RISC design.

We also know the probable Achilles heel of the architecture, namely the need to build a memory system capable of supplying the bandwidth that the processor can absorb. It is -- fortunately or unfortunately -- a problem that has to be faced by all of the newer architectures. Even traditional RISC designs are outstripping the bandwidth capabilities of commercial memory components.

Acknowledgements

Many people have contributed to the results reported here. My special thanks go to: Jim Aylor, Barry Johnson, Anita Jones, Jack Davidson, Andrew Grimshaw, Charlie Hitchcock, Chuck Seitz and his students, Ricky Benitez, Peter Kester, Sally McKee, Ramesh Peri, Max Salinas, and Rohit Wad.

Bibliography

[Ben88] Benitez, M. E. and J. W. Davidson, "A Portable Global Optimizer and Linker", in