

# Annotated Memory References: A Mechanism for Informed Cache Management

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

Department of Computer Science  
Duke University  
Durham, NC 27708 USA  
{alvy,draymond,yangc,mithuna}@cs.duke.edu

## Abstract

*Processor cycle time continues to decrease faster than main memory access times, placing higher demands on cache memory hierarchy performance. To meet these demands, conventional cache management techniques that rely solely on naive hardware must be augmented with more sophisticated techniques. This paper investigates Informed Caching Environments (ICE) where software can assist in cache management. By exposing some cache management mechanisms and providing an efficient interface for their use, software can complement existing hardware techniques by providing hints on how to manage the cache.*

*In this paper, we focus 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 scheduled processors. We show that exposing cache management mechanisms to software can increase the performance of three media benchmarks (*epic*, *pegwit*, *jpeg*) by managing replacement and block size more effectively. We use source code inspection to annotate a subset of the `load` instructions such that we can mark cache blocks for retain (*epic*) in a 4-way set associative cache, utilize a smaller cache block size (*pegwit*, *jpeg*), or default to normal cache operation. For the benchmarks considered, the results indicate that our methods provide between 13% and 20% 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. Technological advances have enabled the implementation of microprocessors with clock cycle times of a few nano-seconds [5,29]. These processors rely on cache memories [22] to satisfy memory accesses at processor speed, and avoid the long delays (200ns-300ns) associated with main memory. Although caches satisfy most memory references, for fast processors the penalty of cache misses can be a significant portion of total execution time.

Conventional cache designs rely solely on hardware to manage the memory hierarchy, and the policies for cache management tend to be relatively naive. Although this simplifies memory system design and does not place any burden on the software, this simplified approach may result in memory hierarchies that do not achieve their full performance potential [1]. Furthermore, these techniques do not exploit any information from the executing pro-

gram about expected reference patterns (the exception being prefetch instructions). These two factors can limit memory hierarchy performance.

These limitations could be overcome by using more sophisticated memory hierarchy management techniques that incorporate information about a program’s memory access pattern. Information on a program’s expected reference pattern can be provided by software or by more sophisticated hardware monitoring past references [12,27]. This paper investigates Informed Caching Environments (ICE) where software provides information to help manage the memory hierarchy, complementing rather than replacing conventional cache management techniques.

The goal of ICE is to exploit software knowledge of a program’s memory access patterns to make more intelligent decisions about cache memory resource allocation than conventional management techniques. The key aspect of ICE is to partially separate cache management policy decisions from the underlying mechanisms, and to expose some of the mechanisms to software so that it can assist in policy decisions. This raises the following questions:

1. What information is obtained from the program?
2. How is information conveyed from the executing program to the memory hierarchy?
3. How is the information exploited by the memory hierarchy for improved performance?

This paper addresses the second question. We present a mechanism—called *annotated memory references*—for software to convey information to the memory hierarchy, and show how it can be used to exploit information on better replacement strategies and better block sizes for improved performance. This paper focuses specifically on a method for statically annotating individual memory reference instructions with one of many possible annotations, while avoiding gross modifications to the instruction set.

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%, and this could be reduced further by scheduling the TAG instructions in unused issue slots [21].

We use the TAG instruction to annotate memory references with one of three possible annotations: *Retain*, *Release* or *WordMode*. The first two annotations allow the application to exercise some control over the cache replacement policy. The last annotation allows the application to exercise some control over the blocksize (and associativity) of the cache. We use the CPROF [13] profiling tool to identify suitable annotations. Using our anno-

tations, we see between 13% and 20% speedups for three media benchmarks (epic, pegwit, jpeg) on a 4-issue dynamically scheduled processor.

The remainder of this paper is organized as follows. Section 2 provides background information on annotated memory references and discusses related work. Our new annotation mechanism is presented in Section 3 and its overheads are evaluated in Section 4. Section 5 describes and evaluates how an application may exploit annotated memory references to manage the cache in a better way. Section 6 concludes the paper.

## 2 Background and Related Work

In this paper we assume a LD/ST architecture and make a distinction between the static instructions that can access memory—called *memory references*—and the dynamic execution of those instructions—called *memory accesses*. A typical memory access is composed of the static memory reference and the dynamic register values used to compute the effective address.<sup>1</sup> Memory access annotations are a series of bits associated with each memory access that are interpreted by the memory hierarchy indicating how to treat the accessed data.

We assume annotations are only performance hints and not required for correct execution of the program. Given the distinction between memory references and memory accesses, there are several possible ways to annotate memory accesses. In particular, annotations can be associated with the memory reference, the address of the accessed data, or the combination of both. Furthermore, the annotations could be generated statically before the program executes, or dynamically during execution. The remainder of this section summarizes several previous annotation techniques.

### 2.1 Related Work

Abraham *et al.* [1] propose static instruction annotations that specify expected source and destination caches in the memory hierarchy. Their mechanism requires significant modification to the instruction set to add the source and destination specifiers to all memory references. They also showed that a small number of memory references are responsible for a large percentage of cache misses. Tyson *et al.* [27] expanded this work and present both static and dynamic memory reference annotation for cache bypassing, thus reducing cache pollution. Gonzalez *et al.* [6] examined the use of static annotations to direct cache blocks into either a spatial or temporal cache. Dynamic memory address annotations were also proposed by McFarling [17] for instruction caches, by Rivers *et al.* [20] and recently by Johnson and Hwu [12] for data caches. Johnson and Merten [11] and Inoue *et al.* [10] also propose dynamic memory address annotations to exploit spatial locality by fetching more than one block.

1. Some effective addresses may be computed from static values.

Temam *et al.* [25] introduce two annotations for conveying information about temporal and spatial locality. Temporal locality is exploited by bouncing data back from the victim cache into the primary cache. Spatial locality is exploited by fetching more than a single cache block.

The SPARC V9 [4] specification includes several types of prefetch instructions. For example, the *prefetch\_read\_once* instruction indicates the specified cache block could be replaced after a single access. The ULTRA SPARC [26] processor also provides a *block\_load* instruction that loads several floating point registers while bypassing the first level cache.

Memory access annotations have also been proposed for reducing coherence overhead in cache coherent shared-memory multiprocessors. For example, read-for-ownership [7], deliver [16], check-in/check-out [8], and associating specific coherence protocols with particular regions of the address space [3].

