PROCEEDING OF THE 1995 28TH ANNUAL
INTERNATIONAL SYMPOSIUM ON MICROARCHITECTURE, NOV. 29–DEC. 1 (ANN ARBOR, MI)

Dynamic Rescheduling: A Technique for Object Code Compatibility in VLIW Architectures

Thomas M. Conte Sumedh W. Sathaye
Department of Electrical and Computer Engineering
North Carolina State University
Raleigh, North Carolina 27695-7911

Abstract

Lack of object code compatibility in VLIW architectures is a severe limit to their adoption as a general-purpose computing paradigm. Previous approaches include hardware and software techniques, both of which have drawbacks. Hardware techniques add to the complexity of the architecture, whereas software techniques require multiple executables. This paper presents a technique called Dynamic Rescheduling that applies software techniques dynamically, using intervention by the operating system. Results are presented to demonstrate the viability of the technique using the Illinois IMPACT compiler and the TINKER architectural framework.

1 Introduction

Lack of object-code compatibility across generations of a VLIW architecture is an often raised objection to its use as a general-purpose computing paradigm [1]. A program binary compiled for VLIW generation x cannot be guaranteed to execute correctly on generations $x + n$ or $x - n$, for a reasonable value of n. This means that an installed software base of binaries cannot be built around a family of VLIW generations. The economic implications of this problem are enormous, and an efficient solution is necessary if VLIW architectures are to succeed. Two classes of approaches to this problem have been reported in the literature: hardware approaches and software approaches. The hardware approaches include split-issue proposed by Rau [2], and the fill-unit proposed by Melvin, Shebenow, and Patt [3] and extended by Franklin and Smotherman [4]. Although these techniques provide compatibility, they do so at the expense of hardware complexity that can potentially impact cycle time. A typical software approach is to statically recompile the VLIW program from the object file. This approach requires generation of multiple executables, which poses difficulties for commercial copy protection and system administration. This paper proposes a new scheme called Dynamic Rescheduling to achieve object-code compatibility between VLIW generations. Dynamic rescheduling applies a limited version of software scheduling during first-time page faults, requiring no additional hardware support. Making this practical requires support from the compiler, the ISA, the operating system, and a fast algorithm for rescheduling. These topics are discussed in detail below. Results are presented that suggest dynamic rescheduling has the potential to effectively solve the compatibility problem in VLIW architectures.

1.1 The VLIW compatibility problem

IALU (1 cycle latency)			IALU (1 cycle latency)			MUL (3 cycle latency)			LD (2 cycle latency)		ST (1 cycle latency)	
d	*s*	*s*	*d*	*s*	*s*	*d*	*s*	*s*	*d*	*addr*	*addr*	*s*
A:R1	R2	R3				D:R6	R2	R3	B:R4	X		
						E:R7	R1	R3				
C:R5	R4	R3	G:R8	R4	R3							
H:R9	R6	R4										
											F: X	R7

Figure 1: Scheduled code for original VLIW machine.

The compatibility problem is illustrated in the following example. Figure 1 shows an example VLIW schedule for a machine with two integer ALUs, and

one unit each of Multiply, Load, and Store. The latencies of the units are as shown. Assume that this represents first generation of the machine. Figure 2 shows the next-generation VLIW where the Multiply and Load latencies have changed to 4 and 3 cycles respectively. The old schedule cannot be guaranteed to execute correctly on this machine due to the flow dependence between operations B and C, between D and H, and between E and F.

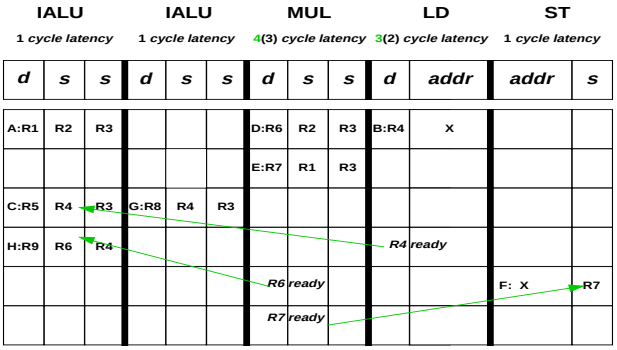

Figure 2: Next generaton VLIW machine: Incompatibility due to changes in functional unit latencies (shown by arrows). The old latencies are shown in parentheses. Operations C, H, and, F are now produce incorrect results because of the new latencies for operations B, D, and E.

Figure 3 shows the schedule for the next-next-generation machine that includes an additional multiplier. The latencies of all FU*s* remain as shown in Figure 1. Code scheduled for this new machine would not execute correctly on the older machines because the scheduler has moved operations in order to take advantage of the additional multiplier. (In particular, operations E and F have been moved.) There is no trivial way to adapt this schedule to the older machines. This is the case of downward incompatibility between generations. In this situation, if different generations of machines share binaries (e.g., via a file server), compatibility requires either a mechanism to adjust the schedule or a different set of binaries for each generation.

