Achieving High Levels of Instruction-Level Parallelism with Reduced Hardware Complexity

Michael S. Schlansker, B. Ramakrishna Rau, Scott Mahlke, Vinod Kathail,

Richard Johnson, Sadun Anik, Santosh G. Abraham

Computer Research Center

HPL-96-120

November 1994

{schlansk, rau, mahlke, kathail, rjohnson, anik, abraham}@hplabs.hp.com

instruction-level parallelism,
VLIW processors,
superscalar processors,
overlapped execution,
out-of-order execution,
speculative execution,
branch prediction,
instruction scheduling,
compile-time speculation,
predicated execution,
data speculation,
HPL PlayDoh

Instruction-level parallel processing (ILP) has established itself as the only viable approach for achieving the goal of providing continuously increasing performance without having to fundamentally re-write the application. ILP processors differ in their strategies for deciding exactly when, and on which functional unit, an operation should be executed. The alternatives lie somewhere on a spectrum depending on the extent to which these decisions are made by the compiler rather than by the hardware and on the manner in which information regarding parallelism is communicated by the compiler to the hardware via the program. HPL PlayDoh is a research architecture that has been defined to support research in ILP, with a bias towards VLIW processing. The overall objective of this research effort is to develop a suite of architectural features and compiler techniques that will enable a second-generation of VLIW processors to achieve high levels of ILP, across both scientific and non-scientific computations, but with hardware that is simple compared to out-of-order superscalar processors. The basic approach is to provide the program (compiler) more control over capabilities that, in superscalar processors, are typically microarchitectural (i.e., controlled by the hardware) by raising them to the architectural level.

1 Introduction

Over the past two and a half decades, the computer industry has grown accustomed to, and has come to take for granted, the spectacular rate of increase of microprocessor performance, all of this without requiring a fundamental rewriting of the program in a parrallel form, using a different algorithm or language, and often without even recompiling the program. The benefits of this have been enormous. Computer users have been able to avail of faster and faster computers while still having access to applications representing billions of dollars worth of investment—an unlikely, if not impossible, occurrence were it not for the fact that the software does not have to be continually re-written to take advantage of the faster, more recent computers. Continuing this trend, of ever-increasing levels of performance without re-writing the applications, is the problem statement that represents the subject of this paper.

Higher levels of performance benefit from improvements in semiconductor technology which permit shorter gate delays and higher levels of integration, both of which enable the construction of faster computer systems. Further speedups must come, primarily, from the use of some form of parallelism. The best known style of parallel processing consists of partitioning an application into multiple sub-tasks which are executed in parallel on multiple processors. A certain amount of speedup can be achieved on loop-intensive applications by having a parallelizing compiler perform this partitioning. However, in general, the application must be manually restructured, typically using a different algorithm, and sometimes requiring re-expression in a parallel language, before enough parallelism is exposed, first, to outweigh the communication and synchronization overhead that is inherent to parallel processing and, second, to provide enough additional speedup to make it all worthwhile.

Another strategy known as **instruction-level parallel (ILP) processing** is a set of processor and compiler design techniques that speed up execution by causing individual machine operations, such as memory loads and stores, integer additions and floating point multiplications, to execute in parallel. The operations involved are normal RISC-style operations, and the system is handed a single program written with a sequential processor in mind. Thus an important feature of these techniques is that like circuit speed improvements, but unlike traditional multiprocessor parallelism and massively parallel processing, they are largely transparent to users. In the long run, it is clear that the multiprocessor style of parallel processing will be an important technology for the main stream computer industry. For the present, instruction-level parallel processing has established itself as the only viable approach for achieving the goal of providing continuously increasing performance without having to fundamentally re-write the application. It is worth noting that these two styles of parallel processing are not mutually exclusive; the most effective multiprocessor systems will probably be built using the most effective ILP processors.

If ILP is to be achieved, the compiler and the runtime hardware must, between them, perform the following functions:

- the dependences between operations must be determined,
- the operations, that are independent of any operation that has not as yet completed, must be determined, and
- these independent operations must be scheduled to execute at some particular time, on some specific functional unit, and must be assigned a register into which the result may be deposited.

ILP processors differ in their strategies for deciding exactly when, and on which functional unit, an operation should be executed. The alternatives lie somewhere on a spectrum depending on the extent to which these decisions are made by the compiler rather than by the hardware and on the manner in which information regarding parallelism is communicated by the compiler to the hardware via the program.

A computer architecture is a contract between the class of programs that are written for the architecture and the set of processor implementations of that architecture. Usually this contract is concerned with the instruction format and the interpretation of the bits that constitute an instruction, but in the case of ILP architectures it extends to information embedded in the program pertaining to the available parallelism between the instructions or operations in the program. The two most important types of ILP processors differ in this respect.

- **Superscalar** processors [33] are ILP processor implementations for sequential architectures—architectures for which the program is not expected to convey and, in fact, cannot convey any explicit information regarding parallelism. Since the program contains no explicit information regarding the dependences between the instructions, if instruction-level parallelism is to be employed, the dependences that exist must be determined by the hardware, which must then make the scheduling decisions as well.

- **Very Long Instruction Word** (**VLIW**) processors [14, 53] are examples of architectures for which the program provides information as to which operations are independent of one another. The compiler identifies the parallelism in the program and communicates it to the hardware by specifying which operations are independent of one another. This information is of direct value to the hardware, since it knows with no further checking which operations it can execute in the same cycle. Since, for any given operation, the number of independent operations is far greater than the number of dependent ones, it is impractical to specify all independences. Instead, for each operation, independences with only a subset of all independent operations (those operations that the compiler thinks are the best candidates to execute concurrently) are specified. These are packaged together as a single VLIW instruction.

In the context of this taxonomy, vector processors [68, 69, 28, 55] are best thought of as processors for a sequential, CISC (complex instruction set computer) architecture. The complex instructions are the vector instructions which do possess a stylized form of instruction-level parallelism internal to each vector instruction. However, attempting to execute multiple instructions in parallel, whether scalar or vector, incurs all of the same problems that are faced by a superscalar processor. Because of their stylized approach to parallelism, vector processors are less general in their ability to exploit all forms of instruction-level parallelism. Vector instructions are only relevant to the innermost loops of a program. Furthermore, these loops cannot contain non-trivial recurrences or complex control flow. Be that as it may, vector processors have enjoyed great commercial success over the past decade.

1.1 Superscalar processors

1.1.1 Overlapped execution and out-of-order execution

In a fully sequential processor, each instruction is issued after the previous one has completed. Not only does this fail to achieve the issuance of multiple instructions per cycle, but it even falls short of achieving an issue rate of a single instruction per cycle, except in the unlikely circumstance that every instruction completes execution in a single cycle.

The first step in increasing the issue rate is to attempt to issue an instruction every cycle, even if prior instructions have not completed. The related techniques of pipelining and overlapped execution, which have been employed since the late 1950s, serve this purpose [20, 7, 9]. Traditionally, overlapped execution refers to the parallelism that results from multiple active instructions, each in a different one of the phases of instruction fetch, decode, operand fetch, and execute whereas pipelining is used in the context of functional units, such as multipliers and floating-point adders, which are able to start subsequent operations before earlier ones have completed execution [12, 37]. (A potential source of confusion is that, in the context of RISC processors, overlapped execution and pipelining, especially when the integer ALU is pipelined, have been referred to as pipelining and superpipelining, respectively [35].)

If the semantics of the program are to be preserved, instruction issue must pause if the instruction that is about to be issued is dependent upon a previous instruction that has not as yet completed execution. To accomplish this, with each instruction that the processor issues, it must check whether the instruction's operands (registers or memory locations that the instruction uses or modifies) coincide with the operands of any other instruction that has been issued but has not yet completed.

If this is the case, instruction issue must be delayed until the instructions, upon which this instruction is dependent, have completed execution. These dependences must be monitored to determine the point in time at which this situation disappears. When that happens, the instruction is independent of all other uncompleted instructions and can be allowed to begin executing at any time thereafter. Also, once an instruction is independent of all other instructions in flight, the hardware must decide exactly when and on which available functional unit to execute the instruction. The Control Data CDC 6600 employed a mechanism, called the scoreboard, to perform these functions [63].

The next step towards achieving the goal of an instruction per cycle is out-of-order execution. Instead of stalling instruction issue as soon as an instruction is encountered that is dependent upon one that is in flight, the dependent instruction is set aside to await the completion of the instructions upon which it is dependent. Once these instructions have completed, the waiting one can begin execution. In the meantime, the processor may issue and begin execution of subsequent instructions which prove to be independent of all sequentially preceeding instructions in flight.

Issued instructions that are waiting on the completion of prior instructions do so because they are flow or output dependent upon those prior instructions. Although there is no option but to wait on the completion of instructions upon which the waiting one is flow dependent, waiting is unnecessary on encountering an output dependence if register renaming is performed. The Tomasulo algorithm [66] is the classical scheme for register renaming and out-of-order execution and has served as the model for subsequent variations [70, 31, 32, 61].

A further step towards the goal of an instruction per cycle, and another instance of out-of-order execution, is to perform memory references out-of-order. Utilizing techniques that are analogous to the scoreboard and Tomasulo's algorithm, loads and stores can be submitted to memory out-of-order once their addresses are known and are found to be distinct from the addresses of any preceeding, unsubmitted stores [1, 33].

1.1.2 Speculative execution

Conditional branches pose a major obstacle to ILP. Even if the above measures have been successful in attaining the rate of an instruction per cycle, the branch, when it is encountered can lead to many lost cycles during which no instruction is issued because it is not clear which path to take after the branch. Two penalties are incurred at this point. First, at the time that the branch is decoded, due to out-of-order execution, the instruction that computes the branch condition might be waiting on the completion of a long chain of flow dependence predecessors. Until the branch condition is disambiguated, no further instruction can be issued. (One could view this as the penalty that was deferred by employing out-of-order execution; many of the issue cycles that we avoided wasting, by using out-of-order execution, are wasted, in a concentrated fashion, at the time of the branch.) Second, even if the branch condition were known, but if it specified that the branch was to be taken, the instructions along this path would have to be fetched, during which time no instructions could be issued. As a result of these two penalties, the objective of an instruction per cycle is seriously compromised.

Three mechanisms must be invoked to overcome the conditional branch problem. First, the processor must make a guess as to which way the flow of control will go once the branch condition is known. Second, instructions along the predicted path must start being prefetched, i.e., fetched speculatively, and early enough so that they are available to be issued right after the branch is issued. Third, the processor must issue and execute these instructions speculatively (i.e., before the branch condition has been resolved) so that, at least in the case when the branch prediction is correct, the goal of an instruction per cycle continues to be achieved.

Of course, if the branch prediction turns out to be incorrect, all of these instructions that were prefetched, issued and executed speculatively, are wasted, and the complete branch penalty is experienced. Furthermore, an additional time penalty must be incurred to restore the processor state before going down the correct path. In view of this, it is important that an extremely accurate branch prediction scheme be used to guide the prefetch and speculative execution and, in response, various static [29, 30, 41] and dynamic [59, 39, 45, 71] schemes of varying levels of sophistication and practicality have been suggested. Dynamic schemes gather execution statistics of one form or another while the program is running and, consequently, are specific to that one program while executing a particular input data set. Static schemes accumulate statistics for a given program over many training runs using a set of hopefully representative input data sets. These statistics are used whenever that program is executed, regardless of the current input data set. As might be expected, dynamic schemes tend to be more accurate, but require relatively expensive hardware. However, static schemes appear to be capable of approaching the accuracy of dynamic schemes [72, 27], but at the expense of code size expansion.

Even if a branch is correctly predicted to be taken, a penalty is incurred unless the fetching of the target path for the branch is started in the cycle following the initiation of the instruction fetch of the conditional branch, i.e., well before the branch instruction has been fetched and decoded. The technique employed is to use a **branch target buffer** (**BTB**) [39] which is a cache that stores branch target addresses which are paired with an associative tag which is the address of the corresponding conditional branch. As each instruction address is issued to the instruction cache to be fetched, it is also presented to the BTB. A matching entry indicates that the instruction, whose fetch is currently being initiated, is a conditional branch which has recently been taken, and the associated branch target address is used to start fetching the target path instead of continuing fetching the fall-through path.

Speculative execution, too, can be viewed as a special form of out-of-order execution in which the instructions following a conditional branch are allowed to execute before the branch has completed. Without it, the benefits of out-of-order execution are restricted to the basic block boundaries and, in light of the generally small size of basic blocks, the pipeline latencies of floating-point and load operations, and the inter-operation dependences, little instruction-level parallelism will be found, typically. It is important, therefore, that operations from multiple basic blocks be executed concurrently if an ILP machine is to be fully utilized. Dynamic schemes for speculative execution must provide ways to:

- terminate unnecessary speculative computation once the branch has been resolved,
- undo the effects of the speculatively executed operations which should not have been executed, or prevent these effects in the first place,

- ensure that no exceptions are reported until it is known that the excepting operation should, in fact, have been executed, and
- preserve enough execution state at each speculative branch point to enable execution to resume down the correct path in the event it turns out that speculative execution proceeded down the wrong path.

Of the various schemes that have been proposed to meet the above requirements, the register update unit [61] stands out as one that is, at the very least conceptually, the simplest and most elegant.

1.1.3 Superscalar execution

The final mechanism to increase the number of instructions executed per cycle is termed **superscalar execution**. The goal of a superscalar processor is to issue multiple, independent instructions in parallel even though the hardware is handed a sequential program. One of the most problematic aspects of so doing is that of determining the dependences between the instructions that one wishes to issue simultaneously. Specifically, it is correct to issue an instruction only if it is independent of all the other instructions that are being issued concurrently but which would have been executed earlier in a sequential execution of the program.

The second challenge for superscalar processors is parallel resource allocation for the independent instructions that are to be issued in parallel. Although the instructions have been proven to be independent, they might have conflicting resource needs, e.g., there may be more integer instructions that can be issued in parallel than there are integer ALU's upon which to execute them. Even if there are sufficient resources for all of the instructions, care must be taken to ensure that multiple instructions do not attempt to use the same resource. When there are multiple resources of the type needed by a particular instruction, the specific resource that gets allocated to it depends on whether the sequentially preceding, but concurrently issued, instructions need a resource of that same kind and, if so, which ones get allocated to them.

1.1.4 The hardware complexity of out-of-order, superscalar processors

Superscalar execution, especially at high levels of ILP, poses the processor designer with some difficult challenges. Since the semantics of the program and, in particular, the essential dependences are specified by the sequential ordering of the instructions, the source and destination registers for each instruction that is a candidate for issue must be compared with those for all of the sequentially preceding instructions which also are candidates for issue. This is required so that one can determine, in parallel, which of those instructions may, in fact, be issued without violating any dependences. If one attempts to issue N dyadic instructions per cycle, one needs 5N(N-1)/2 comparators to check for all possible flow, output and anti-dependences. Parallel resource allocation can be even more expensive, since the resource allocated to each instruction depends on which of the sequentially preceding, but concurrently issued, instructions need a resource of that same kind and, of those that do, which specific resources get allocated to them. Here, we are faced with an inherently sequential decision process which we wish to perform in parallel. The trade-off is between a potentially lengthened cycle time and increased logic complexity.

Even without superscalar execution, a significant amount of complexity is implicit in a processor performing out-of-order execution. As we saw earlier, successful out-of-order execution requires support for register renaming and dynamic branch prediction, a branch target buffer, and a register update unit to support recovery from speculation past a mis-predicted branch. All these mechanisms have significant complexity. An excellent reference on superscalar processor design and its complexity is the book by Johnson [33].

1.2 First-generation VLIW

VLIW processors evolved in an attempt to achieve high levels of ILP without the aforementioned hardware complexity. When discussing VLIW processors, it is important to distinguish between an instruction and an operation. An **operation** is a unit of computation, such as an addition, memory load or branch, which would be referred to as an instruction in the context of a sequential architecture. A (parallel) VLIW **instruction** is the set of operations that are intended to be issued simultaneously. The archetypal VLIW is distinguished by two features: **MultiOp** instructions which each specify the concurrent issue of multiple operations, and operations with architecturally visible execution latencies, which we refer to as **non-unit assumed latencies** (**NUAL**).

1.2.1 Compile-time scheduling

With a VLIW processor, most of the measures that were taken by the superscalar processor at run-time to achieve ILP, are performed by the compiler. Conceptually, the compiler emulates what a superscalar processor, with the same execution hardware, would do at run-time. The compiler takes the sequential internal representation of the program, analyzes the dependences between the operations, performs register renaming, if needed, to eliminate anti- and ouput dependences, delays the scheduled initiation time of operations that are dependent upon others to a time when they will have completed, schedules the out-of-order execution of other operations that are independent, and performs the allocation of the requisite functional units, buses, and registers to the operations as specified by a machine description database. However, it is important to re-emphasize that all of this is being done at compile-time and requires no hardware support. Consequently, it is not only realistic to think of building VLIW processors which have very high levels of ILP, but such processors have, in fact, been built successfully [11, 14, 38, 48, 6].

It is the task of the compiler to decide which operations should go into each instruction. Once a program has been scheduled, all operations that are supposed to begin at the same time are packaged into a single VLIW instruction. Thus, the hardware need make no decisions as to which operation to issue concurrently. The position of each operation within the instruction specifies the functional unit on which that operation is to execute and, so, the hardware need make no decisions regarding resource allocation. A VLIW program is a direct transliteration of a desired record of execution, one that is feasible in the context of the given execution hardware.

1.2.2 Compile-time speculation

As noted, run-time speculation is expensive in the hardware needed to support it. The alternative is to perform speculative code motion at compile-time. The compiler for a VLIW machine specifies that an operation be executed speculatively by performing speculative code motion, that is, scheduling an operation before the branch that determines that it should, in fact, be executed. At run-time, the VLIW processor executes these speculative operations in the exact order specified by the program just as it does for non-speculative operations. These operations will end up being executed before the branches that they originally were supposed to follow; hence, they are executed speculatively in relation to the original sequential code that the scheduler received. Such code motion is fundamental to global scheduling schemes such as trace scheduling [23] and superblock scheduling [30].

When the compiler decides to schedule operations for speculative execution, it ensures that they do not overwrite any of the state of the computation that is needed to assure correct results if it turns out that that operation ought not to have been executed. This is achieved by writing the results of the speculative operations into different destination registers, i.e., performing register renaming at compile-time.

The hardware support needed is much less demanding. First, a mechanism is needed to ensure that exceptions caused by speculatively scheduled operations are reported if and only if the flow of control is such that they would have been executed in the non-speculative version of the code and, second, additional architecturally visible registers are needed to hold the speculative execution state. The former issue was partly addressed in the first-generation of VLIW processors [14].

Just as a superscalar processor must predict which way branches will go to be able to perform speculative execution successsfully, so must a VLIW compiler predict the branch direction so that it can schedule operations speculatively from the more likely path following the branch. Furthermore, since the prediction is being performed at compile-time, dynamic branch prediction is not an option. Instead, profiling runs are used to gather the appropriate statistics and to embed the prediction, at compile-time, into the program. Branch statistics gathered using one set of data have been shown to be applicable to subsequent runs with different data [24]. Although static prediction can be useful for guiding both static and dynamic speculation, it is not apparent how dynamic prediction can assist static speculative code motion.

The VLIW compiler must perform many of the same functions that a superscalar processor performs at run-time to support speculative execution, but it does so at compile-time and, consequently, incurs less hardware penalty.

1.2.3 Shortcomings of first-generation VLIWs

Most of the VLIW features discussed above were incorporated into the first-generation of VLIW processors [14, 6], which were technically successful and extremely cost-effective for the workloads for which they were designed, viz. scientific computing. However, these products were perceived as having certain serious shortcomings in the context of branch-intensive applications, the so-called "general-purpose" codes. First, the fact that all scheduling and allocation decisions are made by the compiler and not by hardware means that a program is good only for the specific machine for which is was compiled. If the number of functional units or the execution latencies change from one machine to the next, the code can be just plain incorrect. This has led to the conventional wisdom that VLIW processors cannot exhibit object code compatibility across a family of machines. Second, even in the context of a single processor, the presence of non-deterministic latencies, as is the case with loads that can experience either a hit or a miss latency, are problematic for a compiler that is attempting to make scheduling decisions when it is unclear, at compile-time, what the latency will be at run-time. Lastly, as a consequence of the fact that first-generation VLIW processors were designed for the scientific computing market, the somewhat unjustified conclusion has been reached that VLIW processors cannot be successful at "general-purpose" computing.