## 2.2 Discussion

From the above discussion we see there are many possible annotations, ranging from cache bypass to sophisticated variants of prefetching. Dynamic annotation schemes may require large, complex state machines to accurately determine each of these possible annotations. In contrast, static annotation techniques only require exposing a set of mechanisms to software, and allowing software to determine when and how to utilize the mechanisms. Therefore, in this paper we focus on static memory reference annotation.

Static annotations allow software to pro-actively manage the memory hierarchy, while dynamic schemes are generally reactive. Another reactive approach is *informing memory operations* [9] that either set a condition code or invoke specific code on a cache miss. This allows software to monitor its cache behavior and potentially modify its access patterns to improve performance. We note that static annotations could be used in conjunction with reactive methods. Investigating combined pro-active and reactive cache management is beyond the scope of this paper, and we leave it as future work.

Recall, static annotations can be associated with either the data address or the memory reference. Data address annotations are easily incorporated into the page table and TLB. Unfortunately, some programs may want different annotations for the same data during different phases of computation. For sufficiently large data sets, the overhead of updating annotations can become a significant portion of execution time. Alternatively, the different phases of computation generally correspond to different portions of the code, making instruction annotation more attractive.

Memory reference annotation can be implemented by adding a new opcode for each annotation. This minimizes execution time overhead, since no additional instructions are executed. Unfortunately, given the large number of possible annotations that may apply to any level of the memory hierarchy, this requires a large number of available opcodes or significant modifications to the instruction set. At the other extreme, we can minimize opcode use by adding a single instruction that annotates the next memory reference executed. While this facilitates an enormous

number of annotations, the overhead of executing the annotations could become prohibitive. In the next section we present a new technique for implementing annotated memory references that minimizes instruction set modification and execution time overhead.

### 3 A Static Instruction Annotation Mechanism

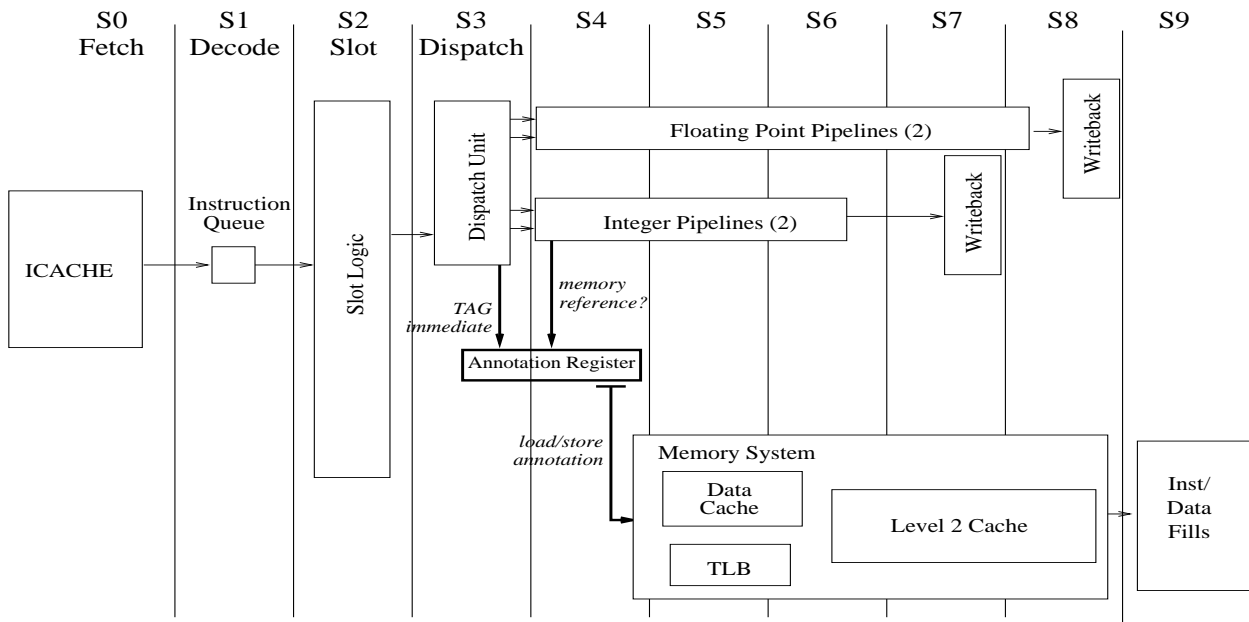
This section presents a new mechanism for passing information from an executing program to the memory hierarchy via annotated memory references. Our goal is to minimize the impact on execution time and avoid gross modifications to the instruction set. We also want to provide a mechanism that can support 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. While conventional microprocessor designs would require a fixed set of annotations, recent studies examining reconfigurable logic [28] may enable applications to efficiently reconfigure cache management mechanisms. 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. This mechanism is similar to that proposed for implementing guarded instructions [18].

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. The remainder of this section describes how to integrate TAG instructions and the annotation register in both statically scheduled and dynamically scheduled processors.

#### 3.1 Annotated Memory References in a Statically Scheduled Superscalar Processor

To provide a concrete example of how to support annotated memory references in a statically scheduled superscalar processor, we use a model similar to the Digital Alpha 21164 [5], a quad-issue, 64-bit RISC architecture. The four functional units include two integer units (which handle all integer operations and memory references), a floating point adder, and a floating point multiplier. Given the right mix of instructions, it is possible for four instructions to begin execution simultaneously in the four functional units. However, because of resource conflicts and dependencies, this is very rare. In fact, in our experiments, the average instructions per cycle (IPC) rarely approached 2.0.



**Figure 1: Integrating the Annotation Register in the 21164 Pipeline**

Figure 1. shows the basic structure of our processor. To understand the operation of TAG instructions in this processor, we concentrate on the first five stages of the pipeline:

**Stage 0:** instruction cache access.

**Stage 1:** instructions are copied into a queue (IQ).

**Stage 2:** the next four instructions in the IQ are examined. Static dependencies are checked and the instructions are slotted into the appropriate functional unit based on the instruction type. Instructions that cannot be slotted during the first available cycle are held in the IQ until the next cycle, when they are examined as part of the next four instruction group.

**Stage 3:** the instruction issue stage which uses a scoreboard for dynamic register and functional unit conflict checking. Instructions are not issued to the appropriate functional until all conflicts are resolved and instructions are not advanced from stage 2 to stage 3 until all instructions in stage 3 have been forwarded to the functional units. All instructions are slotted and issued strictly in program order.

**Stage 4:** instructions begin execution.

In this pipeline, the annotation register is written by TAG instructions in stage 3 and read by memory references in stage 4. To avoid WAR hazards, the annotation register is written in the second half of stage 3 and read in the first half of stage 4.