A scheme which would guarantee correct execution of a VLIW binary on any generation of the machine would suffice to solve the compatibility problem. Such a solution must be efficient in order to be viable. Also, it must be implemented from the very

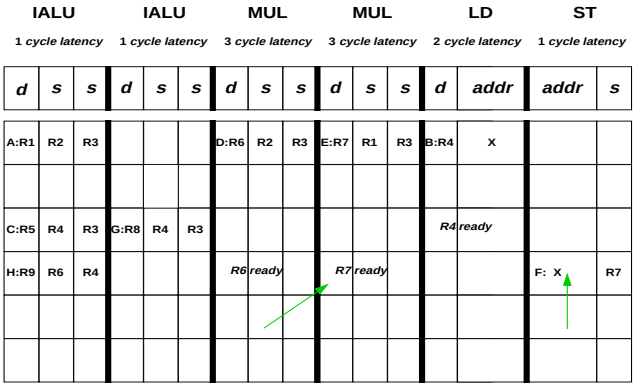

Figure 3: Downward incompatibility due to change in the VLIW machine organization: no trivial way to translate new schedule to older machine.

first generation to ensure upward compatibility with future generations.

Organization of the remainder of this paper is as follows. Section 2 describes some relevant terminology and the previous work done in this area. Dynamic rescheduling technique is described in detail in Section 3. Section 4 presents the experimental evaluation of this technique. The paper ends with concluding remarks in Section 5.

2 Related Work

2.1 Terminology

The terminology used in this paper is originally from Rau [2], and is introduced here for the discussion that follows. VLIW architectures are horizontal machines, with each wide instruction-word, or *MultiOp*, consisting of several operations, or *Ops*. All Ops in a MultiOp are issued in the same execution cycle. VLIW programs are latency-cognizant, meaning that they are scheduled with knowledge of the functional unit latencies. A VLIW architecture which runs latency-cognizant programs is termed a *Non-Unit Assumed Latency* (NUAL) architecture. A *Unit Assumed Latency* (UAL) architecture assumes unit latencies for all functional units. Many superscalar architectures are UAL.

There are two scheduling models for latency-cognizant programs: the *Equals* model and the Less-Than-or-Equals (LTE) model [2]. An *Equals* model schedules for a VLIW architecture on which each op-

eration takes exactly its specified execution latency. In contrast, an LTE model schedules assuming any operation may take less than or equal to its specified latency. The Equals model produces slightly shorter schedules than the LTE model, mainly due to register reuse. However, the LTE model simplifies the implementation of precise interrupts and provides binary compatibility when latencies are reduced. The scheduler in the back-end of the compiler and the dynamic rescheduler presented in this paper follow the LTE scheduling model.

2.2 Previous work

The working principle behind hardware techniques used to support object-code compatibility in VLIW machines is shown in Figure 4. It is similar to superscalars in that both perform run-time scheduling in hardware. The difference, however, is that the schedule presented to superscalar dynamic scheduling hardware is UAL, whereas the scheduling hardware in a dynamically scheduled VLIW processor is presented with a NUAL schedule.

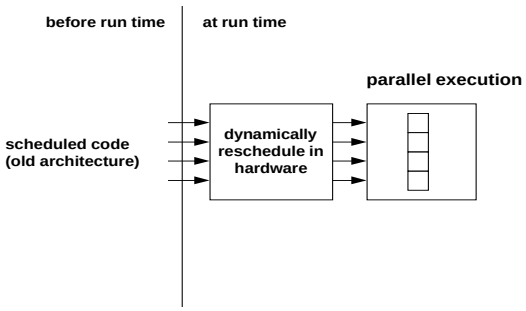

Figure 4: Hardware approach to compatibility.

Rau [2] presented a hardware technique, called Split-Issue, for dynamic scheduling in VLIW processors. In order to handle NUAL programs, it provides hardware capable of splitting each Op into an Op-pair: (*read_and_execute*, *destination_writeback*). *Read_and_execute* uses an anonymous (i.e. a non-architected) register as its destination, whereas *destination_writeback* copies the destination of *read_and_execute* to the destination specified in the original Op. *Read_and_execute* operation is issued in the next available cycle, provided there are no dependence or resource constraints. The *destination_writeback* operation is scheduled to be issued in the latest cycle after (*issue_cycle* (*read_and_execute*) + *original_operation_latency* − 1). To ensure that the *destination_writeback* operation is not issued before the *read_and_execute* completes, support in the form of hardware flags is provided. The splitting of operations and issuing them in the correct time order preserves the program semantics, and correct program execution is guaranteed.

The concept of fill-unit was originally proposed by Melvin, Shebanow, and Patt in [3], and was extended in [4]. Although it was originally not aimed at achieving compatibility in VLIWs, it can be adapted to fulfill this goal. Special hardware consisting of the fill-unit and a shadow cache is used in this technique. It works as follows: the processor routinely executes a UAL program operation stream. Concurrent to the execution, the fill-unit compacts these operations into VLIW-like MultiOps. These newly formed MultiOps are stored in the shadow cache. When an operation requested by the fetch unit is available in the shadow cache, all the operations in the MultiOp containing this operation are issued. The formation of a new MultiOp by the fill-unit is terminated when a branch instruction is encountered.

A limitation of the hardware approaches is that the scope for scheduling is limited to the window of Ops seen at run-time, hence available ILP is relatively less than what can be exploited by a compiler. These schemes also may result in cycle time stretch, a phenomenon due to which many are considering the VLIW paradigm over superscalar for future generation machines.

