*To appear in the*
*Proceedings of IFIP WG 10.3 (Concurrent Sytems) Working Conference*
*on Architectures and Compilation Techniques for Fine and Medium Grain Parallelism*
*Jan. 1993, Orlando, FL.*

# Modelling Instruction-Level Parallelism for Software Pipelining

Ali-Reza Adl-Tabatabai[1], Thomas Gross[1], Guei-Yuan Lueh[1], and James Reinders[2]

[1]School of Computer Science, Carnegie Mellon University, Pittsburgh, PA 15213-3891
[2]Intel Corporation, SSD, 5200 NE Elam Young Parkway Hillsboro, OR 97124-6497

## 1 Introduction

Software pipelining is an attractive method to schedule code for processors that exhibit instruction-level parallelism such as pipelined, super-scalar, and (V)LIW machines. It has been implemented for a variety of processors ( e.g. FPS-164[10], Warp[9], Cydra-5[7]), and a number of pipelining algorithms have been described in the literature. Software pipelining produces a schedule so that the executions of multiple loop iterations are overlapped. Operations from multiple loop iterations are either placed into the same instruction word, or are interleaved, depending on the type of instruction-level parallelism provided in the target architecture. In contrast to other scheduling techniques that rely on loop unrolling to determine the number of iterations to overlap, the degree of overlapping in software pipelining is determined by the exact resource requirements of the loop operations. Thus, accurate modelling of machine resources is especially crucial for software pipelining. Modelling the target machine is easy for idealized machines, but for a real processor, a number of challenges arise. For example:

- If there are several ways to perform a specific operation (e.g., because there are multiple functional units that all implement a common set of operations, but each unit also implements some operations that cannot be performed by another unit), a specific functional unit must be assigned to an operation. A phase ordering problem exists between assigning functional units to operations and scheduling the operations.

- If the instruction set uses implicit registers, such registers impose resource constraints. A value from an implicit register must be used as soon as possible to free up the resource.

- Destructive read instructions (e.g., reading the top of a queue has the effect of dequeuing the value) impose constraints on the scheduler. If a destructively read value is used multiple times, care has to be taken that all such users obtain indeed the same value.

- If the target machine includes different instruction formats with different amounts of parallelism, there may be additional constraints on which operations can be scheduled together.

Existing code generators have often dealt with these problems by employing heuristics or by using ad-hoc methods. This paper discusses a different approach based on abstractions and transformations that simplify the modelling of the target processor and allow efficient resource utilization during code scheduling.

# 2 Target architecture

Some of the material in this paper is based on our experience with building compilers for the iWarp system. iWarp is a microprocessor developed by Carnegie Mellon and Intel Corp. for use in parallel systems. Each processor consists of a computation agent (similar to a microprocessor) that is closely coupled to a communication agent[2,3]. The computation agent is controlled by a long instruction word [5] and provides the source of examples for this paper. Similar issues arise when using software pipelining for other processors with instruction level parallelism (although the details may be different).

The instruction set of the iWarp computation agent contains two instruction formats: short 32-bit instructions that control a single functional unit (similar to the instruction set of commercial RISC processors) and a powerful 96-bit long instruction word (LIW) instruction. An LIW instruction allows operations to be specified for several functional units. The functional units that are available in an LIW instruction are a floating point adder (FPA), a floating point multipler (FPM) and either two operand generation units (AGU1 and AGU2) or the Integer Logic Unit (ILU)[5,6]. All functional units execute concurrently, except for the AGU1 and AGU2, which execute sequentially with respect to each other. The functionality provided by the AGUs is limited. Either AGU can perform simple two address integer addition and subtraction, as well as load operations. AGU2 can also perform store operations. The ILU provides more general integer and logical operations, as well as branches. In addition, all functional units provide a register to register move operation. Both formats also includes a single bit that specifies a loop closing operation ( decrement a counter and jump to the beginning of the loop if the counter is not zero).