Since TAG instructions access only the annotation register, they do not require the use of a functional unit and have dependencies only with memory references. Recall, that data placed in the annotation register apply only to memory references appearing later in program order. However, a TAG instruction could be straddled by two mem-

ory references in the instruction stream. If all three instructions are allowed to proceed into stage 3 simultaneously, the TAG instruction would update the annotation register before the earlier memory reference read its annotation.

To ensure correct annotation, we do not allow TAG instructions to move into stage 3 in the same cycle as earlier memory references. However, we do permit simultaneous slotting with later memory references. Since this is the only instruction conflict, the TAG instructions are almost always scheduled immediately.

### 3.2 Annotated Memory References in a Dynamically Scheduled Superscalar Processor

The above discussion provides a mechanism for annotating instructions in an in-order issue processor, the nature of a dynamically scheduled processor requires a slightly modified implementation. Our statically scheduled implementation assumes that memory references will execute in program order, which is valid in a statically scheduled pipeline. In a dynamically scheduled processor, instructions are not necessarily issued in program order, and there is no way to determine *a priori* the issue order.

#### The Out-of-Order Pipeline

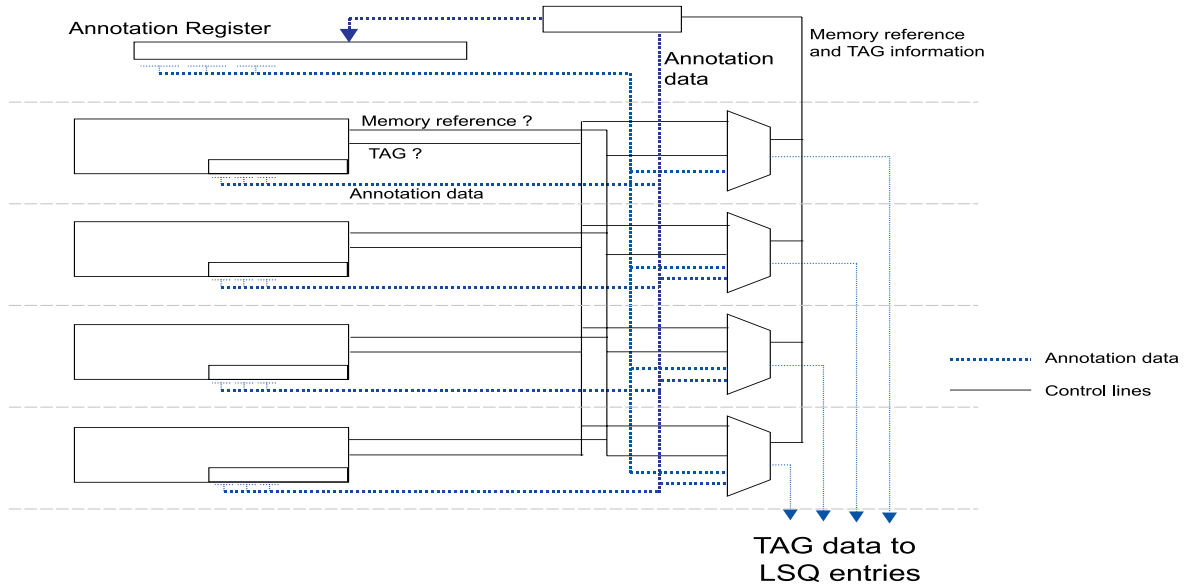
We examine the instruction annotation mechanism in the context of the sim-outorder tool from the simple-scalar [2] toolset which simulates out-of-order issue and execution. It is based on the Register Update Unit [23].

The five SimpleScalar pipeline stages are as follows:

1. **Fetch.** Instructions are fetched from the instruction cache and placed in the fetch queue until the queue is full or an instruction cache miss is suffered.
2. **Dispatch.** Instructions are decoded and each instruction is placed in the RUU queue. Memory references are also given an entry in the load/store queue (LSQ) during this stage.
3. **Issue/Execute.** Execute ready instructions if the necessary functional units are available.
4. **Writeback.** Supply the results of the operation to dependent instructions.
5. **Commit.** Commit results to the register file in program order, free RUU and LSQ entries.

Once an instruction is decoded and linked into the RUU, it sleeps until its source registers are ready and functional unit conflicts are resolved. Once in the RUU, instructions can execute in any order, so annotations must be associated with the memory references during the dispatch stage. Since instructions (including TAGs) are decoded during the dispatch stage, our mechanism associates annotations with memory references during the same pipeline stage and places the annotations in the LSQ. The annotations are then read from the LSQ when necessary by the memory hierarchy.

In a multiple issue machine, where memory references and TAG instructions can be issued in any order in a single cycle, a simple shift register is no longer sufficient to read the annotation data. Here, we introduce a hardware mechanism by which memory references that occur before the TAG get their data from the annotation register,



**Figure 2: Integrating the Annotation Register in a Dynamically Scheduled Processor**

while memory references following the TAG get their data from the TAG immediate field (as shown in Figure 2). The remaining data in the TAG's immediate field is then latched into the annotation register for following memory references to access. This additional hardware adds some complexity to the mechanism which we chose not to include in the statically scheduled implementation.

### 3.3 TAG Instructions and Context Switching

Recall, memory reference annotations are hints to the memory system and do not affect program correctness. An incorrect annotation associated with a memory reference may cause it to be placed in a less than optimal location north memory hierarchy, but its placement will not affect the correct execution of the program. There are therefore two options for handling the contents of the annotation register in the case that a trap or interrupt causes a context switch. The first is to clear the annotation register on a context switch and lose the current annotation data, causing normal memory system behavior for the following group of memory references. Given the relative infrequency of context switches under normal circumstances, the added execution time caused by not taking advantage of those annotations would likely be minimal in most cases. In some situations, however, a synchronous interrupt immediately before a small, frequently executed section of code which makes optimal use of memory references annotations, could cause a large portion of the speedup gained through annotations to be lost. Thus, it is important to understand that annotation register data can be saved and restored during context switches if such an implementation is desired.