1.3. HPL PlayDoh

HPL PlayDoh [36] is a research architecture that has been defined to support research in ILP, with a bias towards VLIW processing. The overall objective of this research effort is to develop a suite of architectural features and compiler techniques that will enable a second-generation of VLIW processors to achieve high levels of ILP across both scientific computations as well as non-scientific computations, i.e., true general-purpose capability. The central thrusts of this effort are:

- to support high levels of ILP, i.e., the ability to issue over eight useful operations per cycle;
- to support “scalar computations” as well as numeric computations, i.e., an emphasis on architectural features that help reduce the critical path through computations that have a high frequency of conditional branches and pointer-based memory accesses;
- to retain hardware simplicity and short cycle times even at high levels of ILP, which translates into a bias away from schemes that need hardware to make complex decisions at run-time;

- specifically, to provide architectural features that eliminate the need for (dynamic) multiple instruction issue and out-of-order execution;
- to provide the program (compiler) more control over capabilities that are typically microarchitectural (i.e., controlled by the hardware) by raising them to the architectural level;
- to focus on containing the increase in code size due to the presence of no-ops within VLIW instructions and as a result of the tendency of ILP compilers to replicate code.

Table 1.1. Play Doh features and the dynamic capabilities they are intended to replace.

PlayDoh Feature	**Dynamic Capability Replaced**
MultiOp instructions	Multiple instruction issue / dynamic parallel dependence analysis
Static resource allocation / explicit specification of resource allocation	Dynamic parallel resource allocation
Static scheduling	Out-of-order execution
Static register renaming / rotating registers	Dynamic register renaming
Static branch prediction Predicated execution	Dynamic branch prediction
Prepare-to-branch instruction	Branch target buffer
Speculative code motion / speculative opcodes / exception tags Predicated execution	Speculative execution
Static disambiguation / static scheduling Statistical disambiguation / data speculation	Out-of-order memory referencing

In the rest of this article, we review those key features of PlayDoh that advance the goals articulated above. PlayDoh is best viewed as a coherent collection of features for achieving high levels of ILP with reduced hardware complexity. It is not intended to represent our concept of the right second-generation VLIW architecture. PlayDoh's features are ones that we believe are valuable and that we think ought to be incorporated into the second generation of VLIW products. Still, many of them have not been thoroughly evaluated as yet and, as is always the case in research, we recognize that in the case of certain features, we might just turn out to be wrong!

PlayDoh's features are intended to offer a viable approach to achieving high levels of ILP for second-generation while minimizing the amount of dynamic analysis, allocation and out-of-order execution. The correspondence of PlayDoh's architectural features to the dynamic microarhitectural capabilities in a superscalar, out-of-order processor is presented in Table 1.1. Some of PlayDoh's features also have benefits that go beyond the replacement of the dynamic capabilities of superscalar processors. Many of these benefits could be availed of, fully or partially, by an in-order superscalar processor were it to extend its architecture to include PlayDoh's features.

The rest of this article is devoted to a detailed discussion of these features. Section 2 treats the two defining aspects of a VLIW processor: its MultiOp instruction format and its non-unit assumed latencies. Section 3 presents the important aspects of PlayDoh's architectural support for predicated execution and their diverse uses and benefits. Section 4 discusses those features that support the software pipelining (specifically, the modulo scheduling) of innermost loops. Section 5 presents the unusual features of PlayDoh's branch architecture. Section 6 discusses the architectural support needed to enable control speculation, i.e., the compile-time motion of code above conditional branches, in order to make run-time speculative execution superfluous. Section 7 treats data speculation, which is the compile-time movement of a load above a store, upon which it could be dependent, when it is known that, in fact, the two operations are rarely to the same memory location. Section 8 discusses those features that assist a static scheduler in coping with the non-deterministic latencies of loads in the presence of a cache, and which give the compiler some degree of control over the staging of data through the cache hierarchy.

2 Uniquely VLIW features

PlayDoh supports VLIW style parallelism by exposing architecturally visible parallelism to the compiler in two basic forms: PlayDoh issues multiple independent operations within a single VLIW instruction and PlayDoh allows non-unit assumed latencies. Since PlayDoh parallelism is architecturally exposed, the compiler needs an accurate semantic model of the processor to generate correct code. The semantic model must define allowed sets of independent operations which may be simultaneously issued within a single instruction and must also describe the latencies of operations which, if violated, produce incorrect code. Definitions for the execution behavior of VLIW instructions with non-unit assumed latencies are described in more detail below.

2.1 Multiple Operation Instruction Issue

VLIW processors exploit wide issue parallelism in a style very similar to horizontal microprogramming. PlayDoh exposes the parallel issue of multiple operations to compilers in a form called **MultiOp**. Multiple operations can be collected into a single MultiOp instruction when a compiler can show that no data dependence or resource usage conflict has been violated. All operations within a single MultiOp instruction can be executed simultaneously without checking for dependence or resource violations. MultiOp can provide four major benefits:

- it can simplify the alignment of operations from the instruction cache into the instruction buffer;
- it can simplify the distribution of operations from the instruction buffer to function units;
- it can simplify or eliminate the checking for dependences among simultaneous or overlapped operations ;
- it can simplify or even eliminate the run-time allocation of resources among simultaneous or overlapped operations.

2.1.1 Instruction alignment and dispersal

Consider Figure 2.1. which illustrates a MultiOp instruction pipeline. An instruction cache supplies MultiOp instructions to be fetched, decoded, and executed. After fetch, a MultiOp instruction must be held in the MultiOp instruction buffer for decode and execution. Instructions are usually right justified within the instruction buffer in order to simplify the subsequent process of transmitting operations to function units. With a conventional superscalar processor, instruction alignment is not guaranteed within the cache and instructions must be aligned prior to storage in the instruction buffer. In the simplest VLIW processors, instruction alignment is guaranteed within a cache which is exactly one VLIW instruction wide. In this case transmission to the instruction buffer is done without any shifting as the instruction emerges from the cache.

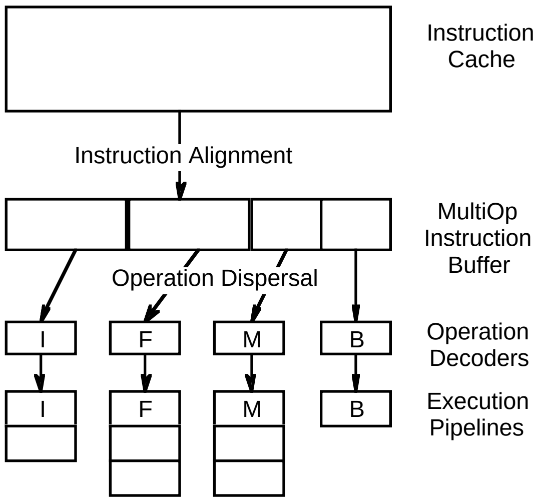


Figure 2.1. MultiOp Instruction Pipeline

The process of moving operations from their location in the instruction buffer to the location where operation decode and execute occurs is called "operation dispersal". In a fully general superscalar architecture which allows operations to lie in arbitrary positions within each instruction, operation dispersal can be very complex and requires expensive multiplexing. In the simplest VLIW processors, operations can flow straight through from the instruction buffer to execution units where final decode occurs. Due to the positional encoding used in such VLIW processors, MultiOp instruction decode can also easily tolerate operations that occupy different numbers of bits in the MultiOp instruction while requiring no alignment or shifting of operations. A one to one correspondence is established between each operation's position across the width of the instruction buffer and the functional unit where final decode and execute occurs.

As issue width is scaled up for parallel execution, instruction dispersal can become very complex. A superscalar processor typically executes a sequence of operations with no constraints on order. When heterogeneous function units are provided which are capable of executing only a subset of all operations, pre-decode and shifting is required to disperse operations to capable functional units. The cost of this shifting in terms of hardware complexity and time can be substantial. Even when all operations are of fixed width, a path may be required from multiple positions within the MultiOp instruction to each of the heterogeneous function units.

When the width of operations vary, dispersal is even more difficult. An operation to be issued must be identified by its origin in the instruction buffer. The operation is issued by routing it, from its origin, through a shifter to an appropriate function unit for further decode and execution. A variety of shift amounts are needed to tolerate an operation's origin after previously issuing a sequence of operations of varying width. The number of operation origins for each function unit is strongly influenced by the variety of allowed operation widths for preceding operations. This can lead to an explosive growth in number of shift paths and dispersal hardware complexity.

As a side benefit, the use of positional encoding in MultiOp instructions allows distinct operations to use the same opcode (bit encoding) if they are not in the same position within the MultiOp instruction. The superscalar model cannot exploit positional encoding because each operation can follow any previous operation eliminating positional significance.

2.1.2 Dependence semantics

Superscalar processors typically allow independent operations to be executed in an arbitrary order. Dependence checking is used to prevent the overlapped issue of dependent operations. The input to a dependence checker consists of issue requests from a set of unissued candidate operations. Dependence checking takes into account both source and destination operand references for all candidate operations as well as any constraints imposed by operations which are still executing. The dependence checker identifies independent operations which can issue within the current cycle. Operations are issued and the state of the dependence checker is updated to correctly reflect constraints imposed by unfinished operations in the next cycle. A new set of candidates for operation issue are identified and the process repeats.

Dependence checking hardware can require complex circuitry which is difficult to pipeline. Dependence checking is particularly complex in implementations which issue many operations requiring the analysis of a large number of operand references on every cycle. Further, for processors with substantial issue width and latency, the enforcement of constraints imposed by a large number of unfinished operations is quite complex. MultiOp instruction issue uses compile time checking of dependences to eliminate hardware dependence checks. The burden of dependence checking is shifted to the compiler and complex circuitry can be eliminated.

2.1.3 Problems with the MultiOp format

MultiOp format advantages are accompanied by a key disadvantage. The MultiOp encoding requires noops in the object program. When, for example, only floating point operations are issued on the simple VLIW processor in Figure 2.1, noops are issued on the integer, memory, and branch units. These noops waste space within the instruction cache and main memory. VLIW processors have been engineered to provide a compromise between the complex circuitry of full superscalar dispersal and the wasted space needed for the noops required by simple MultiOp. Both the Multiflow [14] and the Cydrome [53, 6] processors provided MultiOp instruction formats which more efficiently encode noops while preserving many of the benefits of MultiOp hardware simplicity.

Simple VLIW processors, directly expose their issue width and latencies to a compiler. While this can simplify the processor hardware, it also complicates the architectural specifications as seen by compilers. The parallel architecture is exposed to the compiler using a processor model which provides an abstract description of the processor. With concurrent issue, the architecture must specify the result of simultaneous actions such as simultaneous writes to the same register. The architecture may specify that simultaneous writes are:

- illegal (will abort the program),
- undefined (not guaranteeing any result), or
- legal (writes can be prioritized, guaranteeing a unique result).

The parallelism exposed by MultiOp instruction issue can allow programs to exploit explicit parallelism. The explicit use of parallelism can be illustrated using an exchange copy consisting of two operations r1=r2 and r2=r1 scheduled within the same MultiOp instruction. Using parallel semantics, a single MultiOp instruction can readily exchange two values using these two operations. The two operations must execute in parallel and cannot execute sequentially because each operation is anti-dependent on the other. This exchange would require at least three operations and at least three registers on any processor with a purely sequential execution model.

In many cases, compatibility across a family of processors must be supported by architectures which expose issue width and function unit latencies. Compatibility requires that each implementation run programs compiled for all family members. High performance requires that distinct codes are compiled for distinct processor implementations. The best performance is achieved when code is optimized and scheduled for an implementation using a processor model which describes key implementation parameters such as the number of function units and their latencies. Each implementation must support foreign codes generated using processor models distinct from the native highest performance processor model.

2.1.4 Compatibility with respect to the number of functional units

MultiOp processors of varying width can provide compatibility across a family of implementations. In one scheme, a narrow processor interprets a MultiOp instruction for a wider processor semi-sequentially from left to right, issuing a portion of the MultiOp instruction each cycle. However, semi-sequential interpretation does not always produce correct results. Exchange copy provides an example where concurrent operations are mutually anti-dependent (each has an anti-dependence to the other). Non-concurrent execution of the exchange copy violates anti-dependence and produces incorrect results.

Semi-sequential execution correctly simulates simultaneous execution if results are buffered until all operations in the same instruction have sampled their inputs. However, the additional buffering and multiplexing within processor data paths can be costly in cycle time or latency. MultiOp architectures may exclude bi-directional dependences in order to simplify compatibility. The compiler ensures that allowed dependences (e.g. anti-dependences) within a MultiOp instruction are from left to right. MultiOp instructions can now be interpreted in parallel, but they can also be readily interpreted semi-sequentially or sequentially from left to right without required buffering.

2.2 Non-unit assumed latencies

VLIW processors expose architecturally visible latencies via execution semantics that we call **non-unit assumed latencies** (**NUAL**). In this section, we define NUAL and discuss the advantages and disadvantages of NUAL over the more common sequential execution model. Techniques for handling exceptions, supporting non-deterministic variations in latency and compatibility across families of machines are also discussed in the context of NUAL.

2.2.1 NUAL semantics

Explicit parallelism in both the MultiOp and NUAL forms are expressed in terms of virtual schedule time. The virtual schedule time defines operation timing assumptions which the compiler can use to generate correct code. Virtual schedule time assumes that one MultiOp instruction issues every virtual cycle. On VLIW processors, most virtual cycles complete in a single real time cycle. Virtual schedule time is not, however, identical to actual schedule time. Unusual events such as I/O interrupts, page faults, and cache faults may cause discrepancies between virtual and actual time. These events are not precisely determined at the time the compiler schedules the code and thus force a departure from the compiled virtual schedule.

RISC and superscalar processors usually provide a sequential program interface which does not expose visible latency. In the program's virtual schedule time, measured in units of instructions, the assumed latency of every operation (or, in this case, instruction) is one. This permits each instruction to use the result of a previous instruction as one of its input operands. If, in reality, the previous operation has a latency greater than one, and is directly followed by a dependent operation, the processor must ensure that the correct value is, nevertheless, carried from the first operation to the second operation. Register interlocks, can resolve any apparent latency violation by stalling the execution of dependent operations [33]. Such execution semantics are referred to as **unit assumed latencies** (**UAL**).

VLIW processors expose architecturally visible latencies via execution semantics that we call **non-unit assumed latencies** (**NUAL**). With, NUAL visible latencies are no longer constrained to be one. NUAL allows the dependence checking responsibility to be eliminated from hardware and instead performed by a compiler which ensures that assumed latencies are satisfied. As we will discuss later, multiple definitions for NUAL semantics differ in the strictness with which the assumed latencies are enforced.

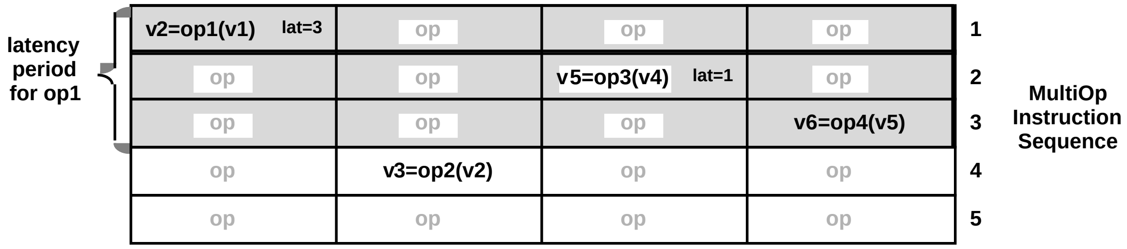


Figure 2.2. Virtual schedule

Consider the schedule for a MultiOp instruction sequence shown in Figure 2.2. In this example, operation op2 depends on the result of operation op1 and must be issued within an instruction which issues after the expiration of the op1 latency. The op1 latency starts with the issue of op1 and expires three cycles later. If this latency is not honored, then NUAL semantics specify that the value of v2 which is input to op2 is not the one computed by op1 but, rather, the previous value of v2.

We call the strictest form of a NUAL operation an "**equals**" (**EQ**) **operation**. The EQ operation samples input operands and stores result operands at precise virtual schedule times. For example, an EQ operation can be defined to sample input operands precisely at issue time and deliver results precisely at latency. The EQ model can also support input operand sampling at times later than issue, as well as differing latencies for each input and output operand. This might occur in a multiply-add operation of the form (a*b)+c which completes the product of a and b before sampling c and performing a final sum. An EQ operation offers the highest degree of determinacy to the compiler and provides the greatest opportunity for the compiler to exploit NUAL.

NUAL can enhance the efficiency of register use and the efficiency of the program schedule by more precisely specifying the duration of values in registers. The example of Figure 2.2 is used to demonstrate the use of EQ operation semantics to reduce register requirements. Assume that operands v1...v6 name virtual registers which must be bound to physical registers for execution. Consider v2 and v5 which carry values from op1 to op2 and op3 to op4 respectively. If op1 were unit latency, the operand lifetime carried by v5 would overlap the operand lifetime carried by v2. A common physical register could not be used a to carry both values. If op1 has EQ semantics with latency 3, op1 will not deliver its result until the end of its latency period. This guarantees that op4 reads v5 prior to op1 writing v2. The v2 and v5 lifetimes do not conflict and can be allocated to a common physical register.

NUAL offers a key hardware advantages in reducing requirements for dependence check, but, this feature is not without cost. If programs never branched, then compiler generated static schedules and the actual execution history might precisely match. However, branches can disrupt static schedules by creating dynamic program behavior. The compiler must ensure that dependences are satisfied on all control flow paths leading to each operation. Consider an operation inside a basic block at a merge point in the program's control flow graph. The operation may be dependent upon operations within multiple preceding basic blocks. If preceding blocks are already scheduled, the static scheduler must accommodate all dependences to the operation within the current block. After calculating the earliest schedule time for each dependence, the scheduler maximizes the value over all dependences to find the earliest schedule time for the operation. As a result, this might be a sub-optimal schedule for all but one path through the current block.

This problem is especially difficult at program locations where the static scheduler has little knowledge of the program history. Consider, for example, an external procedure compiled without inter-procedural information. The basic block reached upon entry to the procedure may make conservative assumptions regarding operand dependences originating within the calling program without seeing the calling program schedule.

2.2.2 Handling exceptions with NUAL

A NUAL processor must support dynamic adjustments to the schedule as required by unexpected events. Page faults, cache faults, clock interrupts and other events require that a processor deviate from its virtual schedule time. The implementation must preserve the selected MultiOp NUAL semantic model in the face of any asynchronous events. Compatibility across a family of processors represents another factor in causing processors to deviate from the virtual schedule. In particular, to achieve higher performance, computer designers may wish to adjust not only to the clock cycle time but also to the issue width and operation latencies. These changes may disrupt the precise correspondence with the virtual schedule used when the code was previously compiled.

Consider a MultiOp NUAL processor which handles interrupts. Assume that when an interrupt is processed the virtual schedule is cut so that when the interrupt handler is entered, the nth instruction in the schedule has completed while the (n+1)th instruction has not yet begun. We call this process "cutting the schedule" which separates a top portion of the schedule from a bottom portion by inserting a large time delay at the cut. When an EQ operation spans the cut, NUAL semantics may be violated if that operation is allowed to go to completion and write its result into the destination register. From the viewpoint of other operations below the cut, this operation finishes early, at a virtual time corresponding to the cut, and EQ semantics are violated.

In Figure 2.2, assume that virtual registers vr2 and vr5 are both bound to the same physical register taking advantage of op1's EQ semantics. If an interrupt occurs between MultiOp instructions 1 and 2, it may force op1 to early completion. If op1 goes to completion early, op2 does not execute with correct virtual operand v2. Instead, op3 overwrites v5 (which is bound to the same register as v2) before v2 is used. The problem is solved by draining the state of the op1 functional unit pipeline into a buffer which we call the "snapshot buffer". The snapshot buffer holds the exact status of the function unit pipeline at the moment the interrupt occurred. This prevents operands from exiting the function unit pipeline and destroying operands within registers. If the state of the function unit pipeline is precisely restored after interrupts, then execution can resume without violating EQ semantics.

An ILP compiler must accurately model timing constraints which ensure that operations are correctly executed. For example, we might wish to describing the timing constraints between operations op1 and op2 after register allocation to support any required re-scheduling after allocation. Since virtual registers v2 and v5 are bound to a common physical register, we have an output dependence from the first operation which writes this register to the second operation which writes the same register. Operation op3 (with latency one) writes virtual register v5 at the end of cycle two. Operation op1 (with latency 3) writes to virtual register v2 at the end of cycle three. Here, we have an output dependence from op3 to op1.

Constraints on instruction issue are due to dependences caused by either the sampling or write back of operands (e.g. output dependences are due to operand write back from two operations). In the example above, we can say that op1 must write its result at least one cycle after op3. However, to provide a uniform view of scheduling constraints, we always describe execution constraints between two operations from their issue cycles. When the write back constraint is translated to an issue constraint, we say that a delay of -1 cycle is allowed from the issue time of op3 to the issue time of op1. That is, as long as op1 is issued one cycle before op3 or later, the writes to the shared register occur in an order consistent with the existing schedule. This illustrates that NUAL with EQ semantics can allow negative delays between the issue times of an operation and its dependent operations.

