Compiler-Directed Resource Management for Active Code Regions

Ravikrishnan Sree, Alex Settle, Ian Bratt, and Dan Connors
Department of Electrical and Computer Engineering
University of Colorado at Boulder

Abstract

Recent studies on program execution behavior reveal that a large amount of execution time is spent in small frequently executed regions of code. Whereas adaptive cache management systems focus on allocating cache resources based on execution access patterns, this paper presents a method of using compiler analysis to manage critical processor resources. With the addition of new architecture techniques to direct the utilization of instruction and data cache resources, the compiler can guard the most active regions of program execution from cache contention issues. The effect is that the overall performance of programs can be improved by either selectively granting each dynamic region a priority level for using cache and memory resources or providing active regions with dedicated cache structures.

1. Introduction

Memory access penalties have become a serious impediment to the delivery of increasingly higher performance microprocessors. The growing disparity between processor and memory performance will continue to make cache misses increasingly expensive. Additionally, data caches are not always used efficiently, resulting in large numbers of data cache misses. In numeric programs there are several known compiler techniques for optimizing data cache performance [16]. However, integer non-numeric programs often have irregular access patterns that are more difficult for the compiler to optimize.

Run-time spatial locality detection and optimization studies show that run-time adaptive cache management can significantly improve the overall performance of integer applications [10][11]. The improvements are due to increased cache hit rates and reduced cache miss handling latencies. However, there is still a large amount of potential improvement available for improving cache management and replacement policies.

As memory latencies increase, the importance of cache performance improvements at each level of the memory hierarchy will continue to grow. Rather than solely using adaptive hardware cache management techniques [10] to improve memory system efficiency, it follows that compilers should be used to assist with cache management. This paper focuses on using the compiler to identify frequently executed code regions, then allowing the memory system to prioritize access to data and instructions from these regions. The term region, in this paper, is analogous to a superblock [9], a sequence of instructions extending beyond a basic block boundary having only one entry point and multiple exit points.

Many performance studies have shown that very small frequently-executed code regions incur the majority of cache miss penalties of a program. Most schemes attempt to aid performance by directing resources based on a single instruction, even though the result may be to harm a neighboring instruction in the same frequently executed trace. For EPIC-style architectures that are based on principles of in-order execution, the overall performance of a region is more critically dependent on the collective efficiency of all cache operations within the region. Only by eliminating all of the cache misses during one invocation of a region of code on an EPIC-style architecture, can the true performance potential of the machine be achieved. At the same time, compiler-directed EPIC architecture features provide an appealing facility for improving interaction between the compiler and the architecture by allowing the compiler to express code region boundaries to the processor. Superscalar architectures may not have as severe of performance problems since they have an inherent ability to withstand memory latencies.

The objective of this research is to improve cache effectiveness by utilizing compiler-directed adaptive cache management techniques, in order to reduce the number of long memory latencies. Optimizations for both cost and performance are explored. Specifically, the aim is to increase data cache effectiveness for integer programs. The paper proposes a profile guided method for conveying data placement information to the run time system through compiler-guided management commands that govern an entire region of program code. Caching and guarding decisions of data and in-

structions are thus based on their access frequency and the presence of closely coupled accesses relationships. As such, the cache management decisions cause a reduction in cache misses for the targeted code regions. Overall, the compiler interacts with the architecture to guide caching decisions that could not have otherwise been made by hardware-only management schemes.

2. Motivation and Background

Several empirical studies indicate the presence of high-frequency code regions during program execution [13][14]. For many of these regions, the dynamic number of taken branches does not heavily outweigh the number of fall-through branches. The unpredictable nature of these branches suggests that these regions are not contained in inner loops. While some regions do contain such loops, many have acyclic control flow and represent a large portion of overall execution time. Although current trends in compiler technology help optimize the amount of time within program regions such as loops, even code that is aggressively optimized by modern, state-of-the-art compilers exhibits large portions of execution time in acyclic regions.

There are several sources of execution repetition within programs. These sources may be based on aspects of the input data, programming model, application domain, and the software distribution model. In addition, virtually all programs go through a series of stages during execution. A stage is characterized by changes in the execution properties for the code, the data, or both. The stages of a particular program depend upon the problem domain and the implementation. Similarly, programs such as compilers, interpreters, and graphics engines exhibit phase behavior, having different modes of operation for different inputs [12].

