Speed Up Results for Software Pipelined Loops on ILP Architectures

Eric Stotzer  
Texas Instruments  
P.O. Box 1443, MS 730  
Houston, TX 77251  
281-274-2404  
estotzer@ti.com

Ernst Leiss  
University of Houston  
Dept. of Computer Science  
Houston, TX 77204-3475  
713-743-3359  
coscel@cs.uh.edu

Abstract

Software pipelining is a technique used to schedule instructions from a loop body so that multiple iterations of the loop are executing in parallel. This can be accomplished by taking advantage of instruction level parallelism (ILP). ILP is a combination of compilation techniques and architectural features that exploits the fine grain parallelism present at the machine instruction level. Software pipelining is a form of instruction scheduling that has the potential for finding the optimal instruction schedule for a loop body. In this paper we present speed up results using software pipelining to schedule the first 14 loop kernels in the Livermore loops benchmark for a sequence of ILP architectures with an increasing number of parallel resources. Our results show that software pipelining is an instruction scheduling technique capable of creating enough fine grain parallelism to justify very wide instruction word ILP architectures.

1. Introduction

The traditional sequential execution model performs one operation for each machine instruction that is issued. Data dependency relations between operations force a sequential ordering on some of the operations, but many operations are independent of each other. Independent operations can potentially execute in parallel and increase the operations per instruction ratio. A combination of compilation techniques and architectural features known as Instruction Level Parallelism (ILP) are used to execute multiple operations for each machine instruction. Since it is transparent to the programmer, the very fine grain parallelism found in ILP is different from that which is seen in vector and multiprocessor architectures [RF93].
There is some doubt as to whether enough fine grain parallelism exists in most programs to justify very long instruction word (VLIW) architectures, which are capable of issuing a large number of operations in parallel. Specifically, Sohi and Vajapeyam, using loop unrolling and list scheduling, showed in [SV89] that for an architecture similar to the Trace [LFK+93] and Cydra 5 [BYA93] VLIW architectures, the most speed up was achieved by issuing only two operations in parallel. Especially for deeply pipelined architectures, they presented results which suggest that there is not enough performance increase to justify the hardware costs of VLIW architectures that issue eight or more operations in parallel.

Although Fisher and Ellis obviously believed that wide architectures were viable, they used trace scheduling and loop unrolling to find ILP, and they questioned whether software pipelining was an effective technique for very wide VLIW architectures [Fisher79] [Ellis85]. Exploiting ILP across loop iterations using unrolling has drawbacks such as increasing code size and determining a good unroll factor. Similar to loop unrolling, software pipelining is a technique used to schedule instructions from an inner loop so that multiple iterations of the loop are executing in parallel. This can be accomplished by initiating new loop iterations before previous iterations are complete by taking advantage of parallel hardware resources. Software pipelining takes the N instructions in an inner loop body and forms an M-stage pipeline as if a vector functional unit were specially designed to execute the loop body.

Finding a valid software pipeline is an instruction scheduling problem. A component of the compiler must perform instruction scheduling by examining data dependencies to determine which operations may execute in parallel. Finding an optimal solution to an instruction scheduling problem reduces to the NP-complete bounded resource scheduling problem [GI79] [Fisher79] [Lam87]; therefore, heuristics based scheduling algorithms are used to approximate solutions. The success of a scheduling algorithm is affected by the schedulability of the target architecture [RG81]. In this paper, we show speed up results attained by using software pipelining on the first 14 loops in the Livermore loops benchmark [McMahon72]. We use a sequence of architectures with an increasing number of operations that can issue in parallel. Our results indicate that when software
pipelining is used, there is a significant speed up attained by increasing the number of operations that issue in parallel.