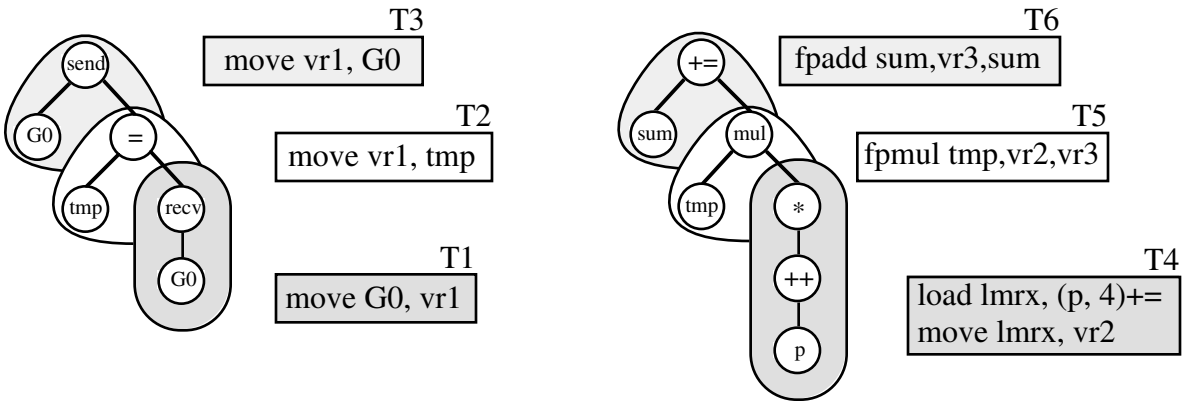



Figure 1: Templates

Although there are two instruction formats, the set of operations that can be performed by the LIW format is a subset of those that can be performed by the short instruction format. Hence, code selection can select LIW format machine operations wherever possible without committing to the LIW format, allowing the selection of instruction format to be postponed until code emission. Modelling the short instruction format is straight forward, therefore we

only discuss the LIW format in the rest of this paper. We note, however, that code selection would be more complicated if certain operations were only available in the LIW format.

To save encoding space in the 96-bit instruction format, AGU operations use *implicit registers* as the destination register for load operations and the source register for store operations. The destinations of the AGU1 and AGU2 load operations are the `lmr1` and `lmr2` (local memory read) registers respectively, while the source of a store operation is the `lmw` (local memory write) register. The value in an `lmr` register is available only until the next load operation that targets it. The value in the `lmw` register must be stored to memory before another operation targets it.

Inter-processor systolic communication is performed by reading from or writing into queues. These queues are accessed via *gates*, i.e., the head or tail of a queue is mapped into the computation agent's register space. Writing to a gate has the effect of enqueuing a (double)word into a communication queue, while reading a gate dequeues a (double)word. If a gate is read by several operations concurrently (in the same cycle within an LIW instruction), all operations receive the same value. We call an operation that targets a gate or implicit register a *producer*, while an operation that uses the value in a gate or implicit register is referred to as a *consumer*.

# 3 Code generator model

The code generator has the task of mapping the intermediate representation (IR) of a program into machine instructions. The IR of a program has been subject to a variety of optimizations and transformations (e.g., see [1] and [11] for descriptions of such optimizations). The intermediate representation is typically a graph (e.g., DAGs, or trees), where edges indicate data or control dependences. In our system, the IR consists of trees in which tree edges indicate flow dependences. Additional inter-tree arcs are added to indicate other dependences (e.g., input, output and anti dependencies as well as loop carried dependencies for inner loops). The additional dependence information is necessary for scheduling and is computed by a data flow analysis phase in the code generator.

Code generation is organized into four phases that are run in series, with no backtracking among the phases. The **code selection** phase maps IR operations to machine operations. The code selector is based on a simple non-backtracking recursive pattern matcher that matches tree patterns against sub-trees in the IR[8]. Each tree pattern contains a sequence of machine operations called a *code template*. A code template includes the resource requirements (such as functional units or instruction fields) of its machine operations. The machine operations produced by this phase reference *virtual* registers that are allocated from an infinite pool. The **code scheduling** phase places the machine operations into a sequence of machine instructions, using software pipelining for inner loops and straight-line code compaction for other regions. **Register assignment** builds live ranges of virtual registers and assigns physical registers to virtual registers using Chaitin-style graph coloring [4]. Coalescing eliminates register move operations. Spills and reloads are added to the schedule for live ranges that are spilled. Finally, the **code emission** phase turns the schedule into assembly code. This phase generates prologue and epilogue code for pipelined loops and assigns functional units to machine operations.

