

Optimization of VLIW Compatibility Systems Employing Dynamic Rescheduling

Thomas M. Conte Sumedh W. Sathaye
Department of Electrical and Computer Engineering
North Carolina State University
Raleigh, North Carolina 27695-7911
(919)-515-5067
e-mail: {conte,swsathay}@eos.ncsu.edu

This is a revised and expanded version of the paper presented by the authors at the 28th Annual International Symposium on Microarchitecture (MICRO-28), Nov. 1995, Ann Arbor, MI.

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 OS: at each first-time page fault, the page of code is rescheduled for the new generation, if required. Results are presented to demonstrate the viability of the technique using the Illinois IMPACT compiler and the TINKER architectural framework.

For the machine models and the workloads used in this study, performance of the rescheduled code compares well with the native scheduled code for a machine. The behavior of a subset of programs in the workload is such that they face a large number of first-time page faults. Due to this, their rescheduling overhead is higher relative to their total execution time. Such programs are called *high-overhead* programs. Caching of translated pages across multiple invocations of the program to reduce the rescheduling overhead, using a *persistent rescheduled-page cache (PRC)* [1] is discussed. It was found that for the workload used in this evaluation, a PRC of size between 512 to 1024 pages, and which uses an *overhead-based* page replacement policy would be effective in reducing the overhead.

Keywords

VLIW, Object-Code Compatibility, Dynamic Rescheduling, Instruction-Level Parallelism

I. 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 [2]. 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 in the marketplace. Two approaches to solve this problem have been reported in the literature: hardware approaches and software approaches. The hardware approaches include split-issue proposed by Rau [3], and the fill-unit proposed by Melvin, Shebenow, and Patt [4] and extended by Franklin and Smotherman [5]. 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 executable. This approach requires generation of multiple executables, which poses difficulties for commercial copy protection and system administration. This paper proposes and optimizes a new scheme called Dynamic Rescheduling to achieve object-code compatibility between VLIW generations [6]. Dynamic rescheduling applies a limited version of software scheduling during first-time page faults, requiring no additional hardware support for scheduling. Making this practical requires support from the compiler, the ISA, the operating system, and the rescheduling algorithm. These topics are discussed in detail in this paper. Results are presented that suggest dynamic rescheduling has the potential to effectively solve the compatibility problem in VLIW architectures.

A. The VLIW compatibility problem

The compatibility problem is illustrated in the following example. Figure 1 shows an example VLIW schedule for a machine with two integer ALUs, and a single unit each for Multiply, Load, and Store. The latencies of the units are as shown. Assume that this represents the 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 because the flow dependence between operations B and C, between D and H, and between E and F will be violated due to the changes in latencies.

Figure 3 shows a schedule for an alternative next-generation machine that includes an additional multiplier. The latencies of all FUs 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 machine generation.

A scheme which would guarantee correct execution of a VLIW binary on any generation of the machine without undue loss of performance over natively compiled code 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 first generation of the architecture to ensure upward compatibility with future generations. Since dynamic rescheduling is a software approach, it can be implemented from the very first generation of an architecture. The performance measurements for the rescheduling algorithm presented in this paper indicate that it is also efficient. Organization of this paper is as follows: Section II describes some relevant terminology used in this paper; the previous work done in this area is described in Section III; the dynamic rescheduling technique is described in detail in Section IV; Section V presents the experimental evaluation of this technique, and the paper ends with concluding remarks in Section VI.

II. TERMINOLOGY

The terminology used in this paper is originally from Rau [3], 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 to their corresponding functional units 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 [3]. Under the equals model, each operation within the schedule takes exactly the specified execution latency. In contrast, under the LTE model, each operation may take less than or equal to its specified execution latency. The equals model produces slightly shorter schedules than the LTE model, mainly due to increased freedom for register reuse. However, the LTE model simplifies the implementation of precise interrupts and could provide binary compatibility when only the execution latencies are reduced across a generation. The scheduler in the back-end of the compiler and the dynamic rescheduler presented in this paper follow the LTE scheduling model.

III. RELATED 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.

Rau [3] presented a hardware technique, called *Split-Issue*, for dynamic scheduling in VLIW processors. In order to handle NUAL programs scheduled for an *Equals* machine (the LTE case is trivial and can be implemented via simple hardware interlocking), it provides hardware capable of splitting each Op into an Op-pair: (*read_and_execute*, *destination_writeback*). An anonymous (i.e., non-architected) register is used as the destination for the *read_and_execute* Op, whereas the *destination_writeback* Op writes back the destination of *read_and_execute* to the destination specified in the original Op. The

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 [4], and was extended in [5]. Although it was originally not aimed at achieving the 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. A specific limitation of the fill-unit approach is its inability to speculate beyond branches. The complexity of 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 requirements 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 to a new machine architecture. Silberman and Ebcioğlu [7] describe in detail an effort to gain performance advantage by translating and running old CISC object code on RISC, Superscalar and VLIW machines. The approach of *Binary Translation* [8], and a combination of binary translation and *emulation* [9] has been used by Digital Equipment Corporation to migrate various software platforms to its Alpha architecture. Apple Computer has also used the technique of emulation to migrate the software compiled for Motorola 680x0 to the PowerPC [10]. Insignia Solutions has presented the effort of emulating Intel x86 architectures on modern RISC machines in [11]. A short survey of related techniques and issues can be found in [12].

IV. DYNAMIC RESCHEDULING

Dynamic rescheduling is illustrated in Figure 6. When a program is executed on a machine generation other than the one 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 a generation mismatch¹. After the first page of the program is loaded for execution, the page fault handler invokes the dynamic rescheduler module, which 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 upon 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 single-entry, acyclic regions (SEAR) [13] called Superblocks or Hyperblocks [14], [15]. Construction of Superblocks and Hyperblocks is shown in Figures 7 and 8, respectively. Implementation of the dynamic rescheduling algorithm presented in this paper uses the TINKER architecture [16]. TINKER is based on the parametric PlayDoh VLIW architecture from Hewlett-Packard Laboratories [17]. 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 the MultiOps provides a *Region-Bit* in an Op to mark an entry point (merge point) of a SEAR. This information is used by the rescheduler to define the scope of rescheduling. More examples are presented later in this section. The discussion in the rest of this section is purposely biased towards acyclic code. While cyclic code schedules generated via various methods of software pipelining are not incompatible with the technique of dynamic rescheduling, a detailed discussion of rescheduling of software pipelined loops is beyond the scope of this paper.

A. Problems and their solutions

A.1 Changes in Code size

A new schedule generated via rescheduling may have a different height as compared to the old schedule. This can happen if the schedule expands due to insertion of empty cycles, or if it shrinks in due to deletion of empty cycles from the old schedule. Thus, variations in code size may be introduced due to rescheduling. 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 have a width of 64 bits. The number of nops in the upper sample schedule in Figure 9 is 24. After the code is rescheduled for a machine with one less IALU and an increased Load latency, the number of nops 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

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

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, any changes in code size due to rescheduling must be avoided.

This problem is solved via a specialized encoding provided in TINKER. The basic idea of the TINKER encoding is to avoid explicit representation of `nops` in the schedule. The `Ops` hold enough information to derive the knowledge of the placement of the `nops` and the empty cycles in the schedule at run-time. Symbolic definition of the TINKER encoding is as follows.

Definition 1 (A VLIW Operation) A VLIW Operation (Op) is defined by a 5-tuple: $\{H, p_n, FUtype, opcode, operands\}$, where, $H \in \{0, 1\}$ is a 1-bit field, p_n is an n -bit field called *pause*, *s.t.* $p_n \in \{0, \dots, 2^n - 1\}$, $FUtype$ uniquely identifies the functional unit instance where the Op must execute, $opcode$ uniquely identifies the task of the Op , and $operands$ is the set of valid operands defined for the Op . \square

Definition 2 (Header Op) An Op , O , is a *Header Op* iff the value of the H field of O is 1. \square

Definition 3 (MultiOp) A VLIW MultiOp, M , is an unordered sequence of Ops $\{O_1, O_2, \dots, O_m\}$, *s.t.* $0 \leq m \leq w$, where w is the number of hardware resources to which the Ops are issued concurrently, and O_1 is the Header Op. \square

Definition 4 (Schedule) A VLIW schedule, S , is an ordered sequence of MultiOps $\{M_1, M_2, \dots\}$. \square

Some discussion of this encoding is now in order. A given schedule consists of a sequence of Ops in which a new MultiOp begins at a Header Op and ends exactly at the Op before the next Header Op in the sequence. The MultiOp fetch hardware uses this rule to identify and fetch the next MultiOp. The p_n field in an Op is referred to as *pause*, because its value in the Header Op is used by the fetch hardware to halt MultiOp issue for those many cycles. This is a mechanism devised to eliminate explicit encoding of empty cycles in the schedule. the $FUtype$ field is used to route an Op to the functional unit indicated by it. The Ops need not be aligned with their functional units thus avoiding the use of *nops*. Design of the specialized instruction fetch hardware required for this encoding is described in [18]. Since the TINKER encoding avoids explicit encoding of *nops*, it ensures that code rescheduling does not trigger any code size changes. Figure 10 shows the TINKER code for the example shown in Figure 9. Note that the code size has not changed; only the *nops* in each MultiOp and the empty MultiOps in the schedule have been “compressed out”. The code size remains 64 bytes for the original code as well as the rescheduled code.

