A Persistent Rescheduled-Page Cache for Low Overhead Object Code Compatibility in VLIW Architectures

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

Abstract

Object–code compatibility between processor generations is an open issue for VLIW architectures. A potential solution is a technique termed dynamic rescheduling, which performs run-time software rescheduling at the first-time page faults. The time required for rescheduling the pages constitutes a large portion of the overhead of this method. A disk caching scheme that uses a persistent rescheduled-page cache (PRC) *is presented. The scheme reduces the overhead associated with dynamic rescheduling by saving rescheduled pages on disk, across program executions. Operating system support is required for dynamic rescheduling and management of the PRC. The implementation details for the PRC are discussed. Results of simulations used to gauge the effectiveness of PRC indicate that (1) the PRC is effective in reducing the overhead of dynamic rescheduling, and (2) due to different overhead requirements of programs, a* split *PRC organization performs better than a* unified *PRC.*

The unified PRC was studied for two different page replacement policies: LRU and overhead-based *replacement. It was found that with LRU replacement, all the programs consistently perform better with increasing PRC sizes, but the* high-overhead *programs take a consistent performance hit compared to the* low-overhead programs. *With overhead-based replacement, the performance of* high-overhead *programs improves substantially, while the* low-overhead *programs perform only slightly worse than in the case of the LRU replacement.*

1 Introduction

Unlike contemporary superscalar processors [1] [2] [3] which employ dynamic scheduling, VLIW processors depend on a schedule of code generated by the compiler. The compiler has full knowledge of the machine model, described in terms of the hardware resources available, and the latencies related to execution on each resource. Correct execution of a program scheduled under one machine model assumptions is guaranteed only on processors that have exactly the same machine model or, its supersets where the assumptions are strictly held. Thus, a program scheduled for a particular generation in a VLIW family cannot be guaranteed to be binary compatible with other generations. This is known as the object–code compatibility problem in VLIW architectures [4]. Lack of object-code compatibility is a commonly cited reason why VLIWs may not become a general–purpose computing paradigm [5]. Solutions to the problem have been suggested and can be classified as hardware or software approaches. Hardware techniques typically employ scheduling hardware [6], [7], [4], [8], [9], which could substantially increase the hardware complexity of the machine. A common software approach is that of off–line recompilation of source programs, which yields excellent performance because the compiler has access to all the necessary information to expose the ILP in the program. The drawback of this technique is that it is cumbersome to use, because access to source code may not always be possible. A variant of this is off–line object-code translation, which is more practical when the source code is unavailable or a large set of programs for the older architecture exists [10]. Alternatively, an interpreter could be used to translate between the architectures at run–time [11] but this approach usually suffers from poor performance.

Another approach is *dynamic rescheduling* [12]. In this technique, a program binary scheduled for the target machine model of a VLIW generation is allowed to directly execute on any other generation of the VLIW. As the execution proceeds, each page-fault generated in the program instruction space results into a special action: the page being fetched is rescheduled for the machine model of the target generation. Per-page rescheduling is performed only at the first instance the page-fault occurs (i.e., only at *first-time*

1072-4451/96 $5.00 © 1996 IEEE

page faults). A page of code is considered as an atomic unit for rescheduling. Implementation of this technique requires support from the operating system, specifically in the page fault service routine. The extra time incurred for rescheduling constitutes the overhead of this technique.

This paper presents a scheme that reduces the overhead associated with dynamic rescheduling by caching on-disk rescheduled code on a per–page basis. It employs a structure called a *persistent rescheduled-page cache (PRC)* that holds rescheduled pages across multiple executions of a program. The organization of this paper is as follows. Section 2 reviews previous work on object–code compatibility and explains dynamic rescheduling and presents performance measurements of code subjected to dynamic rescheduling. Section 3 introduces the PRC as part of an operating system–managed disk caching scheme that reduces the overhead of dynamic rescheduling. The architecture and management of the PRC is detailed, and experimental results that measure its performance are presented. Section 4 presents the conclusions of the study and avenues for future work.

2 Object–Code Compatibility and Dynamic Rescheduling

Several hardware approaches have been reported previously to address the VLIW compatibility problem. Rau presented a technique called split-issue to perform superscalar-style dynamic scheduling of code in hardware [4]. The *fill-unit*, originally proposed by Melvin, Shebanow, and Patt [6], and later extended by Franklin and Smotherman [8] can be adapted to achieve a limited level of compatibility in VLIWs. These approaches are counter to the principle of hardware simplicity, one of the tenets of the VLIW philosophy. An obvious alternative is off–line recompilation and rescheduling. This technique has an advantage in terms of the performance but is extremely cumbersome to use, owing to its off-line nature. Similar to this is off–line binary translation, which was used to migrate VAX software to the Alpha platform [10]. Run–time software emulation can also be used, as implemented in Insignia Solution's SoftWindows product [13]. An approach employing both the emulation and translation techniques has been utilized in the FX!32 product from DEC [14]. May investigated using an interpreter to execute IBM System/370 binaries on an IBM RT PC [11]. These solutions attempt to achieve compatibility between widely varying architectures. Although such large differences will probably not exist between generations of a VLIW family, the techniques used are still of interest.

Interpreters can use caching of translated sections of code across executions as a means of reducing the amount of translation needed; disk storage is used as a cache for the translated code. Caching is an intuitive and effective technique for reducing run–time translation overhead. However, a majority of the literature on translation does not contain details on management of cached code segments.