In an effort to better understand the contribution of regions to the total program execution time, Figure 1 examines the region execution percentage relative to overall program execution. A number of optimizations have already been applied to the code in these simulations. Specifically, Superblock Formation [9] (profile-guided code-straightening optimization) has eliminated many of the taken conditional branches and removed many of the cold blocks from the trace. Furthermore, function inlining has been performed to remove fetch and optimization barriers caused by calls and returns. The numbers vary considerably across the benchmarks, but in several cases more than 35% of program execution time is spent in regions. This suggests that a well developed heuristic for identifying regions can prepare the run-time system for optimizing the most common execution traces.

Aggressive compiler-directed management of programs may enable more performance in future computer systems in the presence of diverse workloads. To direct optimization

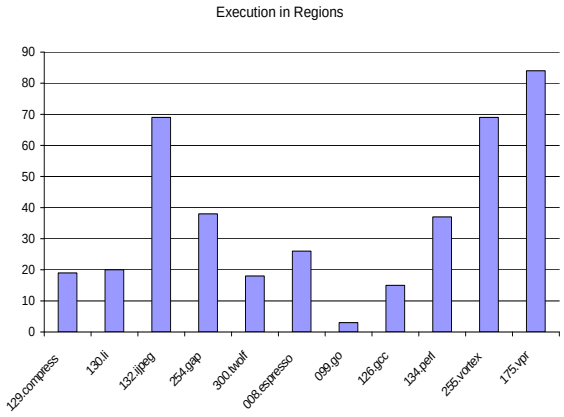

Figure 1. Active region execution percentage.

opportunities, it is critical to identify and optimize the execution hot spots for the current workload. After program regions representing substantial execution have been identified, methods for transforming the selected code region to execute multiple instructions per cycle can be performed.

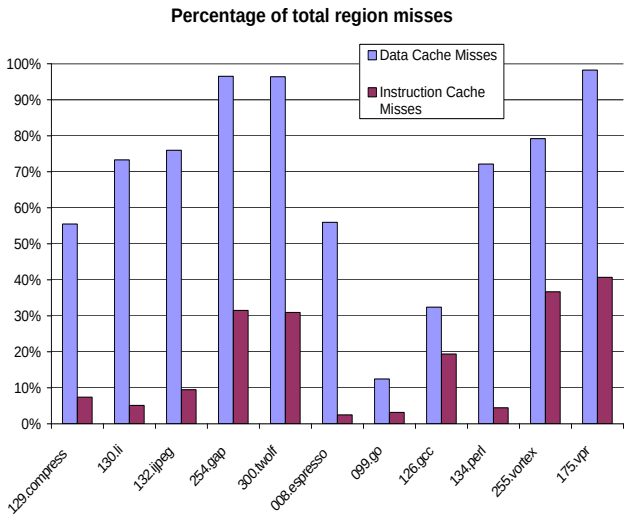

Figure 2. Cache misses from regions.

The graph of Figure 2 illustrates the impact on memory latency that can be achieved by reducing the number of cache misses for a small set of instructions. If the cache system can be made aware of frequently executing instructions,

then it can be enabled to keep these instructions and the associated data in the cache as long as possible. Similarly, delinquent loads can be marked and treated with this same priority scheme. Figure 3 examines a similar result concerning region activity in the memory system. The data shows the percentage of data cache lines associated with high frequency regions that get replaced. Nearly 60% of cache lines replaced from the first level data cache have been accessed by one or more regions. This supports the argument that memory system performance can be improved by enabling the architecture to provide fine-grain control over cache line replacement policies.

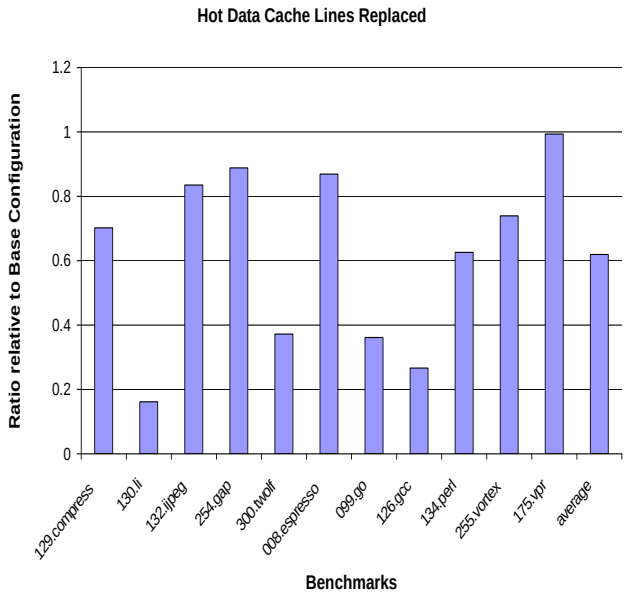

Figure 3. Hot cache lines replaced.