A.2 Speculative code motion

Two problems are introduced by speculative code motion, if any, during rescheduling, and are illustrated with an example shown in Figure 11. The first problem is caused by incorrect execution of the code due to target invalidation. In this example, Op A is the target of a branch from elsewhere in the code. If all Ops A, B, C, D were speculatively moved above the branch (`beq`) which was originally before Op A, then this motion would invalidate the target of the incoming branch. The second problem is caused by the patch-up code which may be necessary to undo the 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 renaming the code at the target of this branch, and/or copies. And the target may very well lie in another page,

which in turn may not be memory resident. Even if the target lies in the current page, such patch-up has the potential to cause overflow/underflow of the code at the page boundaries. For these reasons, this situation must be avoided altogether is possible.

To solve the first problem, the dynamic rescheduling algorithm takes advantage of the property of the SEARs that they have a unique entry point at the top, and no side-entrances (i.e., merge-points). Each SEAR in a page is rescheduled individually. The compiler is page-size cognizant when it forms the SEARs, guaranteeing that they do not span page boundaries. Since speculation does not happen outside a SEAR, 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 patch-up code. To support this, the compiler saves the live-out set info for each branch in the program in the object file (see Section IV-B). 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 the live-out set can be moved.

A.3 Register file architecture

Better hardware implementation techniques sometimes allow for more registers in an advanced generation, 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 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 register-usage 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*. The dynamic rescheduling algorithm presented in this paper currently assumes that the register file architecture does not change across generations.

B. 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 the architectural details such as the number and latency of each functional unit type. Also, the SEAR boundaries are marked using the *Region 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 patch-up 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 I 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. Actual increase in the file-size for the executable of the benchmark 008.espresso was measured, and was found to be 20% with the bit-vector scheme, and slightly more than 33% for the byte-encoded list.

TABLE I
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
126.compress	162,004	1.00	98.00	38.80
eqn	162,801	5.00	117.00	19.53
023.eqntott	9,261	1.00	53.00	21.58
008.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

C. Operating system support

Pieces of the mechanism that 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 machine generation mismatch by the OS loader, and (2) invocation of the dynamic rescheduler by page fault handler at first time page faults. Yet another part of the OS that plays a 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 overhead 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. The *Persistent rescheduled-page cache (PRC)* [1] demonstrates the effectiveness of this approach. Use of the PRC is discussed in detail in Section V-C.

D. The Dynamic rescheduling algorithm

This section describes the core of the dynamic rescheduling algorithm. It is adapted from a simulation algorithm for out-of-order execution processors, described in [19]. 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 the `old_machine` and the `new_machine` are assumed LTE (less-than-or-equals) machines. Two main data structures keep track of the data dependences between Ops: a `scoreboard` of registers helps determine the flow and output dependences, and a `register usage matrix` keeps track of the anti-dependences. 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. This algorithm makes a single-pass over the region of code presented for rescheduling. Incrementally, it builds a database of dependence information on the `scoreboard` and in the `register usage matrix`, and schedules the Ops using the knowledge of FU latencies. In each iteration of the algorithm, it processes a single MultiOp in the old schedule, and completely schedules the operations into the new schedule, in a greedy fashion. The implementation presented in this paper does not, however, make any attempt to control the greed, which results in a reduced overhead of rescheduling. In the terminology presented by Rau [20] [21] and by Ellis [22], this algorithm employs *operation scheduling* as opposed to *instruction scheduling*. The main control structure of the dynamic rescheduling algorithm is shown as Algorithm *DReschedule*. **Algorithm DReschedule**

input

the `old_schedule`;

output

the `new_schedule`;

var

temporary variable `Op`;

T_δ , the *dependence time*;

T_{rc} , the *resource-constraint time*;

the *Scoreboard*;

the *resource_usage_matrix*;

the *register_usage_matrix*;

functions

Routines *anti_dependence_check* (), *pure_dependence_check* (), and *output_dependence_check* () use the *Scoreboard* and the *register_usage_matrix* to perform dependence checks for a given `Op` and modify its *dependence_time*;

begin

for (*each cycle of execution in old_schedule*)

begin

/ Step 1: Resolve the resource constraints */*

for (*each Op from old_schedule that writes back in this cycle*) **do**

begin

Get the T_δ for the `Op`;

Look up *resource_usage_matrix* to determine T_{rc} for the `Op`;

Update *resource_usage_matrix*;

Update *register_usage_matrix*;

Set cycle of the `Op` in *new_schedule* $\leftarrow T_{rc}$;

```

    end
end
/* 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) do
    begin
    for the destination operand of Op do
        begin
            Reserve it on the Scoreboard through the latency of Op on old_machine;
            Mark its most-recent-writer as Op;
        end
    end
end
/* Step 3: Dependence checking. */
for (each Op from old_schedule that initiates in this cycle) do
    begin
        Initialize  $T_\delta = 0$ ;
         $T_\delta \leftarrow anti\_dependence\_check (Op, T_\delta)$ ;
         $T_\delta \leftarrow pure\_dependence\_check (Op, T_\delta)$ ;
         $T_\delta \leftarrow output\_dependence\_check (Op, T_\delta)$ ;
    end
end
end

```

V. EXPERIMENTAL EVALUATION

A. Methodology

The set of benchmarks used for evaluation of dynamic rescheduling is shown in Table II. Most of the benchmarks are from the SPECint92 and SPECint95 suites, 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 benchmarks from SPEC represent the workloads commonly used today in industry to characterize and compare performance of machines. Three TINKER machine models, termed TINKER-A, TINKER-B,

TABLE II
BENCHMARKS USED FOR EVALUATION.

SPECint92 Benchmark	Description	SPECint95 Benchmark	Description	UNIX Utility	Description
026.compress	compression/ decompression utility	129.compress	compression/ decompression utility	cccp	C pre-processor
023.eqntott	Truth Table generator for logic circuits	099.go	AI game playing	eqn	Equation formatter
008.espresso	PLA Optimization	134.perl	practical extraction and report language (PERL)	lex	Scanner generator
022.li	LISP interpreter		interpreter	yacc	Parser generator
085.gcc	GNU C Compiler	130.li	LISP interpreter	tbl	Table formatter
072.sc	Spreadsheet				

and TINKER-C were used in the evaluation. The TINKER-A model has eight functional units and represents a hypothetical first-generation VLIW architecture; TINKER-B has 16 functional units, while TINKER-C has 32 units. The organization and FU latencies of all the models are shown in Table III. 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 and TINKER-C are rough equivalents of four- and sixteen-issue superscalars, respectively. The dynamic rescheduling

TABLE III
TINKER MACHINE MODELS.

TINKER-A	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
TINKER-B	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
TINKER-C	Number of Units	FU Latency
Integer ALU	16	1
Load	4	2
Store	3	1
Branch	1	1
FP Add	2	1
FP Mul	2	3
Predicate Unit	4	1

algorithm has been implemented in a tool called *October*. *October* is designed to interact with the IMPACT [23] 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, TINKER-B, or TINKER-C), using the IMPACT 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 another machine model. Execution time estimate 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 the *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 *October* benchmark consisted of the pages selected, 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, TINKER-B, and TINKER-C. This was found to be 54,272 cycles for the rescheduler executing on TINKER-A, and 51,200 cycles executing on TINKER-B, and 48,108 cycles while executing on TINKER-C. 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: $\text{speedup} = (\text{number of cycles of execution estimated in the experiment}) / (\text{number of cycles of execution estimated for the base model})$. All three parts assumed a page size of 4K bytes, as in contemporary operating systems [24] [25] and processors [?] [26].

B. Results

The experiments described above were run for each benchmark, and are presented in Figures 12, 13, and 14. Figure 12 shows the performance of the code rescheduled to run on TINKER-A. This is compared with the performance of the native compiled code for TINKER-A. Both *no overhead* (without the rescheduling overhead) and *w/overhead* (including the rescheduling overhead) measurements are presented for the TINKER-B to TINKER-A, and TINKER-C to TINKER-A cases. Figures 13 and 14 present the corresponding measurements for the code rescheduled to execute on TINKER-B and TINKER-C respectively.

TABLE IV
OVERHEAD CHARACTERIZATION OF BENCHMARKS.

	Unique Page Count	Overhead Ratio (Percentage)	Overhead Category
008.espresso	137	4.35	low
022.li	47	10.52	moderate
023.eqntott	25	0.64	low
026.compress	8	1.25	low
072.sc	60	6.29	moderate
085.gcc	323	17.50	moderate
cccp	34	50.09	high
tbl	50	64.15	high
grep	4	29.10	high
lex	45	16.51	moderate
yacc	56	12.19	moderate

It can be seen that the *no overhead* speedup compares quite well with that of *Native* in all the cases. The performance of rescheduled code when the overhead is included (the *w/overhead* bars in Figures 12, 13 and 14) is less than the *no overhead* results. This is expected, because the rescheduling algorithm is a limited form of list scheduling, operating with constraints on data speculation and the amount of flow information available at reschedule-time. It is important to note, however, that the loss of performance is not excessive in almost all the cases, suggesting the effectiveness of this technique.