As an example, consider the following C code loop of a vector dot product:

```
for (i = 0; i < N; i++){
        _sendf(tmp = _receivef(GATE0_in),GATE0_out);
        sum += tmp * *p++;
}
```

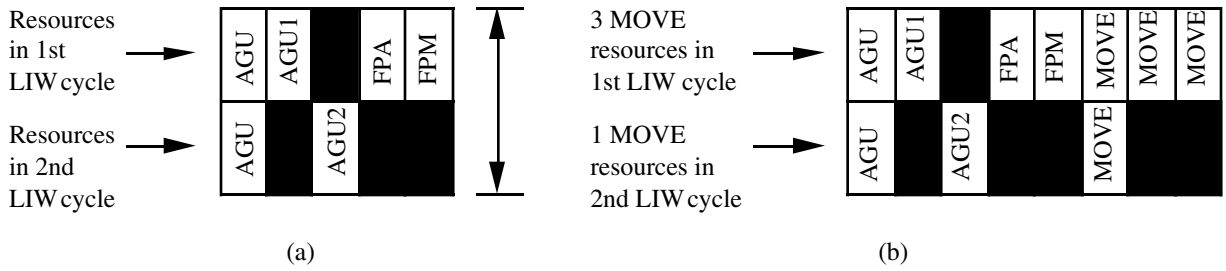



Figure 2: (a) Basic machine model (b) Augmented machine model

This loop computes the dot product of two vectors, where one vector is accessed from memory and the other is received from a systolic communication port (GATE0_in) and subsequently forwarded on a transmission port (bound to GATE0_out). The code was taken from a neural network simulation program written for the iWarp. Figure 1 illustrates the IR and selected code templates for this code. The variable `tmp` is allocated a register (tmp), and GATE0_in and GATE0_out are abbreviated to "G0"; the position indicates if a gate is used for input or output. For brevity we have not shown the counter increment and branch back operations.

# 4 Modelling functional unit resources

A model of the LIW instruction must capture the functional unit resources available in an LIW instruction, as well as the execution order of AGU operations. The resources available in an LIW instruction are modelled by a *resource vector*, which contains a slot for each functional unit. To model the sequential execution of the two AGU resources, an LIW instruction is modelled as two sequential cycles. The first cycle of an LIW contains AGU, FPA, and FPM resources, while the second cycle contains only an AGU resource. The resource utilization of a machine operation $m$ is modelled by specifying the resources that $m$ requires to be scheduled in an instruction. E.g., a floating point add operation specifies that it requires the FPA resource and is scheduled at an instruction $I$ only if $I$ has the FPA resource free.

Since certain operations, such as integer addition, can be performed using either AGU resource, while a store can only be performed using AGU2, the two AGU resources need to be distinguished. To model this distinction, the additional resources AGU1 and AGU2 are added to the resource vector. Figure 2(a) illustrates the basic machine model. An operation that can be performed using either AGU resource requires only an AGU resource, while a store operation requires both an AGU and an AGU2 resource. Thus operations that can be performed using either AGU resource do not commit to a particular AGU during code selection. An ILU operation utilizes the AGU1 and AGU2 resources as well as both AGU resources, since if an ILU operation is scheduled in an instruction $I$, no other AGU operations can be performed by $I$; conversely if an AGU operation is scheduled in an instruction $I$, no ILU operations can be performed by $I$.

# 5 Functional unit assignment

A phase ordering problem that occurs in code generation involves the assignment of machine resources to instructions. A classic example of this is the phase ordering problem caused by register assignment and code scheduling. If register assignment is performed before schedul-

ing, false dependences caused by the reuse of registers impact code scheduling. On the other hand, if code scheduling happens before register assignment, it usually increases register pressure, impacting register assignment. Also, if spill/reload code is required, it must be added to the schedule by the register assigner, thus affecting the quality of the schedule (unless the scheduler is re-run). In iWarp, the problem of resource assignment manifests itself in another form as well. The iWarp instruction set allows move operations to be performed by either the FPA, the FPM, or by the AGU. Similarly, a load operation can be performed in either one of the two AGU cycles. These operations must be assigned functional units during code generation. Since instruction scheduling is constrained by functional unit resources, the assignment of functional units to moves and loads impacts the code scheduler.