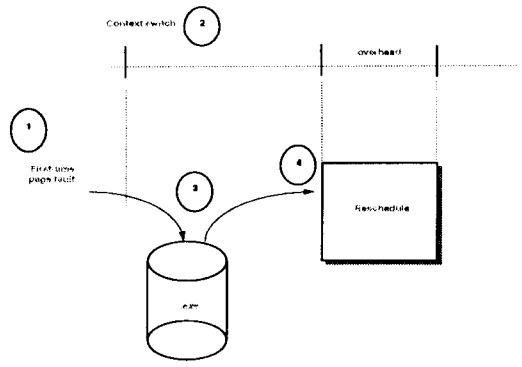

Figure 1. The sequence of events in Dynamic Rescheduling.

Events 1–3 are the detection of the page fault and generation mismatch, the context switch of the process, and the retrieval of the page from disk, respectively; these events are standard. The overhead of dynamic rescheduling is event 4, where the page is dynamically rescheduled.

Another technique for compatibility is *Dynamic Rescheduling (DR)* [12]. At all first–time page-faults, the OS invokes a module called the *dynamic rescheduler* to reschedule the page being accessed, for the host machine model. The sequence of events in dynamic rescheduling is illustrated in Figure 1. The detection of a first–time page fault is shown as Event 1. Events 1, 2, and 3 always take place at a page fault. In the case of dynamic rescheduling, however, this is treated as a special page fault; this is indicated as Event 4. The program binary contains a complete description of the machine for which it was originally compiled. If the page fault handler detects a generation mismatch between the host machine and the program, the rescheduler module is invoked. After the page is rescheduled, program execution resumes. If the rescheduled page is selected for replacement from physical memory before the program terminates, it is written to the text swap space [15], [16]; this eliminates the need to reschedule if the page is re–accessed. The net overhead of the dynamic rescheduling technique can be quantitatively expressed in terms of the following three factors: (1) the time spent in rescheduling pages at run-time, (2) the time to write rescheduled pages to text swap when pages are replaced, and (3) the amount of disk space used to save the rescheduled pages. The time for disk I/O during page replacement is negligible as writing to swap can be performed asynchronously [15]. The true overhead is reduced to the time for rescheduling and the disk space required to

save rescheduled pages.

If dynamic rescheduling performed code motion, it would give rise to code size changes due to the compensation code it may have to insert and/or delete. Since it is not straightforward to handle any changes in code size dynamically, the DR framework avoids the size changes altogether, via a special binary encoding which eliminates explicit use of NOPs from the code. More discussion of the issues involved therein can be found in [12], and are beyond the scope of this paper. The PlayDoh [17] VLIW architecture from Hewlett-Packard Laboratories is used as a testbed in this paper. Also, it is assumed that no modifications other than the DR framework itself are made in the instruction space of the programs (such as those by self-modifying code).

2.1 Performance of Dynamic Rescheduling

The effectiveness of dynamic rescheduling was measured using programs from the SPECint92 suite[1] and several UNIX utility programs in [12], and is expanded here. In order to construct an interactive load, the benchmarks were divided into two categories: tools (cccp, compress, gcc, grep, and tbl) and applications (espresso, eqntott, li, lex, sc, and yacc). It was assumed that tools would be invoked twice as often as applications, but that there would be no other pattern in the workload. Two sets of inputs were used for each benchmark, and were alternated between invocations of the benchmark. Using these assumptions, several workloads were created and used to measure the performance of the benchmarks with and without dynamic rescheduling. Three machine models were used for the evaluation of the performance of the benchmarks in situations that required dynamic rescheduling: Generations 1, 2, and 3 (their organizations are shown in Figure 2). The types of functional units are shown horizontally, while the execution latency assumptions are shown vertically. The unit `pred` is used to perform predicate computation in integer compare operations. Predicate computation in FP compare operations is done in the FPAdd unit. A three part method was used to evaluate the dynamic rescheduling technique. In the first part, intermediate code for a benchmark was scheduled for a given machine model, using a VLIW scheduler; hyperblock scheduling was used for the initial compilation [18]. The intermediate code was then profiled in order to find the worst–case estimate of execution time in terms of the number of cycles. The number of times each page of code is accessed is also recorded, which also indicates each unique code page that is accessed. This was called the *Native* mode experiment. In the second part, the code scheduled for *Native* mode execution was rescheduled for the other machine models. Execution time estimates for the rescheduled code were also generated as described before. This time estimate indicates the performance of the rescheduled code without taking into account the rescheduling overhead incurred by the rescheduler. Hence this part was termed the *no overhead* experiment.

[1] Performance characteristics of none of the FP programs in the presence of the PRC are presented because rescheduling of software pipelined loops has not yet been implemented in the current dynamic rescheduling framework.

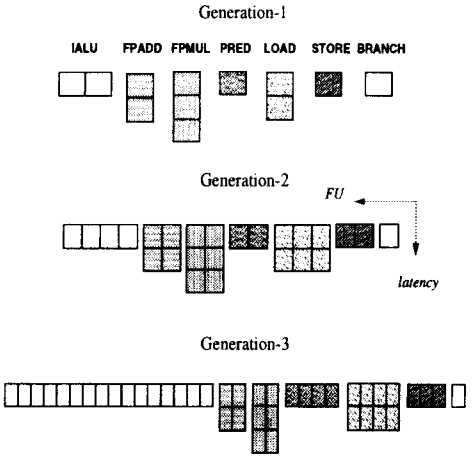

Figure 2. Simulated machine models.