2. Synopsis of related work

Charlesworth coined the term software pipelining [Charlesworth81]. Kogge developed a microcode assembler that implemented software pipelining [Kogge77]. Patel and Davidson developed an algorithm for designing pipelined hardware functional units [PD81]. Rau and Glaeser defined the fundamental framework of a software pipelining technique called modulo scheduling and the polycyclic architecture [RG81]. Fisher developed trace scheduling [Fisher79] and Ellis implemented the Bulldog compiler [Ellis85] for the ELI project at Yale University. Multiflow Corporation produced the VLIW Trace computer based on the research of the ELI project and the Bulldog compiler [LFK+93]. [Touzeau84] describes the software pipelining algorithm used in the compiler for the FPS-164 attached array processor. The VLIW Cydra 5 computer developed at Cydrome Corporation implemented hardware support for software pipelining [BYA93] [DHB89]. The Cydra compiler implemented a software pipelining technique called slack scheduling [DT93] [TLS90]. Huff modified the slack scheduler in the Cydra compiler to minimize register pressure [Huff93]. Lam implemented software pipelining and developed modulo variable expansion in the compiler for the VLIW Warp computer developed at CMU [Lam87] [Lam88]. [RLTS92] describes register allocation algorithms for software pipelined loops and [Rau92] describes code generation methods for generating setup code after a loop has been pipelined. Nicolau and Potasman developed a class of software pipelining algorithms that unroll loop bodies and attempt to detect a repeating pattern [NP90]. [TLS90] describes techniques for software pipelining while-loops. Rau and Fisher provide an excellent description of the history of ILP in [RF93].

3. Software Pipelining

The compiler framework required for exploiting ILP requires extensive dependency analysis to detect instructions which are independent. Much of the sophisticated dependency analysis techniques developed over the years of research into vectorizing compilers is applicable to software pipelining [Leiss95] [Zima90]. The goal of software pipelining is to find an instruction schedule
that increases loop throughput by initiating new iterations before previous iterations complete. Finding this instruction schedule is complicated by resource constraints and loop carried dependencies. In our research we found two very different software pipelining algorithms based on a technique called modulo scheduling:

- Lam's nonbacktracking algorithm [Lam87].
- The Cydra 5 compiler's backtracking algorithm [DT93] [TLS90] [Huff93].

Lam's algorithm is based on list scheduling and hierarchical reduction. Strongly connected components [Tarjan72] in the dependency graph are scheduled to produce a single node which represents the union of the resource and dependency constraints of the connected component. This eliminates recurrences from the dependency graph, and list scheduling with a modulo constraint is then used to schedule the resulting acyclic graph. Register lifetimes are minimized by scheduling in the opposite direction of the execution order.

This Cydra 5 compiler's slack scheduling algorithm is based on operational scheduling. The slack range for each operation is dynamically calculated using the dependency constraints, the partially constructed schedule, and the iteration interval. Operations with the smallest slack range or which use critical resources are scheduled first. Register lifetimes are minimized by using a bi-directional heuristic that places operations which share dependencies closer together.

Our implementation [Stotzer95] is based on the slack scheduler as described in [Huff93]. This algorithm performs bounded backtracking. We modified the algorithm to delay functional unit allocation until scheduling. We implemented a stand-alone loop scheduler called the SP-scheduler. The SP-scheduler takes as its input an unscheduled assembly language loop body, builds a data dependency graph, software pipelines the loop, and outputs the valid schedule. Using this approach, we are able to focus on instruction scheduling and its interaction with the target architecture. The SP-scheduler can be used in designing compilers; it can also be used by assembly language programmers to explore how code was or could be scheduled. The SP-scheduler also allows hardware designers to experiment with different architectures to measure performance on inner loops. The SP-scheduler uses a machine model to define the characteristics of the
resources available for scheduling. The machine model describes the number of functional units, what each functional unit does, and what the functional unit latencies are. It is simple to change the machine model.

4. **Target Architecture**

The interesting features of a VLIW architecture are how many functional units it has, the delay of the functional units, the interconnect between the functional units, the register file, and hardware features that are implemented to aid instruction scheduling.