The notable exceptions, however, are *cccp*, *grep* and *tbl*: in their case, the penalty due to overhead is quite high for all the cases considered for rescheduling. This can be explained using the concept of *overhead ratio* for a program [1]. The overhead ratio is defined as: $O = R/(E + R)$, where E is the execution time of the program, and R , the total rescheduling overhead = (*the Unique Page Count of the program*

* *avg. time required to reschedule a page*). The *unique page count* of a program is defined as the number of first-time page faults that occur during the execution of the program. Values of the unique page counts for a subset of benchmarks are shown in Table IV, along with the percentage overhead ratio for the TINKER-A to TINKER-C rescheduling case. Based on the overhead ratio for a program, it is either categorized as a *high-overhead* (overhead ratio of 20% and above), *moderate-overhead* (5%–20%), and *low-overhead* (overhead ratio less than 5%). *Cccp*, *tbl*, and *grep*, the three programs that incur the most performance loss in the *w/overhead* measurement, are high-overhead programs as indicated. Figure 15 presents the overhead ratios of these programs graphically.

As suggested earlier in Section IV-C, a *Persistent Rescheduled-Page Cache (PRC)* can be used to effectively reduce the overhead of rescheduling, especially for the *high-overhead* programs like *cccp*, *tbl*, and *grep*. The PRC has been extensively studied in [1] and is briefly described in the following section (Section V-C).

C. Caching techniques to reduce rescheduling overhead

A PRC is organized as a system-wide cache of rescheduled pages, managed by the operating system. Since it is a shared resource (like the file system buffer cache in UNIX, for example), all the programs which warrant invocation of the dynamic rescheduling technique can displace pages stored by each other in the PRC.

During the first invocation of the program under the dynamic rescheduling framework, the program pages are rescheduled at first-time page faults as described in Section IV. While the program executes, these rescheduled pages are written to the text swap area of the disk and re-used during the lifetime of the program. Also during the lifetime of the run, the OS keeps the following information: (1) the number of unique page accesses made by the program, (2) the time spent in rescheduling the pages, and (3) the total execution-time of the program. These values are later used to compute the overhead ratio of the program.

When the program completes execution, the PRC manager module considers the pages used by the program for placement into the PRC. If the PRC has enough empty page slots, then the entire set can be placed in it. If not, this situation calls for replacement of pages from the PRC. The policy used for replacement is called *overhead-based replacement*, which works as follows: if the overhead ratio of the page to be placed into the PRC is higher than any of the existing pages in the PRC, then the page with the least overhead ratio is replaced. This strategy ensures that the priority of use of the PRC is dictated by the overhead ratio of the program, guaranteeing fair treatment to the short-running benchmarks while not hampering the performance of long-running and big benchmarks. At each subsequent execution of the program, the PRC is searched (via a mapping of the program text to the PRC) for existence of the rescheduled page. If available, the rescheduled page is used and the overhead of rescheduling is thus avoided.

The *w/overhead* performance of a subset of the benchmark programs for the TINKER-A to TINKER-C rescheduling case in the presence of a PRC with overhead-based replacement is presented in Figure 16. Multiple workloads were created by randomly interleaving the page access patterns for these programs, and

were used to drive a PRC simulator. The average number of page translations incurred for each program for various sizes of PRCs (in number of pages) were measured, and used to compute the *w/overhead* performance of the program. Note that the *PRC-infinite* case is the same as *no overhead* case explained in Section V-B. It is evident from this Figure 16 that the *high-overhead* benchmarks such as *cccp*, *tbl*, and *grep* perform well with increasing PRC sizes. For smaller PRC sizes, they incur larger overhead due to replacement by each other. For larger PRC sizes, however, no overhead is incurred because of complete caching without any replacement between runs. Hence it can be concluded that the PRC is effective in reducing the effect of overhead on their overall performance across multiple executions. The *low-overhead* and *moderate-overhead* programs compete with each other for the space in the PRC that remains after *high-overhead* programs are cached. Depending upon their relative overhead ratios and number of unique page accesses, they affect each other’s performance. For example, *085.gcc* consistently displaces all other *low-* and *moderate-overhead* programs due to its large size (323 unique pages—see Table IV). This effect is also apparent from this graph.

VI. CONCLUDING REMARKS

Dynamic rescheduling is a practical technique to guarantee VLIW-to-VLIW object-code compatibility. It relies on a limited version of software scheduling applied during first-time page faults. It requires support from the compiler, the ISA, the operating system, and a fast algorithm for rescheduling. As demonstrated in this paper, for the machine models under study and for most benchmarks, the performance of the code after rescheduling compares well with native scheduled code, even after the overhead of rescheduling is taken into account. For the benchmarks which have a short execution-time, and induce a large number of unique pages accesses during execution, the overhead of rescheduling dominates their total run-time. A technique which caches the rescheduled pages between runs of the program, called a *Persistent Rescheduled-page Cache (PRC)* [1], can be effectively used to reduce the rescheduling overhead. For the workloads used to evaluate the PRC in this paper, it is apparent that a PRC of size between 512 to 1024 pages, which uses an *overhead-based* replacement policy would be most appropriate.

The rescheduling algorithm operates on a page of code at a time, which implies that bigger page-sizes would present more opportunity for code motion, and hence potentially more ILP. Larger page-sizes would lead to an increased overhead of rescheduling, but if the PRC is designed carefully to accommodate the page access needs of the workload, it can reduce its effect on performance.

Dynamic rescheduling uses a binary encoding to hide the nops and the empty cycles in the VLIW schedule. The changes in code size that could potentially be prompted at rescheduling time can be accommodated via this encoding, effectively keeping the size of code constant. This encoding, however, has a penalty associated with it: specialized instruction-fetch hardware which is aware of the hidden nops in code must be implemented to handle this encoding. Design of various schemes to realize this hardware and an analysis of its performance are presented in [18].

Another approach to handle the changes in code-size at run-time would be as follows: the code generation phase after initial compile inserts a limited number of nops between SEARs, which are not in the execution

path, but are used to account for any changes in the size due to rescheduling. This approach, however, imposes a strict upper bound on the amount of code motion such that the change in code-size is not excessive so as to cause underflow/overflow at page boundaries. Another drawback of this approach is that the *effective page-size* (the number of executable Ops in a page) is reduced, thus reducing the available ILP within the page.

The algorithm used for rescheduling is a limited version of the acyclic list scheduling algorithm. It is invoked as a module of the OS in the current framework, although hardware implementations are also possible. Such an implementation would also be used only at rescheduling time and will not affect the normal execution pipeline of the machine, and hence would not have the same cycle time drawbacks as the superscalars employing the dynamic scheduling hardware.

REFERENCES

- [1] T. M. Conte, S. W. Sathaye, and S. Banerjia, "A Persistent Rescheduled-Page Cache for low-overhead object-code compatibility in VLIW architectures," in *Proc. 29th Ann. Int'l Symp. on Microarchitecture*, (Paris, France), Dec. 1996.
- [2] J. S. O'Donnell, "Superscalar vs. VLIW," *Computer Architecture News (ACM SIGARCH)*, pp. 26-28, Mar. 1995.
- [3] B. R. Rau, "Dynamically scheduled VLIW processors," in *Proc. 26th Ann. Int'l Symp. on Microarchitecture*, (Austin, TX), pp. 80-90, Dec. 1993.
- [4] S. Melvin, M. Shebanow, and Y. Patt, "Hardware support for large atomic units in dynamically scheduled machines," in *Proc. 21th Ann. Int'l Symp. on Microarchitecture*, (San Diego, CA), pp. 60-66, Dec. 1988.
- [5] M. Franklin and M. Smotherman, "A fill-unit approach to multiple instruction issue," in *Proc. 27th Ann. Int'l Symp. on Microarchitecture*, (San Jose, CA), pp. 162-171, Dec. 1994.
- [6] T. M. Conte and S. W. Sathaye, "Dynamic rescheduling: A technique for object code compatibility in VLIW architectures," in *Proc. 28th Ann. Int'l Symp. on Microarchitecture*, (Ann Arbor, MI), Nov. 1995.
- [7] G. Silberman and K. Ebcioglu, "An architectural framework for supporting heterogeneous instruction-set architectures," *Computer*, vol. 26, pp. 39-56, June 1993.
- [8] R. L. Sites, A. Chernoff, M. B. Kirk, M. P. Marks, and S. G. Robinson, "Binary translation," *Comm. ACM*, vol. 36, pp. 69-81, Feb. 1993.
- [9] J. Turley, "Alpha runs x86 code with fx!32," *Microprocessor Report*, vol. 10, Mar. 1996.
- [10] P. Koch, "Emulating the 68040 in the PowerPC Macintosh," in *Proc. Microprocessor Forum*, Oct. 1994.
- [11] P. Stears, "Emulating the x86 and DOS/Windows in RISC environments," in *Proc. Microprocessor Forum*, Oct. 1994.
- [12] 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.
- [13] M. S. Schlansker and V. K. Kathail, "Techniques for critical path reduction of scalar programs," Tech. Rep. HPL-95-112, Hewlett-Packard Laboratories, Palo Alto, CA, 1995.
- [14] 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.
- [15] 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.
- [16] "TINKER machine language manual," 1995. Department of Electrical and Computer Engineering, North Carolina State University, Raleigh NC 27695-7911.
- [17] 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.
- [18] T. M. Conte, S. Banerjia, S. Y. Larin, K. N. Menezes, and S. W. Sathaye, "Instruction fetch mechanisms for VLIW architectures with compressed encodings," in *Proc. 29th Ann. Int'l Symp. on Microarchitecture*, (Paris, France), Dec. 1996.
- [19] T. M. Conte, *Systematic computer architecture prototyping*. PhD thesis, Department of Electrical and Computer Engineering, University of Illinois, Urbana, Illinois, 1992.