Static recompilation is the most prevalent software technique (illustrated in Figure 5). It recompiles the entire program off-line, and hence can take advantage of sophisticated compiler optimizations to attain superior performance. Alternatively, complete recompilation of the program may be avoided by maintaining multiple copies of the program for various target architectures in a partitioned object file. An appropriate module can be scheduled at installation time. The main drawback of these methods is that they involve an extra step to achieve code compatibility. This introduces a deviation from the normal development process for the developer, and from the routine installation process for the user. Also related is the issue of potential copy protection violations. Software licensing is done on a per-copy basis; having multiple specialized copies of the same program, even though the user plans to use only the one for his machine, may well become an expensive proposition. Another problem is that the storage space require-

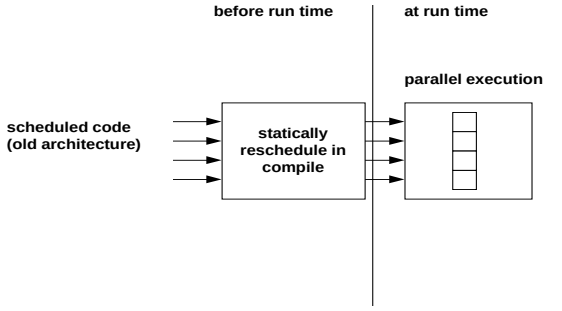

Figure 5: Rescheduling the program off-line for compatibility.

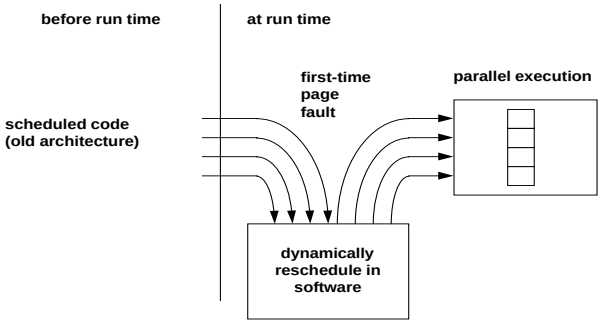

Figure 6: Dynamic Rescheduling.

ments of multiple copies may be excessive. These problems suggest that compatibility via off-line recompilation may not be easy to commercialize.

Of related interest are the techniques used to migrate the software across to a new machine architecture. Silberman and Ebcioglu [5] describe in detail an effort to gain performance advantage by translating and running old CISC object code on RISC, Superscalar and VLIW machines. *Binary Translation* used by Digital Equipment Corp. to migrate from the VAX/VMS environment to its newer Alpha architecture is documented in [6]. Apple Computers used the technique of *emulation* to migrate the software compiled for Motorola 680x0 to the PowerPC. Insignia Solutions has presented the effort of emulating Intel x86 architectures on modern RISC machines in [8]. A short survey of related techniques and issues can be found in [9].

3 Dynamic Rescheduling

Dynamic rescheduling is illustrated in Figure 6. When a program is executed on a machine generation other than what it was scheduled for, the dynamic rescheduler is invoked. The exact sequence of events is as follows: The OS loader reads the program binary header and detects the generation mismatch.[1] After the first page of the program is loaded for execution, the page fault handler invokes the dynamic rescheduler module. The rescheduler reschedules the page for execution on the current host. This process is repeated each time a new page fault occurs. Translated pages are saved to swap space on replacement. Only the pages which are executed during the life-span of the program are rescheduled. The knowledge of architectural details of the executable's VLIW generation is necessary for the dynamic rescheduler to operate, and is retained in the executable image.

Dynamic rescheduling poses some interesting problems which can reduce its effectiveness as a run-time technique. The rest of this section discusses these problems in detail and presents solutions. This paper assumes that the code scheduled for a VLIW machine is logically organized as a sequence of scheduling structures called Superblocks or Hyperblocks [10], [11]. Construction of Superblocks and Hyperblocks is shown in Figures 7 and 8, respectively. The implementation of the dynamic rescheduling algorithm uses the TINKER architecture [12]. TINKER is based on the parametric PlayDoh VLIW architecture from Hewlett-Packard Laboratories [13]. Some of the features in TINKER have been designed specifically to solve the problems faced in dynamic rescheduling. For example, the TINKER bit-encoding for MultiOps provides a *Block-Bit* in an Op to mark a entry point (merge point) of a Hyperblock or a Superblock. This information is used by the rescheduler to define the scope of rescheduling. More examples are presented later in this section.

In the discussions that follow, only acyclic scheduled code is considered. Software pipelining is not incompatible with the techniques presented, but a detailed discussion of dynamic rescheduling for software pipelined loops is beyond the scope of this paper.

[1] The version information is retained as part of the process table entry for the process.

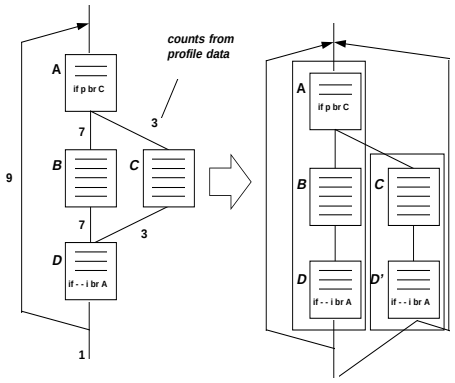

Figure 7: Example of Superblock formation. Note that both Superblocks and Hyperblocks have a single entry, multiple exits and no side entrances. In general, Hyperblocks provide larger scope for ILP than the Superblocks.