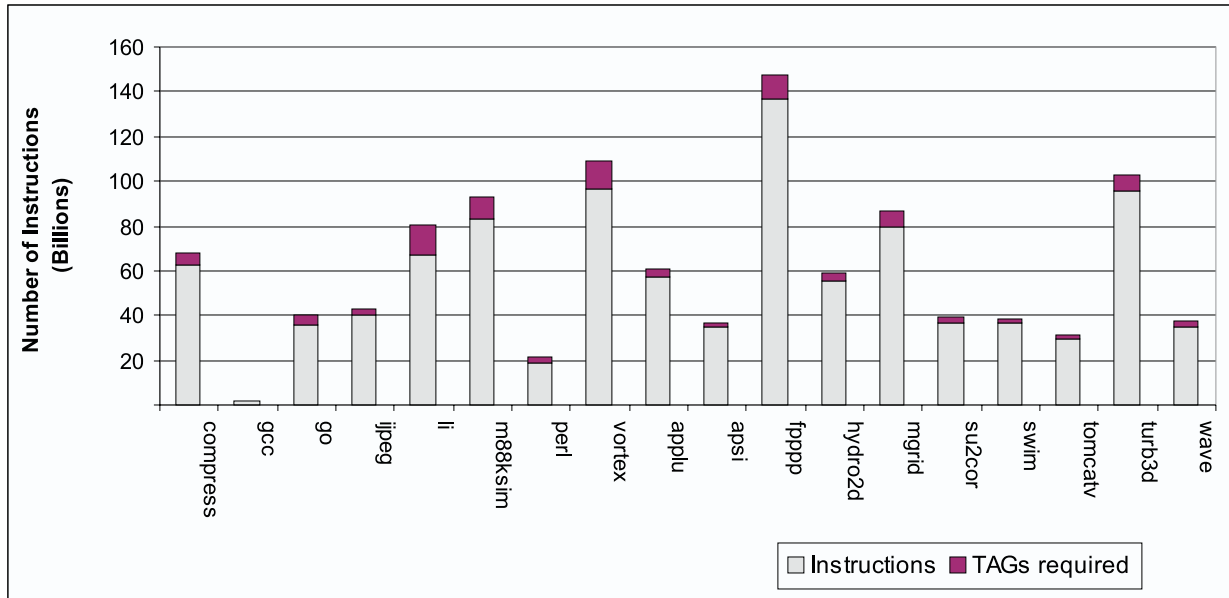
Before the annotation register is saved, it must be restored to the proper state so that when the current process resumes execution, memory references before the next TAG get the correct annotations. For the in-order processor model, the annotation register is updated by shifting the annotations back into the annotation register for the memory references that will be re-issued when the process resumes execution. The out-of-order implementation requires that TAG instructions be given slots in the RUU so the annotation register can be filled with annotation data from the last TAG before the effective PC of the context switch. Annotations for the memory references between the TAG and the context switch PC are then shifted out of the annotation register, and it is saved as part of the process context.

Since no more than six memory references, and probably fewer, can be effected by a single context switch, it is unlikely that a few missed annotations per context switch would have noticeable impact on execution time. In fact, the overall overhead of saving and restoring the annotation register on every context switch might outweigh the benefits. Only in a few pathological cases in which an interrupt occurs regularly before a small section of code which makes optimal use of annotations could the failure to save annotation information cause a significant increase in runtime. Such well defined synchronous cases would normally be associated with deliberate system call traps, in which case using the compiler or profiler to insert TAG instructions immediately after the system call is enough to avoid any adverse affects.

## 4 Annotation Overhead

This section evaluates annotated memory references in the context of both the Digital Alpha 21164 processor and the dynamically scheduled processor described above. The Alpha 32-bit instruction set uses a standard 6 bit opcode and four different types of instructions (PAL Code, Branch, Memory, and Operate Formats). We assume the availability of a single opcode for the TAG instruction. Alternatively, we could use one of the five reserved but unused PAL opcodes. In either case, we assume the remaining 26 bits (32 - 6-bit opcode) are available for annotations. We use 4 bits per annotation allowing up to 16 different annotations, and we can annotate up to 6 memory references with a single TAG instruction, leaving two unused bits. Further analysis of the annotation size and other data not presented here, can be found elsewhere [19].

In this section we assume **all** memory references require annotations in order to show the worst case overhead, although it is unlikely that under normal circumstances all memory references would be annotated. Section 4.1 describes how TAG instructions are added to the instruction stream and examines the number of additional instructions executed by our benchmarks. The number of extra instructions executed does not necessarily reflect TAG overhead in terms of execution time since, as the results presented in Section 4.2 show, TAG instructions are largely hidden by occupying empty instruction issue slots in superscalar machines.



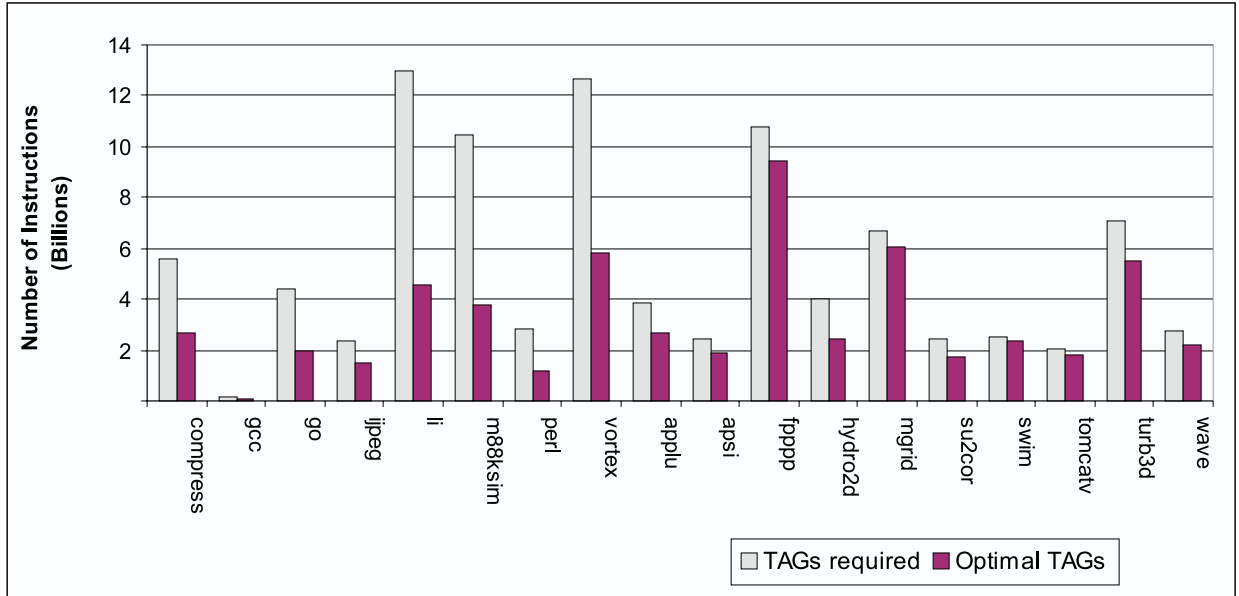
**Figure 3: Instruction Count Overhead**

#### 4.1 Instruction Count Overhead

To measure the overhead of our annotation mechanism we use ATOM [24] to insert TAG instructions in the SPEC95 benchmarks. We currently restrict annotations to within a single basic block, and always insert a TAG instruction at the beginning of basic blocks containing memory references. Although each TAG instruction can annotate up to six memory references, basic block size determines how many memory references are actually annotated by a particular TAG instruction. If the basic block has six or less memory references, the TAG instruction is added as the first instruction in the basic block. If the basic block has more than six memory references, a TAG is added as the first instruction, and then after every sixth memory reference.