- [20] B. R. Rau, "Iterative modulo scheduling: An algorithm for software pipelining loops," in *Proc. 27th Ann. Int'l Symp. on Microarchitecture*, (San Jose, CA), Dec. 1994.
- [21] B. R. Rau, "Iterative modulo scheduling," Tech. Rep. HPL-94-115, Hewlett-Packard Laboratories, Technical Publications Department, 1501 Page Mill Road, Palo Alto, CA 94304, 1995.
- [22] J. R. Ellis, *Bulldog: A compiler for VLIW architectures*. Cambridge, MA: The MIT Press, 1986.
- [23] 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. Int'l Symp. Computer Architecture*, (Toronto, Canada), pp. 266-275, May 1991.
- [24] Hewlett Packard, *How HP-UX works: Concepts for the System Administrator (R9.0)*. Palo Alto, CA: Hewlett Packard, 1991.
- [25] Data General, *Programming in the DG/UX Kernel Environment (R4.11)*. Westboro, MA: Data General, 1995.
- [26] S. Weiss and J. E. Smith, *POWER and PowerPC*. San Francisco, CA: Morgan Kaufmann, 1994.
- [27] *Proc. 29th Ann. Int'l Symp. on Microarchitecture*, (Paris, France), Dec. 1996.

IALU			IALU			MUL			LD		ST	
<i>1 cycle latency</i>			<i>1 cycle latency</i>			<i>3 cycle latency</i>			<i>2 cycle latency</i>		<i>1 cycle latency</i>	
<i>d</i>	<i>s</i>	<i>s</i>	<i>d</i>	<i>s</i>	<i>s</i>	<i>d</i>	<i>s</i>	<i>s</i>	<i>d</i>	<i>addr</i>	<i>addr</i>	<i>s</i>
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

Fig. 1. Scheduled code for original VLIW machine.

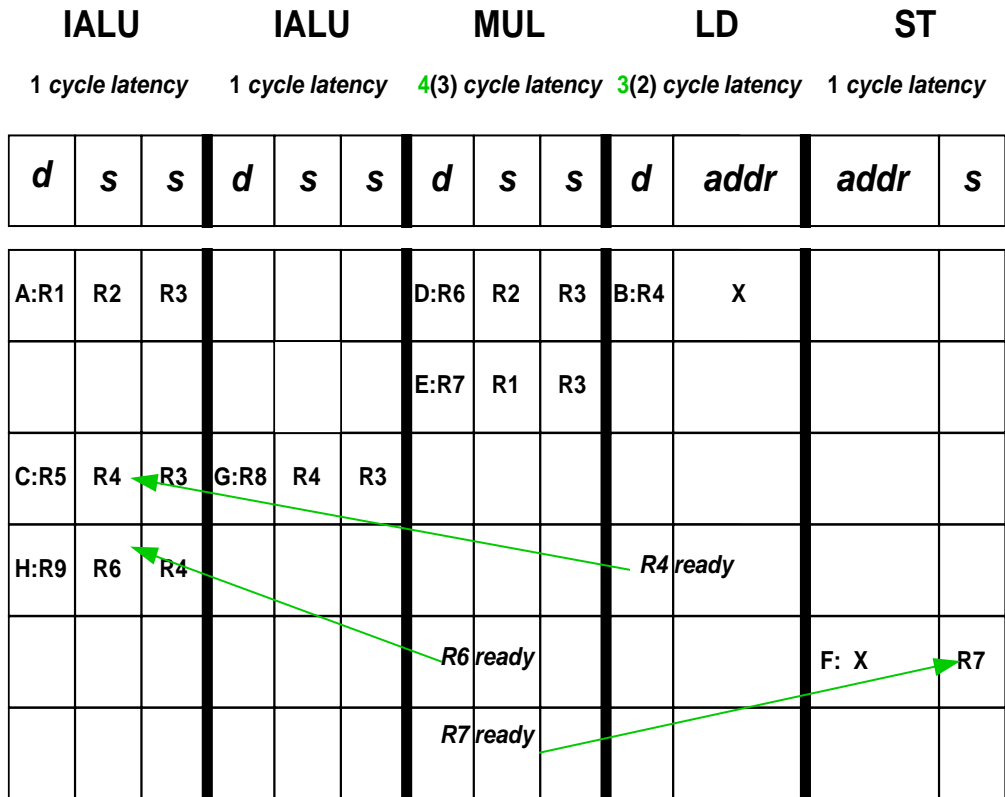


Fig. 2. Next generation VLIW machine: Incompatibility due to changes in functional unit latencies (shown by arrows). The old latencies are shown in parentheses. Operations C, H, and, F now produce incorrect results because of the new latencies for operations B, D, and E.

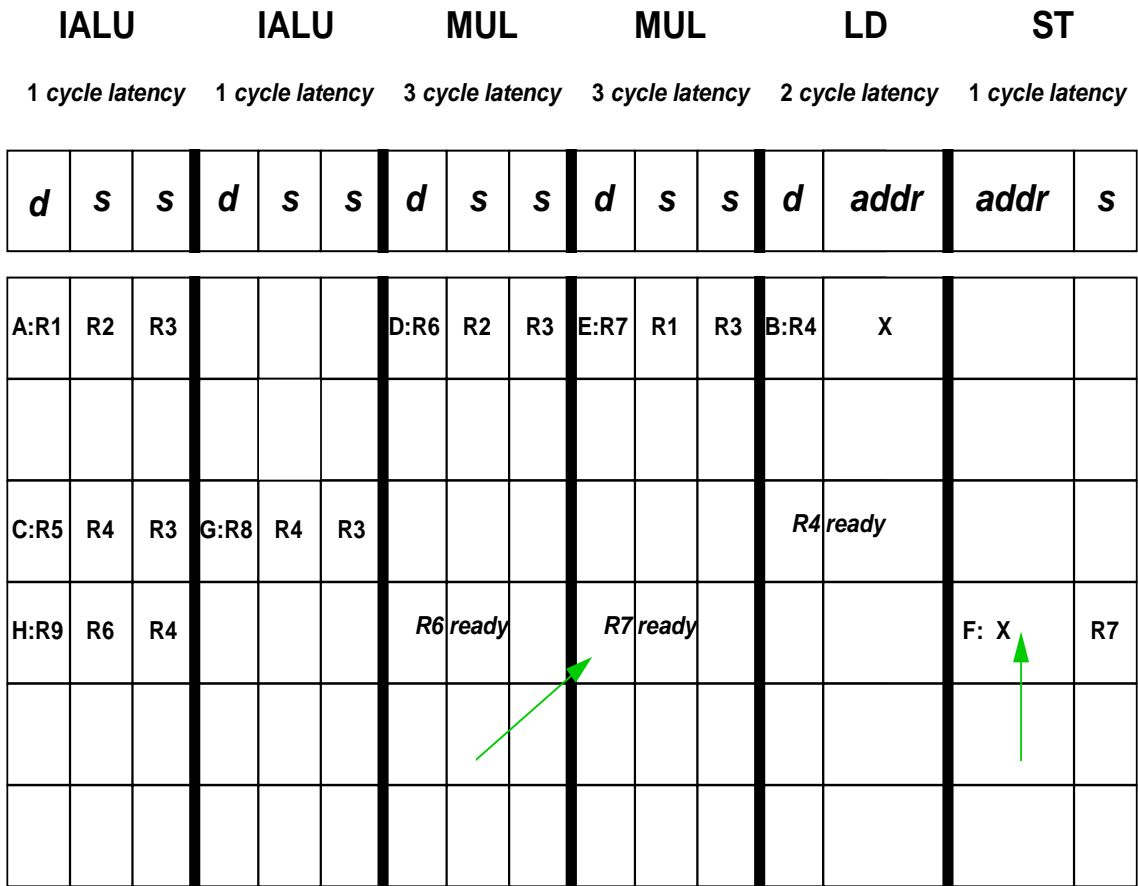


Fig. 3. Alternative next-generation machine: downward incompatibility due to change in the VLIW machine organization: no trivial way to translate new schedule to older machine.