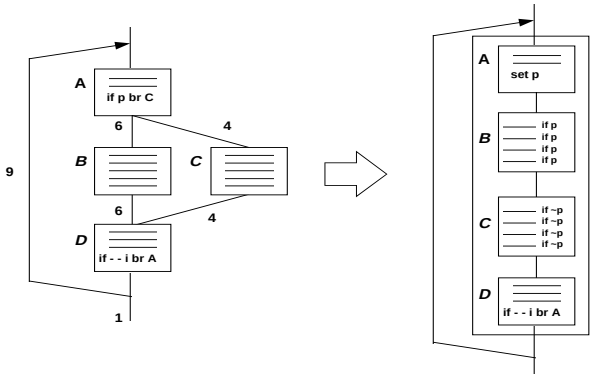

Figure 8: Example of Hyperblock formation.

3.1 Problems and their solutions

3.1.1 Changes in Code size

The dynamic rescheduling algorithm constructs a new schedule from the old schedule, using the knowledge of the old and new machine organizations and the execution latencies of the functional units. The new schedule may grow larger in size due to the insertion of empty cycles, or may shrink in size due to the deletion of empty cycles from the old schedule. This insertion or deletion of empty cycles introduces variation in code size. This phenomenon is illustrated in Figure 9, which assumes two simple machines each having integer ALU (IALU), FP add, FP multiply (FPMul), load, store, branch, and predicate comparison (Cmpp) units. Further, each Op is assumed to be 8-bytes wide. The number of `nop`s in the upper left

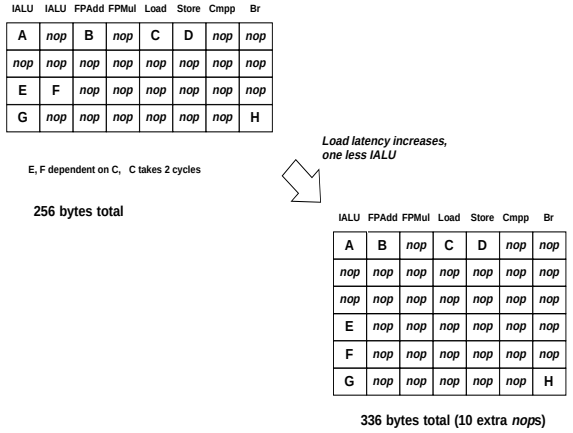

Figure 9: Page size change due to no-ops.

sample schedule of Figure 9 is 24. After the code is rescheduled for a machine having one less IALU and increased Load latency, the number of `nop`s in the rescheduled code becomes 34, resulting in a code size increase of $(34-24)*8 = 80$ bytes. As this example illustrates, any change in the size of the program would cause an overflow or underflow at the page boundary. It is neither easy nor practical to handle such changes in the page boundaries at run time. Hence, changes in code size due to rescheduling must be avoided.

This problem is solved via efficient encoding provided in TINKER (see Figure 10). The first three fields of a TINKER Op are: `Header bit`, `Optype`, and the `Pause`. An Op with `Header Bit` = 1 signifies the beginning of a new MultiOp, and it is called a *Header Op*. `Optype` indicates the type of the functional unit in which the Op will execute, thus bypassing the need for `nop`s within a MultiOp. The `pause` field in each Header Op encodes the number of empty cycles, if any, that would follow the current MultiOp. (No meaning is attached to the value of `pause` in a non-Header Op). The Op encoding in TINKER thus *hides* the *nop*s, and ensures that code rescheduling within a basic block does not trigger any code size changes. Figure 10 shows the rescheduled code previously shown in Figure 9, as it would be encoded in

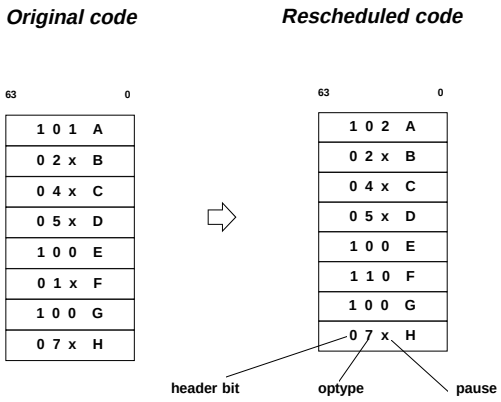

Figure 10: An example of TINKER Encoding scheme. Each Op is a fixed-format 64-bit word. The format includes the *Header Bit*, *optype*, and the *pause* fields, which together eliminate the need for *nop*s in code.

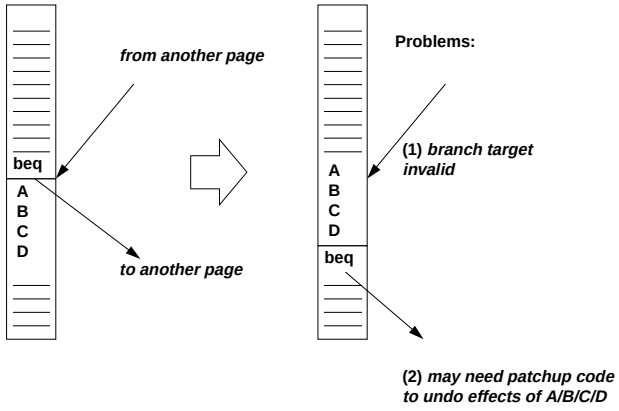

Figure 11: Problems introduced by speculative code motion.

TINKER. It can be noted that the size of code has not changed, the *nop*s in each MultiOp have been squeezed out using the `optype` field, and empty cycles have been squeezed out using the `pause` field. The code size remains 64 bytes in total for both the machines.