In a typical scalar machine, an operation can only be performed by one functional unit. Therefore, assignment of functional units is done at the same time as instruction selection. The trend in LIW and superscalar machines is to include additional functional units or to provide multiple paths to memory, creating more options for the compiler to perform a given operation. Code generators for these types of machines have to handle the additional phase ordering problem caused by functional unit assignment.

There are three ways how a code generator can handle the issue of selecting the functional unit for a given operation : (i) during code selection, (ii) during code scheduling, or (iii) after code scheduling. The first approach is insensitive to the later code scheduling phase and may result in unnecessary resource constraints. E.g., since at code selection time it is not known which operations end up in the same instruction, it is possible to pick functional units such that two move operations cannot be scheduled together. In the second approach, an available functional unit is allocated to a move or load operation as it is being scheduled. Again, unnecessary resource constraints may result since the allocation is ambivalent to the resource requirements of subsequently scheduled operations. To have the least impact on resource constraints, functional unit selection should be done after code scheduling. This approach requires a mechanism that ensures the availability of sufficient functional units for all operations scheduled into one instruction. Otherwise, it may not be possible to satisfy a specific schedule (as we want to avoid iterations of the scheduler).

For brevity, we focus on single precision move operations in the rest of this paper. However, the approach we describe for functional unit assignment solves the general problem and has been used to handle other operations such as loads and double precision moves.

# 6 Generic operations

In the iWarp C compiler, the functional unit problem is solved using the concept of *generic operations*. Generic operations allow the binding of move operations to functional units to be deferred until code emission, the latest possible time. A move is modelled as an operation that is generic; i.e. one that is not bound to any *particular* functional unit from the set of eligible units. A generic operation's resource utilization is modelled in a way that guarantees the availability of resources during and after scheduling without actually committing to a particular resource. Furthermore, generic operations allow *all* the resources of an instruction to be used, regardless of the order in which operations are scheduled. Moreover, any combination of operations that can be scheduled together without resource conflicts are allowed in one instruction.

To implement generic moves, the iWarp machine model is augmented with four generic move resources, since there are four options (FPM, FPA, AGU1, AGU2). The augmented resource vector is shown in Figure 2(b). There is one generic move resource for each functional unit that can perform a move, but there is no correspondence between a generic move

and a particular functional unit; an individual generic move can correspond to any physical functional unit. The generic move resources do not represent additional physical resources. Rather, they are extensions to the machine model that guarantee the availability of resources for move operations.

During code selection, a generic move operation is selected wherever a register to register move operation is required. A generic move operation represents a move operation that is not bound to any particular functional unit, and its resource vector requires only a generic move resource. Since there is one generic move resource for each functional unit that can perform a move, the use of one of these functional units implies one less generic move resource. Therefore, an operation $O$ that requires a specific functional unit $F$ also requires a generic move resource, if $F$ can be used for a move. This means that an instruction must have both a generic move and an $F$ resource available to satisfy $O$'s resource constraints. When the *code emitter* encounters a generic move, it checks the resources used by the other operations scheduled in the same instruction as the move and selects an available functional unit to perform the move. (A practical note: if there are only partial bypasses between the functional units, selection of the unit to perform a move operation must take the structure of the bypasses into account.)

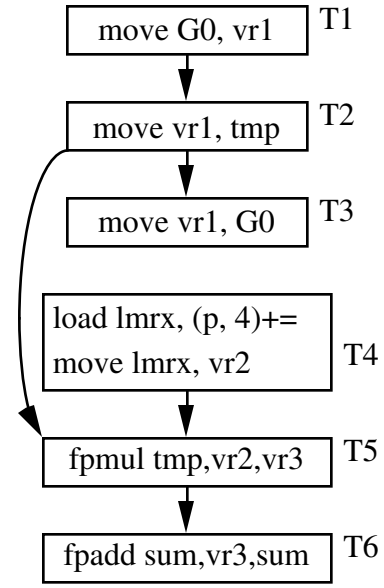
Generic operations are transparent to the code scheduler, and the concept of the generic operation has been employed to model several aspects of the iWarp machine. The concept is a general one and can be applied to any situation where functional units support different operations that have the same net effect. Given an appropriate representation, we can even deal with generic operations where different functional units incur a different latency. For example, AGU1 and AGU2 load operations differ in their latency (among other details), so making a decision early (before scheduling) results in inferior schedules. The iWarp C compiler is able to delay the decision by modelling both options as generic loads.