In the third part, the rescheduler itself was compiled, scheduled for the machine model used in the first part, and then used as a benchmark. The input to the the rescheduler benchmark was pages taken from each of the other benchmarks. The performance of the rescheduler benchmark was used to find the average time to reschedule a single 4K page for each of the three machine models. This was found to be 54,272 cycles for Generation–1, 51,200 cycles for Generation–2, and 48,108 cycles for Generation–3. This was then combined with the number of unique page accesses from the first part of the experiment to estimate the total number of execution cycles for the rescheduling overhead. The rescheduling overhead was added to the execution times from the no–overhead experiment to derive execution times for the *w/overhead* experiment. Finally, to compare the performance achieved in the above three parts, the speedup with respect to a single-unit, single-issue processor model (the *base model*) was calculated. Speedup is: (*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, as is used in many contemporary operating systems [19] [20] and processors [21] [3]. Results from all three parts are shown in Figure 3. It can be observed that the no-overhead speedup of rescheduled benchmark code (marked G_{n-m}^{no}) has comparable per-

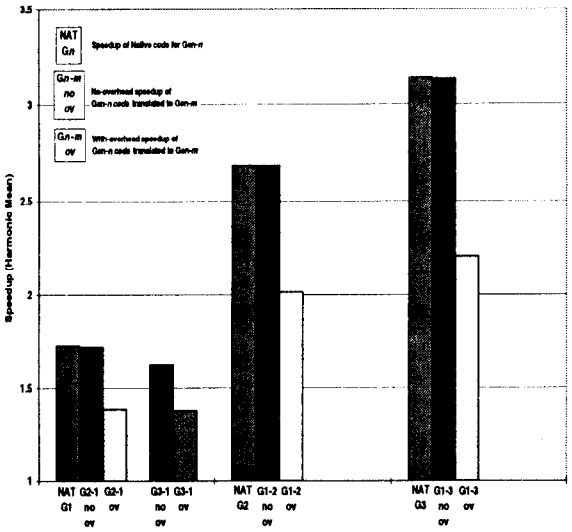

Figure 3. Speedups for programs under dynamic rescheduling framework.

Each bar is the harmonic mean of speedups of benchmarks for a specific generation–to–generation rescheduling. Each set of bars is identified on the x-axis by a Gn-m label, which indicates that code originally scheduled for Generation–n was rescheduled for Generation-m. The first bar in each set is native performance on Generation-m, the remaining bars show rescheduled performance, both with and without the overhead due to dynamic rescheduling.

formance as the native compiled code ($\frac{NAT}{Gn}$), but there is a prominent performance degradation when rescheduling overhead is taken into account ($\frac{Gn-m}{ov}$).

The dominant time in page fault handling is the time spent reading the page from disk if the executable is stored on the local machine. If the page is being accessed over a LAN, as is common in a client-server environment, the network latency is the most dominant factor, and is at least an order of magnitude more than the local disk access. The time spent rescheduling a page adds to this page access latency. To get an estimate of how much overhead DR would add to page fault handling in a local paging environment, an experiment was conducted to measure the page-fault service time when paging from a set of contiguous blocks on a local disk. The SAR performance analysis tool [22] was used to measure average page fault service time on two hardware platforms and was found to be about 2500 μs [2], when averaged over a total of 50 page-faults. Based on average execution times on a 100 MHz machine, dynamic rescheduling increases the page fault service time by about 20%, a significant increase (rescheduling a 4 kilobyte page, with 64-bit operations, from the Generation-2 code to Generation-1 code, for example, would take 54,272 cycles times 10 ns, which is approximately 543μs, more than 20% of the average page-fault service time measured here). It is apparent that in the case of paging over a LAN, the relative overhead of DR will be lower, but probably not small enough to completely neglect it.

There are two approaches that can be used to reduce overhead of DR: (1) improve the performance of the DR algorithm, or (2) reduce the number of pages that require rescheduling over multiple invocations of the program. The second component is the focus of the investigation in this paper. The OS saves rescheduled pages in the text swap if a program is swapped out during execution. These rescheduled pages are effectively *cached* on disk during a program's current execution. Extending this concept, the OS can aid in caching rescheduled pages not only during a single program execution but across multiple program executions as well. The following section introduces an OS–supported caching scheme that achieves this.

3 Disk caching and the Persistent Rescheduled-Page Cache

The significant overhead introduced by dynamic rescheduling can be largely alleviated through the use of a caching scheme that employs a *persistent rescheduled-page cache (PRC)*. A PRC is an OS–managed disk cache that holds binary images of rescheduled pages; if rescheduled pages are cached, they will not need to be rescheduled when re–accessed across program runs. A PRC is defined by its behavior during a program execution and at program termination. During execution, rescheduled pages are stored in text swap space. When a program terminates, its rescheduled pages are written to the PRC. Page placement and replacement policies are implemented within the OS.