If there is one general purpose register file for all functional units, then each unit will require two read ports and one write port into the register file. Since the size of the interconnect into the register file grows in proportion to the product of the number of read ports and write ports [RF93], the interconnect between the register file and functional units can be costly and may lead to complex routing problems. This problem can be solved by splitting the register file among different functional units or reducing the number of functional units. The split can be made in different ways. For example, there could be a floating point register file and an integer register file; alternatively, a cluster of units can have its own register file. Explicit operations must move values between register files in different clusters. Restrictions placed on the allocation of registers complicates register allocation and instruction scheduling for the compiler.

Because of replicated functional units, or because an operation can be performed on different units, there may be multiple functional units that perform a specific operation. For example, multiple units may be able to execute a register move operation or an integer addition. Different functional units may not have direct access to the same registers. The concept of operation locality emerges. It is important to place dependent operations as close to each other as possible to avoid having to schedule explicit copy operations. Therefore, simply adding more functional units complicates instruction scheduling by creating a partitioning problem.

We present a VLIW architecture with four functional units. Figure 1 describes the functional units. We assume a symmetrical instruction set that is similar to a RISC. Some of the functional units perform multiple operations. For example, the S-unit is an ALU that is specialized for performing
a scalar shift; the A-unit is specialized for performing float addition; and the D-unit is specialized for accessing memory. Any of these three units can also be used for performing scalar addition. We assume a single register file is accessible to all functional units on every cycle. The configuration of this architecture allows us to use it as a target for both fixed and floating point algorithms. This exercises the scheduler at extremes since the fixed point operations have short latencies, and the floating point operations have long latencies.

We assume the architecture can conditionally execute any instruction [DHB89]. Therefore, global scheduling techniques can be used to eliminate conditional blocks [AKPW83]. We also assume an unlimited number of registers and that all functional units are connected to a global register file.

<table>
<thead>
<tr>
<th>Unit</th>
<th>Operation</th>
<th>Modest</th>
<th>Deep</th>
</tr>
</thead>
<tbody>
<tr>
<td>M-unit</td>
<td>float multiply</td>
<td>4</td>
<td>8</td>
</tr>
<tr>
<td></td>
<td>scalar multiply</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>A-unit</td>
<td>float add</td>
<td>4</td>
<td>8</td>
</tr>
<tr>
<td></td>
<td>scalar logical</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>scalar addition</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>S-unit</td>
<td>scalar shift</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>scalar logical</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>scalar addition</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>D-unit</td>
<td>load</td>
<td>5</td>
<td>10</td>
</tr>
<tr>
<td></td>
<td>store</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>scalar addition</td>
<td>1</td>
<td>2</td>
</tr>
</tbody>
</table>

Figure 1: Operation Latencies

5. Results

We hand-assembled the first 14 loop kernels in the Livermore loops benchmark [McMahon72] into serialized assembly language; this is similar to [SV89]. We ran SWIPE on the loop kernels using six different architectures, each with an increasing number of functional units that can issue operations in parallel, and an ideal architecture with an unlimited number of resources. The first three architectures are based on the single cluster VLIW described in Section 4; these architectures issue one, two, and four operations in parallel, respectively. The remaining architectures increase
the number of functional units by simply replicating a cluster. We also ran the experiments on modest and deeply pipelined versions of all architectures.

The loop kernels basically consist of an inner and outer loop. According to the structure of the Livermore loops benchmark, each loop is run three times with different trip counts for the inner and outer loops. For each loop kernel, we developed a total execution time formula that is a function of the loop trip counters, and the execution time of the inner loop. Speed up results are presented relative to list scheduled versions of the loop kernels assuming a modestly and deeply pipelined target architecture capable of issuing one operation at a time. Figure 2 shows the speed up results for each of the architectures.