NUAL operations with EQ semantics are more difficult to support when the schedule is cut by an interrupt. This problem is addressed using "less than or equals" (LEQ) operation semantics. A NUAL operation with LEQ semantics is an operation whose latency is constrained to be between one and its assumed latency. Codes scheduled using LEQ operations can be readily cut without a snapshot buffer because schedule cutting yields effective latencies which lie within LEQ schedule constraints. However, from a compiler viewpoint, EQ operation semantics at a given latency is architecturally superior because it provides superior code from the standpoints of optimization, scheduling, and register allocation than LEQ semantics at the same latency. This is true because, EQ semantics falls within the constraints of a LEQ processor, and provides more determinism to the compiler.

However, the benefits of EQ semantics may not justify snapshot buffer hardware complexity.

Note that negative delays between the issue times of dependent operations cannot be supported with LEQ scheduling constraints. When a negative delay separates the issue of two operations, it asserts that the dependent operation may actually issue before the operation upon which it depends. However, LEQ semantics allows the first operation to complete before the second. Thus a dependent operation completes prior to the issue of an operation upon which it depends. From this contradiction, we can see that LEQ semantics will never result in negative delay constraints separating operation issue.

LEQ operation semantics eliminates the need for a function unit snapshot. However, the resulting loss in schedule determinism often unnecessarily penalizes architected performance. Consider the example of two identical operations op1 and op2 executing in sequence on a common function unit both targeting register r1. For example, we may wish to write and then conditionally overwrite r1 before a subsequent use of r1. If op2 issues within the latency period of op1 but after op1, then LEQ semantics requires an undefined result. LEQ semantics would allow op1 to take its full latency while op2 may execute within only a single cycle causing a possible final value of the result to be op1. However, this occurs only when operations were inserted into the function unit in issue order and they emerged from the pipeline in reversed order. If the unit is FIFO, this cannot occur and the final value of r1 is unambiguously defined as the result of op2.

This gives rise the definition of the "order preserving" relation which is defined over operations with LEQ NUAL semantics. Order preserving operations allow overlapped writes to a common register to produce clearly defined results. Operation A is order preserving with operation B if it can be guaranteed that when A is issued before B, then A completes before B. Typically, not all operations are order preserving. For example, consider the case where operation A has long latency and issues on function unit one while operation B has short latency and issues on function unit two; A is not order preserving with B unless complex hardware enforces order.

2.2.3 Compatibility with respect to latency

The simplest NUAL processors do not need hardware interlocks to ensure that dependences satisfy latencies. When assumed latencies and actual hardware latencies match, no interlocks are necessary. When hardware latencies exceed assumed latencies, interlocks can, be incorporated into NUAL processors to support compatibility. Each operation has an assumed latency during which dependence checking is unnecessary. After the assumed latency expires, the processor can use dependence checking to continue issuing independent operations while stalling any dependent operation in a superscalar like manner.

Latency stalling is a technique which can be used by NUAL architectures with LEQ operations semantics to provide compatibility across implementations of differing latency without using register interlocks. With latency stalling, if an operation is scheduled with virtual latency vl, and executed on a function unit with physical latency pl, then MAX(0,pl-vl) stall cycles are inserted during the latency period for the operation. This ensures that the result operand for the operation exits the function unit pipeline prior to any use of the result after the virtual latency period. The key advantage of latency stalling is that it provides compatibility using simplified hardware which does not examine all source and target operands over a wide collection of operations.

Latency stalling implementations need to guarantee correctness but should insert the fewest number of stalls. In particular, when a stall is required on behalf on multiple concurrent operations, stalls can often be shared. That is, a single stall cycle can guarantee the correctness of more than one operation. This is accomplished by delaying the insertion of stall cycles needed for each operation until the last possible moment. Other operations may cause intervening stalls which eliminate or partially eliminate the stall requirement for the given operation.

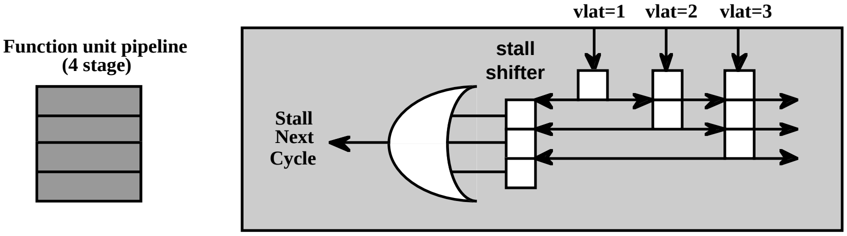


Figure 2.3. Latency stalling circuit

The latency stalling circuit for a four deep function unit pipeline shown in Figure 2.3 illustrates the hardware simplicity of implementing latency stalling. The circuit maintains proper state so as to stall as late as possible and only when necessary to guarantee correctness. Each square represents a 1-bit register with arrows describing shift paths among registers. Each cycle is either a normal cycle which issues a new instruction and advances both the virtual schedule time as well as physical time, or a stall cycle which does not issue a new instruction and advances only physical time. On a normal cycle, shifting is to the left if possible or down. On a stall cycle, shifting is to the right if possible or down. When a 1 enters the stall shifter, one or more stall cycles are required as determined by the result of the OR gate. If, for example, the stall shifter is entered at the top, three stall cycles ensue.

Stalls are forced when a 1-bit is shifted into the appropriate input and reaches the stall shifter. On each stall cycle or when no operation is scheduled on the unit, 0s are injected into all inputs. When an operation is scheduled, a 1 may be injected into the appropriate input while 0s are injected into all other inputs. The selected input depends on the operation's virtual latency. If an operation is scheduled with virtual latency four or larger, 0s are placed on all inputs. Here, the physical pipeline length is less than the virtual pipeline length and stalling can never be necessary.

An input is provided for each virtual latency (1,2, and 3) corresponding to (vlat=1, vlat=2, and vlat=3). Operations scheduled at each of these latencies may cause a stall. However, each stall cycle can alleviate pending stall cycles for other operations. Consider an operation which is scheduled at latency three with no intervening stalls, a 1 shifts to the bottom of the vlat=3 column and into the stall shifter at the bottom causing one stall cycle after three cycles . However, if an intervening stall occurs from another operation, then a right shift eliminates the pending stall.

2.2.4 Specification of the assumed latency

NUAL processors require some way to specify the assumed latencies. The simplest specification, used historically for VLIW processors, defines a unique static latency for each operation code. This approach works when there is no need to change assumed latencies. However, for architectures which provide compatibility across implementations with differing latency, the scheme must be enhanced to allow varying latency specifications on a per program basis. A simple enhancement defines fixed latencies for each implementation. Each object program identifies the implementation that it was compiled for and a table of latencies corresponding to each implementation is consulted to determine the actual latency for each operation.

It is sometimes useful to allow varying the assumed latencies within a single program for a single implementation. This is especially useful when the function unit pipeline has non-deterministic latency. For example, load operations on the critical path may use a short load latency to expedite the critical path while, loads which are off the critical path may use longer latencies to better overlap cache misses with further processing. A fully flexible scheme provides a field to independently specify the latency of each operation. However, latency fields contribute to increased code size. Another approach defines a latency register for each operation type. Latency registers are initialized prior to running a program and used by the hardware (e.g. latency stalling circuitry) to enforce assumed latencies. The contents within latency registers can be changed while the program is running, but the change affects all operations of the same type. The Cydra 5 used this mechanism to allow the assumed latency of load operations to be changed under program control [6].

3 Predicated Execution

3.1 Introduction

Predicated execution (also known as guarded execution) refers to the conditional execution of operations based on a boolean-valued source operand, called a predicate. If the predicate input is 1, the predicated operation is executed normally, just like an unpredicated operation. If the predicate input is 0, the predicated operation is squashed; i.e., it does not affect the architectural state in any way. In particular, the squashed operation does not modify any destination register or memory location, it does not signal any exceptions and, if it is a branch operation, it does not branch. Predicated execution is a method for enforcing the requirements of program control-flow, in a different way than that provided by branch operations. Predicated execution is often a more efficient method for controlling execution than branching, and it also provides much more freedom in code motion than would exist otherwise.

A primary use of predicated execution is to eliminate many of the conditional branches present in a program, using a technique commonly referred to as **if-conversion.** During if-conversion, each operation within a region of interest is guarded by a predicate computed so as to be true if and only if flow of control would have reached that operation [4, 47, 40, 18]. Once this is done, the branch operations in that region become redundant and can be removed without altering the behavior of the program. The predicate used to guard operations in a basic block is computed by generating code which evaluates a boolean expression equivalent to the branch conditions within the region that originally determine execution of that block. If-conversion not only reduces the impact of branch latencies, it has the benefit that guarded operations can be moved freely across branch boundaries. If-conversion is used in software-pipelining of loops with conditionals [49, 51] and in hyperblock scheduling [44]. Predicates may also be used to fill branch delay slots more effectively than would be possible otherwise.

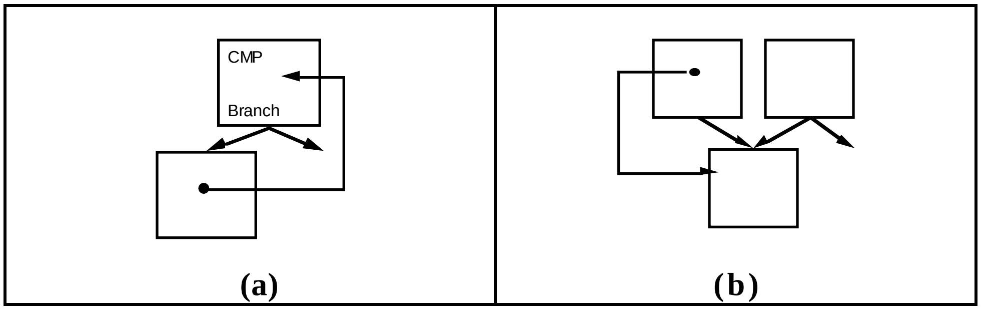


Figure 3.1. Code motion enabled by using predicated execution.
(a) Motion above branch. (b) Motion below merge.

In general, predicated execution provides a non-speculative approach to exploiting instruction-level parallelism. Figure 3.1 illustrates two common uses of predicated execution to allow greater code motion flexibility. Without predication, these motions are unsafe and require special treatment. More specifically, for an unpredicated operation to move above a branch, the operation must execute speculatively. Speculation may require the insertion of sentinel operations in the home block to detect exceptions when execution takes that path. (See Section 6.) If the speculated operation writes to a register that is live on the other path, the destination variable and subsequent uses must be renamed. Since stores and loads that are "maybe dependent" upon earlier stores cannot be speculated, their motion is normally restricted. Furthermore, unpredicated operations cannot be moved below a merge point. On the other hand, properly predicated operations can move freely across block boundaries without the need to speculate, rename, or insert compensation blocks.

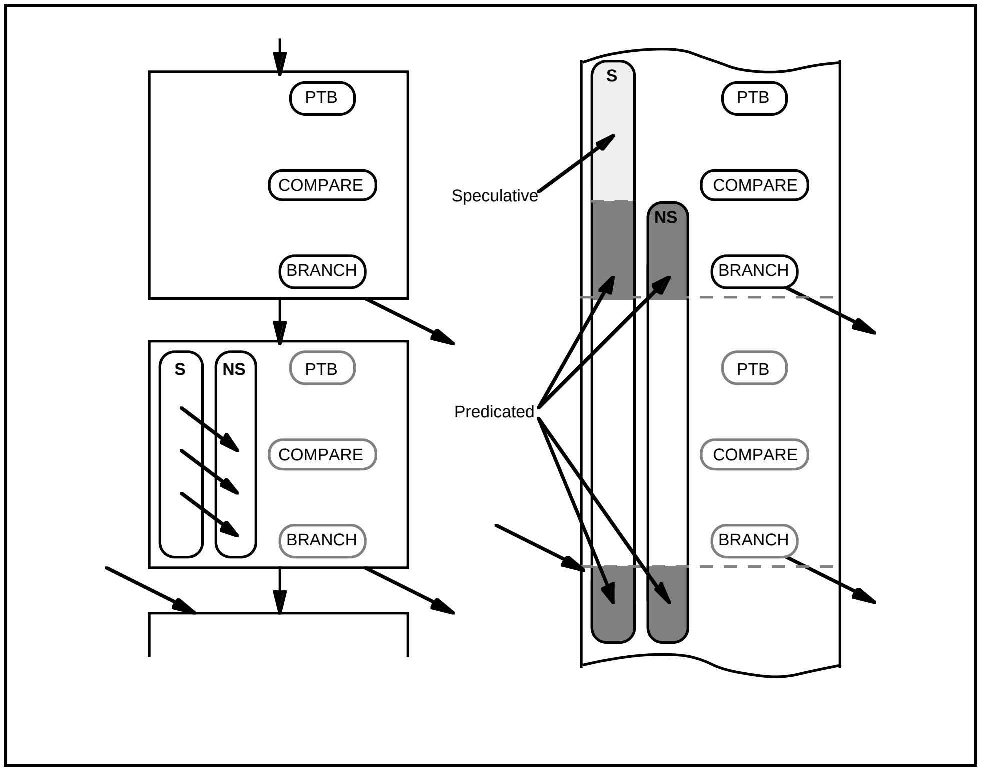


Figure 3.2. Relationship between speculation and predication.

Figure 3.2 illustrates the ranges of motion provided by speculation and predication. For upward motion across a branch, both speculative and predicated operations are free to move. However, predication can be used even in cases where speculation is not possible, such as when moving stores above a branch. On the other hand, predicated operations remain trapped below the compare operation that sets the guarding predicate, whereas speculative operations may move freely above both the branch and the compare operation. Motion below a merge point is normally unsafe, and speculative execution does not address this issue. Predicated operations may move freely below the merge, although the lifetime of the guarding predicate is lengthened.

3.2 Architectural Support for Predicated Execution in PlayDoh

Support for predicated execution provided in PlayDoh is an enhanced version of the predication provided by the Cydra 5 processor [17, 53, 6]. Predication allows the use of a boolean data operand to guard an operation. For example, the generic operation "r1 = op(r2,r3) if p1" executes of p1 is true and is nullified if p1 is false. Omitting the predicate specifier for an operation is equivalent to executing the operation using the constant predicate true.

3.2.1 PlayDoh predicate setting opcode repertoire

PlayDoh introduces a family of compare-to-predicate[1] operations, which are used to compute guarding predicates for operations. A compare operation has the following format:

p1,p2 = CMPP.<cond>.<D1-action>.<D2-action>(r1,r2) if p3

The compare is interpreted from left to right as: "p1" - first destination predicate; "p2" - second destination predicate; "CMPP" - compare op-code; <cond> - the compare condition which is to be evaluated; <D1-action> - first destination action; <D2-action> - second destination action; "(r1,r2)" - data inputs to be tested; and "p3" - predicate input. A data input can be either a register or a literal. A single-target compare is specified by omitting the second destination predicate operand and the second destination action specifier.

The allowed compare conditions are the same as those provided by the HP PA-RISC architecture. These include "=", "<", "<=", and other tests on data which yield a conventional boolean result. The boolean result of a comparison is called its compare result. The compare result is used in combination with the predicate input and destination action to determine the destination predicate value.

Table 3.1. Behavior of compare-to-predicate operations.

Predicate input	Compare result	On result				On complement of result			
		UN	CN	ON	AN	UC	CC	OC	AC
0	0	0	--	--	--	0	--	--	--
0	1	0	--	--	--	0	--	--	--
1	0	0	0	--	0	1	1	1	--
1	1	1	1	1	--	0	0	--	0

[1] The suffix "to-predicate" is intended to indicate that the destination of the compare operation is a 1-bit predicate register and not a 32-bit or 64-bit general-purpose register.

The possible actions on each destination predicate are denoted as follows: unconditionally set (UN or UC), conditionally set (CN or CC), wired-or (ON or OC), or wired-and (AN or AC). The first character (U, C, O or A) determines the action performed on the corresponding destination predicate; the second character (N or C) indicates whether the compare condition is used in "normal mode" (N), or "complemented mode" (C). When an action executes in complemented mode, the compare result value is complemented before performing the action on the destination predicate.

Table 3.1 defines the action performed under each of the allowed destination action specifiers. The result of an action is specified for all four combinations of predicate input and compare result. Each cell describes the result corresponding to the input combination indicated by the row, and action indicated by the column. The cell specifies one of three actions on the destination predicate register: set to zero ("0"), set to one ("1"), or unmodified ("-").

The names of destination action specifiers reflect their behavior. In Table 3.1, we see that with the **unconditional** actions (UN or UC), a compare-to-predicate operation *always* writes a value to its destination predicate. In this case, the predicate input acts as an input operand rather than as a guarding predicate, and the compare-to-predicate operation is never squashed. The value written to the destination predicate register is simply the conjunction of the predicate input and the compare result (or its complement, if the action is UC). On the other hand, cmpp operations using the **conditional** actions (CN or CC) behave truly in a predicated manner. In this case, a cmpp operation writes to its destination register only if the predicate input is 1, and leaves the destination register unchanged if the predicate input is 0. The value written is the compare result (if CN is specified) or its complement (if CC is specified).

The **wired-or** action is named for the familiar circuit technique of computing a high fan-in OR by directly connecting the outputs of (suitably modified) devices, instead of computing the OR of those outputs using an OR gate. In the compare-to-predicate operation, the wired-or action specifies that the operation write a 1 to its destination predicate only if the predicate input is 1 (i.e. the operation is not squashed) and the compare result is asserted (1 if ON, else 0 for OC). Since a wired-or cmpp operation either leaves its destination predicate unchanged or writes only the value 1, multiple wired-or cmpp operations that target the same destination predicate can execute in parallel. That is, the parallel write semantics are well-defined since the multiple values being written are guaranteed to be the same (namely 1). By initializing a predicate register p to 0, the disjunction of any number of compare conditions can be computed in parallel using wired-or cmpp operations all writing to p. The value of p will be 1 if and only if one or more of the compare operations executes with its compare result asserted. The **wired-and** action is similar, but is used to compute the conjunction of any number of compare conditions. The motivation and uses for each kind of compare-to-predicate action will be described next.

3.2.2 Usage of compare-to-predicate operations

Compare-to-predicate operations were designed primarily to address the compiler requirements in three important areas, all of which are related to the broad issue of how to efficiently model the control-flow of a program.

1. **Predicated execution:** In this area, there are two techniques of interest: if-conversion [4, 17, 47, 44] and predicated code motion. If-conversion uses compare operations in the U, C, and O classes. Operations in the U class are suitable only for the if-conversion of "structured" code, i.e., if-then-else control structures which are either concatenated or nested within one another. They are actually the best choice in that case, since they don't require extra operations to initialize predicates. The O and C classes are used to handle "unstructured" code. Choosing between O and C class operations for if-conversion involves making a trade-off between the number of operations required and scheduling freedom. The C class cmpp operations do not require predicates to be initialized to 0, however there can be output dependences between C style cmpp operations targeting a common predicate destination. The use of O class operations gives much more freedom in scheduling; there are no output dependences to honor even though multiple operations target the same register. Table 3.2 summarizes these trade-offs. The compiler requirements for predicated code motion are similar to those for if-conversion.

Table 3.2**.** Pros and cons of using UN, CN and ON classes in if-conversion.

Operation class	**Handles "Unstructured" code**	**Initialization of the predicate**	**False output dependences**
UN	No	No	No
CN	Yes	No	Yes
ON	Yes	Yes	No

2. **Height reduction of control-dependences**: Techniques in this area typically require the efficient computation of boolean conjunction or disjunction of multiple branch conditions in as parallel a fashion as possible; the O and A classes provide the needed functionality.

3. **Efficient computation of complex branch conditions**: Many cases of complex branch conditions can be computed efficiently by reducing them to a combination of boolean reductions. Again, the O and A classes of operations provide a means for fast evaluation of these reductions.