We could potentially reduce the number of TAGs inserted by allowing annotations to cross basic block boundaries, or lifting TAG instructions out of loops. However, this requires more sophisticated static analysis (e.g., branch prediction) and changes to the annotation register to reuse annotations. We chose not to investigate these optimizations in this paper. Nonetheless, our results show that annotation overhead can be quite low.

Figure 3 shows the total number of instructions executed when annotating *every* memory reference in each of the SPEC95 benchmarks. Each bar represents a different benchmark, and is divided into original instructions and TAG instructions. For the floating point benchmarks there is an increase of 6.27% to 7.72% in the number of dynamic instructions. For the integer codes, there are an additional 5.5% to 16.2% instructions executed.



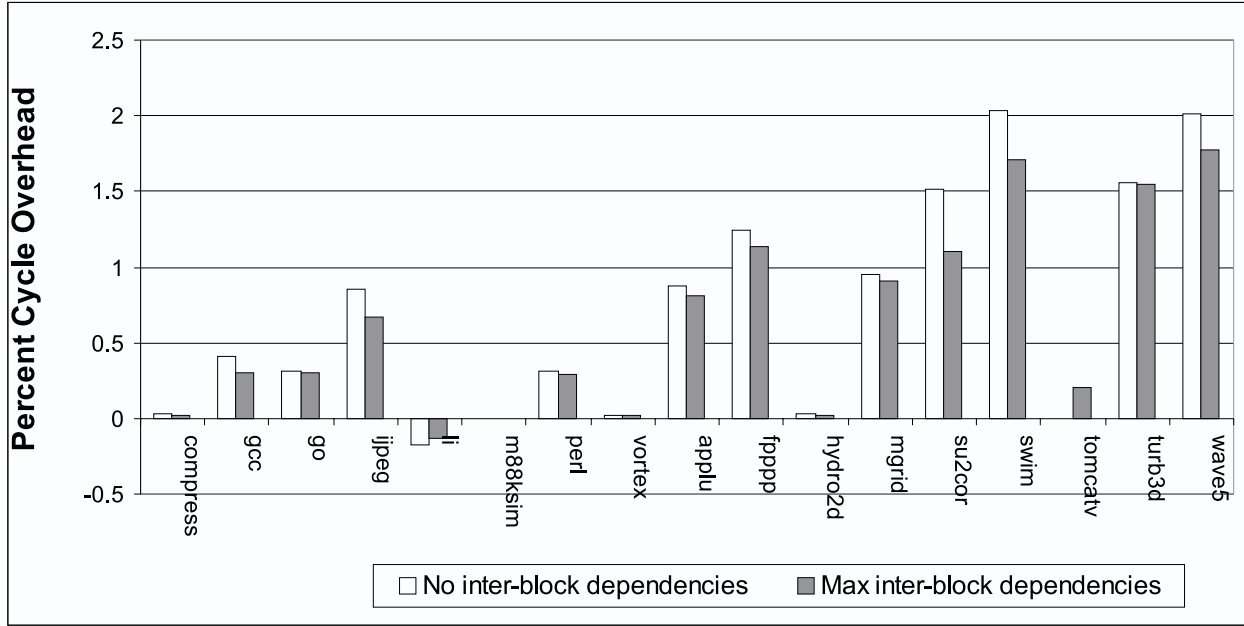
**Figure 4: Required vs. Optimal Tags**

Since our annotations do not cross basic block boundaries, all six annotations in a TAG instruction may not be used. Basic blocks with less than six memory references (which is often the case for integer codes), or anything other than a multiple of six references, result in unused annotation slots. To determine the utilization of TAG instructions, we compare the number of TAG instructions executed to the optimal number. The optimal number of TAG instructions is the total number of memory references executed divided by six (the maximum number of references annotated by a single TAG instruction).

Figure 4 shows the number of TAG instructions executed using our algorithm and the optimal number for each of the benchmarks. From these results we observe that many integer benchmarks execute more than twice the optimal number of TAG instructions, because of their small basic blocks with few memory references. In contrast, the floating point codes generally have larger basic blocks with more memory references. Therefore, they utilize more of the annotations available in a single TAG instruction, and generally incur a lower percentage of additional instructions (as shown in Figure 3). On a single issue machine, the percentage of added execution time would approach the percentage of added instructions. However, the next section shows that TAG instructions add very little overhead to execution time on superscalar processors.

## 4.2 Execution Time Overhead

To determine the impact of TAG instructions on execution time, we use ATOM to compute execution schedules both with and without TAG instructions according to the 21164 issue policies. Cycle overhead is computed by



