Annotated Memory References: A Mechanism for Informed Cache Management

Alvin R. Lebeck, David R. Raymond,
Chia-Lin Yang, and Mithuna S. Thottethodi

Department of Computer Science
Duke University
Durham, NC 27708-0129 USA
alvy@cs.duke.edu

Abstract. As the importance of cache performance increases, allowing software to assist in cache management decisions becomes an attractive alternative. This paper focuses primarily on a mechanism for software to convey information to the memory hierarchy. We introduce a single instruction—called TAG—that can annotate subsequent memory references with a number of bits, thus avoiding major modifications to the instruction set. Simulation results show that annotating all memory reference instructions in the SPEC95 benchmarks increases execution time between 0% and 2% for both statically and dynamically scheduleded processors. We show that exposing cache management mechanisms to software can decrease the execution time of three media benchmarks (*epic*, *pegwit*, *ijpeg*) between 11% and 17% speedups on a 4-issue dynamically scheduled processor.

1 Introduction

One of the key challenges facing computer architects is the increasing discrepancy between processor cycle times and main memory access time. Conventional cache designs rely solely on hardware to manage the memory hierarchy, and the policies for cache management tend to be relatively naive. These limitations could be overcome by using information provided by software to help manage the memory hierarchy [1, 6, 3].

This paper presents a mechanism—*called annotated memory references*—for software to convey information to the memory hierarchy, and shows how it can be used to exploit information on replacement strategies and block size selection for improved performance. Our implementation adds a single instruction (TAG) that can annotate each of the following 6 memory references with a 4-bit annotation. The TAG instruction initializes an annotation register, and the appropriate annotation is extracted for each subsequent memory reference. We analyze our new instruction in a statically scheduled processor using a simplified model of the Digital Alpha 21164 [5] and using the SimpleScalar toolset to model a dynamically scheduled processor [2].

Our simulation results show that using the TAG instruction to annotate **all** memory references increases the number of instructions executed by 5.5% to 16.2% for the programs we studied. The instruction count overhead does not necessarily translate directly into performance loss. Instead, the superscalar processor reduces execution time overheads to at most 2%. Using the TAG instruction to manage cache replacement and block size selection we can reduce execution times by 11% to 17% for three media benchmarks (`epic`, `pegwit`, `ijpeg`) executing on typical embedded system memory hierarchy and a 4-issue dynamically scheduled processor.

2 A Static Instruction Annotation Mechanism

Our goal in designing a mechanism is to minimize the impact on execution time and avoid gross modifications to the instruction set. We also want to provide a mechanism that can support a variety of annotations, and is not specific to any particular use of the annotations. In essence, we are defining an interface between the executing program and the memory hierarchy. For example, one application (or one phase of an application) may want greater control over Level-1 and Level-2 cache replacement decisions, while another application (or phase) may want several forms of sophisticated prefetching. Further discussion of these issues is beyond the scope of this paper. Instead, we focus on the interface for passing information from the executing program to the memory hierarchy.

The mechanism we propose does not require any modification to existing instructions and requires the addition of only a single instruction—*called TAG*. The TAG instruction initializes a new hardware register—*called the annotation register*—from which bits are extracted to annotate subsequently executed memory instructions.

Conceptually, the annotation register is a simple shift register. Every memory reference extracts the right most bits from the annotation register, which is then right shifted with zeros shifted into the left most bits. We assume a zero annotation indicates default cache operation, hence untagged memory references default to normal memory system operation.

We evaluate the overhead of TAG instructions using ATOM [5] to model a statically scheduled processor using the Digital Alpha 21164 issue rules and using SimpleScalar [2] to evaluate a 4-issue dynamically scheduled processor with 64 RUU entries. In both systems we assume a perfect memory system and perfect branch prediction. Our results show that TAG instructions increase execution time by at most 2% even when **all** memory references are annotated. For most applications, we expect to annotate a much smaller number of memory references, thus further reducing overhead.

3 Utilizing Annotated Memory References

The goal of this section is to provide an example of how software can use annotated memory references to help manage the memory hierarchy. We focus on multimedia applications typical of those that would execute on future embedded systems:

- **epic**: a lossy image compression program which is designed for extremely fast decoding on non-floating-point hardware at the expense of slower encoding,
- **ijpeg**: a lossy image compression program, and
- **pegwit**: a public-key encryption and authentication program

We use cache profiling [4] followed by source code inspection to obtain programmer specified annotations. Cache profiling provides key insights into an applications memory system behavior. For `epic`, we found that controlling cache replacement may improve performance. Similarly, cache profiling and source code inspection revealed that control of cache block size may improve the performance of `ijpeg` and `pegwit`.