<table>
<thead>
<tr>
<th>Loop Kernel</th>
<th>Mod</th>
<th>Deep</th>
<th>Mod</th>
<th>Deep</th>
<th>Mod</th>
<th>Deep</th>
<th>Mod</th>
<th>Deep</th>
<th>Mod</th>
<th>Deep</th>
<th>Mod</th>
<th>Deep</th>
<th>Mod</th>
<th>Deep</th>
<th>Mod</th>
<th>Deep</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2.2</td>
<td>4.1</td>
<td>4.0</td>
<td>7.1</td>
<td>5.8</td>
<td>10.2</td>
<td>10.8</td>
<td>18.0</td>
<td>10.9</td>
<td>18.1</td>
<td>19.1</td>
<td>18.1</td>
<td>19.1</td>
<td>18.1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>1.8</td>
<td>3.1</td>
<td>3.4</td>
<td>5.8</td>
<td>3.4</td>
<td>5.8</td>
<td>6.4</td>
<td>10.0</td>
<td>8.8</td>
<td>13.3</td>
<td>8.8</td>
<td>13.3</td>
<td>14.5</td>
<td>13.3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>2.1</td>
<td>3.6</td>
<td>4.0</td>
<td>3.6</td>
<td>4.0</td>
<td>3.6</td>
<td>4.1</td>
<td>3.6</td>
<td>4.1</td>
<td>3.6</td>
<td>4.1</td>
<td>3.6</td>
<td>4.1</td>
<td>3.6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>1.5</td>
<td>2.3</td>
<td>2.5</td>
<td>2.7</td>
<td>2.9</td>
<td>2.7</td>
<td>3.0</td>
<td>2.8</td>
<td>3.1</td>
<td>2.8</td>
<td>3.1</td>
<td>2.8</td>
<td>3.1</td>
<td>2.8</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>1.8</td>
<td>1.7</td>
<td>2.0</td>
<td>1.9</td>
<td>2.0</td>
<td>1.9</td>
<td>2.0</td>
<td>1.9</td>
<td>2.0</td>
<td>1.9</td>
<td>2.0</td>
<td>1.9</td>
<td>2.0</td>
<td>1.9</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>2.7</td>
<td>3.6</td>
<td>4.0</td>
<td>3.6</td>
<td>4.0</td>
<td>3.6</td>
<td>4.0</td>
<td>3.6</td>
<td>4.0</td>
<td>3.6</td>
<td>4.0</td>
<td>3.6</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>1.5</td>
<td>3.0</td>
<td>2.8</td>
<td>5.3</td>
<td>4.2</td>
<td>7.7</td>
<td>7.9</td>
<td>13.8</td>
<td>9.5</td>
<td>16.3</td>
<td>12.2</td>
<td>20.1</td>
<td>26.6</td>
<td>26.2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>1.1</td>
<td>1.3</td>
<td>2.0</td>
<td>2.5</td>
<td>2.9</td>
<td>3.4</td>
<td>5.4</td>
<td>6.5</td>
<td>8.4</td>
<td>9.8</td>
<td>9.7</td>
<td>11.2</td>
<td>44.8</td>
<td>27.6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>1.3</td>
<td>2.5</td>
<td>2.4</td>
<td>4.4</td>
<td>3.4</td>
<td>6.1</td>
<td>5.9</td>
<td>10.1</td>
<td>8.2</td>
<td>13.5</td>
<td>10.5</td>
<td>16.5</td>
<td>21.5</td>
<td>20.9</td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>1.6</td>
<td>2.7</td>
<td>2.5</td>
<td>4.1</td>
<td>2.5</td>
<td>4.1</td>
<td>4.7</td>
<td>7.5</td>
<td>6.4</td>
<td>9.9</td>
<td>8.5</td>
<td>12.6</td>
<td>24.5</td>
<td>21.5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>2.3</td>
<td>2.7</td>
<td>2.9</td>
<td>2.7</td>
<td>2.9</td>
<td>2.7</td>
<td>2.9</td>
<td>2.7</td>
<td>2.9</td>
<td>2.7</td>
<td>2.9</td>
<td>2.7</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>2.1</td>
<td>3.7</td>
<td>4.1</td>
<td>7.0</td>
<td>4.1</td>
<td>7.0</td>
<td>6.1</td>
<td>10.1</td>
<td>11.5</td>
<td>10.2</td>
<td>11.5</td>
<td>10.2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>1.1</td>
<td>1.6</td>
<td>2.1</td>
<td>2.9</td>
<td>2.8</td>
<td>3.8</td>
<td>4.9</td>
<td>6.1</td>
<td>7.0</td>
<td>8.3</td>
<td>8.0</td>
<td>9.2</td>
<td>18.2</td>
<td>13.6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>14</td>
<td>1.5</td>
<td>2.2</td>
<td>2.4</td>
<td>3.5</td>
<td>2.4</td>
<td>3.5</td>
<td>4.5</td>
<td>6.4</td>
<td>7.0</td>
<td>9.7</td>
<td>8.6</td>
<td>11.8</td>
<td>27.5</td>
<td>20.6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Avg.</td>
<td>1.8</td>
<td>2.7</td>
<td>2.9</td>
<td>4.1</td>
<td>3.4</td>
<td>4.7</td>
<td>5.2</td>
<td>7.4</td>
<td>6.7</td>
<td>8.8</td>
<td>8.1</td>
<td>9.8</td>
<td>16.0</td>
<td>13.3</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 2: Using software pipelining, speed up attained by increasing the number of operations that issue in parallel for architectures with modest and deep hardware pipelines.