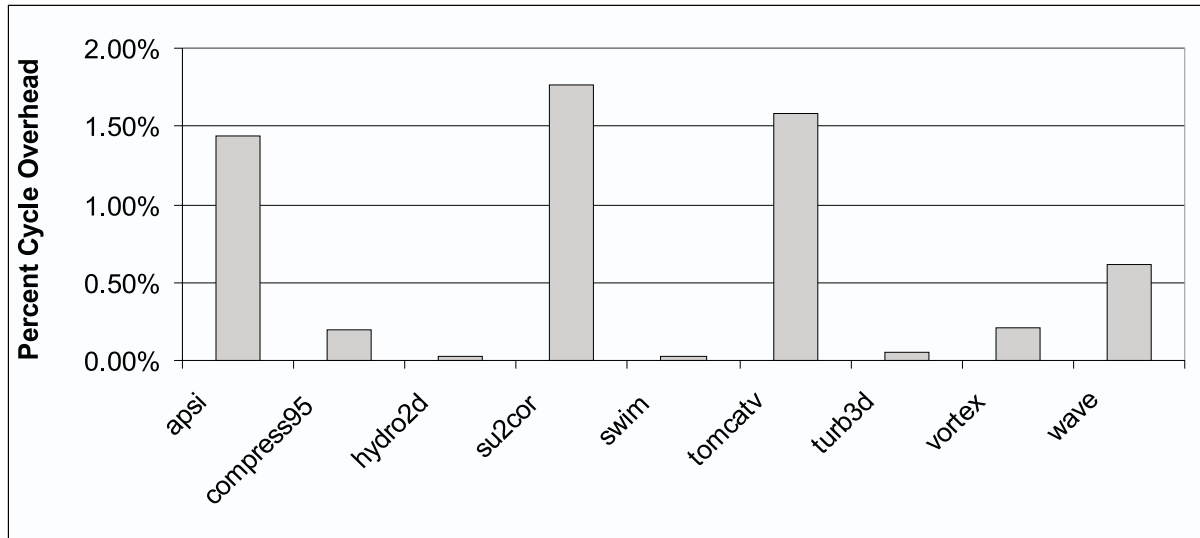
**Figure 5: TAG Overhead for Statically Scheduled Processor**

dividing the number of additional cycles executed in the annotated version of the executable by the number of cycles the unannotated version executes. We do not schedule instructions within a basic block. TAG instructions are always the first instruction in a basic block and immediately after every sixth memory reference.

We assume perfect branch prediction and an ideal memory system in which all memory references complete in one cycle. Adding instructions can affect instruction cache performance. However, for our benchmarks the dynamic code expansion is at most 1.16. Using the model developed by Lebeck and Wood [14], this could increase the number of instruction cache misses by approximately 25% for ill-behaved codes. Again, this is a worst case scenario where all memory references are annotated. In practice, we do not expect to annotate every memory reference and additional instruction cache misses would decrease.

For our experiments, we do not accurately model dependencies across basic block boundaries. Instead, we compute the upper and lower bounds on inter-block dependencies. The upper bound is computed by stalling the start of each basic block until all instructions from the previous basic block complete (long basic block). The lower bound is computed by assuming no inter-block dependencies, and starting each basic block as soon as the preceding block issues its last instruction (short basic block).

Figure 5 and Figure 6 show the execution time overhead for static and dynamic scheduled processors, respectively. For each benchmark in Figure 5, the left bar represents long basic blocks and the right bar represents short basic blocks. The actual value for TAG overhead, excluding the impact of memory system cycles, lies between the two overheads presented for each benchmark. As shown, this produces a very tight bound. For most of the benchmarks, overhead is slightly higher for short basic blocks, because the additional cycles lost to TAG instructions



**Figure 6: TAG Overhead for Dynamically Scheduled Processor**

delay completion of the basic block, whereas their execution can often be overlapped with other instruction execution in the case of long basic block. The exception is `tomcatv`, which shows higher overhead for long basic blocks. This is because the insertion of TAG instructions delays the start of execution of long running instructions (floating point divide) causing even longer delays before the next basic block can begin.

The TAG instructions generally add very little to the execution time of the benchmarks tested. For integer codes, additional cycles range from zero to a maximum of 0.8%, and the floating point overheads range from 0 to 2%. The floating point codes exhibit higher overhead, even though the number of TAG instructions is close to optimal, because these programs do not have as many free issue slots. In contrast, the integer codes have small basic blocks and more available issue slots.

Figure 6 shows the overheads for a 4-issue dynamically scheduled processor with 64 RUU entries and 32 LSQ entries. We examine a smaller set of the SPEC95 benchmarks, due to extensive simulation time. In general, the dynamically scheduled processor incurs lower overhead. However, in some cases the overhead is comparable to that of the statically scheduled processor.

Although these cycle counts do not include memory system cycles, we expect relatively small increases due to instruction cache misses, and increasing data cache stall cycles will generally decrease tag overhead. These results show that even when **every** memory reference is annotated, TAG instruction overhead is relatively low, ranging from 0 to 2%.

## 5 Utilizing Annotated Memory References

The results in the previous section show that annotated memory references can be implemented with low overhead. In this section, we show how to use annotated memory references to improve memory hierarchy performance in a dynamically scheduled processor. As discussed in Section 2.2, there are many possible sources of information that could be used to help manage the memory hierarchy. Since this paper focuses primarily on the mechanism for passing information from the executing program to the memory hierarchy, we limit the analysis in this section to two types of information: information regarding preferable cache replacement policies and information regarding preferable block sizes. 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 propose a set of annotations and demonstrate the usefulness of the proposed annotations by showing between 13% and 20% speedup on the following three benchmarks.

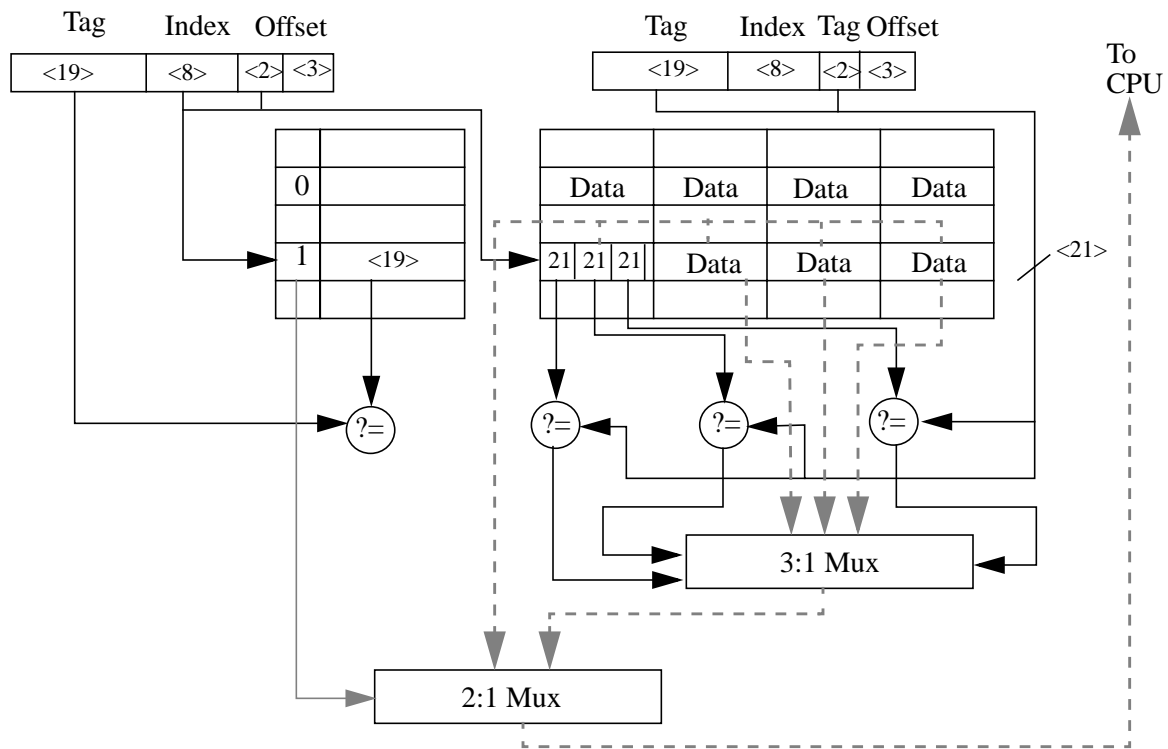
- **epic:** This benchmark is part of the MediaBench [15] benchmark suite. It is a lossy image compression program which is designed for extremely fast decoding on non-floating-point hardware at the expense of slower encoding.
- **jpeg:** This benchmark is part of the SPEC95 benchmark suite. It is a lossy image compression program.
- **pegwit:** This benchmark is part of the MediaBench benchmark suite. It is a public-key encryption and authentication program.