3.3 Unconditional Compare-to-Predicate Operations

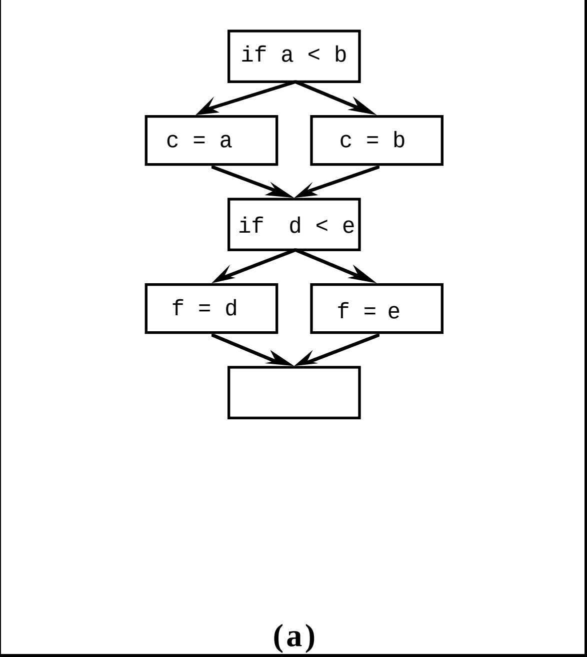
The UN and UC compare-to-predicate operations can be used to permit more general code motion, but their main use is in if-conversion. If-conversion is a technique for replacing branching control flow with predicated execution, so that an acyclic control flow region is converted into a straight-line sequence of predicated operations. This process eliminates hard-to-predict branches, thereby reducing branch mispredict and latency costs [43]. Moreover, the if-converted code provides a larger block of code to the scheduler. In this section, we describe if-conversion of structured control flow regions, which requires only unconditional compare-to-predicate operations.

3.3.1 If-conversion of structured code

Figure 3.3(a) shows an example control flow graph in which predicated execution can be used to exploit greater amounts of ILP. In this example, we have two min functions which are independent and therefore could be executed in parallel. Such a sequence is common -- it results from unrolling a loop and renaming variables. To exploit ILP in this code, we must either have support for multiple independent processes (multithreading), apply code replication techniques, or use predicated execution. Code replication quickly becomes problematic, because the code size in this example grows exponentially with the number of min functions being executed in parallel. Moreover, some method is needed to quickly branch to the correct code block; predicated execution avoids both of these problems.

If-converted code for this example is shown in Figure 3.3(b). In the resulting predicated code, branches have been eliminated and independent operations can execute in parallel. For such structured regions, if-conversion requires only the unconditional class of compare-to-predicate operations. Note that the compare operations can execute in parallel, and then all four predicated assignments can execute in parallel, since only one from each min function will write its value, while the others will be squashed.

This example also illustrates a potential liability of predicated execution. Although predication provides new opportunities for exploiting ILP, it creates new challenges for compilation: in order to exploit the parallelism exposed by predication, a compiler must be able to recognize that no dependences exist between the disjoint assignments having the same destination register. These compiler challenges are discussed briefly at the end of this section.



```
p1 = cmpp.w.<.UN.UC (a, b)
p2 = cmpp.w.>=.UN.UC (a, b)
q1 = cmpp.w.<.UN.UC (a, b)
q2 = cmpp.w.>=.UN.UC (a, b)
c = a    if p1
c = b    if p2
f = d    if q1
f = e    if q2
```

(b)

Figure 3.3. Example of if-convertion of structured control flow graph.
(a) Structured control flow graph. (b) If-converted code.

3.3.2 Two-target compare-to-predicate, normal & complement modes

Compare-to-predicate operations with two destinations provide a means to optimize certain common uses of these operations. For example, if-conversion of structured code regions typically associates two predicates with each branch, one for the then-clause and one for the else-clause. In these cases, the use of dual-destination operations reduces the number of predicate-setting operations by half. For operations with two destinations, it becomes important to make complementary actions available in the opcode repertoire (e.g., UN and

UC) so that the action on one destination can be controlled by the result of the comparison and the action on the other by the complement of the result. For example, the UN-UC combination together with a two-target compare-to-predicate operation is used to halve the number of comparisons required in the example of Figure 3.3(b).

3.4 Conditional Compare-to-Predicate Operations

Unlike structured if-then-else control flow regions, in which every basic block has precisely one branch upon which it is control dependent, unstructured control flow regions are characterized by blocks having multiple control dependence predecessors. When if-converting such regions, the predicate corresponding to an unstructured merge block will have multiple compare operations which potentially set its predicate, one per control dependence predecessor. These compare operations must execute conditionally (since the block in question is executed if *any* one of its control dependence predecessors executes and sets its predicate), and therefore the unconditional compare operation is not adequate.

For example, consider a block B which follows an unstructured merge of control flow; i.e. B is control dependent on multiple branch conditions C1, .., Ck. During if-conversion, a predicate p will be associated with block B, and the proper value for p in order to guard operations from block B is the OR of conditions C1, .., Ck. In if-conversion of structured code, we simply replaced each branch with an unconditional compare-to-predicate operation. This approach fails for structured code, because multiple cmpp operations will be inserted, all of which target predicate p. In the resulting code, these unconditional cmpp operations will appear in some order; the final value of p will be determined only by the last of these cmpp operations, since each one *always* writes a value. This illustrates the necessity of conditionally executing compare-to-predicate operations in order to compute the OR of the multiple conditions C1, .., Ck needed to handle unstructured control flow.

One solution to this problem is to use *conditional* cmpp operations that write to predicate p in place of each branch upon which block B is control dependent [22, 47]. In the resulting predicated code, two conditional compare-to-predicate operations are ordered with respect to one another if the branch corresponding to one of them was control dependent, directly or indirectly, upon the branch corresponding to the other. Consequently, the final value of p is correctly determined: in the original control flow, the last branch in a chain of control dependent branches leading to block B would have transferred control to B; in the predicated code, the corresponding conditional cmpp is the last one that writes to p, and it is the value written by that conditional cmpp that determines whether operations guarded by p execute or are squashed.

3.5 Wired-OR Compare-to-Predicate Operations

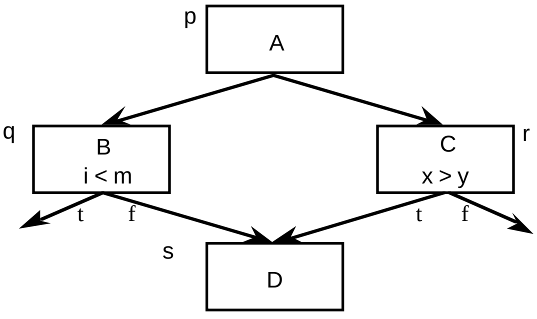
In this section, we describe another solution, using wired-OR compare-to-predicate operations. The advantage of this solution is that the output dependences between the multiple compare-to-predicate operations, that can exist if conditional cmpp operations are used, are eliminated.

3.5.1 If-conversion of unstructured code

Figure 3.4(a) shows a control flow region containing an unstructured merge block (block D). Operations in block D execute if either of the predecessor blocks, B or C, executes and then branches to D. In terms of predicates, operations in blocks A, B, C and D are predicated upon predicates p, q, r and s, respectively. Predicate s is conceptually computed as follows.

$$s = \mathrm{OR}\ (\ q \wedge !(i < m),\quad r \wedge (x > y)\ )$$

The predicated code is shown in Figure 3.4(b).



(a)

```
     A                         if p
q, r = ...                     if p
s = 0                          if p
     B                         if q
s = cmpp.w.<.OC (i,m)          if q
     C                         if r
s = cmpp.w.>.ON (x,y)          if r
     D                         if s
```

(b)

Figure 3.4. Example of if-convertion of an unstructured control flow graph.
(a) Unstructured control flow graph. (b) If-converted code.

The code works as follows. The predicate register s is initialized to 0. Each compare operation writes the value 1 if the (normal or complemented) compare result is true and if its predicate input (q or r, respectively) is true. Otherwise, it leaves the predicate s unchanged. In other words, if one or both of the wired-or compare-to-predicate operations is not squashed, and evaluates to 1, then register s is set to 1; otherwise it retains the initial value 0. This correctly evaluates the boolean expression. The conditional update of the destination predicate is essential for correct if-conversion of unstructured code; PlayDoh also defines simultaneous-write semantics (described next) to support efficient parallel computation of the guarding predicate.

3.5.2 Simultaneous writes to registers

Most architectures don't permit multiple operations to write into a register at the same time; the result is defined to be indeterminate. PlayDoh permits multiple operations to write into a register at the same time; the result is well-defined in special cases. The semantics of simultaneous writes to a register are as follows. When multiple operations write the same value at the same time, that value is written to the register. However, if multiple operations write different values at the same time, the result stored in the register is indeterminate.

Because the wired-or compare operations all conditionally write the value 1 and otherwise do not write a value, the results are always well-defined. The most important point to note is that in this case, there are no output dependences between the compare operations even though they all write to the same register. Therefore, the compare operations can be scheduled in any order, and can even execute concurrently. In Figure 3.4(b), operations from blocks B and C, including the wired-or compare-to-predicate operations, can execute in parallel.

3.6 Wired-AND Compare-to-Predicate Operations

The wired-AND action is similar to the wired-OR action, and it is used to compute efficiently high fan-in AND operations as follows:

- the result predicate is initialized to value 1;
- any number of compare operations execute, each of which uses the AN or AC action to conditionally set the result to 0;
- after all compares have executed, the correct conjunction is available as the result.

The wired-AND compares can execute in any order or in parallel since each conditionally clears the result if the compare condition is false. When multiple compares execute in the same cycle, the multiported register file insures that whenever more than one wired-AND compares simultaneously clears the result, the result is in fact cleared.

3.6.1 Control height reduction

In this section, we illustrate the use of wired-AND compare-to-predicate operations in reducing the height of a control dependence chain. Figure 3.5(a) shows a typical situation where control height can be reduced. Using branch frequency information, a dominant execution trace is selected, forming a superblock; the superblock is a single-entry multiple-exit sequence of basic blocks intended to be scheduled as a single unit [30]. Unfortunately, instruction-level parallelism is limited in this example, and store operations are trapped below branches, limiting the scope of code motion. Because branching off-trace is unlikely, we would like to expedite the fall-through path.

In Figure 3.5(b), we have used boolean AND operations to compute guarding predicates for operations on the fall-through path. Predicates p1, p2, p3, and p4 fully guard operations below the corresponding branches; i.e. predicated operations are independent of *all* previous branches. There is redundancy in the computation of these predicates due to the height reduction of the boolean compuration for each predicate. In the resulting code, predicates are available early through boolean height reduction, stores may move as early as permitted by data dependences, aliasing memory operations move up with stores, and operations move up as early as permitted by their data dependences and live-out constraints at the side-exits. Because operations have greater freedom of motion, resource contention is smoothed over the entire superblock. Often, the low probability branches can all move off-trace, once an additional branch on condition ~p4 is introduced to branch out to a compensation block holding the original branches.

3.7 Features That are Made Less Exotic

The use of predicated execution reduces the dependence on other architectural and microarchitectural features for achieving high performance. In particular, the branch architecture and the dependence on speculative execution are affected.

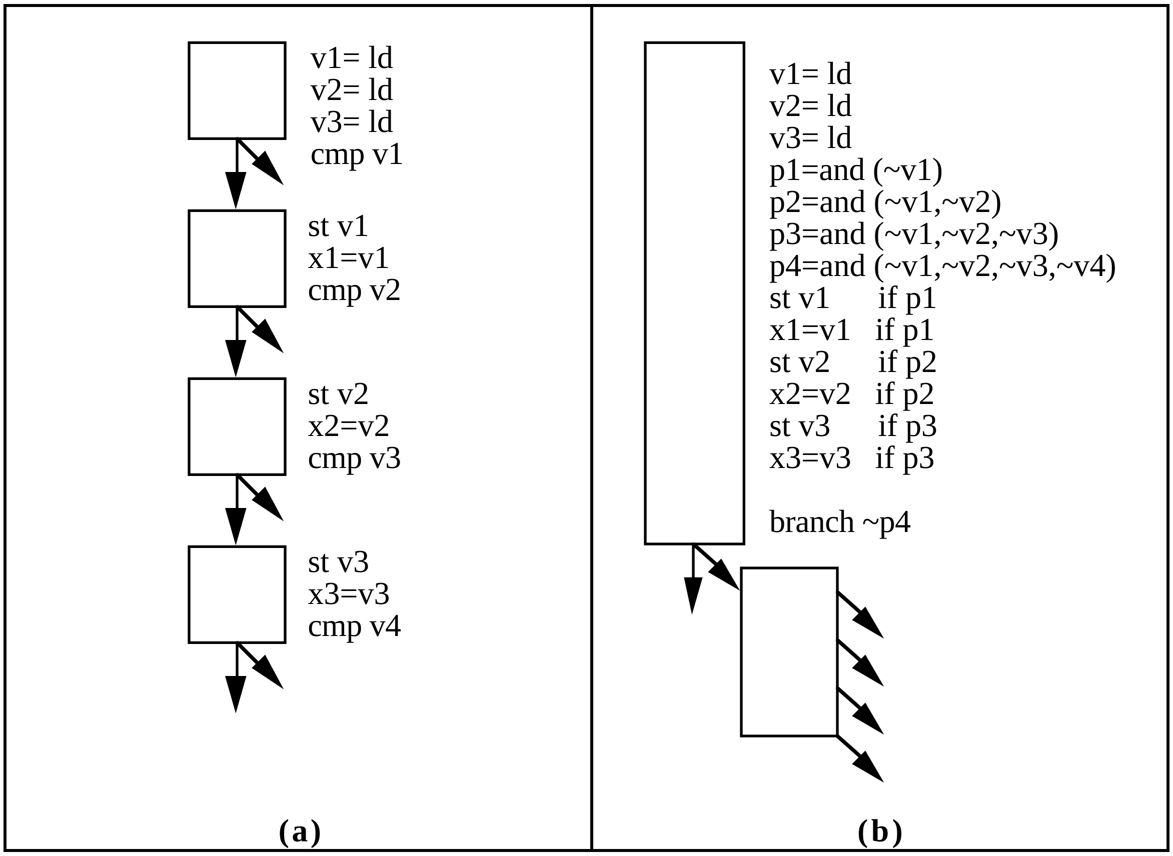


Figure 3.5. Example of control height reduction using wired-and cmpp operations. (a) Superblock with on-trace branches. (b) Height-reduced code.

3.7.1 Branch architecture

If-conversion replaces explicit branches in acyclic control flow with predicated execution. Because branches are eliminated, the branch misprediction rate generally decreases [43]. There are two reasons for this. First, hard-to-predict branches are often the ones eliminated through if-conversion -- highly biased branches need not be if-converted, since trace selection or superblock formation works well in this case. As branches are eliminated, dynamic prediction may improve for the remaining branches, since they no longer compete with the eliminated branches for dynamic prediction resources, such as entries in the branch target buffer.

As we have seen, predicated execution allows greater scope for code motion. Predicated operations can fill branch delay slots more often, thereby reducing branch penalty even for correctly predicted branches.

To reduce the prefetch penalty due to I-cache misses, one can do multipath instruction prefetch in the absence of predicated execution. This places a heavy demand on the prefetch unit. For instance, with a sequence of if-then-else clauses, even though flow of control repeatedly reconverges, the instruction text that is prefetched grows exponentially with the number of branches, because it is hard to avoid prefetching the same operations when they are reached via different control flow paths. In predicated code, the branches and merges are eliminated, and so redundant prefetching is not an issue.

Finally, when exploiting instruction-level parallelism between control-independent code regions, predication reduces the need for high branch bandwidth, multiple branches per cycle, prioritized branching, multiway branching, or tree branches that guard intermediate operations.

3.7.2 Speculative execution

There is a cost associated with speculative execution, whether static or dynamic. Predicated execution lessens the impact by providing a non-speculative alternative when performing code motion. The alternative to predicated execution is to issue branches as early as possible, which increases the code size and the number of small compensation blocks. (As branches are moved up across operations, the operations are duplicated below each exit of the branch. If a branch exit flows directly to a merge point, a compensation block is introduced.) Compensation blocks are undesirable because they are often small and have little ILP. Small compensation blocks also put extreme demands on branch hardware.

3.8 Compiler issues

As we have seen in this section, predicated execution offers many opportunities for exploiting increased amounts of instruction-level parallelism. However, in our experience, adding general compiler support for predicated code is a major challenge. Here, we describe briefly some of the required compiler support for predicated execution.

3.8.1 Opportunities

If-conversion algorithms. Existing algorithms [47, 40, 18] for if-conversion eliminate all internal branching within a single-entry single-exit acyclic control flow region. The IMPACT algorithm handles a restricted case of multi-exit region, which after if-conversion is termed a hyperblock, where "side" exits are allowed, provided that the region has a single post-dominating final exit when these side exits are ignored [44]. By conceptually ignoring side exits during if-conversion, the current algorithms do not fully eliminate control dependences within the if-converted region; operations below side exits remain control dependent on dominating branches.

We are currently investigating a new approach to if-conversion of multi-exit regions in which all predicates within the region are *fully resolved*. In this case, the side-exit fall-through conditions are incorporated into the guarding predicates of subsequent (dominated) operations, so that all control dependences are eliminated within the if-converted region [56].

Predicate promotion. Up to this point, we have discussed predicated execution as an alternative to speculative execution. Operations are either fully guarded by a predicate or they execute unconditionally (but potentially speculatively) when control flow reaches them. In practice, we find cases where a guarding predicate p can be relaxed, or promoted, to another predicate q such that the operation executes speculatively but not unconditionally. That is, q is true whenever p is true (but not vice versa). A common situation where this arises is from if-conversion of nested if-then-else statements. Promotion is analogous to hoisting an operation from a deeply nested if-then statement to an outer scope. The motivation is that the relaxed predicate may be available earlier than the non-speculative guarding predicate. The normal concerns about overwriting live values and detecting exceptions apply.

Critical-path reduction. Just as predicated execution is an important tool for exploiting ILP, it is also useful for exposing additional ILP through height-reduction of computations on the critical path. Height-reduction transformations can take many forms, but the central theme is to reorganize computations so that the longest dependence path through a computation is shortened. This critical path may contain flow, anti, or output data dependences, or it may contain control dependences. Variable renaming is a simple example of height reduction, where output and anti dependences are removed, thereby exposing parallelism between previously sequentialized (but independent with respect to value flow) computations. A more traditional example is the reassociation of a linear recurrence into a logarithmic-height tree computation, as can be done with an unrolled reduce-to-scalar loop. Predicate promotion, described above, is another example of height reduction, where a flow dependence is relocated from a critical-path compare-to-predicate operation to one of lesser height, resulting in speculation of the guarded operations. Predication is also useful for height reducing boolean expressions, by using the wired-OR and wired-AND operations. These reduced-height expressions may be data expressions, guarding predicates in if-converted unstructured code, or they may be fully-resolved branch conditions which lie on the critical path of some computation [56].

New twists on conventional optimizations. Predicated execution generalizes code motion and gives an alternative to control flow for determining execution. Conventional optimizations such as common subexpression elimination, partial redundancy elimination, and partial dead code elimination can exploit this new degree of freedom. Traditionally, these algorithms relocate operations in order to eliminate redundancies. With predication, operations can be guarded as well as moved. The best choice will probably require an understanding of resource usage, register pressure, and critical path length.

3.8.2 Challenges

Predicate analysis. Conventional compiler analysis techniques produce incorrect or overly-conservative results when applied to predicated code. To aggressivly target machines with predicated execution, a compiler must understand run-time relationships between predicates and incorporate this information into data flow analysis and optimization. For example, in register allocation, temporally overlapping live ranges are said to interfere, and their corresponding variables are assigned to different physical registers. In predicated code, temporal overlap of live ranges is necessary but not sufficient to cause interference, since overlapping live ranges may be predicated upon disjoint predicates. That is, in any execution, it may be the case that only one of the two variables is ever written or read. By understanding the disjointness between guarding predicates, a predicate-sensitive register allocator can use fewer registers in allocating predicated code [21, 34].

Predicate-cognizant data flow analysis. In general, many phases of analysis may require understanding the semantics of predicated execution. By extending data flow analysis to handle predicated code, all modules that make use of data flow analysis (e.g. optimizations) can operate on predicated code.

Pervasive impact on compiler design. In our experience, adding compiler support for predicated execution affects the overall design of the compiler. Predicated code may be introduced quite early in the compilation process, through inlining of hand-coded intrinsics, if-conversion, schematic transformations, modulo-scheduling, height-reduction, or optimization. Clearly it is desirable that predicated code be treated as effectively and easily as conventional code. Without such a facility, predication will be avoided rather than aggressively pursued.

4 Architectural support for innermost loops

It is generally understood that there is inadequate instruction-level parallelism (ILP) between the operations within a single basic block and that higher levels of parallelism can only result from exploiting the ILP between successive basic blocks [65, 26, 54, 46, 10, 67]. In the case of innermost loops, the successive basic blocks are the successive iterations of the loop. One method that has been used to exploit such inter-iteration parallelism has been to unroll the body of the loop some number of times and to overlap the execution of the multiple copies of the loop body [25]. Although this does yield an improvement in performance, the back-edge of the unrolled loop acts as a barrier to parallelism. Software pipelining, in general, and modulo scheduling, specifically, are scheduling techniques which attempt to achieve the performance benefits of completely unrolling the loop without actually doing so.