2.1. Related Work

Both static and dynamic methods exist to identify the most frequently executed regions of a program. Methods which make use of statistical sampling depend on static analysis to identify active regions of code. Hardware methods use some form of auxiliary structures to keep track of dynamic program behavior. Several different works have explored utilizing execution frequency, gathered either statically or dynamically, to implement novel optimizations. The technique developed in this paper relies upon statically determined execution frequencies. The execution information is then utilized by the compiler and conveyed to the architecture in order to improve the efficiency and performance of the memory system.

2.1.1 Hardware

Conte et al [7] uses dedicated hardware for profiling. Their work is dependent upon branch execution frequencies to capture frequently occurring regions of code. The major goal of their work is in improving the accuracy of the profile obtained. Satish Narayanasamy et al [17] uses multiple hash tables to accurately capture important events occurring in the lifetime of a program. Hardware counter assisted profiling systems such as DCPI [2] take advantage of the hardware performance monitoring counters to collect profiles.

Merten et al [14] collects dynamic program behavior information with the help of a set of hardware tables placed in the retirement stage of the program. The primary purpose of the Merten et al paper [15] was to detect frequently executing regions of code for the purpose of dynamically changing the code layout.

The Non-Temporal Streaming (NTS) Cache proposed by Rivers [18] dynamically identifies loads with a non-temporal nature. Once identified, values accessed by non-temporal loads bypass the L1 cache and are placed in a fully associative buffer accessed in parallel with the L1 cache. By placing non-temporal values in the buffer rather than the L1 cache, frequently accessed data has a lower probability of being removed from the cache. The NTS Cache assumes a direct mapped L1 cache.

2.1.2 Compiler-Directed

Methods of compiler-directed cache management [4] have been explored in which the compiler interacts with the cache hardware to manage the cache placement and replacement policy. The generation of appropriate cache hints can be based on both locality of the instructions and profile information [1].

Wu et al [19] relies on the compiler to direct intelligent cache bypassing for high performance EPIC architectures. Wu's work uses compile time schedule information to determine which loads can bypass the lowest-level cache without sacrificing performance. This information is then conveyed to the architecture through the load instruction.

Generally, cache management approaches similar to ours, [19] [4] [18], cannot substantially improve caching decisions because decisions need to be made based upon their ability to help an entire region of code. Simply caching the data accessed by a load operation may not help a section of code if it forces a transfer of cache misses to another load in the near vicinity of the original load. Basically, the compiler can help in the decision making process of cache management by deciding to keep a cache block if such a data item is important to a frequently occurring region of code. Our goal is to direct memory system management around entire regions of code, not just single values.

3. Approach

3.1. Active Regions Detection

The motivation behind detecting active regions is to enable the processor to give priority treatment to selected instructions. This requires that the region characteristics should be congenial to handling by the processor. The region sizes should be small enough so that giving preferential treatment to these regions would not hurt the program performance but should be large enough to influence the performance when resources are dedicated to them.

3.1.1 Resource Guarding

In order to observe the effects of providing preferred cache access to instructions or data belonging to regions, two techniques were studied. The first, Resource Guarding, is a way to modify the cache replacement policy of an associative cache based upon region information stored in each cache line. In this case, the Least Recently Used (LRU) replacement algorithm is modified so that cache lines that are associated with a region are not chosen for replacement. The exception is when all cache lines in a set are related to a region, in this case the normal LRU policy is restored. The goal behind this approach is to keep data or instructions that account for a dominant percentage of overall execution time in the highest possible cache levels.

3.1.2 Resource Caches

This approach provides small fully associative caches, Figure 4, that can store cache lines related to particular regions. These caches are accessed in parallel to the level one icache and dcache. A new line is brought in after each level one replacement. In this way, the region caches extend the amount of time select cache lines reside at the first level. The resource cache that a cache line is stored in is mapped to the region id associated with the cache line. Since the size of each region cache is quite small (2KB), it is feasible to add several of these to a cache system with little hardware overhead.

Figure 4 shows the simulated cache hierarchy used for the experiments. The dcache used was a 16KB 2 way set associative cache with one port. The icache was the same configuration as the dcache. Each of these caches were accessed in parallel to the corresponding victim and resource caches. The victim caches were 1KB fully associative. Each resource cache was 2KB fully associative. The number of resource caches was changed from for different experiments. The values tested were: 4, 8, and 16. The second level cache was a unified 8 way set associative 256KB cache, while the third level was a 1MB fully associative.