The concept behind the persistent rescheduled-page cache originated from software–managed disk caching, a proven method for reducing I/O traffic [15], [16]. The idea is to cache the most recently accessed disk blocks in memory in the hope that they will be accessed again in the near future. Typical Unix systems implement a file system buffer cache to hold the most recently accessed disk blocks in memory using a LRU algorithm for replacement [15], [16]. The buffer cache effectively operates as a fragmented page cache also

[2] SAR was run on a 133 Mhz Pentium–based Data General computer running DG–UX, and a 99 MHz Hewlett–Packard 9000/715 computer running HP–UX.

and reduces the amount of I/O during page faults [3]. Variants of this have been used in distributed file systems such as Sprite [23] and Andrew [24], for the purposes of remote file caching. Disk caching has also been used to reduce translation overhead in architectural interpreters [11], and can be used effectively with dynamic rescheduling.

3.1 Persistent Rescheduled-Page Caches

A PRC can be organized as a system–wide cache: a portion of the file system that holds only rescheduled pages and managed by the OS. Pages from different programs can displace each other in this case. The primary configuration parameters for the PRC are the size, placement and replacement policies, and are discussed below.

During initial program execution on a non–native host, text pages are rescheduled at first–time page faults, as described in Section 2. If a rescheduled page is displaced from physical memory during program execution, it is written to the text swap space on disk; this prevents rescheduling a page multiple times during program execution. At the end of the initial execution, all of the rescheduled pages are written to the PRC (the page placement policy will be explained shortly). During subsequent executions, when a page fault occurs for a text page with a generation mismatch, the PRC is probed to check for the presence of a rescheduled version of the page. If a rescheduled version is present, it is retrieved and loaded into physical memory, and the overhead of rescheduling is not incurred. If a rescheduled version is not available, the page is retrieved from the binary image of the program, rescheduled, and then loaded into physical memory. If this page is replaced during program execution, it is written to the text swap space. At program termination, all rescheduled pages of a program are written to the PRC. A rescheduled page can be placed into any entry in the cache, which implies that the cache is effectively fully associative with a replacement policy such as LRU. An outline of the algorithm used for PRC management is shown in Figure 4.

Probing the PRC on disk for the presence of a page is an expensive operation. This can be eliminated by modifying a program's disk block pointers during execution and program exit. When a page is rescheduled and subsequently written to the PRC at program termination, a separate set of disk block pointers (*PRC pointers*) for the program are set to point to the rescheduled versions of pages in the cache. A PRC pointer is annotated to indicate which original page the rescheduled page replaces. This scheme implements a PRC probe as an examination of a program's disk block pointers in one central location rather than multiple locations in the PRC on disk. Disk block pointers can be cached by the OS, which can further reduce the number of disk probes. The

[3]The term "fragmented" is used because all of the disk blocks that comprise a page might not be in the buffer cache at the same time.

- Load program: if rescheduled pages exist, set page table entries to disk addresses in PRC.
- Run program.
- At program termination, write all rescheduled pages into PRC. If needed, displace LRU pages from PRC.
- Update program PRC disk block pointers to point to PRC. Update PRC pointers to point to program file data structure.

Figure 4. Persistent Rescheduled-Page Cache (PRC) management algorithm.

rescheduled version of a page is accessed without probing the PRC on disk and perhaps without any disk accesses at all, if the program's disk block pointers are in the in–memory cache. Rescheduled pages that are stored in swap during program execution are managed by the OS using known methods for managing text swap space [15].

The probing of multiple PRC pointers during page faults can also be eliminated. At program load time, the disk block and PRC pointers for the program are examined to determine if rescheduled versions of pages exist in the PRC. If a page has a rescheduled version, the loader modifies the page table entry (PTE) for that page to point to the rescheduled version. When a page fault occurs for a page that has a rescheduled version, the rescheduled version is accessed directly, using the updated PTE. No disk accesses or in–memory searches are required to implement a PRC probe: if the PTE for a page points into the PRC, a rescheduled version exists. The re-mapping of the PTE entries can be done at program load time [15], [16]. As rescheduled pages are displaced from the use of a PRC due to replacement, the PRC pointers to the displaced pages must be nullified. This can be accomplished with OS support. A table can be maintained by the PRC manager that lists the locations of the PRC pointers associated with each page in the cache. When a page is replaced, its PRC pointer is set to null.

3.2 PRC performance

Figure 5 presents the performance of PRCs of different sizes for rescheduling across four generation–to–generation combinations. The metric used is speedup over a single universal–unit, single–issue processor. Each set of bars shows the harmonic mean of speedups for a rescheduling

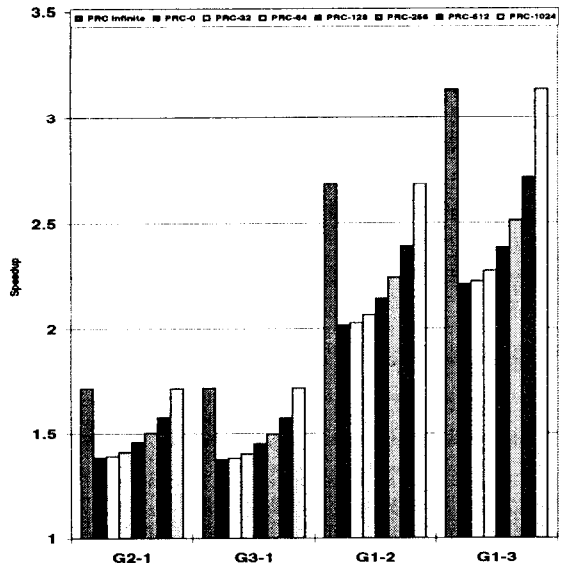

Figure 5. Speedups of benchmarks for PRC performance, unified PRC, LRU replacement

Each bar is a harmonic mean of speedups of the benchmarks for a particular generation–to–generation rescheduling and PRC size. Each set of bars is identified on the x-axis by a Gn-m label, which indicates that code originally scheduled for Generation–n was rescheduled for Generation-m. The first bar in each set of bars is the no–overhead performance, the second bar is the worst–case overhead (no PRC), and subsequent bars are performance with the indicated PRC size; PRC size is the maximum number of pages the cache holds.