# 7 Implicit registers and gates

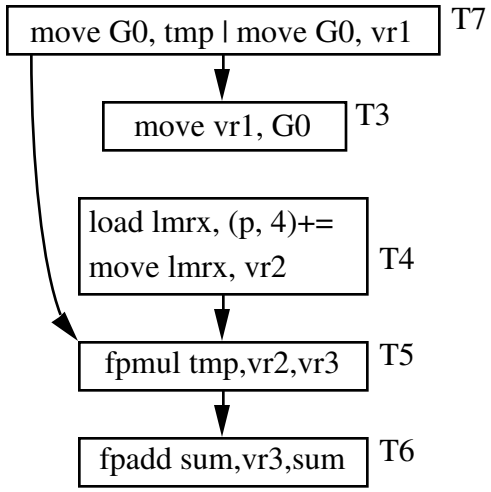
Implicit registers and gates are similar in that in both cases, it is desirable to schedule all uses of a value in a gate or implicit register in one instruction. The use of implicit registers causes additional storage conflicts to occur, since all loads and stores implicitly reference the same storage location. The resulting anti and output dependences constrain code scheduling and cannot be removed by renaming. Storage conflicts caused by implicit registers can be alleviated by placing all consumers of an implicit register's value in the instruction immediately following the producer. This means that all instructions using a loaded value must be scheduled together in the instruction following the load. If this is not possible, the implicit register's value can be buffered in a register to free up the implicit register resource. Similarly, if the value in a gate register $G$ is read by an operation $O$, then ideally $O$ should use $G$ as a source register. Moreover, if $G$'s value is used by several operations, these operations should all be scheduled in the same instruction. If this is not possible, the gate's value must be buffered in a register. The destructive gate reads cause input dependences, which are captured by dependence arcs added to the IR by the data flow analysis phase.

Generating such code in the code selector results in ad hoc case analysis; the number of cases (or tree patterns in our system) that must be considered by the code selector explodes. Furthermore, intermediate representations typically buffer common subexpressions in a temporary location. Therefore, if a gate or implicit register value is used by more than one operation, it is copied into a register before use. The code selector cannot place all uses of the value in one instruction unless it increases the number of cases it considers.

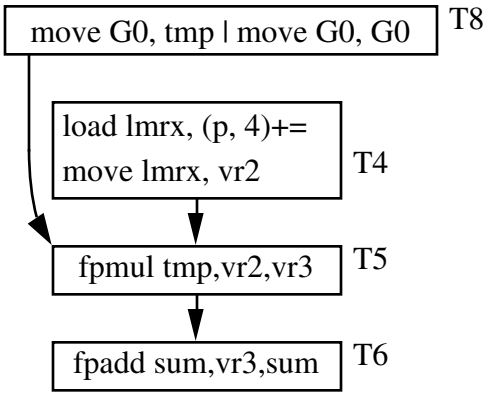
To avoid exhaustive case analysis, our code selector generates moves that buffer implicit