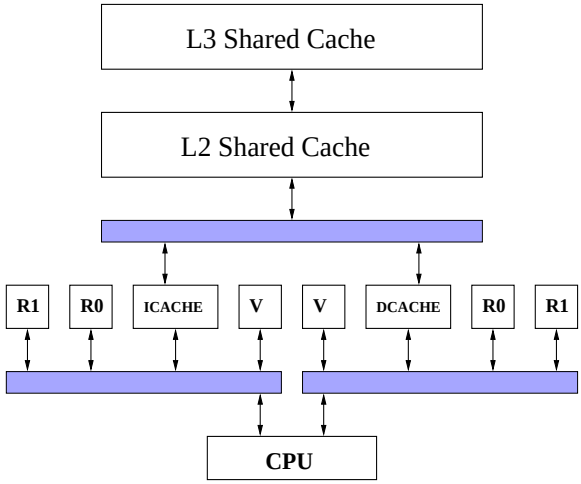

Figure 4. Experimental cache hierarchy.

3.1.3 Case Study

The region formation and its behavior in 099.go is an interesting case to examen. The experiments of Figure 1 show that the execution of the program inside the regions for 099.go is only about 3%. While the data of Figure 9 shows that there is a 69% reduction in the data cache misses and a corresponding performance improvement of 19% (Figure 8). Although these results are counter-intuitive, close examination of the source code for 099.go reveals some interesting facts.

The two most active regions discovered by the compiler occur inside the function $Lresurrect$. Figure 5 shows that this function occurs inside a loop in the function $Ldndate$. The arrays used across the functions are the same and it turns out that these arrays are global structures used by the majority of the program. Normally these arrays get thrashed from the data caches, as the number of such structures used across the program is large, making it impossible for them to be preserved inside the data cache. Being part of the regions enables them to be protected in the resource caches. Thus, every access to any of these arrays, region or non-region, results in a hit at the first level. The process of guarding globally accessed arrays within a region has the added indirect effect of guarding those arrays for non-regions. This results in the drastic reduction of total dcache misses. This explains the fact that although the program executes only 3% of the time in regions, guarding the region data inside the resource caches results in a performance improvement of 19%. This example also sheds light on the indirect and unexpected benefits to other parts of the program from providing the active regions with exclusive resources.

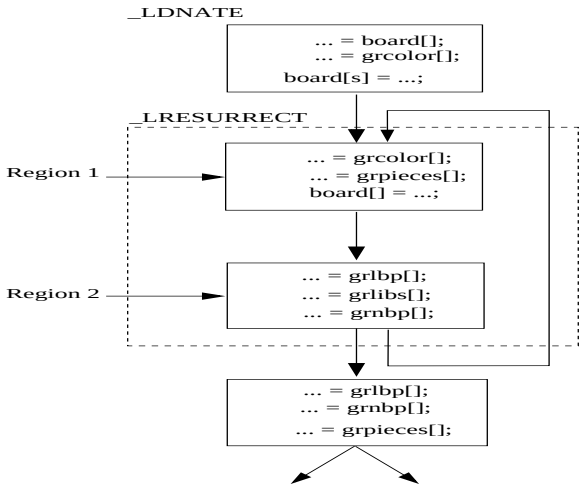

Figure 5. Example of 099.go regions.

3.2. Profile-Guided Active Region Formation

By profiling an application on a set of sample inputs, representative run-time information can be conveyed to the compiler. This enables an optimizing compiler to increase application performance by transforming code to achieve better execution efficiency for sections of the program with high execution frequency. Optimizations based upon run-time value invariance [3][5] offer great potential for exploiting run-time behavior. Other dynamic techniques have focused on discovering invariant relationships between variables from execution traces [8]. Invariant value profiling has also been used to improve the effectiveness of compiler-directed computation reuse [6].

Due to the limited amount of parallelism inherent to basic blocks within non-numeric programs, compilers must optimize and schedule instructions across basic block code boundaries to achieve higher performance. An effective code structure for instruction-level parallelism compilation is the superblock [9]. The formation and optimization of superblocks increases the ILP to the scheduler along important execution paths by removing constraints due to infrequently executed paths. Superblocks have a single entrance and represent paths with high potential of reuse behavior. Since branches are controlled by program data, the nature of the flow of control through a frequently executed path directly relates to the cache locality being exercised by the code's decision and data access components.

Our initial compiler-based approach selects superblocks that represent a large portion of overall execution time. Next, rather than selecting a single load or single instruction to have better management in the data cache or instruction cache, our technique guides cache management for an entire active region of code. An active region is defined as an arbitrary, connected subgraph of the program control flow graph that has been determined to have execution characteristics that warrant cache guarding of all data and instruction contents accessed by the region.