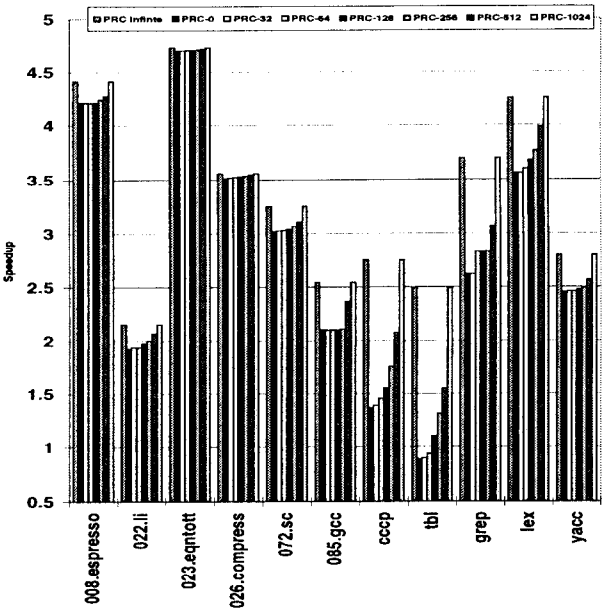

Figure 6. Generation–1 to Generation–3 rescheduling, unified PRC, LRU replacement

Each bar is the speedup for a Generation–1 to Generation–3 rescheduling for the specified PRC size. Each set of bars corresponds to an individual benchmark, as indicated by the labels on x-axis. The first bar in each set of bars is the no–overhead performance, the second bar is the worst–case overhead (no PRC), and subsequent bars are performance with the indicated PRC size; PRC size is the maximum number of pages the cache holds.

combination, for various PRC sizes. As indicated here, PRC-n means a PRC of size n pages. PRC-infinite indicates the performance when rescheduling of the unique page accesses was performed only at the initial invocations. This is essentially a PRC without an upper bound on its size. PRC-infinite speedups are the same as the no-overhead case speedups mentioned in Section 2.1. PRC-0 indicates the performance when no PRC is used, and all the pages uniquely accessed by the program are rescheduled at each invocation. This provides a measure of the worst-case overhead of rescheduling. A page can displace any other page in the PRC, based on the LRU replacement policy. This organization is called a *unified PRC*.

Performance of all of the rescheduling combinations benefits from the use of a PRC. The trend is that a larger PRC provides greater speedup. Note that for PRC-1024, perfect speedup (equal to the PRC-infinite) case is achieved. This happens because the total number of pages in the workload is less than the size of the PRC and all the programs completely reside in a PRC of that size without any requirement to reschedule once the PRC is populated.

All of the benchmarks do not benefit equally from the presence of a PRC. To illustrate this, Figure 6 presents speedups for individual benchmarks for Generation–1 to Generation–3 rescheduling. Some benchmarks – such as *008.espresso*, *023.eqntott*, *026.compress* – show only a small improvement with even a large PRC. Others, such as *cccp*, *tbl*, *grep* show substantial improvement with the increasing PRC size. Moderate improvement is shown by *022.li*, *072.sc*, *085.gcc*, *lex*, and *yacc*. The reason behind this behavior can be explained using the *overhead ratio* metric for a program. 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*

Table 1. Unique page counts of the benchmarks.

BENCHMARK	UNIQUE PAGE COUNT
008.espresso	137
022.li	47
023.eqntott	25
026.compress	8
072.sc	60
085.gcc	323
cccp	34
tbl	50
grep	4
lex	45
yacc	56

Table 2. Overhead Ratio (O): Generation–1 to Generation–3 rescheduling.

BENCHMARK	OVERHEAD RATIO (percentage)	OVERHEAD CATEGORY
008.espresso	4.35	low
022.li	10.52	moderate
023.eqntott	0.64	low
026.compress	1.25	low
072.sc	6.29	moderate
085.gcc	17.50	moderate
cccp	50.09	high
tbl	64.15	high
grep	29.10	high
lex	16.51	moderate
yacc	12.19	moderate

*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 are shown in Table 1. Table 2 shows the percentage values of the overhead ratio for Generation-1 to Generation-3 rescheduling. A high overhead ratio (20% and above in this case) indicates that the amount of time taken to reschedule is relatively high. Such programs benefit the most from the use of a PRC – *cccp, tbl,* and *grep* as shown here – and are termed *high-overhead* programs. For programs which showed the least performance improvement, the overhead ratio is relatively small (less than 5%). These programs are termed *low-overhead* programs (*008.espresso, 023.eqntott*, and *026.compress*). All the others (*022.li, 072.sc, 085.gcc, lex*, and *yacc* – they all have a value of O between 5% and 20%) are referred to as *moderate-overhead* programs.