3.1.2 Speculative code motion

Two problems are introduced by speculative code motion, if any, during rescheduling, and are illustrated with an example in Figure 11. The first problem is caused by incorrect execution of code due to target invalidation. In the example, Op A is the target of a branch from elsewhere in the code. If all Ops A, B, C, D are speculatively moved above the branch (`beq`) which was originally before Op A, then this motion invalidates the target of the incoming branch. The second problem is caused by patchup code which may be necessary to undo effects of code motion. For example, if the outgoing branch (`beq`) is taken, the effects of speculating Ops A, B, C, and D, may need to be undone by inserting patchup code at the target of this branch. This target which may very well lie in another page, which in turn may not be memory resident. Even if the target lies in the current page, code insertion will cause overflow of code at the page boundary. For these reasons, generation of patchup code must be avoided during rescheduling.

To solve the first problem, the dynamic rescheduling algorithm takes advantage of the property of Superblocks/Hyperblocks that they have a unique entry point at the top of the block, and no side-entrances (i.e. merge-points). Each Superblock/Hyperblock in a page is rescheduled individually. The compiler is page-size cognizant when it forms Hyperblocks or Superblocks, guaranteeing that they do not span page boundaries. Since speculation does not happen outside a Superblock/Hyperblock, branch target invalidation is eliminated.

The second problem could be solved by performing speculation only during the initial compilation and confining rescheduling to basic blocks (i.e. no speculation during rescheduling). But this would limit the opportunity to expose more ILP in a more parallel VLIW architecture. Instead, dynamic rescheduling performs limited speculation, but only if it requires no patchup code. To support this, the compiler saves the live-out set info for each branch in the program in the object file (see Section 3.2). During rescheduling, if the rescheduler detects that a speculatable Op modifies a register in the live-out set of the Branch, it cancels the move. Any other Op not modifying a member of live-out set can be moved, since it does not require generation of patchup code.

3.1.3 Register file changes

Better hardware implementation techniques sometimes allow for more registers in an advanced gen-

eration, thus providing more opportunity to reduce register pressure. Although it is not traditional to allow a change in the register file size in the ISA, it is interesting to consider this possibility. From the perspective of dynamic rescheduling technique, such a change poses additional problems. When code is rescheduled for a machine with a different register file architecture, the rescheduler must perform register re-allocation. It is likely that spill code will be generated during this process, causing an increase in code size. This will violate the requirement that the page-size cannot change at run-time. The dynamic rescheduling algorithm presented in this paper currently assumes that the register file architecture does not change across generations.[2]

Table 1: Live-out set sizes for branches in various benchmarks.

Benchmark	#Branches	Liveout set size		
		min	max	average
cccp	103,405	1.00	116.00	25.00
compress	162,004	1.00	98.00	38.80
eqn	162,801	5.00	117.00	19.53
eqntott	9,261	1.00	53.00	21.58
espresso	22,257	1.00	73.00	14.76
lex	55,320	1.00	58.00	18.89
tbl	19,487	1.00	139.00	34.00
wc	3,616	2.00	86.00	19.38
yacc	139,166	1.00	142.00	39.60
Average	75,257	1.56	98.00	25.73

3.2 Additional object-file information

The following is a review of the information included in the object file to support dynamic rescheduling. The version of the VLIW architecture for which the code is scheduled is encoded in the header of the object file, along with information about the architecture (i.e., the number and latency of each functional unit type). Also, the block boundaries (for Hyperblocks and Superblocks) are marked using the `Hyperblock Bit` in Ops. The live-out register sets for each branch in the code are included if the rescheduler is allowed to perform speculative code motion without the patchup code hazard. The live-out sets are organized in a non-loadable segment of the object-file, and hence do not interfere with layout of the code or data segments. The live-out segment is read and used by the rescheduler only when needed.

The increase in the size of the object file due to the live-out sets could be of concern, especially if the program is known to contain a large number of branch instructions. To address this concern, Table 1 presents live-out set sizes measured for the branch instructions in the benchmarks. For each benchmark, the minimum and maximum of live-out set sizes are shown. A clever encoding can be used to efficiently store this information in the object file. If the live-out set is small, a byte-encoded register list is constructed. If the set is large, a bit vector encoding is used.

[2]The dynamic rescheduling algorithm can withstand the changes in register file architecture in a limited way. An *increase* in the number of registers with no change in compiler's subroutine calling conventions, can be handled by the algorithm without generation of spill-code. This is true only for the programs originally scheduled for the machine with a smaller register file, being rescheduled for the machine with a larger register file, and not *vice versa*.

3.3 Operating system support

The pieces of the mechanism which invokes the dynamic scheduling algorithm constitute the OS support for this technique. Some of these were mentioned earlier in this section, for example, (1) the detection of the machine generation mismatch by the OS loader (2) invocation of the dynamic rescheduler by page fault handler at first time page faults, and (3) the machine architecture database maintained in the OS tables. Yet another part of the OS that plays crucial role is the file system buffer cache. The buffer cache routinely holds the pages that were used in the recent past. This is a standard mechanism available in modern operating systems, which directly helps amortize the cost of rescheduling over the first-time page accesses. The penalty of rescheduling is therefore not incurred for every page access made during the life-span of the program.