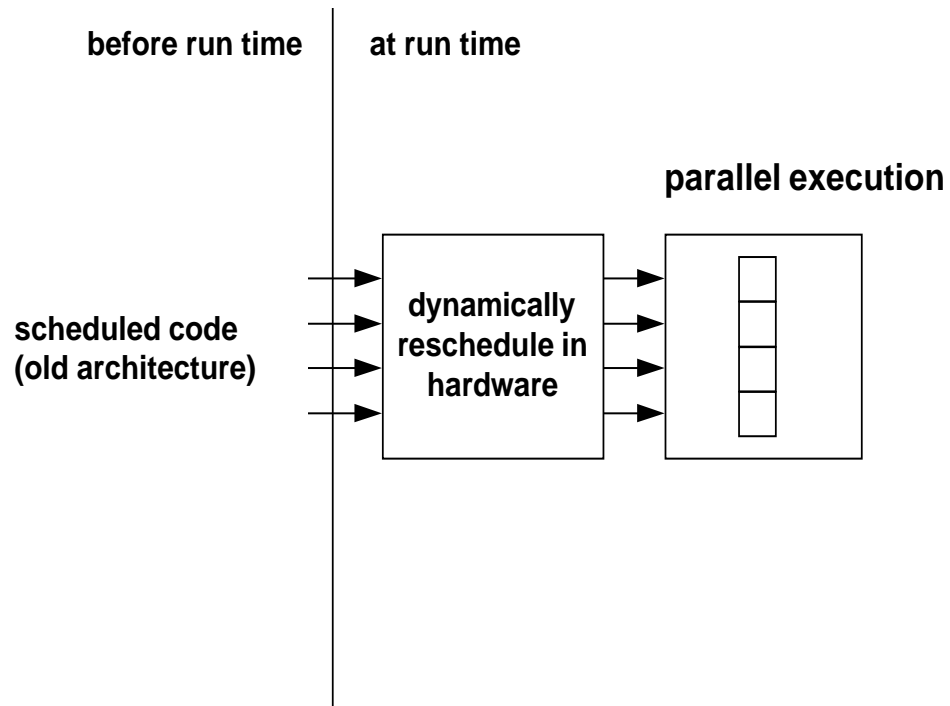


Fig. 4. Hardware approach to compatibility.

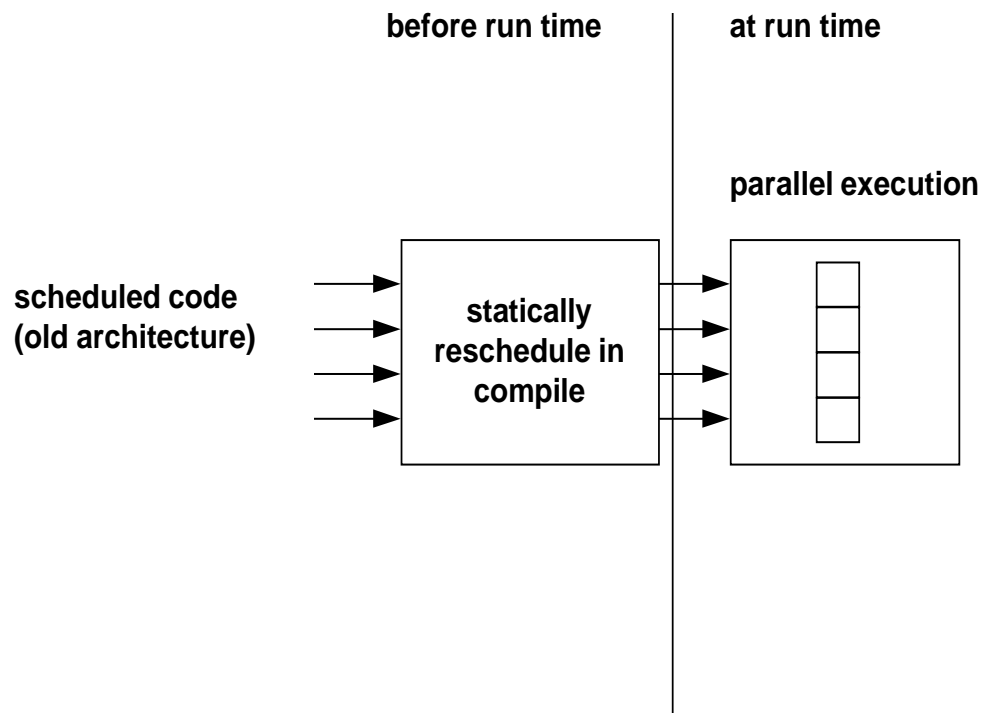


Fig. 5. Rescheduling the program off-line for compatibility.

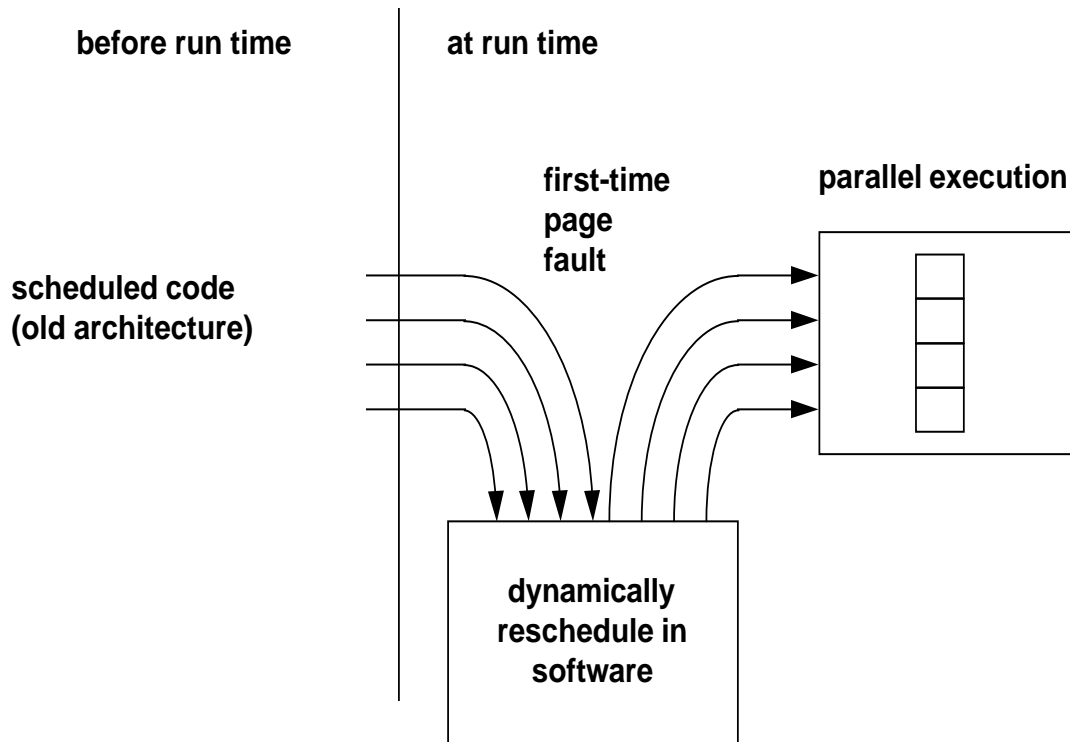


Fig. 6. Dynamic Rescheduling.

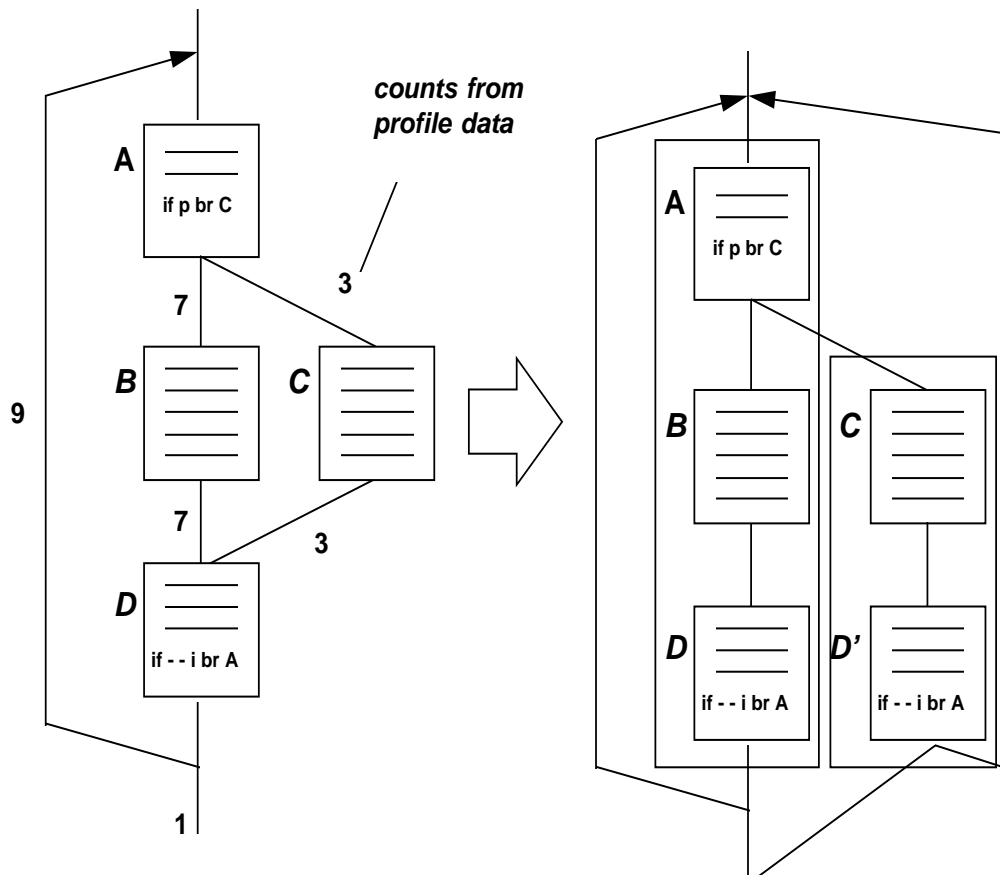


Fig. 7. Example of Superblock formation. Note that both Superblocks and Hyperblocks have a single entry, multiple exits and no side entrances.