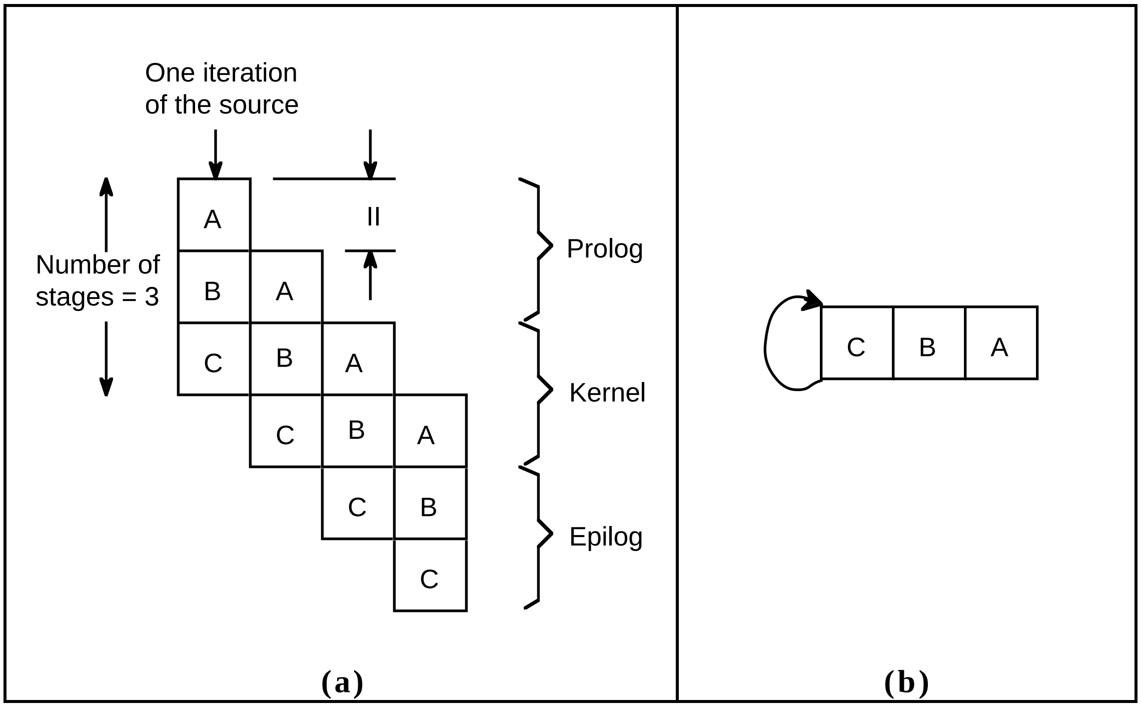


Figure 4.1: (a) Record of execution for a modulo scheduled loop. (b) Kernel-only code schema

Modulo scheduling engineers the schedule for a loop so that successive iteration of the loop are issued at a constant interval, called the **initiation interval** (II). Typically, the initiation interval is less than the time that it takes to execute a single iteration. As a result, the execution of an iteration can be overlapped with that of other iterations. This overlapped, parallel execution of the loop iterations can yield a significant increase in performance. Figure 4.1(a) shows the record of execution for a modulo scheduled DO loop in a schematic form. The figure shows the execution of 4 source iterations of the loop. The schedule for an iteration can be divided into stages consisting of II cycles each. The number of stages in an iteration is termed the **stage count** (SC). In the figure, the schedule for each iteration contains three stages, represented by A, B and C. The execution of the loop consists of three distinct phases: ramp-up, steady-state and ramp-down. The ramp-up phase "fills the pipeline" by initiating a new source iteration every II cycles. The steady-state phase represents that part of the execution during which an iteration completes and a new iteration is initiated every II cycles. The ramp-down phase "drains the pipeline". It completes the iterations that have been initiated, but initiates no new iterations.

The code to achieve this can be partitioned into three parts. The **prologue** is the code that is executed during the ramp-up phase. The **kernel** is the loop that when executed yields the repeating part of the record of execution in Figure 4.1(a). The **epilogue** is the code that is executed during the ramp-down phase. In Figure 4.1(a), indicated on the right hand side of the figure is the code that is being executed at each point in time.

Architectural support for modulo scheduling (*e.g.*, as in PlayDoh or the Cydra 5) enables generation of high performance, compact code for modulo scheduled loops. For example, the kernel code is sufficient to generate the record of execution shown in Figure 4.1(a) on processors with support for modulo scheduling. This is called the **kernel-only** code schema for counted loops [52]. Figure 4.1(b) shows the kernel-only code for the example. The advantages of this schema include its small code size and ease of engineering. A single piece of code equivalent in size to the original loop is sufficient to execute the entire modulo scheduled loop. In contrast, other code schemas require complex code replication to achieve the same effect [52].

The support for modulo scheduling includes rotating register files, specialized loop control operations, predicated execution and support for compile-time speculative code motion. For brevity, we focus on modulo scheduling of counted loops and describe the use of many of these features in the kernel-only code schema. A detailed description of modulo scheduling of while-loops can be found elsewhere [64, 52].

4.1 Rotating register file to address overlapped life-times

A fundamental problem in generating modulo schedules is to prevent successive lifetimes for a loop-variant virtual register from being assigned to the same physical register. The successive lifetimes correspond to the successive definitions of the loop-variant virtual register in successive iterations.

Consider the example in Figure 4.1. Suppose the loop contains two operations, a load and an add, which are scheduled in the stages A and C, respectively. Figure 4.2 shows the execution of the two consecutive iterations of the loop. The execution of the stage A in iteration i will load a value in vr1. The lifetime of this value extends to the cycle in which add is scheduled in the stage C. However, II cycles later the load operation will be executed again on behalf of the next iteration and will overwrite the value in vr1 while it is still live, thereby yielding an incorrect result. An approach to address this problem is to provide some form of register renaming so that successive definitions of vr1 actually use distinct registers. The rotating register file provides such a capability.

Iteration i	**Iteration i+1**
vr1 = Load(vr2);	
…	vr1 = Load(vr2);
vr3 = Add(vr1, 4);	…
	vr3 = Add(vr1, 4);

Figure 4.2: Example to illustrate overlapped lifetimes

Registers in a conventional register file are addressed using an absolute address, *e.g.*, register number 32. In contrast, registers in a rotating file are addressed using a base plus offset model. The register number specified in an instruction, either as a source or a target, is added to the **rotating register base** (RRB) to derive the physical register number in the file. The addition is a modulo sum with the number of physical register in the file as the modulus. Thus,

physical register address = (RRB + instruction-specified address) **mod** file size

Special branch operations which are used in modulo scheduling decrement RRB each time a new iteration starts, thereby giving each loop iteration a distinct physical register from that used by the previous iteration.

Iteration i	**Iteration i+1**	**Physical registers accessed by operations**
rr15 = Load(vr2);		(0 + 15) mod 64 = 15
…	rr15 = Load(vr2);	(0 -1 + 15) mod 64 = 14
vr3 = Add(rr17, 4);	…	(0 -2 + 17) mod 64 = 15
	vr3 = Add(rr17, 4);	(0 -3 + 17) mod 64 = 14

Figure 4.3: Use of rotating registers to handle overlapped lifetimes

Consider the example in Figure 4.2 again. Assume that vr1 is allocated to a rotating register file. Figure 4.3 shows how the use of rotating registers addresses the problem of overlapped life-times. The first two columns show the execution of the two consecutive iterations of the loop. The third column shows the actual register that is accessed assuming that RRB is set to 0 at the beginning of the ith iteration and that the rotating register file contains 64 physical registers. The load operation specifies rr15 as its target. In the ith iteration it writes to the physical register number 15. At the end of the stage, a special branch operation executes which decrements RRB. As a consequence, the load operation in the next iteration writes to physical register 14, though it still specifies rr15 as the target. Thus, the distinct life-times originating in different iterations are given distinct registers. Since RRB is decremented at the end of each stage, proper producer-consumer relationship must be maintained by specifying appropriate offsets. For example, the add operation specifies rr17, and not rr15, as the source operand in order to access the value produced by the load, since the add operation is scheduled two stages later than the load operation, and RRB is decremented twice between the two operations. Thus, the add operation in iteration i reads physical register 15, which is written by the load operation in iteration i. Similarly, the add operation in iteration i+1 reads physical register 14, the one written by the load in that iteration.

A rotating register file is quite similar in concept to vector registers. Instead of moving the pointer every cycle as in the case of vector registers, it is decremented once per kernel iteration, and instead of having multiple vector registers, they are all pooled into one circular register file.

Rotating register files provide dynamic register renaming but under the control of the compiler. Their availability in the architecture allows the compiler to generate compact code for modulo scheduled loops in the form of the kernel-only code. Rau, et al. [52] discuss code schemas that don't rely on rotating registers, all of which require unrolling of the kernel to correctly handle overlapped lifetimes. It is important to note that conventional hardware renaming schemes are inadequate and cannot be used in place of rotating register files. In a modulo scheduled loop, successive definitions of a variable (see vr1 in the above example) are encountered before the uses of the prior definitions. Thus, it is impossible even to write correct kernel-only code with the conventional model of register usage.

From the perspective of micro-architecture, rotating registers add a modulo sum in the register access path. The modulo sum can be implemented as an adder of the appropriate size given by the number of registers in the file. Its impact on pipeline depth or cycle time is hard to quantify without a detailed pipeline design for a given cycle time. From the perspective of the compiler, rotating registers both simplify and complicate the task. Generating kernel-only code is simpler to engineer than other code schemas that require complex code replications. On the other hand, the compiler has to allocate rotating registers. Rotating register allocation is somewhat different from the traditional allocation and it is easy to engineer it as a separate module. The traditional allocator must be enhanced to understand rotating register allocation at the entries to, and the exits from, modulo scheduled loops.

4.2 Use of predicated execution

Predicated execution is used for two distinct purposes in the modulo scheduling of loops. First, it is used to handle loops that have control-flow within the body. Explicit control-flow can be replaced by predicated code in such loops by using if-conversion. This simplifies code generation for loops with control-flow since the body of the loop no longer contains any branches; as far as the modulo scheduler is concerned, it is a single basic block. (Predicated execution and its use in if-conversion were discussed in Section 3.) Second, predicated execution is essential for the kernel-only code schema to work as described below. Consider the record of execution in Figure 4.1(a). Each stage during the ramp-up or ramp-down phase is a subset of the kernel-only code shown in Figure 4.1(b).

Thus, the prologue and epilogue can be swept out by executing the kernel with the appropriate operations disabled by predicated execution.

Source Iteration								Predicate Registers	LC	ESC
-2	-1	0	1	2	3	4	5	p2 p1 p0		
								0 0 1	3	2
C	B	A						0 0 1	3	2
	C	B	A					0 1 1	2	2
		C	B	A				1 1 1	1	2
			C	B	A			1 1 1	0	2
				C	B	A		1 1 0	0	1
					C	B	A	1 0 0	0	0
								0 0 0	0	-1

Figure 4.4: Prologue and epilogue generation using predicated execution. Shaded iterations are spurious iterations.

Figure 4.4 shows the dynamic creation of prologue and epilogue computation using predicated execution. All operations from the ith stage are logically grouped together by predicating them on the same predicate, specifically, the rotating predicate register specified by the predicate specifier i relative to the RRB. In this example, operations in stages A, B and C are predicated on rotating predicate registers p0, p1 and p2, respectively. The code outside the loop sets the staging predicate, p0, for the first stage to 1 and sets the predicates for all other stages to 0. As a consequence, only operations in stage A execute in the first II cycles; operations in other stages are nullified. (Nullified operations are shown as shaded boxes in the figure.) The execution of the branch operation at the end of II cycles decrements RRB, which has the effect of shifting the staging predicates, i.e., the predicates that used to be referenced as p0, p1 and p2, are now referenced as p1, p2 and p3, respectively. In addition, the branch operation sets the staging predicate p0 for the first stage, the effect of which is to initiate a new source iteration. This continues until all the source iterations have been issued. Note that once p0, p1 and p2 are all 1, the loop is in the steady-state phase. Once the last source iteration has been initiated, the branch operation thereafter sets the staging predicate p0 to 0 in order to disable all operations in stage A. On each subsequent iteration of the kernel, one additional staging predicate becomes 0. This continues until all the staging predicates are 0, at which point the last source iteration has completed. The manner in which these phase transitions are triggered is discussed in Section 4.3.

4.3 Branch operations for modulo scheduled loops

As pointed out in the last two sections, the kernel-only code schema for counted loops relies on special branch operations. In PlayDoh, there are a number of such operations, generically called the BRF family of operations. In this section, we summarize the semantics of these operation using BRF.B.B.F (called brtop in the Cydra 5). The reader is referred to the PlayDoh architecture specification [36] for more details about other operations, including the ones to support while-loops, and for the exact semantics of these operations.

Branch operations for counted loops use two architecturally visible registers—the **loop counter** (**LC**) and the **epilogue stage counter** (**ESC**). Prior to entry into the loop, LC is initialized to one less than the trip count of the loop, and ESC is initialized to one less than the number of stages, *i.e.* SC-1, which is the number of additional kernel iterations needed to drain the pipeline once the last source iteration has been initiated. The way that the BRF.B.B.F operation behaves is as follows. While LC is not zero, the loop is in either the prologue or the kernel phase. In this case, the branch operation decrements LC, decrements RRB, sets the predicate p0 to 1, and branches to the beginning of the loop. If LC is 0 when the branch operation is issued, the loop is either about to enter, or is already in, the epilogue phase. In this case, the operation decrements ESC, decrements RRB, sets p0 to 0, and branches to the top of the loop. If both LC and ESC are 0, the execution of the loop is over. In this case, the branch operation falls through to the code after the loop.

5 Branch architecture

5.1 Simultaneous and ovelapped branches

The exposure of MultiOp and NUAL parallelism has important implications for the definition of branch operations. The simplest MultiOp instructions issue simultaneous operations without dependence checking for both arithmetic and branch operations. Operations within an instruction containing a taken branch execute irrespective of their position relative to the branch. Here, even a unit latency branch has a branch shadow consisting of the operations "after the branch" (i.e., to the "right" of the branch) but within the same MultiOp instruction. Operations positioned after the branch in the instruction text but within this shadow execute irrespective of the branch condition. When the branch latency is increased beyond one, the branch shadow also includes one or more complete MultiOp instructions which execute unconditionally prior to the completion of the branch operation at the branch latency.

A more traditional definition of branch semantics where each branch dismisses all operations after the branch is not as attractive when used with MultiOp instructions. MultiOp instruction encodings are often non-uniform allowing only specific operations within specific positions within the instruction. The traditional branch definition leads to unappealing architectures where, for example, an integer operation which always lies to the left of a branch in a MultiOp is not guarded by the branch while a floating point operation which is to the right of the branch is guarded by the branch. Such inconsistencies are eliminated when either all operations or none of the operations are guarded by a branch within the current MultiOp.

PlayDoh defines semantics for simultaneous and overlapped branches using a definition which is both compatible with MultiOp execution semantics, and requires the least hardware to implement. PlayDoh specifies that if two branches take simultaneously then the branch target is undefined. Here MultiOp branch operations are treated like MultiOp data operations: when two operations write the same register (in this case, the program counter) on the same cycle they produce an undefined result.

An alternative strategy is to define a priority between the multiple branches in a MultiOp instruction [14]. When concurrent branches are prioritized each branch dismisses all branches having lower priority. However, prioritized branches are dependent branches requiring the flow of signals between multiple branches within a single cycle. Further, when branches have latency greater than one, priority must be extended to dismiss all branches within the branch shadow.

PlayDoh treats branches differently from other operations in one respect: branch operations with LEQ semantics are not allowed. A LEQ branch might (or might not) execute each of the operations within its shadow. This makes the effective use of a branch shadow difficult. Branches with EQ semantics are more useful allowing the branch shadow to be filled with operations which execute irrespective of the branch condition. PlayDoh, therefore, only provides branches with EQ semantics.

It can be very complex to schedule NUAL EQ branches where, a taken branch executes in the shadow of a taken branch. A discontinuity in scheduling semantics arises when a subsequent branch is moved from outside the shadow of a preceding branch into the shadow and both branches take. When the subsequent branch is outside the shadow, the first branch dismisses the second. When the subsequent branch moves into the shadow, the final branch target reached is not that of the first taken branch but instead that of the subsequent taken branch within the shadow. It is very complex to develop program schedules which benefit from these complex interactions among overlapped pipelined branches.

Predicates can be used to simplify the interaction of simultaneous or overlapping MultiOp NUAL branches. In particular, multiple branch predicates can be computed with a guarantee that only one is true. This in turn can guarantee that multiple overlapped predicated branches do not take. Mutually exclusive branches can be scheduled simultaneously or in an overlapped fashion without introducing any of the significant complexities arising from pipelined branches. Methods for calculating predicates providing this form of exclusion have been introduced [57].

5.2 "Unbundled" branch architecture

Many traditional branch operations are really compound operations. PlayDoh decomposes branch semantics into component operations (in a fashion similar to the Pipes architecture [73]) allowing component operations to issue separately. The PlayDoh branch components are:

1) A prepare-to-branch operation has two inputs which provide the branch target address as well as a static branch hint. The result operand targets a register in the branch target register (BTR) file.

2) A compare operation computes the branch condition targeting a result in the predicate file.

3) A branch operation specifies two inputs: a predicate which determines the branch condition and the BTR operand which corresponds to a branch which has been previously prepared.

These operations must be scheduled so that the branch is prepared and the branch condition computed before executing the actual branch.

The decomposition of branches provides additional efficiency within the branch unit pipeline. For example, the prepare to branch may be issued prior to the actual branch operation. This allows the instruction unit to prefetch text at the target of the branch prior to the issue of the actual branch operation. Similarly, the compare operation may also be issued well before the actual branch. This separates the latency of the compare from the branch thus reducing the branch latency.

Further, the separation of branches into components allows branch optimization at a more fine-grained level. A common example is the optimization of a loop closing branch where, the address of the branch target is loop invariant and may be prepared outside of the loop. While the branch condition must be tested on each loop iteration, branch preparation occurs once. This allows fewer operations within the loop body and a more timely calculation of the loop branch target address.

6 Control Speculation

A major impediment to exploiting ILP is the control dependences imposed by branch operations. An operation is control dependent on a branch if the branch determines whether control flow will actually reach the operation. As a result, the branch determines whether the operation will be executed at run-time. For example, with a conventional if-then-else statement in C, all operations in the "then" clause are control dependent on the if-statement evaluating to true. Correspondingly, all operations in the "else" clause are control dependent on the if-statement evaluating to false. Thus, the operations in the "then" and "else" clauses may not be executed until the condition is evaluated. Control dependences impose this kind of ordering constraint on operations with respect to a branch to ensure that operations are executed at the proper times. As a result, the number of independent operations available each cycle is limited by the presence of control dependences. This problem becomes extremely serious for non-numeric applications which contain a high frequency of branches. For these applications, very little ILP may be extracted due to the large number of control dependences.

Control speculation refers to the process of executing operations before one or more of their control dependences are satisfied. In this manner, operations are executed before it is certain their execution is required. Static control speculation support allows the compiler to ignore control dependences, thereby speculating the operations. The purpose of static control speculation is to increase the ILP that the compiler exposes in an application. A compiler can utilize static control speculation to increase ILP in several ways. First, it allows the compiler to employ an aggressive global scheduling algorithm to move operations across basic block boundaries. The ability to overlap the execution of operations from many different basic blocks enhances the ILP that the compiler can expose by a large amount. Second, long latency operations, such as memory loads, can be initiated early to overlap their execution with useful computation. Finally, instructions which start long dependence chains can be executed early to reduce the length of critical dependence chains.

In discussing control speculation, it is useful to define the notion of the **execution condition** for an operation to be the same as the predicate for that operation, *in the original, unspeculated version of the code*, except that it need not be explicitly computed; it is implicit in the values of the set of branch conditions which, together, either guide flow of control to that operation or away from it. Thus the execution condition for an operation determines whether or not that operation gets executed in the original program, prior to any control speculation.

In order to correctly perform static control speculation, several obstacles must be overcome. These problems occur because speculative operations are executed more often than is required for proper program execution. Therefore, the side effects of speculative operations require special handling to ensure correctness and good performance. In particular, the side effects of speculative operations must be reflected in the processor state when their execution is indeed required. However, their side effects must be suppressed for execution instances that were not required. The problem is made more difficult by the fact that the execution condition of an operation may not be known for many cycles after the operation completes. Thus, there is this period of unknown execution status for control speculative operations which must be handled. To assist with the discussion in this section, the terms necessary speculative operation (NSO) and unnecessary speculative operation (USO) will be used to refer to an execution instance of a speculative operation which is required and is not required by the original program semantics, respectively.