There are several ways to obtain information on memory access patterns (e.g., compiler, programmer, or profiling). In this paper we use CPROF profiling followed by source code inspection to obtain programmer specified annotations, we plan to investigate other methods in the future.

We use CPROF to give us key insights about the memory system behavior of the applications. From the CPROF analysis we found that, for epic, there was scope for improvement in performance if the application could control the replacement policy in the cache. We use two annotations: *Retain* and *Release* for this purpose. The memory system behavior associated with these annotations is described in Section 5.1. A similar analysis of pegwit and jpeg revealed that applications could benefit if they had control over the cache blocksize. We use one annotation: *WordMode*, for this purpose. The memory system behavior associated with this annotation is described in Section 5.2.

### 5.1 Managing Replacement in Set-Associative Caches

This section describes the memory system behavior of the annotation types (*Retain* and *Release*) proposed to let the application manage the replacement policy in set-associative caches. A memory reference with the *Retain* annotation marks the cache block it touches. A cache-block marked *Retain* is not chosen as the block to be replaced



**Figure 7: Implementation of Variable Block Sized Caches:**

32 byte, 1-way sets or 8-byte, 3-way sets

even if it is the least recently used block. Such a block remains in the cache till it is accessed by a memory reference instruction with the *Release* annotation. In a  $k$ -way associative cache, if all blocks in a set are marked *Retain* subsequent blocks which map to this set are not allowed to become resident. Further accesses which map to this set have to bypass the cache.

## 5.2 Changing Block Size

The block size that gives the best performance may vary with data-structure and the phase of the program. There is a compromise in performance because current caches have fixed block sizes. We propose a way to use the cache in a more flexible way so that the block size can be varied.

Our scheme details a way to use the bits in the data-array as additional tag-storage bits. This section describes the memory system behavior of the annotation type (*WordMode*) proposed to let the application control the block size of the cache. We propose associating an extra bit with each set called the “mode bit”.

The mode bit indicates whether a set is in *Word-mode* or in *Block-mode*. There is no change in the operation of the cache when the set being accessed is in *Block-mode*. In *Word-mode*, a set which formerly acted as a single large

frame now behaves as a set containing smaller frames and the additional tag storage required for the additional frames. Also, the set has higher associativity than before because of the increased number of smaller frames in the set. There are two benefits of such a cache organization. The cache transfers are faster and there is less bus contention because of the smaller block size. There is also the added benefit of increased associativity. (Care must be taken, however, to ensure that the blocks transferred in *Word-mode* have very little or no spatial locality. Otherwise the effect of having smaller block size could hurt performance.)

This scheme is illustrated in Figure 7 for one specific case where the larger frame is of size 32 bytes and the smaller frames are of size 8-bytes. Note that the associativity increases to three when the set is in *Word-mode*. The bits from one 8-byte frame are used to store the additional tag information. A memory reference with the *Word-Mode* annotation sets the mode bit for the set it accesses. A memory reference without the *WordMode* annotation clears the mode bit for the set it accesses.

### 5.3 Experiments and 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 and perfect branch prediction.

For epic, we compare performances of systems with an 8KB, 32 byte block, 4-way associative cache with and without *Retain/Release* annotations. (See Figure 8.) We see a 17% reduction in execution time. The miss rate drops from 26.5% to 20.8%.

For pegwit and ijpeg, we compare performances of systems with an 8KB, 32 byte block, direct mapped cache with and without *WordMode* annotations. (See Figure 9.) We see a 11.6% and a 12.4% reduction in execution times for pegwit and ijpeg respectively. The miss rates also drop from 16.8% to 12.8% for pegwit and from 5.7% to 5% for ijpeg.

### 5.4 Summary

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.