To establish the effectiveness of the proposed approach to compiler-directed cache management and replacement, execution profiles were collected for benchmarks from *SPECINT98* and *SPECINT2000*, applications using both the training and reference input sets. Region formation steps were applied to programs annotated with execution frequency information to detect superblocks. Two region identification methods, *execution frequency* and *cache accuracy* were proposed and investigated. Both techniques are based on execution profiling, a generally accepted technique used in modern optimizing compilers. Figure 6 illustrates the percentage of program execution for the training input attributed to candidate superblocks. Candidate regions are those superblocks that exhibit at least a minimum selected frequency of execution. Specifically, Figure 6 shows regions selected based upon execution frequencies of 100000 (region1), 50000 (region2), 25000 (region3), and 15000 (region4). These results indicate that a significant percentage of program execution is attributed to candidate regions and can be exposed without the aid of significant profiling.

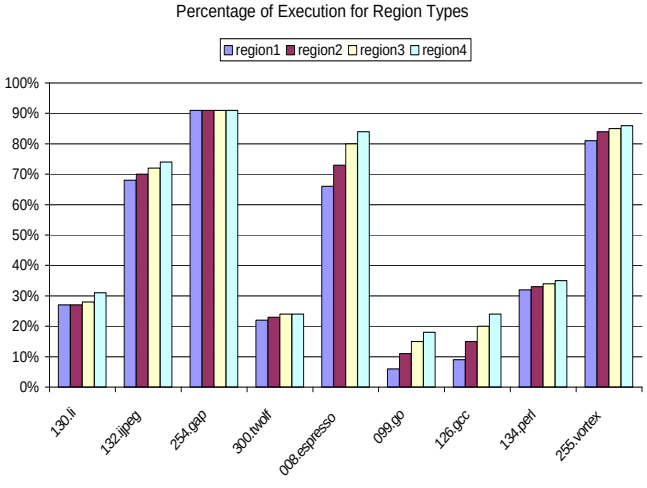

Figure 6. Percentage of execution spent in regions.

The number of candidate regions increases as the minimum execution threshold for region selection is decreased. The number of regions selected directly determines the resources required of the cache management system. In section 4.2 of this paper, we evaluate the performance increase

while varying the number of superblocks marked as regions. Similarly, we summarize the results for selecting regions based upon a high incident of the overall program cache misses.

4. Experimental Evaluation

4.1. Methodology

The active cache management experiments were implemented on a modified version of the IMPACT EPIC simulator. The victim and resource caches were included in the cache hierarchy in parallel with the data and instruction caches. The simulator's LRU replacement policy was also modified in order to activate cache line guarding. Region collection was implemented as a profile guided optimization phase of the IMPACT compiler. Candidate region instructions were marked with attributes, enabling the simulator to identify regions.

Each cache line was modeled to contain a bit indicating the presence of region data. After a cache hit, this bit was set or cleared depending on the requesting instruction's region status. In order to perform guarding of region cache lines, this bit is checked during LRU replacement and if set, the corresponding cache block is not considered a replacement candidate. If however, all of the blocks in a set have the region bit set, then the normal LRU replacement policy is followed.

A base simulation configuration was established from which to compare the experiments on active guarding. This base processor modeled a three level cache hierarchy with an EPIC-style execution engine, as in Figure 4. The base EPIC processor modeled can issue in-order six operations up to the limit of the available functional units: four integer ALU's, two memory ports, two floating point ALU's, and one branch unit. An LRU replacement algorithm was used for all set associative caches in the system. In order to measure the upper bounds of possible performance gains, the cache system was first modeled as perfect for two different circumstances. The first experiment treated all caches as perfect for any accesses. The second only exhibited perfect behavior when regions were requested, as shown in Figure 7. This case provided a more realistic view of the maximum allowable speedup due to reducing cache misses associated with regions. An important result shown by the perfect cache access for regions is that many of the benchmarks have the potential to achieve performance improvements greater than 10%. Thus, enabling the run time system to facilitate cache accesses associated with these regions has the potential to cause significant speedup.

After establishing the boundary conditions for the experimental model, a number of tests were performed in order to deduce the best method for optimizing cache requests related to regions. First, victim caches were introduced to the base system, then region caches with and with out the victim caches. Next, the cache line guarding technique was employed as a basis for comparison against the region caches. Lastly, the number of resource caches used was varied in order to identify the cost to performance ratio associated with region caches.