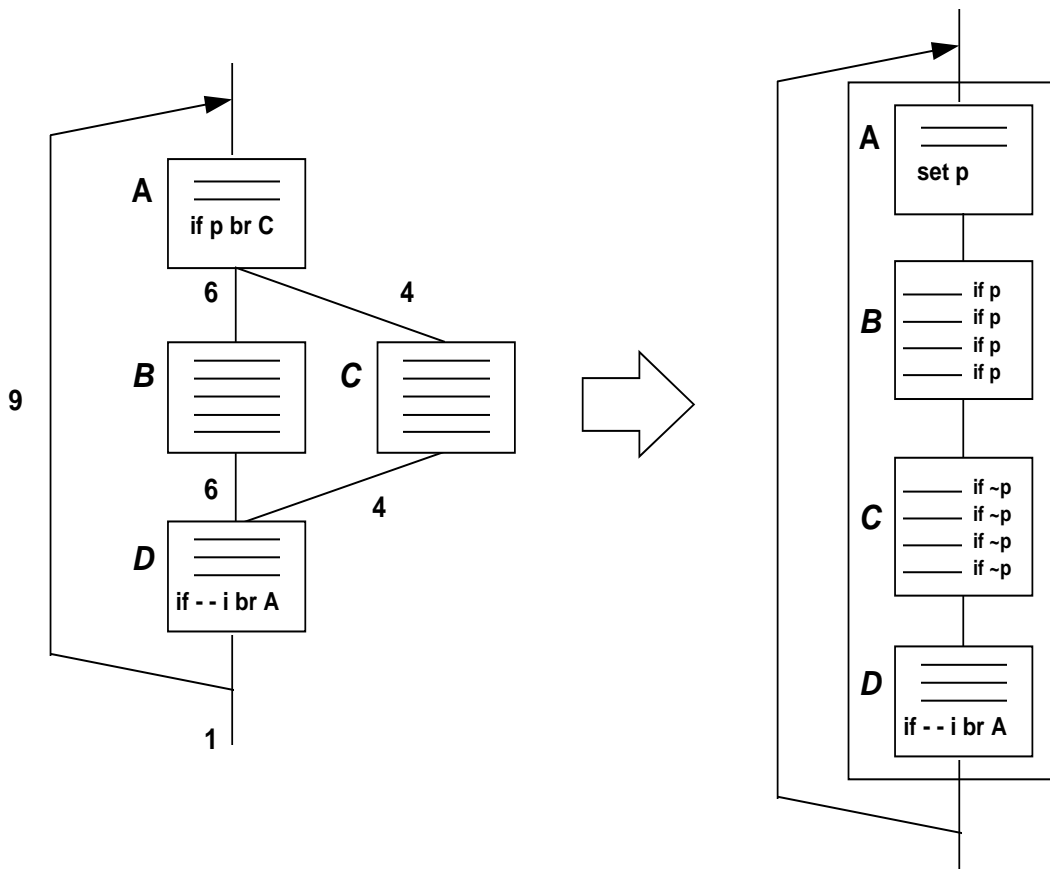


Fig. 8. Example of Hyperblock formation.

IALU	IALU	FPAAdd	FPMul	Load	Store	Cmpp	Br
A	<i>nop</i>	B	<i>nop</i>	C	D	<i>nop</i>	<i>nop</i>
<i>nop</i>	<i>nop</i>	<i>nop</i>	<i>nop</i>	<i>nop</i>	<i>nop</i>	<i>nop</i>	<i>nop</i>
E	F	<i>nop</i>	<i>nop</i>	<i>nop</i>	<i>nop</i>	<i>nop</i>	<i>nop</i>
G	<i>nop</i>	<i>nop</i>	<i>nop</i>	<i>nop</i>	<i>nop</i>	<i>nop</i>	H

E, F dependent on C, C takes 2 cycles

256 bytes total

*Load latency increases,
one less IALU*



IALU	FPAAdd	FPMul	Load	Store	Cmpp	Br
A	B	<i>nop</i>	C	D	<i>nop</i>	<i>nop</i>
<i>nop</i>	<i>nop</i>	<i>nop</i>	<i>nop</i>	<i>nop</i>	<i>nop</i>	<i>nop</i>
<i>nop</i>	<i>nop</i>	<i>nop</i>	<i>nop</i>	<i>nop</i>	<i>nop</i>	<i>nop</i>
E	<i>nop</i>	<i>nop</i>	<i>nop</i>	<i>nop</i>	<i>nop</i>	<i>nop</i>
F	<i>nop</i>	<i>nop</i>	<i>nop</i>	<i>nop</i>	<i>nop</i>	<i>nop</i>
G	<i>nop</i>	<i>nop</i>	<i>nop</i>	<i>nop</i>	<i>nop</i>	H

336 bytes total (10 extra nops)

Fig. 9. Page size change due to no-ops.

Original code

63	0
1 0 1	A
0 2 x	B
0 4 x	C
0 5 x	D
1 0 0	E
0 1 x	F
1 0 0	G
0 7 x	H

Rescheduled code

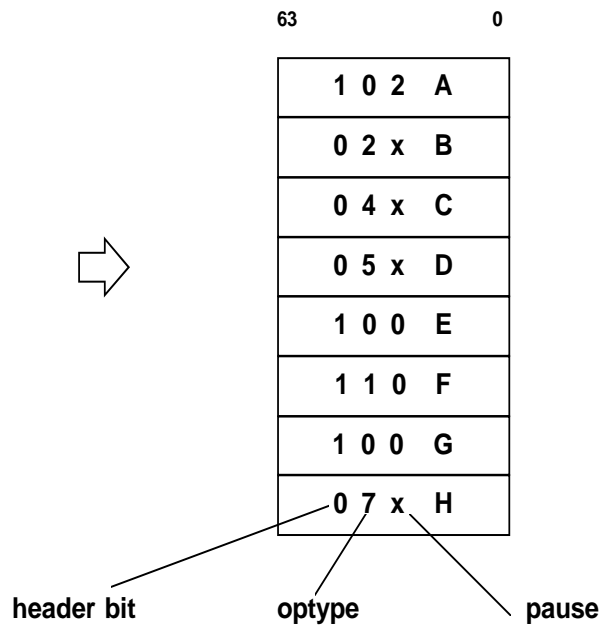


Fig. 10. An example of the TINKER instruction encoding scheme. Each Op is a fixed-format 64-bit word. The format includes the *Header Bit*, *optype*, and *pause* fields, which together eliminate the need for *nops* in the code.

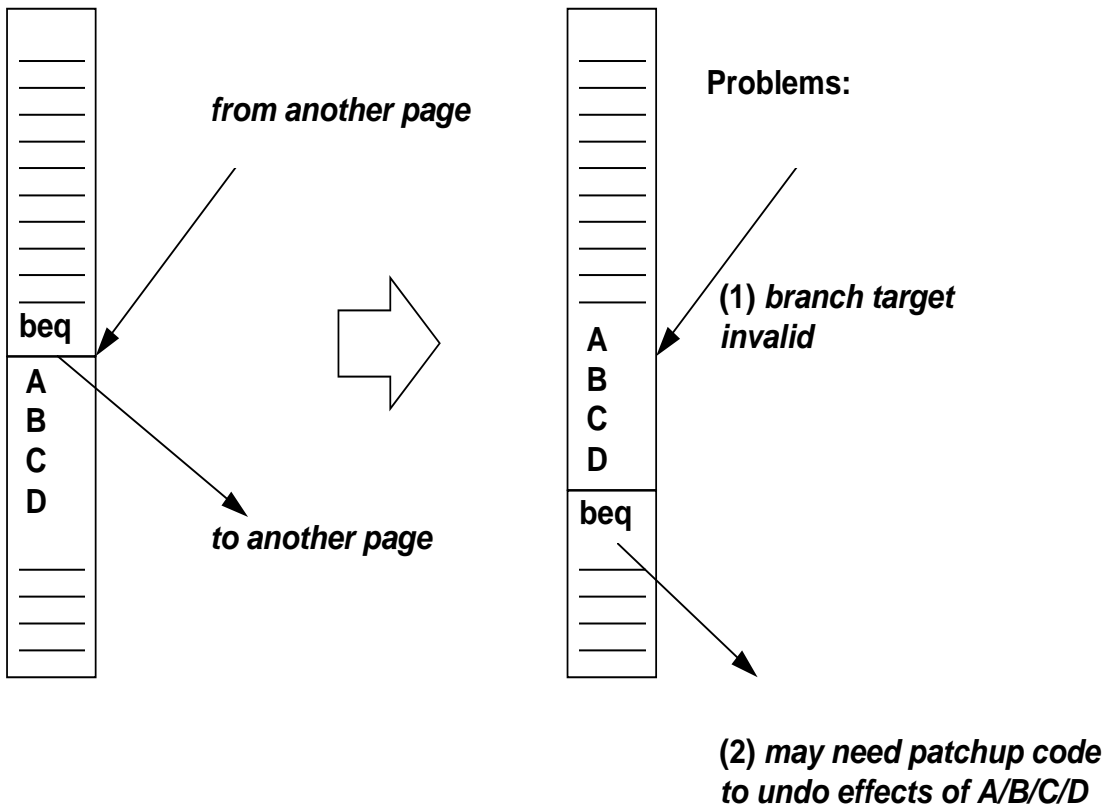


Fig. 11. Problems introduced by speculative code motion.