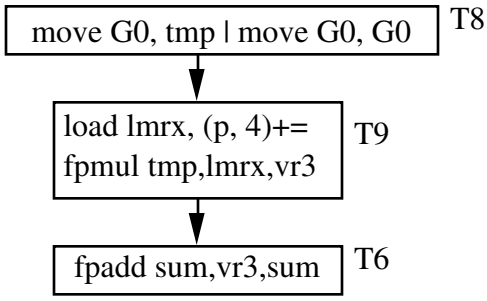

(a) Canonical execution order



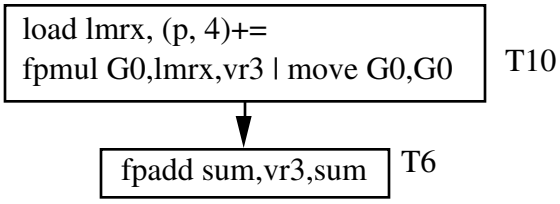

(b) Grouping T1 & T2 into T7




(c) Grouping T3 & T7 into T8




(d) Grouping T4 & T5 into T9




(e) Grouping T8 & T9 into T10

Figure 3: Pre-scheduling

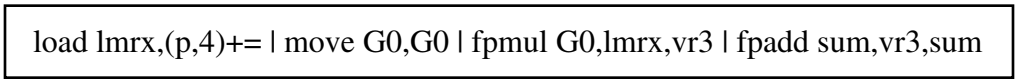



Figure 4: Steady state of software pipeline schedule

register and gate values. This results in a simple code selection phase that produces correct code but with additional move operations for every load, store or gate access. These move operations are removed by a separate *pre-scheduling* pass before code scheduling.

# 8 Pre-scheduling

The additional move operations required for buffering implicit register and gate values add latency, increasing critical execution paths. Furthermore, the additional moves utilize extra resources and thus impact scheduling. Pre-scheduling attempts to eliminate these move operations by *grouping* all consumers of an implicit register's value in the instruction immediately following the producer, or grouping all consumers of a gate's value into one instruction. Thus certain code templates are grouped into pre-scheduled sequences of operations that are scheduled as a unit by the scheduler. Note that this copy eliminating optimization is different from coalescing [4]. Coalescing is performed *after* scheduling as part of register assignment. Pre-scheduling eliminates extraneous copies *prior* to scheduling. Hence, pre-scheduling shortens the critical path and decreases the resource requirements of the operations to be scheduled, allowing the code scheduler to produce a better schedule.

Pre-scheduling works by grouping code templates together to produce templates that are maximal sequences of machine operations. First, the code templates in a basic block are ordered in *canonical execution order*, which represents a legal execution order of the templates. An execution order is legal if a template $T_i$ does not have an intra-iteration dependence on a template $T_j$ that appears after $T_i$ in the execution order. The templates are then visited in canonical execution order and checked against previous templates to find opportunities for grouping. There are two ways of grouping code templates:

1. Let $m$ be a move operation that copies the value of a register $R_s$ into a register $R_d$, and let $u$ be a subsequent use of $R_d$ appearing after $m$ in the same basic block. $u$ can be scheduled in parallel with $m$, and $u$'s use of $R_d$ can be changed to $R_s$. If $R_d$ becomes dead after this transformation, $m$ can be eliminated.

2. Let $m$ be a move operation that copies the value of a register $R_s$ into a register $R_d$, and let $d$ be the definition of $R_s$ that reaches $m$ in the same basic block. If $m$ is the only use of $R_s$, then $m$ can be eliminated and the target register of $d$ can be changed to $R_d$.

Grouping of code templates is subject to resource and dependence constraints. The former ensures that machine resources are not oversubscribed, while the latter ensures that the ordering remains a legal execution order, thus enforcing data dependences (such as anti or output dependences) and preventing cycles in the dependence graph, which make scheduling impossible.

Figure 3 shows the sequence in which pre-scheduling groups the operations from the example in Figure 1. Figure 3(a) show the templates ordered in canonical execution order, with intra-iteration flow dependence arcs. Template T2 uses register vr1, whose value is produced by a move from gate G0 in T1. Therefore, T2 is grouped with T1, and the use of vr1 is changed to G0. This transformation produces template T7 (Figure 3(b)). T3 also uses

the value in vr1 and can be grouped with T7 in a similar manner. However, when T3's use of vr1 is changed to G0, vr1 becomes dead and the first move operation is deleted, resulting in T8 (Figure 3(c). The next template that is a candidate for grouping is T5. This template uses values in registers tmp and vr2, which are both targets of previous move operations. This template is grouped with T4 creating T9, eliminating the move operation of T4 (Figure 3(d)). Finally, T9 is grouped with T8 to form T10, and the move operation that targets tmp is eliminated (assuming variable `tmp` is dead after execution of this loop). The final set of code templates (Figure 3(e)) make efficient use of gate and memory values: all uses of gate G0's value are pre-scheduled in one instruction, and the value produced by the load instruction is immediately used in the instruction following the load. Therefore, no superfluous move operations are necessary to buffer gate or implicit register values, and the instructions can now be scheduled into a software pipeline with a steady state of one instruction (Figure 4). Note that if any additional operations existed, or if move or load operations were prematurely bound to functional unit resources, then resource constraints would prohibit a single instruction steady state. Also, the loop count and branch operations are performed using the loop closing operation.