3.4 Dynamic rescheduling algorithm

This section briefly describes the core of the dynamic rescheduling algorithm. It is adapted from a simulation algorithm for out-of-order execution processors, described in [14]. It is assumed that the VLIW program is latency-cognizant. The algorithm has full knowledge of the functional unit latencies of the original machine (called the `old_machine`), and the machine for which the program will be rescheduled (called the `new_machine`). Both `old_machine` and `new_machine` are assumed LTE (less-than-or-equals) machines. Two main data structures keep track of data dependencies between Ops: A `scoreboard` of registers helps determine the flow and output dependencies, and a `register usage matrix` keeps track of anti-dependencies. The resource constraints are

handled using a `resource usage matrix`. Memory accesses are modeled as accesses to single pseudo-register, and all LOADs and STOREs source and sink this register, respectively. This ensures that their relative ordering is not altered while rescheduling.

A two-pass implementation of rescheduling would work as follows. In the first step, build a dependence graph for the hyperblock under consideration. In the second step, perform list scheduling. This approach is slow, and hence is more suitable for an off-line to compatibility. Dynamic rescheduling, instead takes a single-pass approach: it implicitly builds the dependence information on the `scoreboard` and in the `register usage matrix`, and schedules Ops with the knowledge of FU latencies. The main control structure of the algorithm is shown in the Appendix.

4 Experimental Evaluation

4.1 Methodology

The set of benchmarks used for evaluation of dynamic rescheduling is shown in Table 2. Three benchmarks are from SPECint92 suite (026.compress, 008.espresso, 023.eqntott), while the others are Unix text-processing utilities (tbl, eqn), development tools (lex, yacc) and a language processor (cccp). This set of benchmarks was chosen because it represents the typical non-numeric workloads in various user-environments. The three benchmarks from SPEC represent the workloads commonly used today in industry to characterize and compare performance of machines.[3]

Table 2: Benchmarks used for evaluation.

Benchmark	Description
cccp	C pre-processor
eqn	Equation formatter
lex	Scanner generator
yacc	Parser generator
tbl	Table formatter
026.compress	compression/decompression utility
023.eqntott	Truth Table generator for logic circuits
008.espresso	PLA Optimization

Two TINKER machine models, termed TINKER-A and TINKER-B, were used in the evaluation. The TINKER-A model has eight functional units and represents a hypothetical first-generation VLIW architecture. Its organization and FU latencies are as shown in Table 3. Table 4 shows the organization and FU latencies for TINKER-B. Although it is difficult to draw direct comparisons, TINKER-A is roughly equivalent to a two-issue out-of-order execution superscalar (due to the two IAlu units). Similarly, TINKER-B is roughly equivalent to a four-issue superscalar.

Table 3: TINKER-A Machine Model.

FU	Number of Units	FU Latency
Integer ALU	2	1
Load	1	2
Store	1	1
Branch	1	1
FP Add	1	1
FP Mul	1	3
Predicate Unit	1	1

Table 4: TINKER-B Machine Model.

FU	Number of Units	FU Latency
Integer ALU	4	1
Load	4	2
Store	2	1
Branch	1	1
FP Add	1	1
FP Mul	1	3
Predicate Unit	3	1

The dynamic rescheduling algorithm has been implemented in a tool called *October*. October is designed to interact with the IMPACT [15] framework from University of Illinois. The IMPACT front-end compiles the benchmarks, profiles, optimizes, if-converts the code to do hyperblock formation, and presents the code to October in a suitable intermediate format. A three part method was used in order to evaluate the dynamic rescheduling technique, and is described as follows. In the first part, intermediate code for a benchmark was scheduled for a given machine model (either TINKER-A or TINKER-B), using the TINKER scheduler. It was then profiled in order to find the worst case estimate of execution time of the benchmark, in terms of the number of cycles. This was called the *Native* mode execution of the program. This experiment also measured the number of unique page accesses for the benchmark, as well as the frequency of access for each page of code. In the second part, the code scheduled for Native mode execution was rescheduled by October for the other machine model. Execution time estimate

[3] No results have been presented for the floating-point benchmarks, because the rescheduler does not yet implement rescheduling for software pipelined loops. Comparing dynamic rescheduling results would be unfair in their case as such comparison would be based on their performance with scheduling of acyclic code only.

for this rescheduled code was also generated as described before. This time estimate indicates the performance of the rescheduled code without taking into account the rescheduling overhead incurred by October. Hence this part is termed as *no overhead* experiment.

In the third part, October itself was compiled, scheduled for the machine model used in the first part, and then used as a benchmark. The input to the Octber benchmark were pages taken at random from each of the other benchmarks. The performance of the October benchmark was used to find the average time to reschedule a page on each of TINKER-A and TINKER-B. This was found to be 54,272 cycles for the rescheduler executing on TINKER-A, and 51,200 cycles for TINKER-B. This was then combined with the number of unique page accesses from the first step to estimate the total number of execution cycles for the rescheduling overhead. The execution time of the no-overhead experiments are stretched by this figure and termed the *w/overhead* experiment. Finally, in order to compare the performance achieved in the above three parts, the speedup w.r.t a single-unit, single-issue processor model (called the *base model*) was calculated. It is defined as: speedup = (number of cycles of execution estimated in the experiment) / (number of cycles of execution estimated for the base model). All three parts assumed a page size of 4K bytes.