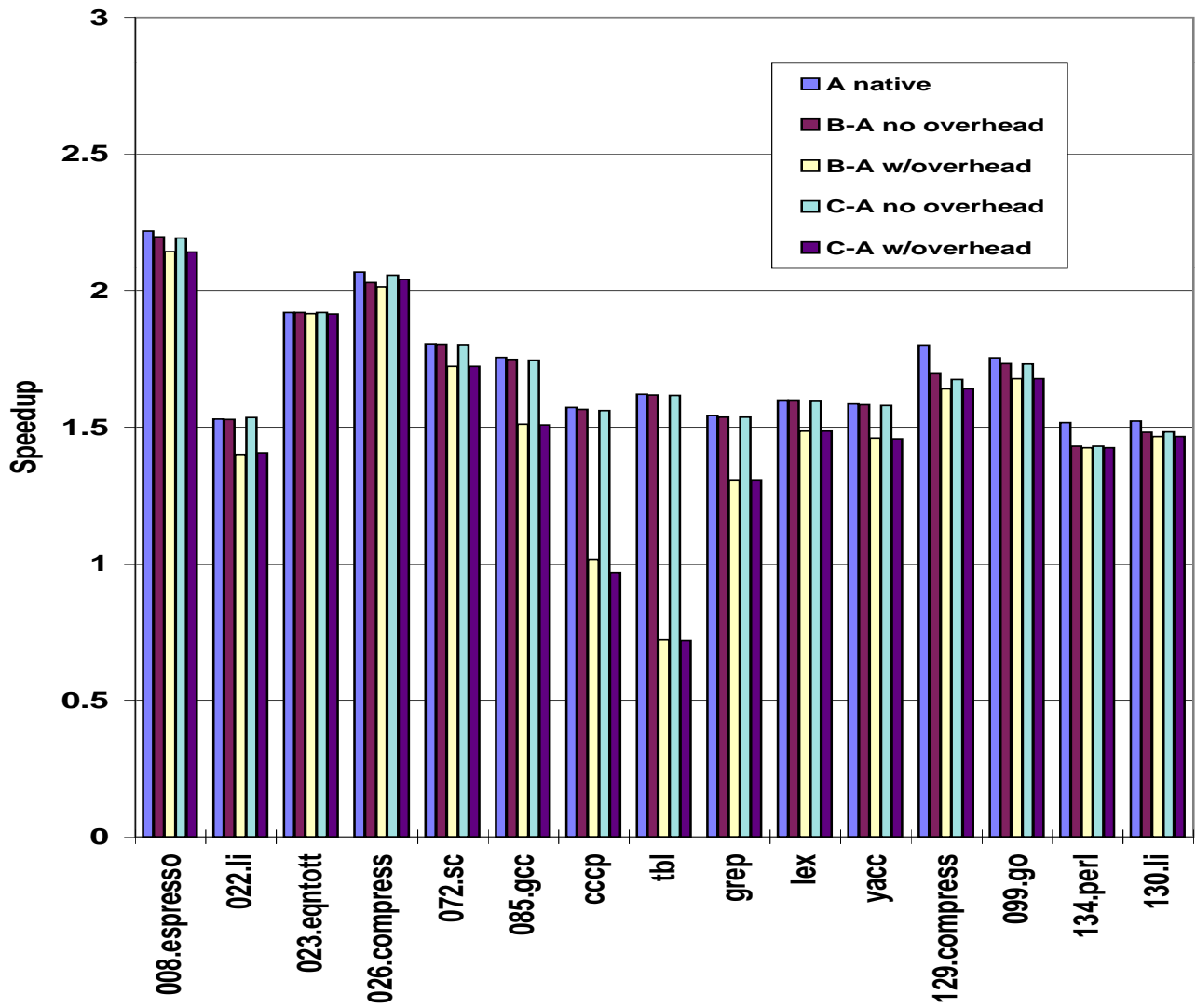


Fig. 12. Performance of dynamic rescheduling to TINKER-A machine model.

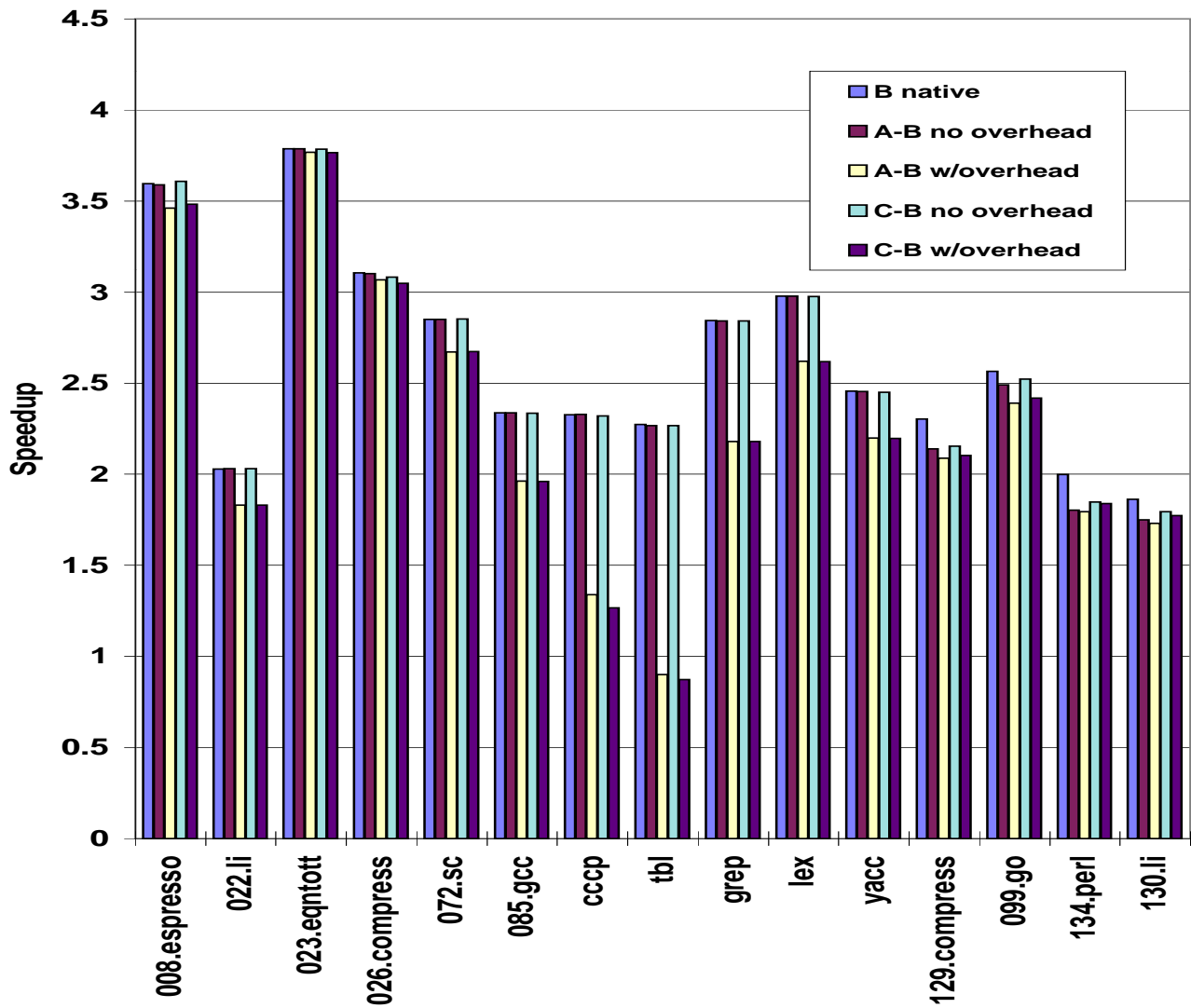


Fig. 13. Performance of dynamic rescheduling to TINKER-B machine model.