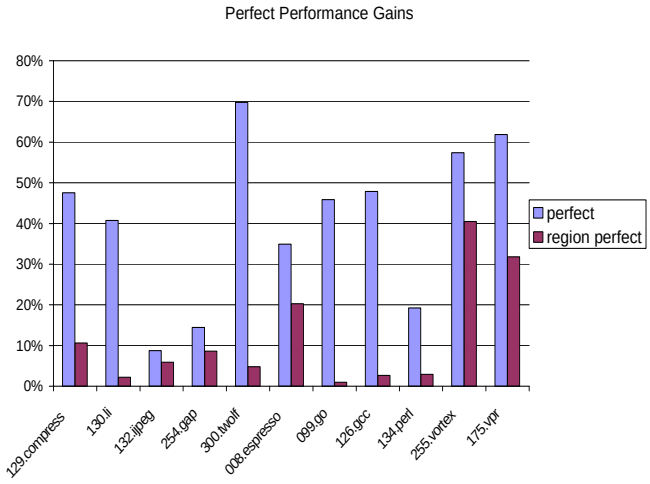

Figure 7. Perfect cache accesses for regions.

4.2. Results and Analysis

Figure 8 shows the speedup from each of the active cache management techniques relative to the base memory organization. For these tests it appears that region caches make the most significant contribution to performance. Upon comparison between the case of using victim caches only and using both resource and victim caches, one can see that the contributions from the victim caches are minimal. Also, the performance graph closely maps to the data cache miss reduction graph of Figure 9. The performance improvements can thus be attributed to the corresponding reduction in cache misses that result from reducing the number of replacements to higher cache levels.

Figure 9 displays the percentage of data cache reductions realized when using a set of cache management techniques. The variations in data cache misses map closely to the performance improvements for each of the benchmarks. The average miss reduction across the benchmarks is roughly 20%, this corresponds to a 10% execution speedup. Thus it appears that for most benchmarks, improvements in data cache performance for regions significantly affect overall

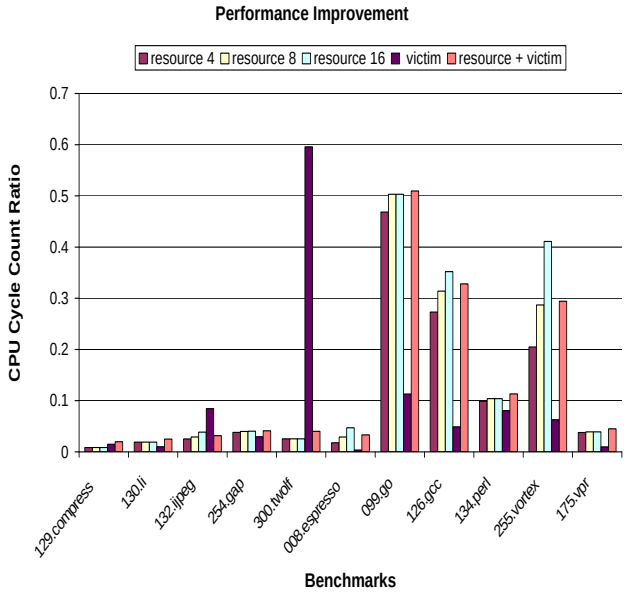

Figure 8. Performance gains of compiler-directed cache management.

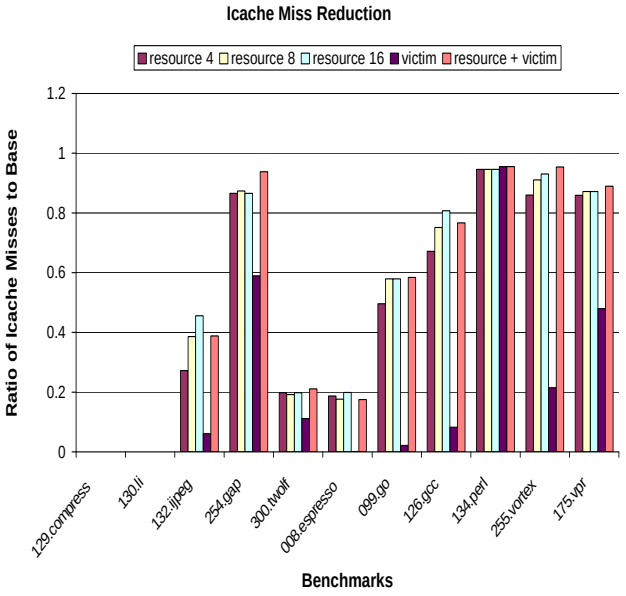

Figure 10. Instruction cache miss reduction for cache management approach.