The two main obstacles that must be addressed for speculative operations are the results that are produced and any exceptions that are generated. First, the results produced by USOs (architecture registers and memory system updates) must not corrupt the input operands of subsequent NSOs and nonspeculative operations. As far as the execution of subsequent operations is concerned, a USO should appear to not produce any results. The second obstacle is ensuring proper exception handling for speculative operations.

Exceptions that occur for USOs must be ignored, whereas exceptions for NSOs must be signaled and handled in the appropriate manner.

Potential control speculative operations may be categorized at compile time as either safe or unsafe. Safe speculative operations are those operations that the compiler can guarantee will produce no undesirable side effects when unnecessarily executed. The compiler can also perform transformations to make more operations safe, such as compile-time renaming or data structure padding. For example by renaming the destination register of an operation to a new temporary register, the compiler can guarantee that the operation will not corrupt a source operand of a subsequent operation. The other category of potential control speculative operations is unsafe. Unsafe speculative operations are all those remaining operations that the compiler cannot classify as safe. For unsafe operations, static control speculation cannot be performed without underlying support provided in the architecture. Architecture support generally alleviates either or both of the side effect obstacles to some extent and allows a larger fraction of operations to be candidates for control speculation. The remainder of this section describes the support that is provided in the PlayDoh architecture for static control speculation.

6.1 Static control speculation in PlayDoh

In the PlayDoh architecture, the support focuses on overcoming the exception handling obstacle to enable control speculation of unsafe operations. The philosophy behind this approach is that it is generally believed that the result generation obstacle can be efficiently overcome by conventional compilers via renaming transformations. However, it is very difficult to prove operations such as, loads or floating-point adds, will not cause an exception when they are unnecessarily executed. As a result, many potential speculative operations must be classified as unsafe by the compiler because of potential exceptions. The architecture support provided in PlayDoh allows potentially excepting operations to be safely speculated by the compiler.

The basic idea of the PlayDoh speculation model is delayed exception handling. Exceptions for control speculative operations are recorded in a structure at the time of occurrence, but no immediate action is taken to report or handle the exception. Rather, exception processing is postponed to a later time when the execution condition of the operation is known. At that point, the speculative exception can be properly processed or discarded. This form of delayed exception handling is accomplished by symbolically providing a *check* for each potentially excepting instruction which is speculated. The check may be an explicit or implicit operation. It is the responsibility of the check to flag any exceptions that are caused by the speculative operation. The check is placed in the speculative operation's original basic block, or home block, so the exception is only flagged if the excepting operation would have been executed in the original program. For the cases where execution does not reach the speculative operation's home block, the check is never executed, thereby correctly ignoring the speculative exception.

The PlayDoh architecture provides two major extensions of the mechanisms provided in the Multiflow machines [14] to support delayed exception handling for control speculative operations [19, 16, 36, 42]. First, each architecture register is extended to contain an additional field called the exception tag. The **exception tag** indicates that the register value is not correct because the operation responsible for creating the value, or some flow dependence predecessor of it, excepted. The second extension is an additional bit for each opcode to differentiate speculative and nonspeculative versions of all opcodes that can be speculated. For purposes of this discussion, the extensions are referred to as the E bit and S bit, respectively. The E and S bits are used to determine the execution semantics for operations in PlayDoh. Table 6.1 summarizes the semantics of control speculative and nonspeculative operations.

Table 6.1**:** PlayDoh execution semantics to support static control speculation.

Operation's S bit	**E bits of source registers**	**Operation generates an exception**	**Destination's E bit**	**Other actions**	**Signal exception if enabled**
1 (Speculative)	0 for all sources	No	0	Update destination register with the result	No
		Yes	1	Record PC and other state information for exception reporting	No
	1 for one or more sources	Don't care condition	1	Record that an exception was propagated if necessary	No
0 (Non-speculative)	0 for all sources	No	0	Update destination register with the result	No
		Yes	0	---	Yes
	1 for one or more sources	Don't care condition	0	---	Yes

6.1.1 Execution of a control speculative operation

Control speculative operations execute normally when the E bits of all their source operands are reset and the operation does not generate an exception itself. For the normal case, the speculative operation performs two actions: the result of the operation is written to the destination register and the destination register's E bit is cleared. For the other execution scenarios, special actions are required to deal with the exception conditions. The first case is that the speculative operation has all of its source operands' E bits reset, but causes an exception itself. The exception is recorded by setting the E bit of the operation's destination register. Additionally the program counter (PC) and any other state information, necessary for the later processing of the exception, is recorded. The actual mechanism used to record the exception is discussed further in Section 6.3. The second case is that the speculative operation has one or more of its source operands' E bits set. For this case, the speculative operation propagates a pending exception down the chain of dependent operations. As a result, the operation itself is not executed, but rather sets its destination E bit and records the exception propagation. The exception propagation information is just some additional data that may be required by the exception handling mechanism to reconstruct the chain of dependent operations.

6.1.2 Execution of a nonspeculative operation

Nonspeculative operations execute in a conventional manner when the E bits of all their source operands are reset. For this case, the operation computes its result, and writes that into the destination register. Additionally, the destination register's E bit is cleared. If an exception occurs for such an operation, traditional exception processing is used. The exception is immediately signaled and the handler is invoked. The case where the E bit of one or more source operands is set for a nonspeculative operation occurs when a pending speculative exception requires processing. The exception is therefore signaled using the recorded state information. If multiple source registers have their E bit set, the exception corresponding to the first operand is reported. Again, the destination register's E bit is cleared for this case. Therefore, any nonspeculative operations which are flow dependence successors of speculative operations behave as a check in the PlayDoh architecture, and an explicit check operation is unnecessary.

A limitation of the PlayDoh static control speculation support is that potentially excepting operations which do not write their results into a destination register may not be speculated.

Therefore, the model does not allow unsafe stores to be speculative. In general, stores were not viewed as important candidates for control speculation in the PlayDoh architecture. Stores most commonly occur at the end of computation chains, so little is gained by increasing the scheduling freedom for stores. Note that those stores that the compiler determines are safe operations may be speculated when profitable just as with any operation type.

The compiler has several important responsibilities in the PlayDoh control speculation model. Although, the specific requirements vary based on the mechanism used to record and handle speculative exceptions (see Section 6.3), the generalized compiler support is discussed here. First, the compiler must ensure that a check is present for every potentially excepting operation which is speculated. The check must directly or indirectly source the speculative operation's destination register, and it must be located in the speculative operation's home block. Second, the compiler must correctly mark all operations which are control speculative by setting the S bit. The execution semantics, as previously described, are dependent on the value of the S bit. Finally, the compiler must preserve any program variables which are required by the exception handling mechanism to recover from a speculative exception. For most schemes, the program variables that require preservation are the source operands in the chain of operations starting from a speculative operation and leading to its check.

6.2 Static control speculation example

The static control speculation support provided in PlayDoh is best illustrated with an example. Throughout this section, the example presented in Figure 6.1 will be used. Figure 6.1(a) shows the C source code for the example loop which traverses a linked list, conditionally incrementing the variable "sum". The corresponding assembly code for a simplified version of the PlayDoh architecture is given in Figure 6.1(b). In Figure 6.1(b), the loop body is unrolled twice and register renaming is applied to remove as many data dependences as possible. There are a total of 18 operations in the unrolled loop body: 6 loads, 4 adds, 4 comparisons, and 4 branches. The assembly mnemonics are of the form: opcode, destination, source1, source2. For example, op1 loads the contents of "r1+0" into "r5".

```
while (ptr != NULL) {
   count++;
   if (ptr->data != NULL)
      sum += ptr->weight;
   ptr = ptr->next;
}
```

(a)

```
L1:   (op1)    l        r5, r1, 0
      (op2)    add      r2, r2, 1
      (op3)    cmp_eq   r6, r5, 0
      (op4)    brct     r6, L3
L2:   (op5)    l        r7, r1, 4
      (op6)    add      r3, r3, r7
L3:   (op7)    l        r11, r1, 8
      (op8)    cmp_eq   r8, r1, 0
      (op9)    brct     r8, L7
L4:   (op10)   l        r15, r11, 0
      (op11)   add      r2, r2, 1
      (op12)   cmp_eq   r16, r15, 0
      (op13)   brct     r16, L6
L5:   (op14)   l        r17, r11, 4
      (op15)   add      r3, r3, r17
L6:   (op16)   l        r1, r11, 8
      (op17)   cmp_ne   r18, r1, 0
      (op18)   brct     r18, L1
L7:
```

(b)

Figure 6.1: Example code segment, (a) C source code, (b) assembly code after unrolling the loop twice and register renaming.

Applying a global scheduling technique, such as trace scheduling [23], to the assembly code exposes the available ILP to the processor. For this example, the underlying processor architecture is assumed to have "adequate" resources, except that at most one branch may be issued each cycle. Also, it is assumed that load operations have a two cycle latency, and all other operations have a one cycle latency. The resulting schedule without any architecture support for control speculation is shown in Figure 6.2(a). For this model, the compiler is only allowed to speculate those operations which are safe. The resulting schedule in Figure 6.2(a) is extremely sparse, taking a total of 14 cycles. The major difficulty with this example is the inability to speculate the load operations, namely op5, op10, op14, and op16. The compiler is not capable of proving these load operations will never cause an exception because of the possibility of walking off the end of the linked list and outside the program's virtual address space. Therefore, the loads must be classified as unsafe. Also, these loads start the critical dependence chain for each basic block in the loop. The inability to speculate these loads in order to overlap their execution with prior operations leads to the serial schedule.

cycle	operations issued
0:	op1, op2, op7
1:	-
2:	op3, op8
3:	op4
4:	op5
5:	-
6:	op6, op9
7:	op10, op11, op16
8:	-
9:	op12, op17
10:	op13
11:	op14
12:	-
13:	op15, op18

(a)

cycle	operations issued
0:	op1, op2, op5(S), op7
1:	-
2:	op3, op8, op10(S), op14(S), op16(S)
3:	op4
4:	op6(C), op12(S), op17(S)
5:	op9
6:	op11, op13(C)
7:	op15(C)
8:	op18(C)

(b)

Figure 6.2: Resulting schedules for example code segment, (a) without any support for control speculation, (b) with PlayDoh control speculation support.

With the support for control speculation provided in PlayDoh, the code motion restrictions for the critical loads are removed. The resulting schedule is shown in Figure 6.2(b). With control speculation support, the schedule length is reduced from 14 to 9 cycles. The scheduler speculates all of the problem loads to reduce the length of the critical dependence chains. In addition to speculating the critical loads, the compiler also speculates two of the comparison operations which use the results of the loads, namely op12 and op17. All the control speculative operations are marked with a "S" in Figure 6.2(b). In order to facilitate delayed exception handling in PlayDoh, each potentially excepting operation which is speculated requires a check located in the operation's home block. The operations which serve as checks are marked with a "C" in the figure. As previously discussed, a check is a nonspeculative operation that is the direct or indirect flow dependence successor of a potentially excepting, control speculative operation. For example, op13 serves as the check for op10 because it indirectly uses *r15* via op12.

To illustrate the proper detection of speculative exceptions, consider an execution scenario for the code in Figure 6.2(b) where op10 causes an exception. In cycle 2, op10 causes an exception. Using the semantics in Table 6.1, the exception tag of op10’s destination register, *r15*, is set and the necessary exception handling information is recorded. The next important event occurs when op12 is executed because it uses the result of op10. Since op12 is speculative and the exception tag of one of its source operands is set, op12 propagates the exception by setting the exception tag of its destination register, *r16*. The execution condition of op10 is still unknown, thus the speculative exception may not be signaled at this point. In the next cycle, the branch (op9), which determines the execution condition of op10, is executed. A taken branch means that control leaves the loop body, so op10 was unnecessarily executed. For this case, the check operation is never executed, and the pending speculative exception is correctly ignored. When the branch falls through, control remains in the loop and, op10 does indeed require to be executed. For this case, the exception must be signaled. The exception is properly detected when op13 is executed. Since op13 is nonspeculative and the exception tag of its source operand, *r16*, is set, an exception is signaled and the exception handler is invoked. The issues associated with processing speculative exceptions are discussed in the next section.

6.3 Exception handling issues

Exception handling in a broad sense refers to the steps the computer system goes through when an exception is detected. For purposes of this discussion, exceptions are limited to operating system generated and handled exceptions. Based on how exceptions are handled, they may be placed into two categories: nonrecoverable and recoverable. Nonrecoverable exceptions result if the program violates some semantic of the system. Common examples of nonrecoverable exceptions are loading from an illegal address or dividing by zero. For these exceptions, program execution is aborted. The other category of exceptions are recoverable exceptions. These exceptions only interrupt program execution for repair, and program execution resumes after exception repair is complete. The user is typically not aware these exceptions even occurred. Common examples of recoverable exceptions are page faults and TLB misses.

With static control speculation, the handling of both types of exceptions needs to be addressed. Speculative nonrecoverable exceptions offer little freedom since they must be handled in an exact manner to prevent spurious exceptions from terminating program execution. These exceptions are handled by aborting program execution after the execution condition of a speculative operation that excepted is known to be true. The other category of exceptions, recoverable exceptions, offers more handling freedom for speculative operations. Since these exceptions are not visible to the user, they may be handled immediately or they may be delayed. The major tradeoff here is performance versus complexity. Handling speculative exceptions at the time of occurrence will likely result in repairing spurious exceptions caused by USOs. As a result, performance is sacrificed by handling unnecessary exceptions. Delaying recoverable exceptions removes the performance overhead since only necessary exceptions will be repaired. However, since the exception is delayed, operations which directly or indirectly use the result of the excepting operation may have already executed. Therefore, these operations must be re-executed after the exception is repaired. This undoubtedly increases the complexity of handling speculative recoverable exceptions. For purposes of this discussion, it will be assumed that recoverable exceptions are delayed.

One effective exception handling approach for the PlayDoh model is to use recovery blocks. A recovery block is a sequence of operations generated by the compiler that are used in the treatment of an excepting speculative operation. Each check operation has a recovery block associated with it. The recovery block will be entered whenever a check detects a pending speculative exception. The recovery block contains the operations necessary to regenerate the speculative exception. In addition, any operations which require re-execution if the exception condition is repaired are contained in the recovery block. The major advantage of using recovery blocks is that the execution condition of the excepting speculative operation is now known to be true. Therefore, the exception may be regenerated as a nonspeculative exception and conventional exception handling (whatever would have been used in the original non-speculative version of the program) may be used. For non-recoverable exceptions, the program is terminated after the speculative exception is regenerated. For recoverable exceptions, the exception is regenerated, program state is repaired by the system, and execution resumes in the recovery block to re-execute any dependent operations which use the result of the excepting operation.

In the simplest recovery block scheme, each check has its own specialized recovery block. The recovery block consists of all unsafe speculative operations for which the check detects pending exceptions. In addition, the recovery block contains all of the speculative operations which are the flow dependence successors of each of the unsafe speculative operations that is a flow dependence predecessor of the check operation. At the end of the recovery block, a return-from-interrupt (rti) operation is placed which returns execution from the recovery block to the appropriate location. The mechanisms used for resuming execution at the correct location of the rti operation are the same as those used for accomplishing conventional interupt returns. For the example scheduled in Figure 6.2(b), the recovery block for the check, op13, would consist of the following:

op10' (copy of op10)
op12' (copy of op12)
op13' (copy of op13)
rti (exit exception handling mode and reenter user code)

Considering the execution scenario of Figure 6.2(b) previously discussed where op10 excepts and requires re-execution, the recovery block is entered when op13 is executed. The exception tag of one of the source operands for op13, r16, is set, therefore it transfers control to the appropriate recovery block. In the recovery block, the operations are not marked as speculative. As a result, when op10' is executed, the exception condition is regenerated. If the exception is nonrecoverable, program execution is aborted. Whereas, if the exception is recoverable, the program state is repaired and execution continues. The next operation, op12', correctly re-executes a dependent operation that had previously been executed. Next a copy of the check, op13', is re-executed to complete the set of operations which use the result of op10. Finally, the rti operation is executed to exit the recovery block and resume execution in the scheduled loop segment after op13.

The details of exception handling with recovery blocks in PlayDoh is beyond the scope of this chapter. Issues, such as generating recovery blocks, combining recovery blocks, transferring control to recovery blocks, and preserving source operands of all the operations in the recovery blocks, are open problems. The interested reader is referred to [5] for one approach to exception handling with recovery blocks. Other approaches to exception handling for PlayDoh are also possible. Inline recovery has been proposed to avoid the code expansion overhead of recovery block schemes [42]. The major issue with inline recovery is the increased compiler complexity to accomplish the proper re-execution during exception recovery. Alternatively, hybrid schemes may also be possible to balance the effects of code expansion and compiler complexity.

6.4 Discussion

In this section, the support for static control speculation in the PlayDoh architecture has been described. The purpose of static control speculation is to increase the compiler's freedom to overlap operations. Compilers are typically limited by the control dependences imposed by branches. Static control speculation allows the compiler to selectively ignore control dependences and allow operations to move across branches. As a result, compilers can employ aggressive global scheduling techniques to expose higher levels of ILP. Traditional superscalar processors do not use static control speculation. Rather, a combination of dynamic scheduling and dynamic speculation is utilized to achieve operation overlap across branches at run-time. Superscalar processors rely extensively on these dynamic features to expose ILP. Static control speculation allows the compiler to effectively take over many of the scheduling responsibilities for the processor. Therefore, ILP processors can be built without complex hardware support for dynamic scheduling and dynamic speculation.

The static control speculation model provided in PlayDoh offers two distinct advantages. The biggest advantage is the ability to support generalized control speculation of unsafe operations. With the PlayDoh model, the compiler is able to simultaneously speculate operations along multiple paths of control. Additionally, the distance that an operation may be speculated is not limited by the architecture. The combination of these features enables the compiler to utilize powerful code motion and scheduling strategies. A second advantage is the relatively small hardware overhead to support static control speculation with proper exception handling. Most alternative strategies sacrifice exception handling to enable generalized control speculation [18, 41]. Other strategies that have been proposed to handle speculative exceptions, require significant hardware overhead and/or impose limitations on the distance an operation may be speculated [60, 8].

There are many open problems that need to be investigated in the area of static control speculation. One of the most important areas is speculative exception handling. In Section 6.3, the recovery block model was briefly described. However, many of the tradeoffs and detailed issues are not understood for recovery blocks or alternative exception handling strategies. Future research is required in static control speculation to answer these questions.

7 Data Speculation

Potential dependences between load and store operations can prevent a compiler from exploiting ILP in VLIW or superscalar architectures. Through static analysis of the address expressions of memory operations, a compiler can attempt to disambiguate memory references. If compile-time disambiguation is successful and indicates that two memory operations reference distinct memory locations, the compiler can remove dependences between the two operations. This results in more opportunities for code motion and enhances ILP. Both the load operations and the computation dependent on the loads can move above the independent store operations. On the other hand, if compile-time memory disambiguation is unable to prove that the two memory accesses are to distinct memory locations, the compiler must conservatively assume that the two references may be to the same location. This results in a greater number of sequentialized memory accesses, degrading the ILP of a program.

Superscalar architectures overcome this limitation by performing out-of-order execution of memory accesses. Simple superscalar implementations expose a limited amount of ILP by executing potentially dependent memory instructions concurrently and sequentializing memory accesses when a conflict is detected. A **memory access conflict** occurs among two or more memory accesses when they potentially access the same memory location. Aggressive implementations of superscalar architectures which employ out-of-order execution can use more sophisticated hardware for run time memory disambiguation. By comparing addresses of memory operations in flight, the hardware can determine which load operations are independent of executing store operations and allow early execution of load operations [33].

Due to the limitations of compile time program analysis, run-time memory disambiguation can be more accurate. If the code scheduled by a compiler is to reflect the record of execution (ROE), then the compiler has to consider the worst case of all possible constraints. In the case of memory dependences, the worst case constraints on memory operation ordering can be highly sequential. The PlayDoh architecture provides a set of primitives which permit a compiler to ignore selected memory dependences, while relying on the hardware to check whether those ignored dependences have been violated at run-time. This results in a schedule which corresponds to an optimistic ROE. In the cases where the actual dependences differ from those assumed by the compiler, a performance penalty is paid to honor these additional constraints. Because there is significant perfomance penalty for ignoring memory dependences which occur frequently, a compiler should be selective in choosing the subset of dependence constraints to be ignored. Techniques like memory dependence profiling can provide statisitical information to a compiler to identify memory dependences which are not likely to occur during execution. This results in the compile time schedule very closely approximating the ROE.

The PlayDoh architecture supports two run-time memory disambiguation mechanisms. First, memory ports are defined as a prioritized resource, i.e., in case there is a conflict between memory operations in the same instruction, the memory operations are executed in their left to right order. This mechanism achieves an effect similar to the concurrent memory operation issue capability of superscalar implementations, and takes a small step towards some of the parallel dependence checking complexities of superscalar processors. The difference, here, is that the parallel dependence checking need only be done across the memory operations, which are relatively few in number.