The rate at which new loop iterations are initiated is bounded by the number of available resources that can execute in parallel and recurrences. Recurrences appear as cycles in the data dependency graph and result from data dependencies that are loop carried. At a point determined by the recurrence bound, adding more resources will not speed up loop kernels with recurrences. This is evident in the speed up results for loop kernels 3, 4, 5, 6, and 11, each of which contains a recurrence. For the modestly pipelined ideal resource architecture, the upper bound on speed up is
sometimes double the deeply pipelined ideal resource architecture. This occurs when the software pipelined loop kernel reduces to one instruction that issues all loop operations in parallel.

We find it very encouraging that as more functional units are added, speed up continues to increase significantly. This also seems to suggest that software pipelining does scale as more functional units are added. The results are especially encouraging for deeply pipelined architectures as contrasted with those obtained when using list scheduling and loop unrolling [SV89]. Assuming there is not a loop recurrence, loop speed up is limited by the number of functional units, the delay through the functional units, and the number of operations in the loop. As the number of functional units increases, it becomes beneficial to increase the number of operations in the loop by performing some degree of loop unrolling before the loop is software pipelined [LKB91]. Our results are presented with no loop unrolling.

Assuming that a deeply pipelined architecture could execute at a clock rate that is twice the speed of a modestly pipelined architecture, then one can compare all of the speed up results on the same scale. This suggests that doubling the clock rate and pipeline latencies is comparable to doubling the number of functional units. For example, compare the speed up of the four-operation deeply pipelined architecture with the eight-operation modestly pipelined architecture (4.7 vs 5.2). The values are quite close. Intuitively this makes sense, since the total number of operations issued in a single time unit is the same (four operations in 0.5 cycles and eight operations in 1 cycle). The advantage of the deeply pipelined architecture is four operations issue in parallel which reduces the number of read and write ports into the register file. This would simplify the interconnect between the register file and functional units.

6. Conclusion

We implemented a standalone software pipelining scheduler (SP-scheduler) that takes an architecture description and a loop kernel and emits a software pipelined loop. We defined a target architecture with four functional units. We hand-assembled the first 14 loop kernels in the Livermore loops benchmark and used the SP-scheduler to schedule them on a sequence of modestly and deeply pipelined architectures. Each architecture had the potential to issue an increasing
number of parallel operations. As summarized by Figures 3 and 4, we found speed up results which indicate clearly that using software pipelining, wide instruction word ILP architectures can significantly increase performance for inner loops. However, loops with recurrences do not benefit from the additional resources, although other optimization techniques could be used to reduce or eliminate the recurrence. We also found that software pipelining on deeply pipelined architectures with faster clock rates is a potential solution for the interconnect bottleneck encountered by wide instruction word ILP architectures.

![Deep Pipelines](image)

**Figure 3: Average speed up for architectures with deep hardware pipelines**

![Modest Pipelines](image)

**Figure 4: Average speed up for architecture with modest hardware pipelines**
7. References