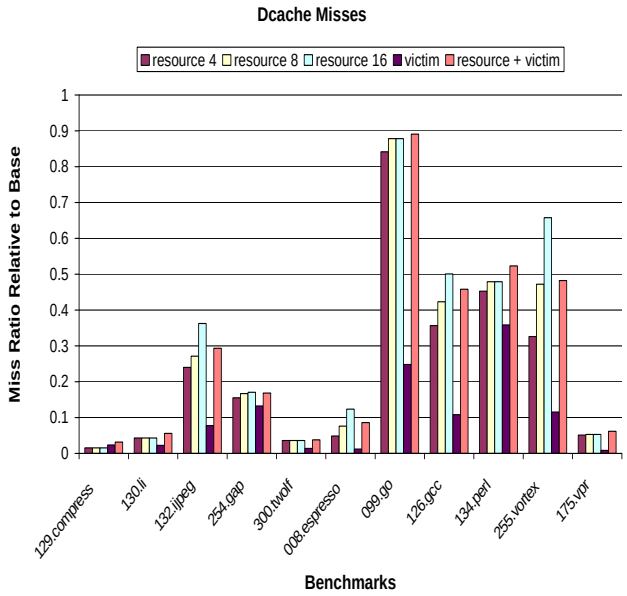

Figure 9. Data cache miss reduction for cache management approach.

performance. The benchmark 099.go has very large data cache miss reductions, the details of which are described in section 3.1.3. Overall the configuration with 16 resource caches appears to outperform the other cache organizations

Figure 10 shows the percentage of instruction cache misses that are reduced after employing the set of cache management techniques. The vertical axis corresponds to the percentage of miss reductions relative to the base model of a simple 2 way 16KB set associative instruction cache using a normal LRU replacement policy. Each of the vertical bars represents a different configuration of resource caches, victim caches, and active guarding. It is interesting to note that the benchmarks 134.perl, 254.gap, and 175.vpr experience very large reductions in icache misses, roughly 90%, but comparatively small increases in both data cache and system performance. This indicates that the overall system performance is heavily dependent upon data cache performance.

Although not reported in the above graphs, the results based on cache line guarding were shown to produce an average data cache miss reduction of roughly 5% across the benchmarks. The effect of cache line guarding proved to be relatively insignificant when contrasted against the effects of using region caches.

The graph of Figure 11 measures the difference in performance based upon region creation criteria. For each of the test cases, both the region caches and the victim caches were enabled. The performance numbers are relative to the base cache system of section 3.1.2. Each bar in the graph corresponds to a method of region generation, all of which are based upon profiled execution frequency. The results suggest that rather than searching for an optimal execution frequency as a criteria for region creation, different heuristics should be explored. In addition to the regions selected by

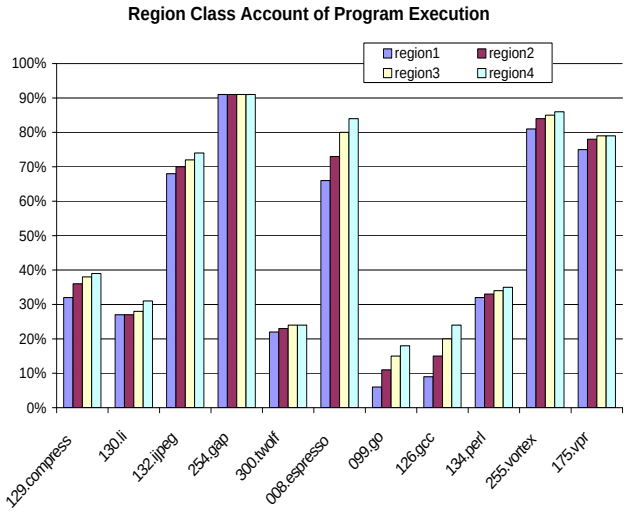

Figure 11. Performance comparison of region types against base architecture.

execution frequency, we also simulated regions which were selected based on the occurrence of a high number of cache misses. Generally the results of these regions performed worse than the execution frequency-selected regions. It appears that cache management performs better for highly executing regions because several different data items are inherently cached by their selection. By caching several different data items, other regions that operate on the same data items can also be improved.

5. Summary

Whereas traditional hardware and compiler techniques provide hints for guiding cache management based on individual instruction behavior, this paper illustrates the initial rationale for further increasing the interaction between compilers and architectures. We present an approach to improving cache effectiveness, taking advantage of the growing chip area, utilizing compiler-directed adaptive cache management techniques. By selecting regions with high execution potential and collectively guarding all of the instruction and data elements used by such regions, the cache effectiveness for integer programs can be significantly improved. This paper examines the initial rational of the compiler-directed region management and found an average of 10-12% performance improvement with some integer applications approving by as much as 25%.