In the unified PRC, the low- or moderate-overhead programs can evict the high-overhead programs completely if their unique page count is large. Thus, for example, even if the unique page count of a high-overhead program such as *cccp* is small (35 in this case), a moderate-overhead program such as *085.gcc* can replace all the pages allocated to *cccp* because it's unique page count is relatively high (323 in this case). If these two programs are run in an alternate fashion (which one would expect, because one is a C preprocessor, and the other is a C compiler), the number of pages for *cccp* that will have to be rescheduled could be sizable, thus increasing its overhead, especially for smaller PRC sizes. A better organization of the PRC which reduces this effect is presented next.

3.3 Split PRC

Because program behavior in the presence of a PRC varies directly with a program's overhead ratio, one approach is to *partition* the cache to hold pages for different classes of programs, based on the program overhead ratio. This should prevent *cache pollution*: a benchmark that benefits very little from a PRC displacing the pages of a program whose performance is substantially enhanced by a PRC. OS–gathered statistics can be dynamically used to compute the overhead ratios of the programs, and to determine which PRC partition a program should use. The OS needs to record the unique page counts, the program execution time, along with the average time take to reschedule a page, for this purpose.

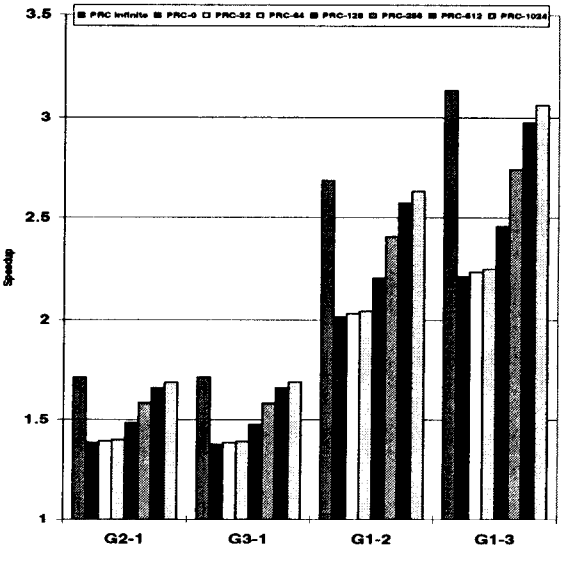

Figure 7. Speedups for PRC performance: 2-way split PRC, LRU replacement.

A PRC with two partitions can be labeled a *2-way split*

PRC: one partition to hold the low- and moderate-overhead programs, and the other for the exclusive use of the high-overhead programs. Figure 7 presents results for such a PRC across various generation–to–generation reschedulings and PRC sizes. (Here, both the partitions are of the same size, though this is not a requirement – the partition sizes can be varied depending on the program unique page counts.) Performance for practically all generation–to–generation rescheduling combinations improved using the 2-way split PRC over a unified PRC. In particular, for the Generation–1 to Generation–3 rescheduling using a PRC-512, the speedup was 91% of the speedup when using an infinite PRC (infinite PRC corresponds to the no–overhead experiment as described in Section 2.1). The Generation–1 to Generation–2 rescheduling also showed improvement with a split cache, particularly for PRC-256 and PRC-512. The general trend was that larger PRCs performed better, as was expected. The speedups of individual benchmarks with a 2-way split PRC, for Generation-1 to Generation-3 translation is shown in Figure 8. All of the high-overhead benchmarks fared well, compared to their performance under a unified PRC (Figure 6). Their low performance for the smaller PRC sizes is because they compete with each others in the same partition. It can also be observed that the cache pollution effect still persists, but is substantially reduced (for example, observe the improvement in performance for *cccp* and *tbl* as compared to the unified PRC, for all PRC sizes). Based on these experiments, an N-way split cache with M pages per partition would do better than a unified cache with $M * N$ entries. In an actual implementation, a PRC can be partitioned with as much granularity as the OS allows.

3.4 Unified PRC with overhead-based replacement

Another technique to reduce the cache pollution effect observed in the unified PRC is the use of an *overhead-based* PRC page replacement policy instead of LRU. This works as follows. With each page in the PRC is associated the overhead ratio for the program to which the page belongs. A page is not allowed to replace another page from the PRC unless its overhead ratio is higher than the overhead ratio of that page. This replacement policy ensures that more of the high-overhead programs stay PRC-resident, once placed in the PRC. The priority of use of the PRC is governed by the overhead ratio: the higher the overhead ratio, the higher the priority. Consequently, the low- and moderate- overhead programs incur higher rescheduling overhead, since a relatively less number of their pages get cached between invocations. Intuitively, this scheme is similar to the multiple dynamically resizable partitions within a PRC.

Figure 9 shows the performance of individual benchmarks for Generation–1 to Generation–3 translation for this

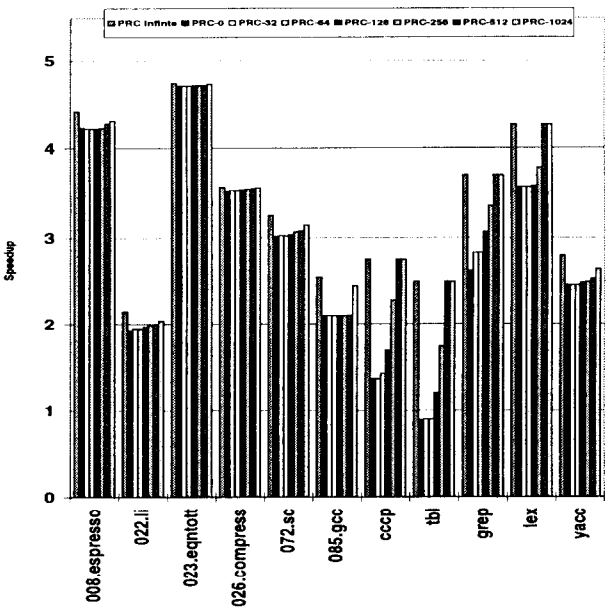

Figure 8. Generation–1 to Generation–3 rescheduling: 2-way split PRC, LRU replacement.