# 9 Concluding remarks

Resource constraints can significantly impact the quality of a software pipelined schedule. Therefore, the instructions to be software pipelined must make the most efficient use of machine resources. In practice, the target machine model is non-ideal. This makes it difficult to accurately model the machine resources. Furthermore code selection is complicated, and the compiler writer is faced with yet another phase ordering problem. New approaches and models are required that can hide non-ideal machine features without sacrificing accuracy and efficiency. In this paper we have discussed how several features in an LIW architecture cause such problems in the compiler and have shown how to deal with these problems by using abstractions that hide non-ideal machine features.

*Generic operations* capture the common functionality provided by a group of functional units. By extending the resource model of the target machine to include the concept of generic resources, we can delay the decision about which unit to use until after scheduling. At the same time, no resource is wasted, since the scheduler is able to place operations into instructions until all resources are used. *Pre-scheduling* groups operation sequences so that copy operations are eliminated before scheduling (i.e., software pipelining). There are many sources of copy operations: to buffer volatile values, to implement source-level assignments, or to copy temporaries. Prior approaches to eliminating copy operations operated after code scheduling. Eliminating copy operations via pre-scheduling reduces the resource requirements of code sequences and potentially shortens the critical path, allowing the software pipeliner to produce a better schedule.

# References


[1] A. V. Aho, R. Sethi, and Ullman J. D. *Compilers*. Addison-Wesley, 1986.

[2] S. Borkar, R. Cohn, G. Cox, S. Gleason, T. Gross, H. T. Kung, M. Lam, B. Moore, C. Peterson, J. Pieper, L. Rankin, P. S. Tseng, J. Sutton, J. Urbanski, and J. Webb. iWarp: An integrated solution to high-speed parallel computing. In *Proceedings of*

*Supercomputing '88*, pages 330–339, Orlando, Florida, November 1988. IEEE Computer Society and ACM SIGARCH.

[3] S. Borkar, R. Cohn, G. Cox, T. Gross, H. T. Kung, M. Lam, M. Levine, B. Moore, W. Moore, C. Peterson, J. Susman, J. Sutton, J. Urbanski, and J. Webb. Supporting systolic and memory communication in iWarp. In *Proc. 17th Intl. Symposium on Computer Architecture*, pages 70–81. ACM, May 1990.

[4] G. J. Chaitin. Register allocation and spilling via graph coloring. In *Proc. of the SIGPLAN 1982 Symposium on Compiler Construction*, pages 98–105, June 1982. In SIGPLAN Notices, v. 17, n. 6.

[5] R. Cohn, T. Gross, M. Lam, and P. S. Tseng. Architecture and compiler tradeoffs for a wide instruction word microprocessor. In *Proc. Third. Intl. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS III)*, pages 2–14, Boston, Apr 1989. ACM/IEEE.

[6] Intel Corp. *iWarp Microprocessor (Part Number 318153)*. Hillsboro, OR., 1991. Technical Information, Order Number 281006.

[7] J. D. Dehnert, P. Y. Hsu, and J. P. Bratt. Overlapped loop support in the Cydra 5. In *Proc. Third. Intl. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS III)*, pages 26–38, Boston, Apr 1989. ACM/IEEE.

[8] S. C. Johnson. *A Tour Through the Portable C Compiler*. Bell Laboratories, January 1981. in Documents for UNIX, Vol. 2.

[9] M. Lam. Software pipelining: An effective scheduling technique for VLIW machines. In *Proc. of the ACM SIGPLAN '88 Conf. on Programming Language Design and Implementation*, pages 318–328, June 1988.

[10] Roy F. Touzeau. A Fortran compiler for the FPS-164 scientific computer. In *ACM SIGPLAN 84 Symp on Compiler Construction*, pages 48–57. ACM, June 1984.

[11] M.J. Wolfe. *Optimizing Supercompilers for Supercomputers*. PhD thesis, University of Illinois at Urbana-Champaign, October 1982.