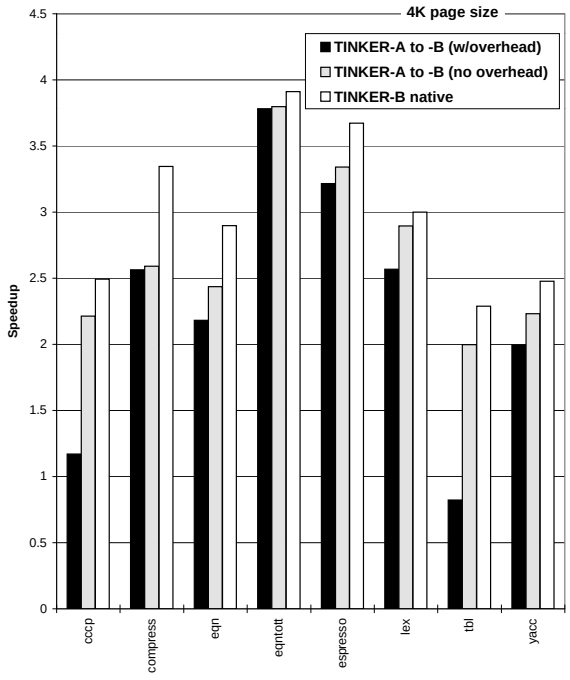

Figure 12: Performance of dynamic rescheduling: TINKER-A to TINKER-B.

4.2 Results

Table 5: Measurements of Unique Page Accesses.

Benchmark	Tinker-16 Native	Tinker-8 Native
cccp	33	32
compress	8	8
eqn	39	41
eqntott	21	19
espresso	132	129
lex	39	39
tbl	45	44
wc	2	2
yacc	50	49
Avg.(rounded)	41	41

The experiments described above were run for each benchmark, and are presented in Figures 12 and 13. Figure 12 shows the speedups for rescheduling TINKER-A code to TINKER-B, with rescheduling overhead (*w/overhead*) and without the rescheduling overhead (*no overhead*), and compares them with the speedup achieved by the Native mode execution on TINKER-B (*Native*). Figure 13 presents the corresponding numbers for rescheduling of TINKER-B code to TINKER-A.

It can be seen that the *no overhead* speedup compares quite well with that of *Native*. The only exception to that is *compress* in TINKER-A to TINKER-B (Figure 12), where the limitations placed on speculation during rescheduling are too restrictive.

The performance of rescheduled code when overhead is included (*w/overhead* in Figures 12 and 13) is less than the *no overhead* results, as should be expected. This reflects the effectiveness of dynamic rescheduling as a run-time technique. The performance compares well, suggesting the technique is effective in most cases. The notable exceptions are *cccp*, and *tbl*: in their case the penalty due to overhead is quite high. This phenomenon can be explained by observing the unique page access counts presented in Table 5, which is identical to the number of first time page faults. Cccp and tbl are relatively short-running benchmarks, and the first time

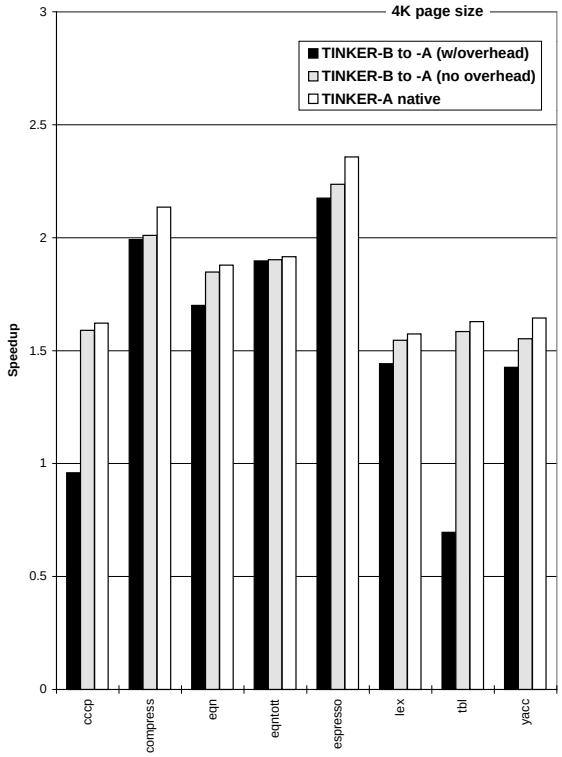

Figure 13: Performance of dynamic rescheduling: TINKER-B to TINKER-A.

page faults are higher compared to their overall execution time than for the other benchmarks. This implies that dynamic rescheduling may not execute short-running programs efficiently.

Overhead can be reduced for both long-running programs and short-running programs by providing a segment in the executable file to cache translated pages between runs. The operating system can be used to maintain this segment, updating it on program exit. Table 5 suggests a size between 32 to 64 pages for this table, although smaller sized tables may perform acceptably depending on the placement and replacement policies used for page cache updates. Such a caching scheme could improve the performance of dynamic rescheduling from the *w/overhead* performance to the *no overhead* performance. This topic is currently being studied by the authors.

5 Concluding Remarks

This paper has presented a technique to guarantee VLIW-to-VLIW object code compatibility across generations without the need for hardware or multiple executables. It relies on a a limited version of software scheduling applied during first-time page faults. The technique requires support from the compiler, the ISA, the operating system, and a fast algorithm for rescheduling. Results were presented that suggest dynamic rescheduling has the potential to effectively solve the compatibility problem in VLIW architectures. Future work in this area includes the use of dynamic rescheduling for cyclic code schedules such as modulo scheduled loops, and a further development of the translated page cache described above.