scheme. It can be observed that the top three high-overhead programs – *cccp, tbl, grep* – perform much better for this replacement policy than for LRU (Figure 6). They show a perfect speedup for PRC sizes 128 and above. (The low speedups for small PRC sizes are due to a large number of capacity misses.) On the other hand, the rest of the programs show a decrease, albeit small, in the performance for smaller PRC sizes compared to LRU. For the larger PRCs, their performance has, however, improved [4]. This is based on the fact that for smaller PRC sizes, they too encounter a large number of capacity misses, effectively displacing each other from the PRC and incurring a larger rescheduling overhead.

In the case of *008.espresso, 022.li, 023.eqntott, 026.compress, 072.sc*, and *lex*, however, there is little or no performance gain for any of the PRC sizes. This is explained by considering the case of *lex*. *Lex* has a relatively small unique page count of 45 (refer to Table 1), compared to the 323 of *085.gcc*. It's overhead ratio (16.51), however, is only slightly lower than that of *085.gcc* (17.50) (refer to Table 2). *Lex* gets displaced from the PRC after every invocation of *085.gcc*, and incurs the overhead of rescheduling all of it's pages at each subsequent invocation. It is speculated that

[4] As mentioned earlier, all the programs have a perfect speedup for PRC-1024 because it is large enough to hold the entire workload throughout the experiment.

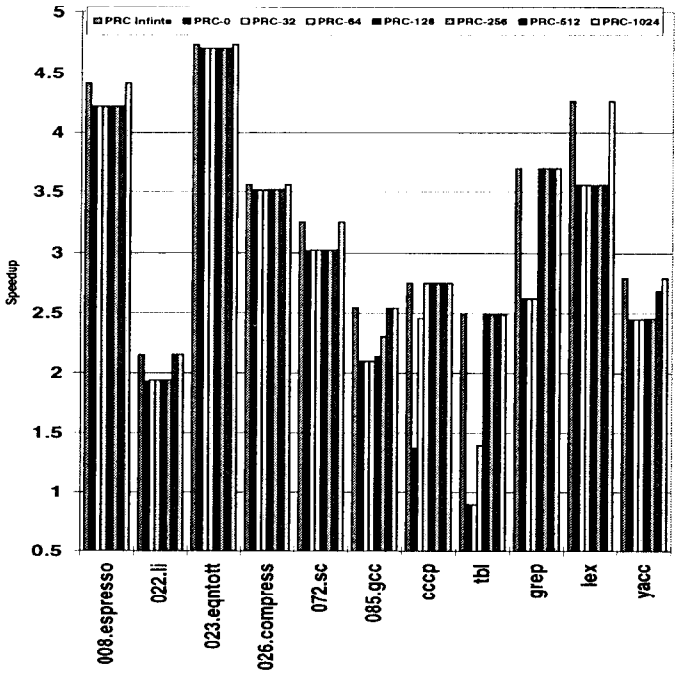

Figure 9. Generation–1 to Generation–3 rescheduling, Unified PRC, overhead-based replacement.

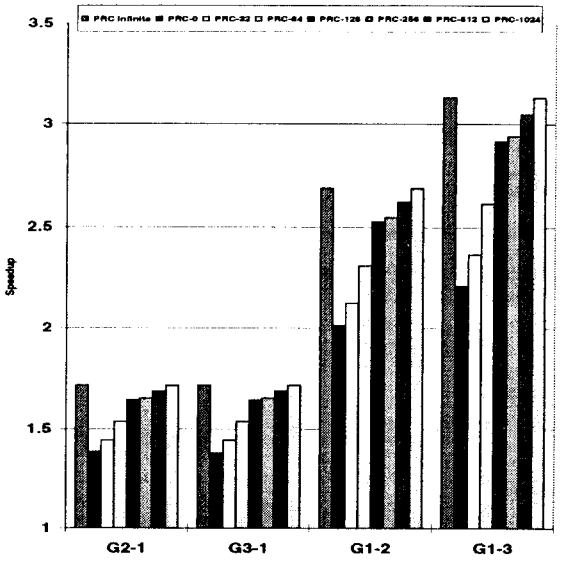

Figure 10. Speedups for PRC performance: Unified PRC, overhead-based replacement.

the impact of this phenomenon may be reduced if *085.gcc* was allowed to displace these programs only if they were invoked in the relatively remote past. A combination of the LRU and the overhead-based schemes may prove effective in such cases, and is a topic of future research.

Nonetheless, all of these programs (*008.espresso, 022.li, 023.eqntott, 026.compress, 072.sc*, and *lex*) being low- or moderate-overhead, the overall performance across all programs is better than the previous two schemes. This is apparent from Figure 10. When compared with the performance of the unified PRC with LRU replacement (Figure 5), and the 2-way split PRC with LRU replacement (Figure 7), the overhead-based replacement performs better across all the PRC sizes. The general performance trend observed for the overhead-based replacement policy confirms the intuition that the priority of use of the PRC is dictated by the overhead ratio.