Second, the architecture exposes the hardware support for detection of memory access conflicts to the compiler by providing opcodes which access the address comparison hardware. This allows the compiler to decide which memory operations need to be monitored for memory conflicts and then delegate to the run-time hardware the task of determining if an address conflict occurs within a window of execution marked by these special operations, and react depending on the outcome. These mechanisms allow a compiler to make optimistic assumptions concerning memory dependences. The compiler can then aggressively schedule and optimize code for the most common case without precise memory disambiguation information.

`r3 = ADD(r1,r2)`		
`S(a1,r3)`		
`r4 = L(a2)`		
`r5 = ADD(r1,r4)`		

Figure 7.1: An example of memory dependency constrained schedule

7.1 Memory port priorities

Consider the sequential schedule of operations in Figure 7.1 where load operations have a two cycle latency and other operations have unit latency.

When the compiler can not disambiguate the two addresses `a1` and `a2`, it needs to schedule the load operation one cycle after the store operation. The total time it takes to execute these four operations is then five cycles. Using PlayDoh's support for prioritized memory ports, the compiler can schedule the load operation in the same cycle as the store operation. Using the left-to-right priority semantics, the load operation has to be scheduled to the right of the store operation. The memory operations take effect as if they are executed in left to right order in an instruction. As a result, the total schedule length for the same set of operations is four cycles as shown in Figure 7.2.

`r3 = ADD(r1,r2)`		
`S(a1,r3)`	`r4 = L(a2)`	
`r5 = ADD(r1,r4)`		

Figure 7.2: Reducing schedule length using memory port priority

During program execution, memory access hardware can detect the existence of an address conflict between the two memory operations and sequentialize the accesses. The address comparison can be conservative. The performance implication is that, in the case where two addresses `a1` and `a2` do not conflict, the instruction sequence executes in four cycles. In the case there is a conflict, one or more extra cycles may be needed for sequentialized memory access. Figure 7.3 illustrates the ROE in the case where memory accesses are sequentialized. The shaded issue slots in this figure illustrate the wasted resources due to the memory access conflict.

r3 = ADD(r1,r2)		
S(a1,r3)	▨	
▨	r4 = L(a2)	▨
r5 = ADD(r1,r4)		

Figure 7.3: Illustration of stall cycles in ROE in case of memory address conflict

If memory address conflicts are known to be frequent, the compiler can accommodate this by scheduling problematic operation pairs in different instructions so that the extra cycle(s) added to the schedule can be overlapped with other computation. This results in a ROE identical to the schedule.

7.2 Description of data speculative loads

Port priorities allow a compiler to reduce the latency of a potential memory dependence to zero cycles. However, in cases where a potential memory dependence is not likely to occur at run-time, it is desirable to speculatively execute a load operation before a store operation that it may be dependent on. Superscalar processors with out of order execution capability perform this type of re-ordering at run-time if the addresses of the store and load operations are known and are found to be different. PlayDoh provides architectural support for run-time memory disambiguation which enables compile time load speculation.

The architectural support provided in PlayDoh for run-time memory disambiguation is similar to the memory conflict buffer proposed by Chen [13]. Silberman et. al. [58] also describe a similar mechanism. The key idea is to decouple a load operation from its dependences on store operations. This is achieved by replacing a load operation with two operations, one load operation which is freed from its dependences and a check operation which at runtime checks if any of the ignored dependences existed. If none of the ignored dependences existed during the execution, the check is successful. In the case one or more of the ignored dependences existed, the check fails and the machine state is updated to reflect the program execution which honors the dependences.

This mechanism is called "data speculation". A data speculative load operation executes like a regular load operation except that the data deposited to its destination register may not be correct. Execution of a corresponding check operation ensures that the data in the destination register is correct.

PlayDoh provides two kinds of architectural support for data speculation of memory operations. The first kind enables the compiler to move a load operation above multiple, potentially aliased stores. The two operations provided are the LDS, data speculative load, operation and the LDV, data verify load, operation. These operations are used as a pair with identical source and destination fields. The LDV operation performs the check operation described above. In the case the check is successful, LDV operation does nothing. In the case of the check fails it ensures that the destination register contains correct data by reloading it from memory. During this load operation, the program execution is frozen and the pipeline is stalled.

The second kind of architectural support enables the compiler to move both a load operations and the operations which are data dependent on it above potentially aliased stores. The LDS operation once again performs the data speculative load operation. In the case of an alias, the machine state that needs to be corrected is not limited to the destination register of an LDS operation. In order to perform arbitrary state update PlayDoh provides a BRDV, data verify branch, operation. The BRDV operation checks whether an alias has occurred and, if that is the case, branches to a block of code constructed by the compiler for machine state fixup. This code would reexecute the load that was speculated and all the computation that used the result of the load data speculatively.

The PlayDoh technical report [36] defines the semantics of these operations. The next two sections illustrate the usage of LDS/LDV and LDS/BRDV operation pairs to shorten critical path length.

7.3 Data speculation examples

For both examples, we will use the code segment in Figure 7.1 where the store operation may alias with the load operation.

7.3.1 LDS/LDV use example

Figure 7.4 shows the code generated for the example using an LDS/LDV operation pair. The original load operation is replaced by an LDS and a corresponding LDV operation. The LDS operation does not depend on the store operation and can move up in the schedule. The LDV operation has the same scheduling constraints as the original load operation. The LDS operation loads the data from memory location `a2` into `r4.` In the case where the store does not alias with the load, the LDV operation does nothing. The add operation which uses `r4` uses the value loaded by the LDS operation. Note that the use of `r4` is scheduled two cycles later than the LDS operation - the assumed load latency.

`r3 = ADD(r1,r2)`	`r4 = LDS(a2)`	
`S(a1,r3)`	`r4 = LDV(a2)`	
`r5 = ADD(r1,r4)`		

Figure 7.4: Use of LDS/LDV operations to move a load above a store

Figure 7.5 displays the ROE for the case where the store operation aliases with the LDS operation, as a consequence of which the data loaded into `r4` by the LDS operation is incorrect. Due to the prioritization between memory operations the execution of the LDV operation is delayed one cycle after the store since they are both to the same address. The LDV operation detects the aliasing between the LDS and the store operation, and re-loads the data from memory to `r4`. Since the LDV operation is where the original load operation used to be, i.e. after the store operation, the data loaded from memory has the correct value. Since the load takes two cycles, an additional stall cycle is interposed after the LDV operation. When the load completes the ADD operation can execute with correct input data.

Note the large number of unavailable issue slots (shown greyed out in Figure 7.5) when the LDS and the store alias, in comparison to the schedule in Figure 7.1 which, too, takes five cycles. If the probability of a1 being equal to a2 is low, it is preferable to use the schedule of Figure 7.4, otherwise the schedule of Figure 7.1 is better.

r3 = ADD(r1,r2)	r4 = LDS(a2)	
S(a1,r3)	▨	
▨	r4 = LDV(a2)	▨
▨	▨	▨
r5 = ADD(r1,r4)		

Figure 7.5: Illustration of stall cycles for data reload in case of a memory access conflict

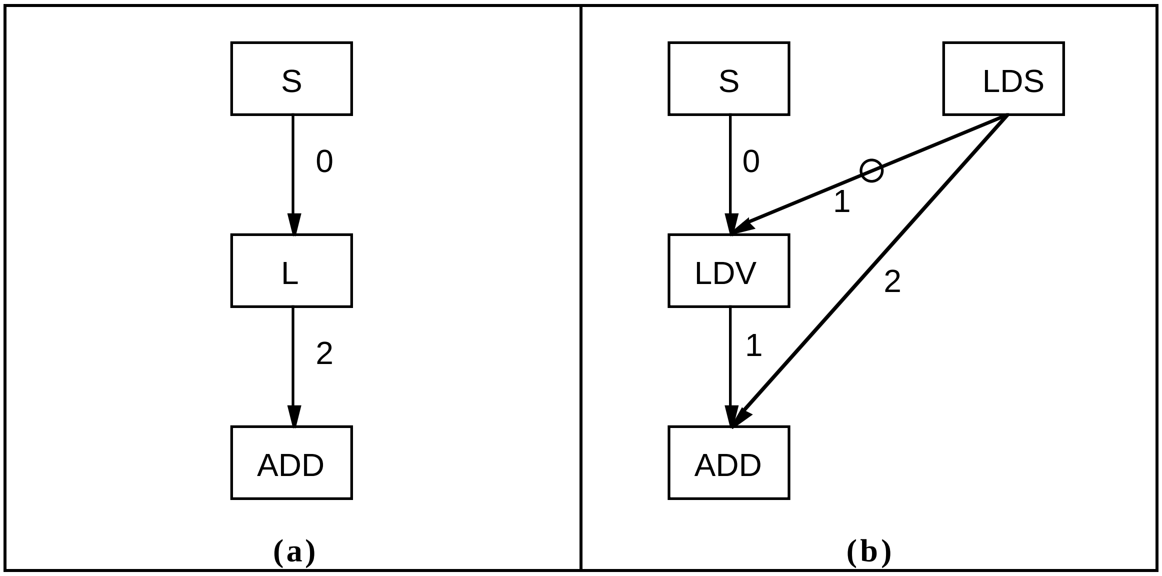


Figure 7.6. (a) Original dependence graph. (b) Dependence graph after data speculation.

Figure 7.6(a) illustrates the dependences of the original code. There is a potential flow dependence from the store to the load operation and there is a flow dependence from the load operation to add operation. Using prioritized memory operations, the store to load dependence latency can be zero cycles. The load to add latency is the latency of the load operation.

Figure 7.6(b) illustrates the dependences of the code with the LDS/LDV pair. The LDS is not dependent on the store operation. This allows early scheduling of the LDS operation to accommodate the memory access latency. In the previous figure, the load to add latency was a contributor to the schedule length. Using data speculation, the load latency can be overlapped with other computation. In Figure 7.6(b), the load to add latency is replaced by the LDV to add latency. Since the LDV operation can be implemented to have as little as a one cycle assumed latency, the add operation can be scheduled in the cycle following the store operation which, in the statistically likely case, results in a shortened length for the path through the store operation.

`r4 = LDS(a2)`		
`r3 = ADD(r1,r2)`	`r5 = ADD(r1,r4)`	
`S(a1,r3)`		
	`BRDV(r4)`	
Operations to check speculative exceptions		

(a)

`r4 = L(a2)`		
`r5 = ADD(r1,r4)`	`RTI`	

(b)

Figure 7.7 (a) Use of LDS/BRDV to move a load and its use above a store
(b) Fixup code to restore machine state from a memory access conflict.

7.3.2 LDS/BRDV use

The LDS/LDV pair of operations allow the motion of load operations across potentially aliasing stores, thereby reducing schedule length. However, the computation that is dependent on an LDS operation is still trapped under the potentially aliasing stores. The LDS/BRDV permits the compiler to move not only a potentially conflicting load operation above a store, but also any computation that uses the result of the speculative load. In the case where the LDS operation aliases with the store operations, the BRDV operation allows the execution of fixup code which re-executes the load operation and the other computation that was data speculatively executed. The purpose and the construction of the fixup code for data speculation is very similar to the recovery block in the case of control speculation. The fixup code contains all operations which are present in the program slice which starts with the data speculated load operation and which moved above the BRDV.

Figure 7.7(a) shows the code generated for the example using the LDS/BRDV operation pair. The original load operation is replaced by an LDS and a corresponding BRDV operation. The LDS operation does not depend on the store operation and can move up in the schedule. The BRDV operation, like the LDV operation in the previous section, has the same scheduling constraints as the original load operation. The LDS operation loads the data from memory location `a2` into `r4`. The add operation which uses `r4` is executed speculatively, i.e. before `r4` is known to contain the correct value. In the case where the store does not alias with the load, both the values in `r4` and in `r5` are correct, hence the BRDV operation does nothing.

When the BRDV operation detects an alias between the load and the store operation, it causes an exception which results in control being transferred to the entry point, specified by the BRDV operation, for the fixup code (Figure 7.7(b)). This block contains the sequence of operations which recompute the correct machine state. In general the fixup code would contain all the operations dependent upon the LDS operation that have been moved above the BRDV operation from below. Once the program state is restored, a return from interrupt, RTI, operation can continue the execution of the program.

In the case where LDS/BRDV pair is used for data speculation, the exceptions generated by data speculative operations other than the LDS operation need special attention. Since these operations are executing with speculative operands, any exceptions that are generated by them need to be considered speculative. The techniques discussed in Section 6, for delayed processing of control speculative exceptions is applicable here. Any unsafe operation that executes data speculatively needs to generate speculative exceptions which can be checked after verification of computation by a BRDV operation. In the case where an alias is detected, the fixup code is executed. Any exception that occur is then handled in the fixup code. In the case where no alias is detected, it is necessary to check for speculative exceptions explicitly. A recovery block can then be utilized to recover from any exceptions that are raised.

The fixup code for data speculation and the recovery block for handling speculative exceptions are very similar. One important difference between them is that, the recovery block does not contain the load operation which was executed data speculatively whereas the fixup code does. This is because the input operands of an LDS operation which starts a data speculative program slice are not speculative. Any exceptions raised by such an LDS operation can be handled immediately.

There are a number of open problems in the area of data speculation. One important problem is the generation of fixup code for LDS/BRDV operations and the corresponding recovery block generation for detection of speculative exceptions. Since data speculation allows a compiler to move code around more freely, it is also likely that some data speculative operations move across conditional branches resulting in both control and data speculative operations. Future research is required to address code generation issues concerning interactions between control and data speculation.

8 Programmatic management of the data cache hierarchy

The memory hierarchy has a large and growing impact on overall performance. This section describes architecturally visible levels of the memory hierarchy, motivates the need for architectural mechanisms to control data movement in the memory hierarchy, and describes the actual mechanisms available in the PlayDoh architecture.

8.1 Motivation

Data cache miss stalls account for a large and growing fraction of overall execution time, especially in large commercial database applications and in scientific/numeric applications that work with extremely large arrays. Data cache stalls are proportional to the latency of a data cache miss and the number of misses in a program. The miss latency in processor cycles is increasing as processor speed increases relative to that of main memory. The number of misses (or equivalently, the miss ratio) is also increasing with larger data sets. The miss ratio cannot be reduced sufficiently by increasing cache size because of cycle time, size and cost constraints. Default hardware cache management strategies can sometimes exacerbate miss ratios. For instance, a data stream with little temporal locality can displace other data with better temporal locality. The increasing impact of cache misses on overall performance and the potential for better performance using software strategies motivates architectural mechanisms for cache management.

8.2 The architecture of PlayDoh's memory hierarchy

The architecturally visible levels of the PlayDoh architecture are as follows: at the first-level closest to the processor, there is a conventional first-level cache and a data prefetch (or streaming) cache. At the next level, there is a conventional second-level cache which is also architecturally visible. Beyond the second-level, there may be further levels of caching or just main memory, but these levels are not architecturally distinct to the processor. The exact structure of each cache depends upon the implementation and is not architecturally visible. The conventional first- and second-level caches may be either a unified (instruction and data) cache or separate instruction and data caches.

The data prefetch cache is used to prefetch large amounts of data having little or no temporal locality while bypassing the conventional first-level cache. When such prefetching is employed, the first-level cache does not have to replace other data potentially having better temporal locality. Typically, the data prefetch cache is a fully-associative cache much smaller in size than the first-level cache.

The PlayDoh architecture supports the standard set of load and store operations. For integer values, the operation repertoire include byte, half-word and word operations at all levels of the memory hierarchy. For floating-point values, the repertoire includes both single-precision and double-precision operations, again at all levels of the memory hierarchy. All load and store operations have a predicate input that guards their execution. Load operations can be issued speculatively, but there are no speculative stores. Memory operations within an instruction are executed in an order that is consistent with their sequential left-to-right order of execution. The PlayDoh architecture also supports run-time memory disambiguation through three related families of operations, called data speculative load (LDS), data verify load (LDV), and data verify branch (BRDV), which were described in more detail in Section 7.

The standard memory operations take a fully-resolved virtual memory address as an argument. Post-increment operations are similar to the standard load/store operations, but they have the additional capability to compute new addresses for subsequent load/store operations. The new address is the sum of the accessed memory address and a displacement, which can be either a literal or a value stored in a GPR. The new address is deposited in the specified destination register, which may or may not be identical to the source address register. Pre-increment operations are not supported because they disrupt the flow of operations through the memory pipeline.

8.3. Novel PlayDoh features

The PlayDoh architecture provides architectural mechanisms to explicitly control caches. These mechanisms selectively override the usual simple default hardware policies. The mechanisms are used when the compiler has sufficient knowledge of the program's memory accessing behavior and when significant performance improvements are obtainable through software control. If not, the default hardware policies apply.

8.3.1 Source cache (and latency) specifier

There are two specifiers associated with each load operation. The first specifier, the latency and source cache specifier, is used by the compiler to indicate its view of where the data is likely to be found. Consequently, it also implicitly specifies the load latency assumed by the compiler. The available latency specifier choices are V1, C1, C2, C3 for the prefetch cache, first-level cache, second-level cache and other levels of the memory hierarchy respectively. Subsequently, we refer to a load with a C1 specifier as a short-latency load and a C2 specifier as a long-latency load.

Before describing the latency specifier further, we distinguish between a latency-stalled machine and a use-stalled machine. In a latency-stalled machine, each load specifies a latency and the processor stalls if the data is not available at the prescribed latency, regardless of whether or not the data is, in fact, used immediately by an operation in the instruction being issued (Section 2). In a use-stalled machine, the processor stalls on coming to an instruction which uses data which has not yet returned from memory. In the spectrum from superscalar to VLIW machines, latency-stalling is philosophically closer to the VLIW end whereas use-stalling is closer to the supercalar end.

In a latency-stalled machine, the compiler is responsible for ensuring that a load operation and its use operations are separated by at least the latency of the load. Otherwise, when the use operation is issued before the load completes, the use operation may incorrectly use the old value in the register being loaded. The choice of the latency specifier has important performance implications. When a load is likely to miss in the first-level cache and when sufficient parallelism exist to pad the slots between a long-latency load and its uses, choosing a long-latency load can eliminate potential stalls. On the other hand, when a load is likely to hit in the first-level cache, choosing a short-latency load can eliminate potential empty operation issue slots.

In a use-stalled machine, the processor is responsible for stalling the machine to ensure that a use gets the value that is being loaded. Therefore, in a use-stalled machine, the latency specifier has a bearing on neither correctness nor performance. However, as in the latency-stalled machine, the compiler must control the separation between a load and its uses to reduce stall cycles.

8.3.2 Target cache specifier

The second specifier, the target cache specifier, is used by the compiler to indicate its view of the highest level to which the loaded data should be promoted for use by subsequent memory operations. This specifier can take on the same values as the source cache specifier: V1, C1, C2, C3. This specifier is used by the compiler to control the cache contents and manage cache replacement.

A store operation can only have a target cache specifier. As with the load operations, it is used by the compiler to specify the highest level in the cache hierarchy at which the stored data should be installed to be available for use by subsequent memory operations.

Non-binding loads (conventionally referred to a prefetches) cause the specified promotion of data in the memory hierarchy without altering the state of the register file. In a non-binding load, the destination of the load is specified as register 0. (In PlayDoh, this is a register which is hardwired to contain the value 0.) In order to distinguish between the two first-level caches, we use the terms pretouch and prefetch, respectively, to refer to non-binding loads that specify the target cache as C1 and V1. In contrast to regular loads, prefetches and pretouches bring the data closer to the processor without tying up registers and thereby increasing register pressure.

8.4 Usage and compiling

The source cache or latency specifiers are used by the compiler to schedule loads to reduce cache miss stalls while containing any increase in schedule length, operation count, or register pressure. The target cache specifiers are used by the compiler to reduce misses in the first- and second-level caches by controlling the contents of the caches at various levels.

Specifically, the compiler may route large data streams with little temporal locality through the prefetch cache to prevent replacement of other data with greater temporal locality from the first-level cache.

8.4.1 Hiding the load latency

Our analysis indicates that the behavior of individual load operations in integer as well as floating-point benchmarks is favorable to compiler-directed cache management [3, 2]. Firstly, a small number of load operations are responsible for a majority of the data cache misses in a program. Thus, the number of load operations that have to scheduled with the miss-latency are small, potentially mitigating the effect on schedule length. Secondly, loads tend to have bi-modal behavior with most loads always hitting in the data cache while a second set of loads tend to miss with an individual miss ratio much higher than the global miss ratio. Typically, only a small fraction of accesses are accounted for by the remaining loads with an intermediate miss ratio close to the global miss ratio. This behavior is also promising, because the costs of scheduling with the miss latency are incurred primarily by those loads most likely to benefit from reduced stalls.