Acknowledgements

We would like to thank the anonymous referees for their excellent comments and suggestions. We would also like to gratefully acknowledge the support of the IMPACT group at University of Illinois at Urbana-Champaign. In particular, we would like to thank Richard Hank and John Gyllenhaal for patiently answering all our queries. We would also like to thank the other members of the TINKER group: Sanjeev Banerjia, Sergei Larin, Kishore Menezes, and Ashutosh Singla. This research has been supported by AT&T, Intel Corporation, and the National Science Foundation under grant MIP-9410377.

Appendix: the algorithm

```
Dynamic_Reschedule (input: old_schedule)
{
 for (each cycle of execution
       in the old_schedule) {
   /* Step 1:
    * resolve the resource constraints.
    */
   for (each Op from old_schedule that
         writes back in this cycle) {
     /* Tdelta is stored in the Op:
      * see Step 3
      */
     Get the Tdelta for the Op;
     Look up resource usage matrix to
     Determine Trc for the Op;
     Update Resource_Usage_Matrix;
     Update Register_Usage_Matrix;
     Set cycle of the Op in new_schedule
       = Trc;
```

```
  }

  /* Step2: Update the Scoreboard:
   * reserve destination registers
   * through the latencies of execution
   * according to the old schedule.
   */
  for (each Op from old_schedule that
       initiates in this cycle) {
    Reserve the dest operand
    of Ops on the Scoreboard through
    the latency of the old_machine;
    Set most recent writer of the
    destination operand = this op;
  }

  /* Step 3: Dependence checking. */
  initialize Tdelta = 0;
  Tdelta
    = anti_dependence_check (Op, Tdelta);
  Tdelta
    = pure_dependence_check (Op, Tdelta);
  Tdelta
    = output_dependence_check (Op, Tdelta);
  Store Tdelta into the Op;
 }
}
```

References

[1] J. S. O'Donell, "Superscalar vs. VLIW," *Comp. Arch. News*, vol. 23, pp. 26–28, Mar. 1995.

[2] B. R. Rau, "Dynamically scheduled VLIW processors," in *Proc. 26th Ann. International Symposium on Microarchitecture*, (Austin, TX), pp. 80–90, Dec. 1993.

[3] S. Melvin, M. Shebanow, and Y. Patt, "Hardware support for large atomic units in dynamically scheduled machines," in *Proc. 21th Ann. International Symposium on Microarchitecture*, (San Diego, CA), pp. 60–66, Dec. 1988.

[4] M. Franklin and M. Smotherman, "A fill-unit approach to multiple instruction issue," in *Proc. 27th Ann. International Symposium on Microarchitecture*, (San Jose, CA), pp. 162–171, Dec. 1994.

[5] G. Silberman and K. Ebcioglu, "An architectural framework for supperting heterogeneous instruction-set architectures," *Computer*, vol. 26, pp. 39–56, June 1993.

[6] R. L. Sites, A. Chernoff, M. B. Kerk, M. P. Marks, and S. G. Robinson, "Binary translation," *Comm. ACM*, vol. 36, pp. 69–81, Feb. 1993.

[7] P. Koch, "Emulating the 68040 in the PowerPC Macintosh," in *Proc. Microprocessor Forum*, Oct. 1994.

[8] P. Stears, "Emulating the x86 and DOS/Windows in RISC environments," in *Proc. Microprocessor Forum*, Oct. 1994.

[9] R. Cmelik and D. Keppel, "SHADE: A fast instruction-set simulator for execution profiling," in *Fast Simulation of Computer Architectures* (T. M. Conte and C. E. Gimarc, eds.), Boston, MA: Kluwer Academic Publishers, 1994.

[10] W. W. Hwu, S. A. Mahlke, W. Y. Chen, P. P. Chang, N. J. Warter, R. A. Bringmann, R. G. Ouellette, R. E. Hank, T. Kiyohara, G. E. Haab, J. G. Holm, and D. M. Lavery, "The superblock: An effective structure for VLIW and superscalar compilation," *The Journal of Supercomputing*, vol. 7, pp. 229–248, Jan. 1993.

[11] S. A. Mahlke, D. C. Lin, W. Y. Chen, R. E. Hank, and R. A. Bringmann, "Effective compiler support for predicated execution using the Hyperblock," in *Proc. 25th Ann. Int'l. Symp. on Microarchitecture*, (Portland, OR), pp. 45–54, Dec. 1992.

[12] "TINKER machine language manual," 1995. Department of Electrical and Computer Engineering, North Carolina State University, Raleigh NC 27695-7911.

[13] V. Kathail, M. Schlansker, and B. R. Rau, "HPL PlayDoh architecture specification: version 1.0," Tech. Rep. HPL-93-80, Hewlett-Packard Laboratories, Technical Publications Department, 1501 Page Mill Road, Palo Alto, CA 94304, Feb. 1994.

[14] T. M. Conte, *Systematic computer architecture prototyping*. PhD thesis, Department of Electrical and Computer Engineering, University of Illinois, Urbana, Illinois, 1992.

[15] P. P. Chang, S. A. Mahlke, W. Y. Chen, N. J. Warter, and W. W. Hwu, "IMPACT: An architectural framework for multiple-instruction-issue processors," in *Proc. 18th Ann. International Symposium Computer Architecture*, (Toronto, Canada), pp. 266–275, May 1991.