To control cache replacment, we use annotations that allow an application to specify that the referenced cache block should be *retained* in a 4-way set-associative cache. This prevents the block from being replaced until the application uses a *release* annotation to unlock the cache block. Cache block size is controlled by using an annotation that specifies *wordmode*, causing only the accessed word to be fetched from main memory. For the blocksize studies, we use a direct-mapped cache, and by reusing the data portion of a 32-byte block we can dynamically increase the associativity so the given cache set can hold three 8-byte blocks.

3.1 Results

We use the SimpleScalar simulator in our experiments to evaluate the benefits of annotated memory references. We simulate a 4-way issue out-of-order processor with 64 RUU entries and 32 LSQ entries, perfect branch prediction, and a cache miss penalty of 28 cycles.

For `epic`, we compare the performance of systems with an 8 KB, 32-byte block, 4-way set-associative cache with and without *retain/release* annotations. Using the *retain/release* annotations provides a 17% reduction in execution time. The miss ratio drops from 26.5% to 20.8%.

For `pegwit` and `ijpeg`, we compare performances of systems with an 8 KB, 32 byte block, direct mapped cache with and without *wordmode* annotations. Using the *wordmode* annotations yields a 11.6% and a 12.4% reduction in execution times for `pegwit` and `ijpeg`, respectively. The miss ratio also decreases from 16.8% to 12.8% for `pegwit` and from 5.7% to 5% for `ijpeg`.

The results in this section show that annotated memory references can be used to improve memory hierarchy performance. We provided two examples of how software could assist in memory hierarchy management by conveying information about replacement strategies and block size requirements. However, the flexibility of annotated memory references permits a variety of annotations. We are currently investigating other sources of information and techniques for obtaining the appropriate annotations (including compile-time) and ways to exploit that information in the memory hierarchy.

4 Conclusion

Cache memories are an important component of the overall solution to the ever increasing discrepancy between processor clock cycle time and main memory access time. Unfortunately, conventional cache designs do not utilize any information that software may provide about expected reference patterns. This paper introduces a mechanism that enables software to provide hints about future memory references and assist in cache management. We propose adding a single instruction—called TAG—that can annotate six memory references with 4-bit annotations to be interpreted by the memory system. Our analysis showed that when annotating all memory references, execution time overhead ranges from zero to 2% on either a 4-issue dynamically scheduled or 4-issue statically scheduled processor. Using annotated memory references to exploit information on better cache replacement strategies and better block sizes, we can reduce execution time by 11% to 17%.

As the demands on cache performance increase, new techniques that complement, rather than replace, hardware cache management become an attractive alternative. The relatively low overhead of TAG instructions combined with the ability to support many different types of annotations can enable new approaches to cache management. In particular, new cache organizations and management techniques that utilize software assistance can easily be supported without requiring significant modifications to an instruction set.

References

1. S. G. Abraham, R. A. Sugumar, D. Windheiser, B. R. Rau, and R. Gupta. Predictability of load/store instruction latencies. *Proceedings of the 26th Annual International Symposium on Microarchitecture*, pages 139–152, December 1993.
2. D. C. Burger, T. M. Austin, and S. Bennett. Evaluating future microprocessors-the simplescalar tool set. Technical Report 1308, University of Wisconsin–Madison Computer Sciences Department, July 1996.
3. A. Gonzalez, C. Aliagas, and M. Valero. A data cache with multiple caching strategies tuned to different types of locality. In *ACM 1995 International Conference on Supercomputing*, pages 338 – 347, 1995.
4. A. R. Lebeck and D. A. Wood. Cache profiling and the spec benchmarks: A case study. *IEEE COMPUTER*, 27(10):15–26, October 1994.
5. A. Srivastava and A. Eustace. Atom a system for building customized program analysis tools. In *Proceedings of the SIGPLAN '94 Conference on Programming Language Design and Implementation*, pages 196–205, June 1994.
6. G. Tyson, M. Farrens, J. Matthews, and A. R. Pleszkun. A modified approach to data cache management. In *Proceedings of the 28th Annual International Symposium on Microarchitecture*, Dec. 1995.