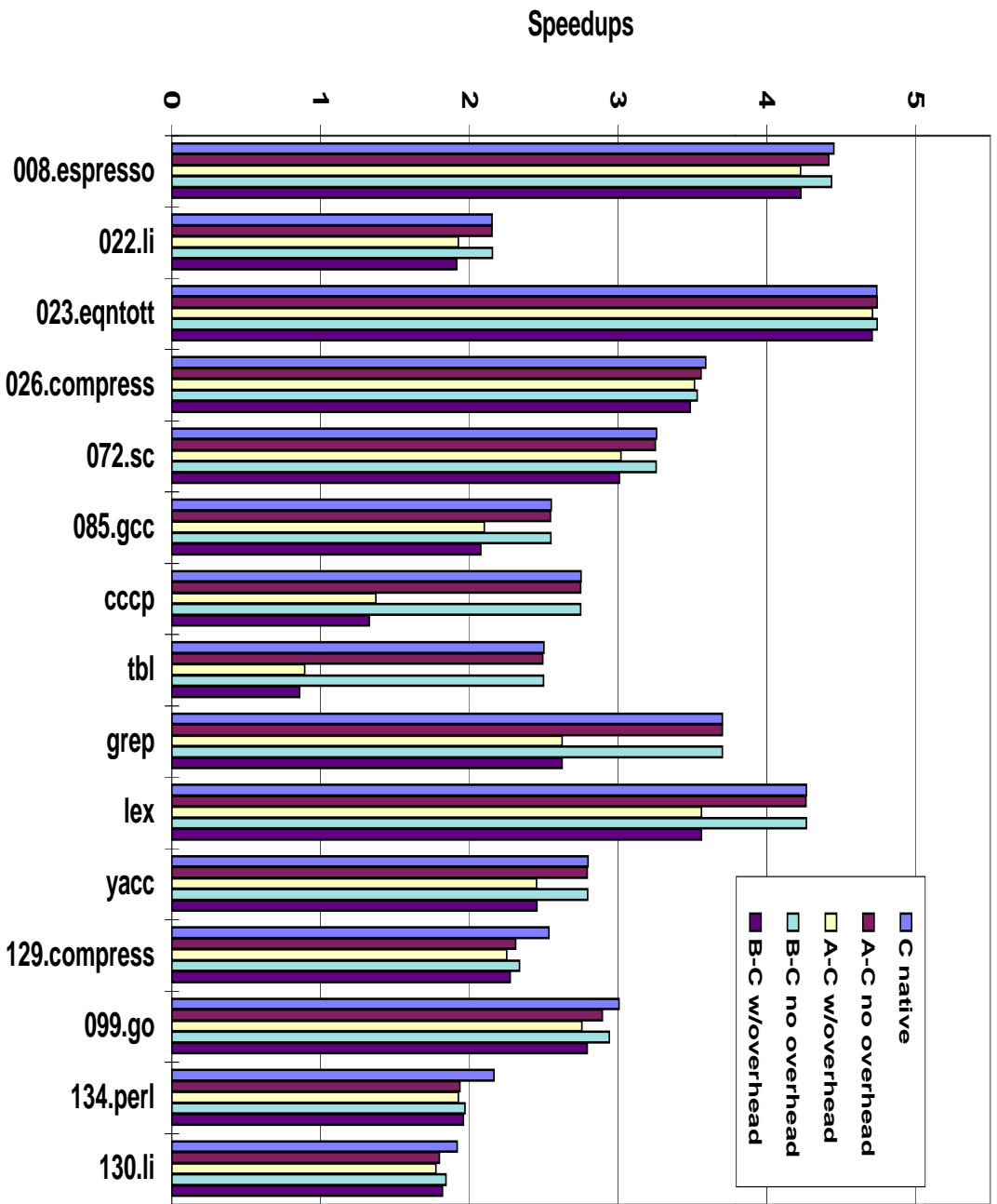


Fig. 14. Performance of dynamic rescheduling to TINKER-C machine model.

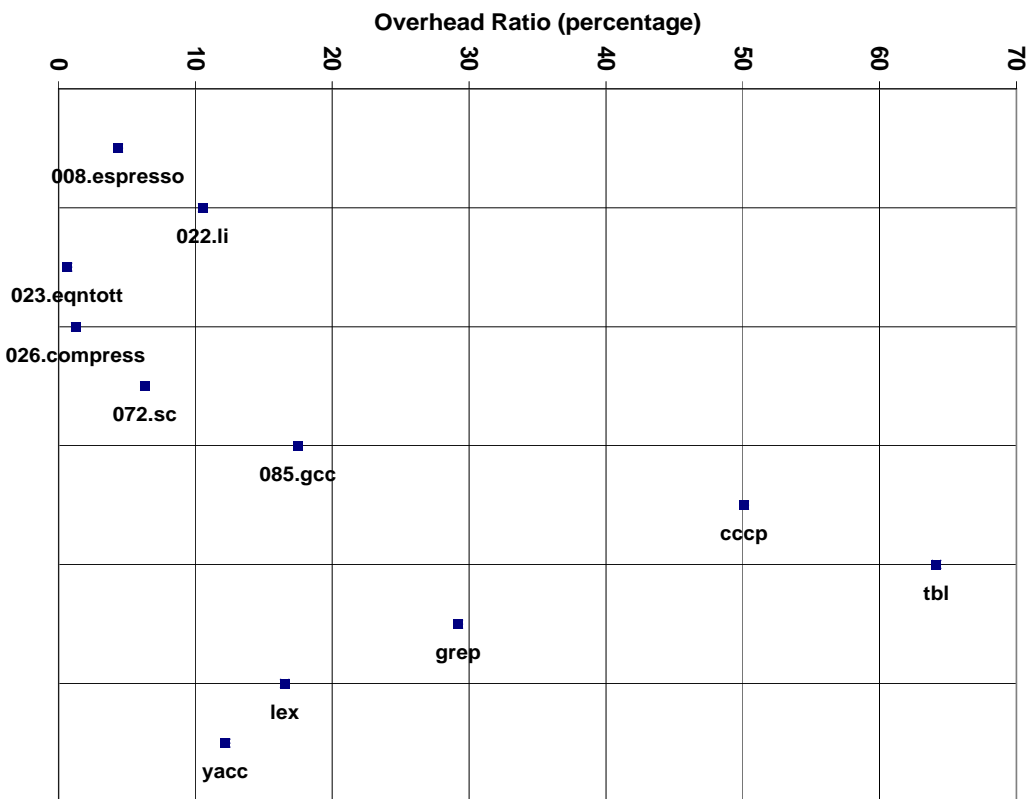


Fig. 15. Overhead Ratios (percentage) for TINKER-A to TINKER-C rescheduling case.

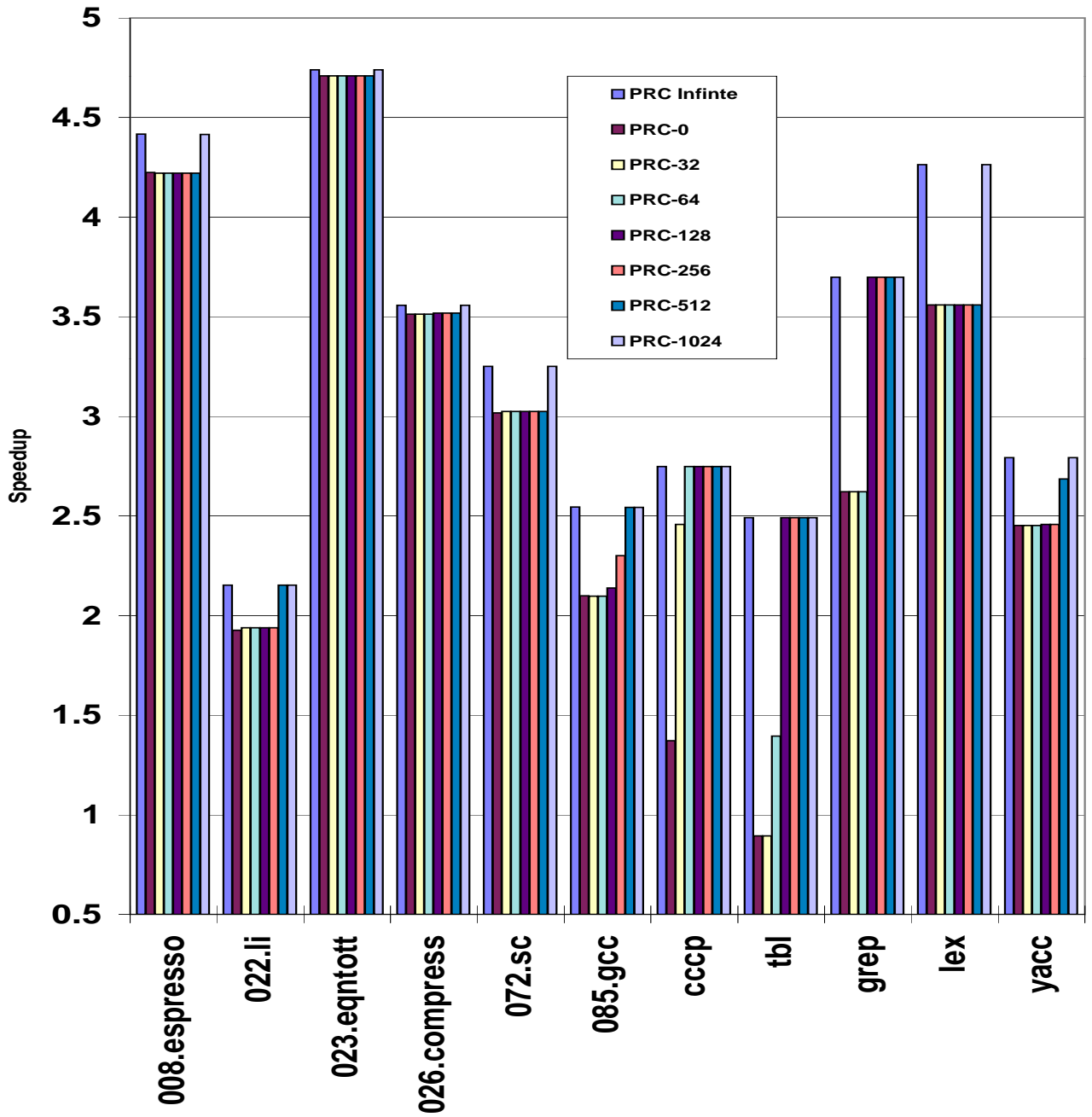


Fig. 16. Performance of the Persistent Rescheduled-Page Cache (PRC) for the TINKER-A to TINKER-C translation. The PRC- n labels indicate a PRC of size n pages. The organization is a single, shared PRC with *overhead-based* replacement policy.