4 Conclusion

The object–code compatibility problem for VLIWs can be approached by either software or hardware methods. A technique called dynamic rescheduling which uses operating system support to reschedule a program on a page–by–page basis during first–time page faults has been reviewed. The overhead of this technique is the time required to reschedule and the space required to save the rescheduled pages. The number of times a page has to be rescheduled can be reduced by saving the rescheduled version on disk, across program executions.

The persistent rescheduled-page cache was introduced as part of a scheme that saves rescheduled pages by using disk caching. The PRC reduces the number of times a page is rescheduled across multiple program executions. The architecture and management of the PRC was described. Specifically, the use and implementation of a system-wide, shared PRC was investigated. Unified caches were simulated to measure performance. The conclusion was drawn that PRCs in general are effective at reducing the overhead associated with dynamic rescheduling, but the effectiveness varies across programs and is dependent on the overhead ratio associated with a program. It was identified that the high-overhead programs are effectively displaced by the low overhead program from the PRC dependent on the invocation pattern in the workload. A partitioned cache organization was then introduced that classified programs based on their overhead ratios. A bi–partitioned scheme called a *Split* PRC was simulated and found to have improved performance over a unified PRC. In the third scheme, an overhead-based page replacement policy was implemented in a unified PRC, and it was found that the performance of the

high-overhead programs improved substantially compared with the two previous schemes, while the low-overhead programs fared only slightly worse than the previous schemes.

Future research dealing with PRCs will investigate the use of other cache organizations. For example, determination of the number of partitions ideal for a split PRC and an investigation of a per–program PRC structure are underway. Also, a study of a combination algorithm for page replacement, based on the LRU and the overhead-based schemes is being conducted.

Acknowledgments

Discussions with Mary Ann Hirsch, Kishore Menezes, and the rest of the TINKER group have proved useful to the development of ideas in this paper. Mary Ann Hirsch also helped collect data from the Data General machines. This research was supported by the National Science Foundation under grants MIP-9696010 and MIP-9625007, in addition to research funding from Intel Corporation, Hewlett-Packard, and the International Business Machines Corporation. We especially thank the University of Illinois IMPACT group for use of the IMPACT compiler. The comments from the anonymous referees are also appreciated.

References

[1] D. B. Papworth, "Tuning the Pentium Pro microarchitecture," *IEEE Micro*, vol. 16, pp. 8–15, Apr. 1996.

[2] D. Alpert and D. Avnon, "Architecture of the Pentium microprocessor," *IEEE Micro*, vol. 13, pp. 11–21, June 1993.

[3] S. Weiss and J. E. Smith, *POWER and PowerPC*. San Francisco, CA: Morgan Kaufmann, 1994.

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

[5] J. S. O'Donnell, "Superscalar vs. VLIW," *Computer Architecture News (ACM SIGARCH)*, pp. 26–28, Mar. 1995.

[6] 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.

[7] G. Silberman and K. Ebcioğlu, "An architectural framework for supporting heteregeneous instruction–set architectures," *Computer*, vol. 26, pp. 39–56, June 1993.

[8] 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.

[9] G. Silberman and K. Ebcioğlu, "An architectural framework for migration from CISC to higher performance platforms," in *Proc. 1992 International Conference on Supercomputing*, pp. 198–215, 1992.

[10] 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.

[11] C. May, "MIMIC: A Fast System/370 Simulator," in *Proc. ACM SIGPLAN '87 Symp. Interpreters and Interpretive Techniques*, pp. 1–13, June 1987.

[12] T. M. Conte and S. W. Sathaye, "Dynamic rescheduling: A technique for object code compatibility in VLIW architectures," in *Proc. 28th Ann. International Symposium on Microarchitecture*, (Ann Arbor, MI), Nov. 1995.

[13] Insignia Solutions, *SoftWindows for UNIX. Administrators's Guide, Version 2.0*. Mountain View, CA: Insignia Solutions, 1995.

[14] J. Turley, "Alpha runs x86 code with fx!32," *Microprocessor Report*, vol. 10, Mar. 1996.

[15] S. J. Leffler, M. K. McKusick, M. J. Karels, and J. S. Quarterman, *The Design and Implementation of the 4.3BSD Unix Operating System*. Addison-Wesley, 1989.

[16] M. J. Bach, *The design of the UNIX operating sustem*. Englewood Cliffs, NJ: Prentice–Hall, 1986.

[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] 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.

[19] Hewlett Packard, *How HP–UX works: Concepts for the System Administrator (R9.0)*. Palo Alto, CA: Hewlett Packard, 1991.

[20] Data General, *Programming in the DG/UX Kernel Environment (R4.11)*. Westboro, MA: Data General, 1995.

[21] G. Kane, *MIPS RISC architecture*. Englewood Cliffs, NJ: Prentice-Hall, 1992.

[22] Data General, *Analyzing DG/UX System Performance (R4.11)*. Westboro, MA: Data General, 1995.

[23] M. Nelson, B. Welch, and J. K. Ousterhout, "Caching in the Sprite network file system," *ACM Trans. Comput. Sys.*, vol. 6, pp. 134–154, Feb. 1988.

[24] J. H. Morris, M. Satyanarayanan, M. H. Conner, J. H. Howard, D. S. H. Rosenthal, and F. D. Smith, "Andrew: a distributed personal computing environment," *Comm. ACM*, vol. 29, no. 3, pp. 184–201, 1986.