This scheme is compatible with existing schemes of locating hotspots in hardware at run time. In future work, a run-time selection mechanism for deciding which region caching decision have the highest priority will be considered. Likewise, similar techniques can be used to manage resources wisely in complex execution environment, such as multithreading and multiprocessors. Hardware profiling technique may provide additional opportunities for hardware, OS, and compiler working together to achieve the optimal solution on run-time performance tradeoffs.

References

[1] S. G. Abraham, R. A. Sugumar, D. Windheiser, B. R. Rau, and R. Gupta. Predictability of load/store instruction latencies. In *Proceedings of the 26th International Symp. on Microarchitecture*, pages 139–152, December 1993.

[2] J. Anderson, L. M. Berc, J. Dean, S. Ghemawat, M. R. Henzinger, S. Leung, R. L. Sites, M. T. Vandevoorde, C. A. Waldspurger, and W. E. Weihl. Continuous profiling: Where have all the cycles gone? In *Proc. of the 16th ACM Symposium of Operating Systems Principles*, pages 1–14, October 1997.

[3] J. Auslander, M. Philipose, C. Chambers, S. Eggers, and B. Bershad. Fast, effective dynamic compilation. In *Proceedings of the ACM SIGPLAN 1996 Conference on Programming Language Design and Implementation*, volume 31, pages 149–159, June 1996.

[4] K. Beyls and E. H. D'Hollander. Compile-time cache hint generation for epic architectures. In *Proceedings of the Second Workshop on Explicitly Parallel Instruction Computing Architectures and Compiler Techniques*, November 2002.

[5] B. Calder, P. Feller, and A. Eustace. Value profiling. In *Proceedings of the 30th Annual International Symposium on Microarchitecture*, pages 259–269, December 1997.

[6] D. A. Connors and W. W. Hwu. Compiler-directed dynamic computation reuse: Rationale and initial results. In *Proceedings of 32nd Intl. Symp. on Microarchitecture*, November 1999.

[7] T. M. Conte, M. K. N., and M. A. Hirsch. Accurate and practical profile-driven compilation using the profile buffer. In *Proceedings of the 29th Annual International Symposium on Microarchitecture*, pages 36–45, Paris, France, 2–4 December 1996. ACM Press.

[8] M. D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin. Dynamically discovering likely program invariants to support program evolution. In *Proceedings of the 19th International Conference on Software Engineering*, pages 213–224, May 1999.

[9] 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 technique for VLIW and superscalar compilation. *The Journal of Supercomputing*, 7(1):229–248, January 1993.

[10] T. L. Johnson and W. W. Hwu. Run-time adaptive cache hierarchy management via reference analysis. In *Proceedings of the 24th International Symposium on Computer Architecture*, June 1997.

[11] T. L. Johnson, M. C. Merten, and W. mei W. Hwu. Run-time spatial locality detection and optimization. In *International Symposium on Microarchitecture*, pages 57–64, 1997.

[12] D. C. Lee, P. J. Crowley, J. L. Baer, T. E. Anderson, and B. N. Bershad. Execution characteristics of desktop applications on windows nt. In *Proceedings of the 25th International Symposium on Computer Architecture*, pages 27–38, June 1998.

[13] M. H. Lipasti and J. P. Shen. Value locality and load value prediction. In *Proceedings of 7th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 138–147, September 1996.

[14] M. C. Merten, A. R. Trick, C. N. George, J. C. Gyllenhaal, and W. W. Hwu. A hardware-driven profiling scheme for identifying program hot spots to support runtime optimization. In *Proceedings of the 1999 International Symposium on Computer Architecture*, May 1999.

[15] M. C. Merten, A. R. Trick, E. M. Nystrom, R. D. Barnes, J. C. Gyllenhaal, and W. W. Hwu. A hardware mechanism for dynamic extraction and relayout of program hot spots. In *Proc. 2000 Int'l Symp. on Computer Architecture*, pages 136–147, June 2000.

[16] T. C. Mowry, M. S. Lam, and A. Gupta. Design and evaluation of a compiler algorithm for prefetching. In *Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 62–73, Oct 1992.

[17] S. Narayanasamy, T. Sherwood, S. Sair, B. Calder, and G. Varghese. Catching accurate profiles in hardware. In *9th International Symposium on High Performance Computer Architecture*, February 2003.

[18] J. Rivers and E. Davidson. Reducing conflicts in direct-mapped caches with a temporality-based design. In *Proceedings of the 1996 International Conference on Parallel Processing*, page 154, August 1996.

[19] Y. Wu, R. Rakvic, L.-L. Chen, C.-C. Miao, G. Chrysos, and J. Fang. Compiler managed micro-cache bypassing for high performance epic processors. In *Proceedings of the 35th International Symp. on Microarchitecture*, November 2002.