In compiler-directed miss-sensitive scheduling, the compiler schedules likely-to-miss (missing) loads with the cache-miss latency and schedules other loads with the cache-hit latency. Since the experimental analysis indicates that loads tend to have a bi-modal behavior, let us first consider the case where some loads are known to always miss the cache and other loads always hit the cache. In a latency-stalled machine, the compiler uses the long-latency (C2) specifier for the missing loads and the short-latency (C1) specifier for the other loads. As with other operations, the scheduler ensures that uses are scheduled after the prescribed latency of the loads. In a use-stalled machine, the compiler may use the C1 specifier for all loads, but ensure that uses of missing loads are scheduled after the miss latency. Since a use-stalled processor does not stall till the use operation, the missing loads do not cause stalls because their uses are scheduled at the miss latency. In a use-stalled processor, the compiler implicitly specifies an arbitrary latency by appropriately scheduling the uses of the load.

Now consider the problem of scheduling loads with an intermediate miss ratio for a latency-stalled machine. The higher latency of a long-latency load can increase the critical path of a scheduling region and increase schedule length. Also, since the register targeted by the load has to be reserved from the issue of the load till its use by a subsequent operation, register lifetimes are lengthened. The choice between short-latency (C1) and long-latency (C2) for these loads depends on the tradeoff between reduced stalls that result from the use of a long-latency load versus the increased schedule length and register pressure associated with long-latency loads. The benefit of reduced stalls is directly proportional to the estimated miss ratio. Thus, the compiler considers all these factors in deciding on a suitable latency specifier. The factors involved in choosing a latency for these loads is similar for use-stalled machines. Since the load latency is implicitly specified by the separation between a load and its first use, the compiler may choose any latency from the short- to the long-latency. For instance, the compiler may determine the slack (or scheduling freedom) associated with a load in choosing its latency. Thus, the compiler attempts to choose the maximum possible latency without adversely affecting overall schedule length.

Though our work indicates that miss-sensitive scheduling can be an effective compiler technology, there are still issues left to be resolved. Firstly, missing loads need to be identified prior to or during compilation. Though such loads can be identified in regular matrix-based programs, there are only broad heuristics for integer applications. These heuristics need to be refined further. Currently, we use cache profiling to identify such loads. Secondly, it is not clear whether adequate parallelism and scheduling flexibility is present to schedule missing loads with the cache-miss latency in integer applications. Techniques to relax scheduling constraints associated with missing loads are being developed. For instance, converting a long-latency load to a pretouch liberates it from memory dependence constraints. The compiler may maintain auxiliary data structures to predict load addresses in a timely manner. For instance, if a list is traversed several times, the compiler may maintain a forward pointer with each element that points to an element further down the list that can be brought into the cache. In conclusion, though memory behavior analysis has indicated that miss-sensitive scheduling is promising, there are still significant open issues to be resolved.

8.4.2 Controlling the cache hierarchy

The target cache specifiers are used to control the contents of the caches at various levels. By excluding data streams with little temporal locality form the highest levels of the memory hierarchy, and by removing data streams from the appropriate level when it is last used, software cache management strategies using the target cache specifiers can reduce data traffic between levels of the memory hierarchy and thereby improve miss ratios.

Misses may be divided into four components: compulsory, capacity, mapping and replacement. Replacement misses are the additional misses in a practical cache managed using a replacement policy such as LRU over a similar cache managed using the unrealizable OPT replacement policy. In a typical two-way set-associative cache, replacement misses contribute approximately 20% of the misses on the SPEC benchmarks [62]. Hence, there is the potential to reduce miss ratios by up to 20% using target cache specifiers to override the default LRU replacement strategy.

In order to further characterize the benefit of target-cache specification, we describe the following profile-driven optimization [62]. A profiling run was used to determine the reuse ratio of each load/store instruction. The reuse ratio is the number of times the data referenced by a load/store instruction is reused while it is resident in the data cache over the execution count of that instruction. A threshold on the reuse ratio, can be used to label instructions with a target specifier of C1 or C2. In the experiments conducted, instructions were labeled either as high or low reuse. Data touched by low reuse instructions were labeled with a lower priority in the cache and the replacement policy preferentially removed this data from the first-level cache.This scheme reduced miss ratios by an average of 8.6% for the SPEC89 benchmarks on reasonable on-chip first-level cache configurations.

9. Conclusion

Over the past few years, it has become apparent that the choice between VLIW and superscalar is a false one. Neither approach, in its most extreme form, is desirable and most machines of interest will almost surely lie somewhere on the continuum between the two extreme versions of these approaches. However, there is a real distinction that can be made between the philosophical positions of of the proponents of these two approaches which is quite reminiscent of the differing positions taken by the RISC and the CISC camps. Early VLIW processors evolved from a belief that high levels of ILP were best achieved by making the expensive decisions at compile-time, thereby permitting the hardware to be relatively simple. This is still the philosophical underpinning of PlayDoh. Most of the architectural features in PlayDoh are there to facilitate the compiler making decisions which would otherwise have to be made by the microarchitecture at run-time.

The two features that best distinguish a VLIW processor from any other kind of ILP processor are MultiOp and NUAL. In this article, we have articulated the benefits of these two features, their problems, and some possible solutions to those problems. In our opinion, MultiOp is the only practical approach for issuing large (more than eight) operations per cycle. We also believe that latency stalling, which implies NUAL, is preferable to use stalling at high levels of ILP since the decision to stall can be made locally by each processor. However, a conclusive arguement in favor of NUAL and latency stalling requires analysis at the detailed circuit level.

The rest of the PlayDoh features could, in principle, be used in either VLIW or superscalar processors, although some of them are better motivated by, or are more synergistic with, VLIW and its absence of run-time scheduling and dependence checking. For instance, predicated execution poses difficulties for processors with interlocking or any form of dynamic scheduling. The problem is that predicated execution squashes an operation if its predicate is false and allows it to execute normally otherwise. Thus, the value of the predicate determines whether an operation modifies its destination register. All of the dynamic scheduling approaches available to a UAL architecture require knowledge of which register is modified by each operation. At the time of issuing an operation, if the value of the predicate is unknown, instruction issue must be stalled.

Alternatively, instruction issue can proceed using the conservative assumption that the predicate is true and that the destination register, therefore, has a pending write to it. Subsequently, if the predicate turns out to be false, the invalid bit for the destination register must be reset, but the resulting temporary and spurious flow and output dependences will have compromised performance. Register renaming, cannot be used since subsequent operations will be waiting for the tag corresponding to an operation that will never execute. Latency stalling, which applies only to NUAL architectures, does not depend upon knowledge of which operation is modifying which register. It merely focuses on the discrepancy between the actual and the assumed latencies. Consequently, the presence of predicated execution has no impact upon it.

One issue that we have not touched upon in this article is that of object code compatibility. In view of the widespread perception that this is the Achilles' heel of VLIW processors, a couple of points are worth making. First, the run-time mechanisms for performing dynamic scheduling, and thereby achieving object code compatibility, are now understood for VLIW processors [50] as they have been for superscalar processors. So, this is a non-issue. Second, dynamic scheduling is almost as expensive for VLIW processors as it is for superscalar processors; the complexity of dynamic scheduling is primarily a function of the number of operations that the processor attempts to issue per cycle, and less a function of the nature of the ILP processor.

Third, the importance of the debate, over whether static scheduling or dynamic scheduling is better, is greatly overestimated. In addition to scheduling, a high quality ILP compiler must perform a number of optimizations and code transformations, such as if-conversion, control speculation, data speculation and critical path reduction, all of which are quite machine-specific and cannot be performed at run-time. Dynamic scheduling alone cannot solve the problem of executing code that has been compiled for one ILP processor on another processor and at performance levels that are comparable to those achieved with re-compilation. In view of these considerations, the choice appears to be between achieving object code compatibility via dynamic scheduling, but at low levels of ILP, or to attain high levels of ILP, but to alter the notion of object code compatibility to include concepts such as dynamic translation [15].

References

1. (Special issue on the System/360 Model 91). IBM Journal of Research and Development 11, 1 (January 1967).

2. S. G. Abraham and B. R. Rau. Predicting Load Latencies Using Cache Profiling. Technical Report HPL-94-110. Hewlett-Packard Laboratories, November 1994.

3. S. G. Abraham, *et al.* Predictability of load/store instruction latencies. Proc. 26th Annual International Symposium on Microarchitecture (December 1993), 139-152.

4. J. R. Allen, K. Kennedy, C. Porterfield and J. Warren. Conversion of control dependence to data dependence. Proc. Tenth Annual ACM Symposium on Principles of Programming Languages (January 1983), 177-189.

5. D. I. August, B. L. Deitrich and S. A. Mahlke. Sentinel Scheduling with Recovery Blocks. Technical Report CRHC-95-05. Center for Reliable and High-Performance Computing, University of Illinois at Urbana-Champaign, February 1995.

6. G. R. Beck, D. W. L. Yen and T. L. Anderson. The Cydra 5 mini-supercomputer: architecture and implementation. The Journal of Supercomputing 7, 1/2 (May 1993), 143-180.

7. E. Bloch. The engineering design of the STRETCH computer. Proc. Eastern Joint Computer Conference (1959), 48-59.

8. R. A. Bringmann, *et al.* Speculative execution exception recovery using write-back suppression. Proc. 26th Annual International Symposium on Microarchitecture (Austin, Texas, December 1993), 214-223.

9. W. Buchholz (Editor). Planning A Computer System: Project Stretch. (McGraw-Hill, New York, 1962).

10. M. Butler, *et al.* Single instruction stream parallelism is greater than two. Proc. Eighteenth Annual International Symposium on Computer Architecture (Toronto, 1991), 276-286.

11. A. E. Charlesworth. An approach to scientific array processing: the architectural design of the AP-120B/FPS-164 Family. Computer 14, 9 (1981), 18-27.

12. T. C. Chen. Parallelism, pipelining, and computer efficiency. Computer Design 10, 1 (January 1971), 69-74.

13. W. Y. Chen. Data Preload for Superscalar and VLIW Processors. Ph. D. Thesis. Department of Electrical and Computer Engineering, University of Illinois at Urbana-Champaign, 1993.

14. R. P. Colwell, *et al.* A VLIW architecture for a trace scheduling compiler. IEEE Transactions on Computers C-37, 8 (August 1988), 967-979.

15. T. M. Conte and S. W. Sathaye. Dynamic rescheduling: a technique for object code compatibility in VLIW architecture. Proc. 28th Annual International Symposium on Microarchitecture (Ann Arbor, Michigan, November 1995), 208-218.

16. Cydrome. Internal memo. (1988).

17. J. C. Dehnert, P. Y.-T. Hsu and J. P. Bratt. Overlapped loop support in the Cydra 5. Proc. Third International Conference on Architectural Support for Programming Languages and Operating Systems (Boston, Mass., April 1989), 26-38.

18. J. C. Dehnert and R. A. Towle. Compiling for the Cydra 5. The Journal of Supercomputing 7, 1/2 (May 1993), 181-228.

19. K. Ebcioglu. Some design ideas for a VLIW architecture for sequential-natured software, in Parallel Processing (Proc. IFIP WG 10.3 Working Conference on Parallel Processing, Pisa, Italy), M. Cosnard, M. H. Barton and M. Vanneschi (Editor). (North Holland, Amsterdam, 1988), 3-21.

20. J. P. Eckert, J. C. Chu, A. B. Tonik and W. F. Schmitt. Design of UNIVAC-LARC System: I. Proc. Eastern Joint Computer Conference (1959), 59-65.

21. A. E. Eichenberger and E. S. Davidson. Predicated register allocation. Proc. 28th Annual International Symposium on Microarchitecture (Ann Arbor, Michigan, November 1995).

22. J. Ferrante, K. J. Ottenstein and J. D. Warren. The program dependence graph and its use in optimization. ACM Transactions on Programming Languages and Systems 9, 3 (July 1987), 319-349.

23. J. A. Fisher. Trace scheduling: a technique for global microcode compaction. IEEE Transactions on Computers C-30, 7 (July 1981), 478-490.

24. J. A. Fisher and S. M. Freudenberger. Predicting conditional jump directions from previous runs of a program. Proc. Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (Boston, Mass., October 1992), 85-95.

25. J. A. Fisher, D. Landskov and B. D. Shriver. Microcode compaction: looking backward and looking forward. Proc. 1981 National Computer Conference (1981), 95-102.

26. C. C. Foster and E. M. Riseman. Percolation of code to enhance parallel dispatching and execution. IEEE Transactions on Computers C-21, 12 (December 1972), 1411-1415.

27. N. Gloy, M. D. Smith and C. Young. Performance issues in correlated branch prediction schemes. Proc. 28th Annual International Symposium on Microarchitecture (Ann Arbor, Michigan, November 1995), 3-14.

28. R. G. Hintz and D. P. Tate. Control Data STAR-100 processor design. Proc. COMPCON '72 (September 1972), 1-4.

29. W. W. Hwu, T. M. Conte and P. P. Chang. Comparing software and hardware schemes for reducing the cost of branches. Proc. 16th Annual International Symposium on Computer Architecture (May 1989), 224-233.

30. W. W. Hwu, *et al.* The superblock: an effective technique for VLIW and superscalar compilation. The Journal of Supercomputing 7, 1/2 (May 1993), 229-248.

31. W. W. Hwu and Y. N. Patt. HPSm, a high performance restricted data flow architecture having minimal functionality. Proc. 13th Annual International Symposium on Computer Architecture (Tokyo, Japan, June 1986), 297-306.

32. W. W. Hwu and Y. N. Patt. Checkpoint repair for out-of-order execution machines. IEEE Transactions on Computers C-36, 12 (December 1987), 1496-1514.

33. M. Johnson. Superscalar Microprocessor Design. (Prentice-Hall, Englewood Cliffs, New Jersey, 1991).

34. R. A. Johnson and M. S. Schlansker. Analyzing Predicated Code. Technical Report HPL-96-?? Hewlett-Packard Laboratories, 1996.

35. N. P. Jouppi and D. Wall. Available instruction level parallelism for superscalar and superpipelined machines. Proc. Third International Conference on Architectural Support for Programming Languages and Operating Systems (April 1989), 272-282.

36. V. Kathail, M. Schlansker and B. R. Rau. HPL PlayDoh Architecture Specification: Version 1.0. Technical Report HPL-93-80. Hewlett-Packard Laboratories, February 1993.

37. P. M. Kogge. The Architecture of Pipelined Computers. (McGraw-Hill, New York, 1981).

38. J. Labrousse and G. A. Slavenburg. A 50 MHz microprocessor with a VLIW architecture. Proc. ISSCC '90 (San Francisco, 1990), 44-45.

39. J. K. F. Lee and A. J. Smith. Branch prediction strategies and branch target buffer design. Computer 17, 1 (January 1984), 6-22.

40. D. C. Lin. Compiler Support for Predicated Execution in Superscalar Processors. M.S. Thesis. University of Illinois at Urbana-Champaign, 1992.

41. P. G. Lowney, *et al.* The Multiflow trace scheduling compiler. The Journal of Supercomputing 7, 1/2 (May 1993), 51-142.

42. S. A. Mahlke, *et al*. Sentinel scheduling: a model for compiler-controlled speculative execution. ACM Transactions on Computer Systems 11, 4 (November 1993), 376-408.

43. S. A. Mahlke, *et al*. Characterizing the impact of predicated execution on branch prediction. Proc. 27th International Symposium on Microarchitecture (San Jose, California, November 1994), 217-227.

44. S. A. Mahlke, *et al*. Effective compiler support for predicated execution using the hyperblock. Proc. 25th Annual International Symposium on Microarchitecture (1992), 45-54.

45. S. McFarling and J. Hennessy. Reducing the cost of branches. Proc. Thirteenth International Symposium on Computer Architecture (Tokyo, Japan, June 1986), 396-403.

46. A. Nicolau and J. A. Fisher. Measuring the parallelism available for very long instruction word architectures. IEEE Transactions on Computers C-33, 11 (November 1984), 968-976.

47. J. C. H. Park and M. S. Schlansker. On Predicated Execution. Technical Report HPL-91-58. Hewlett Packard Laboratories, May 1991.

48. C. Peterson, J. Sutton and P. Wiley. iWarp: a 100-MOPS, LIW microprocessor for multicomputers. IEEE Micro 11, 3 (June 1991), 26.

49. B. R. Rau. Cydra 5 Directed Dataflow architecture. Proc. COMPCON '88 (San Francisco, March 1988), 106-113.

50. B. R. Rau. Dynamically scheduled VLIW processors. Proc. 26th Annual International Symposium on Microarchitecture (Austin, Texas, December 1993), 80-92.

51. B. R. Rau. Iterative modulo scheduling. International Journal of Parallel Processing 24, 1 (February 1996), 3-64.

52. B. R. Rau, M. S. Schlansker and P. P. Tirumalai. Code generation schemas for modulo scheduled loops. Proc. 25th Annual International Symposium on Microarchitecture (Portland, Oregon, December 1992), 158-169.

53. B. R. Rau, D. W. L. Yen, W. Yen and R. A. Towle. The Cydra 5 departmental supercomputer: design philosophies, decisions and trade-offs. Computer 22, 1 (January 1989), 12-35.

54. E. M. Riseman and C. C. Foster. The inhibition of potential parallelism by conditional jumps. IEEE Transactions on Computers C-21, 12 (December 1972), 1405-1411.

55. R. M. Russell. The CRAY-1 computer system. Communications of the ACM 21 (1978), 63-72.

56. M. S. Schlansker and V. Kathail. Critical path reduction for scalar programs. Proc. 28th Annual International Symposium on Microarchitecture (Ann Arbor, Michigan, November 1995), 57-69.

57. M. S. Schlansker, V. Kathail and S. Anik. Parallelization of control recurrences for ILP processors. International Journal of Parallel Processing 24, 1 (February 1996), 65-102.

58. G. M. Silberman and K. Ebcioglu. An architectural framework for supporting heterogeneous instruction-set architectures. Computer 26, 6 (June 1993), 39-56.

59. J. E. Smith. A study of branch prediction strategies. Proc. Eighth Annual International Symposium on Computer Architecture (May 1981), 135-148.

60. M. D. Smith, M. Horowitz and M. Lam. Efficient superscalar performance through boosting. Proc. Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (Boston, Massachussetts, October 1992), 248-259.

61. G. S. Sohi and S. Vajapayem. Instruction issue logic for high-performance, interruptable pipelined processors. Proc. 14th Annual Symposium on Computer Architecture (Pittsburgh, Pennsylvania, June 1987), 27-36.

62. R. A. Sugumar and S. G. Abraham. Multi-configuration Simulation Algorithms for the Evaluation of Computer Architecture Designs. Technical Report CSE-TR-173-93. Department of Electrical Engineering and Computer Science, University of Michigan, August 1993.

63. J. E. Thornton. Parallel operation in the Control Data 6600. Proc. AFIPS Fall Joint Computer Conference (1964), 33-40.

64. P. Tirumalai, M. Lee and M. S. Schlansker. Parallelization of loops with exits on pipelined architectures. Proc. Supercomputing '90 (November 1990), 200-212.

65. G. S. Tjaden and M. J. Flynn. Detection and parallel execution of parallel instructions. IEEE Transactions on Computers C-19, 10 (October 1970), 889-895.

66. R. M. Tomasulo. An efficient algorithm for exploiting multiple arithmetic units. IBM Journal of Research and Development 11, 1 (January 1967), 25-33.

67. D. W. Wall. Limits of instruction-level parallelism. Proc. Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (April 1991), 176-188.

68. W. J. Watson. The Texas Instruments Advanced Scientific Computer. Proc. COMPCON '72 (1972), 291-293.

69. W. J. Watson. The TI ASC -- a highly modular and flexible super computer architecture. Proc. AFIPS Fall Joint Computer Conference (1972), 221-228.

70. S. Weiss and J. E. Smith. Instruction issue logic for pipelined supercomputers. Proc. 11th Annual International Symposium on Computer Architecture (1984), 110-118.

71. T. Y. Yeh and Y. N. Patt. Alternative implementations of two-level adaptive branch prediction. Proc. Nineteenth International Symposium on Computer Architecture (Gold Coast, Australia, May 1992), 124-134.

72. C. Young and M. Smith. Improving the accuracy of static branch prediction using branch correlation. Proc. 6th Annual International Conference on Architectural Support for Programming Languages and Operating Systems (October 1994).

73. H. C. Young and J. R. Goodman. A simulation study of architectural data queues and prepare-to-branch instruction. Proc. IEEE International Conference on Computer Design: VLSI in Computers, ICCD'84 (Port Chester, New York, 1984), 544-549.