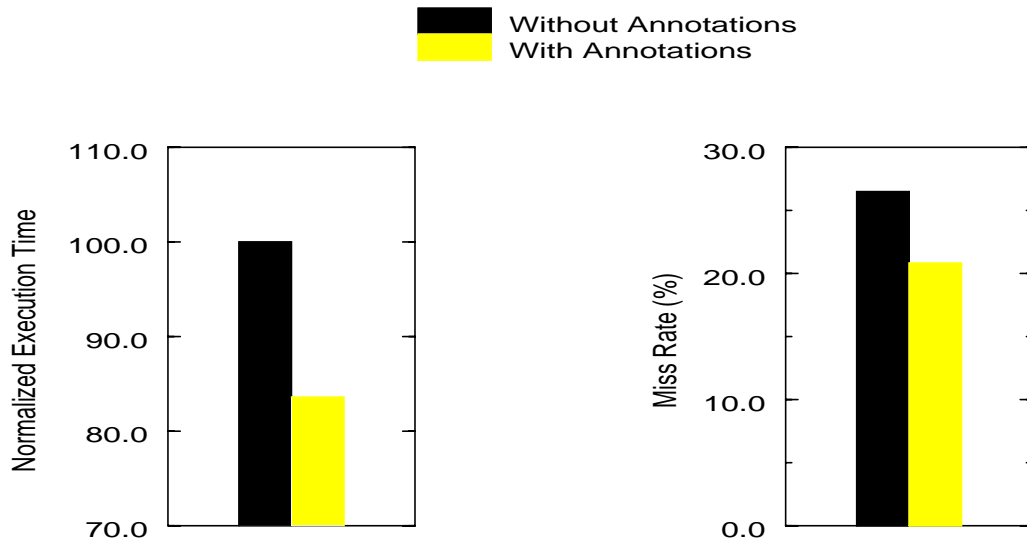


Figure 8: epic: Normalized Execution Times and Miss rates

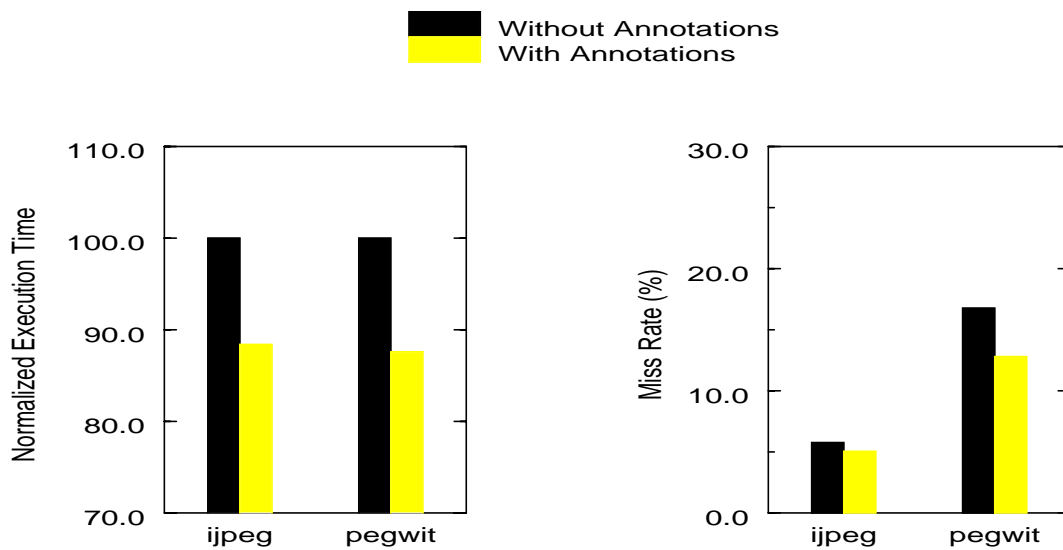


Figure 9: pegwit and ijpeg: Normalized Execution Times and Miss rates

## 6 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 Informed Caching Environments (ICE) where software can provide hints about future memory references and assist in cache management. We focused specifically on annotated memory references—a general mechanism that enables software to pass information to the memory hierarchy.

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, TAG instruction count overhead is between 5.5% and 16.2%. Execution time overhead (assuming a perfect memory system) is even lower, ranging from zero to 2% on a 4-issue dynamically scheduled processor. Using annotated memory references to exploit information on better 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.

## 7 References

- [1] Santosh G. Abraham, Rabin A. Sugumar, Daniel Windheiser, B. R. Rau, and Rajiv Gupta. Predictability of Load/Store Instruction Latencies. *Proceedings of the 26th Annual International Symposium on Microarchitecture*, pages 139–152, December 1993.
- [2] Doug C. Burger, Todd M. Austin, and Steve Bennett. Evaluating Future Microprocessors—the SimpleScalar Tool Set. Technical Report 1308, University of Wisconsin–Madison Computer Sciences Department, July 1996.
- [3] John B. Carter, John K. Bennett, and Willy Zwanepoel. Implementation and Performance of Munin. In *Proceedings of the Thirteenth ACM Symposium on Operating System Principles (SOSP)*, pages 152–164, October 1991.
- [4] Brian Case. SPARC V9 Adds Wealth of New Features. *Microprocessor Report*, 7(9), February 1993.
- [5] John H. Edmondson, Paul Rubinfeld, Ronald Preston, and Vidya Rajagopalan. Superscalar Instruction Execution in the 21164 Alpha Microprocessor. *IEEE Micro*, 15(2):33–43, April 1995.
- [6] Antonio Gonzalez, Carlos Aliagas, and Mateo 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.
- [7] Mark D. Hill, Susan J. Eggers, James R. Larus, George S. Taylor, et al. SPUR: A VLSI Multiprocessor Workstation. *IEEE Computer*, 19(11):8–24, November 1986.
- [8] Mark D. Hill, James R. Larus, Steven K. Reinhardt, and David A. Wood. Cooperative Shared Memory: Software and Hardware for Scalable Multiprocessors. *ACM Transactions on Computer Systems*, 11(4):300–318, November 1993. Earlier version appeared in ASPLOS V, October. 1992.
- [9] Mark Horowitz, Margaret Martonosi, Todd C. Mowry, and Michael D. Smith. Informing Memory Operations: Providing Memory Performance Feedback in Modern Processors. In *Proceedings of the 23rd Annual International Symposium on Computer Architecture*, pages 260–270, May 1996.

- [10] K. Inoue, K. Kai, and K. Murakami. Dynamically Variable Line-Size Cache Exploiting High On-Chip Memory Bandwidth of Merged DRAM/Logic LSIs. In *Proceedings of the Fifth International Symposium on High-Performance Computer Architecture*, pages 218 – 222, 1999.
- [11] T. Johnson and M. Merten. Run-time Spatial Locality Detection and Optimization. In *Proceedings of the 30th Annual International Symposium on Microarchitecture*, pages 57 – 64, December 1997.
- [12] Teresa L. Johnson and Wen mei W. Hwu. Run-time Adaptive Cache Hierarchy Management via Reference Analysis. In *Proceedings of the 24th Annual International Symposium on Computer Architecture*, page to appear, June 1997.
- [13] Alvin R. Lebeck and David A. Wood. Cache Profiling and the SPEC Benchmarks: A Case Study. *IEEE COMPUTER*, 27(10):15–26, October 1994.
- [14] Alvin R. Lebeck and David A. Wood. Active Memory: A New Abstraction for Memory System Simulation. *ACM Transactions on Modeling and Computer Simulation (TOMACS) an extended version of a paper that appeared in Proceedings of the 1995 ACM Sigmetrics Conference on Measurement and Modeling of Computer Systems.*, 7(1):42–77, Jan 1997.
- [15] C. Lee, M. Potkonjak, and W. Mangione-Smith. MediaBench: A Tool for Evaluating Multimedia and Communications Systems. In *Proceedings of the 30th Annual International Symposium on Microarchitecture*, pages 330 – 335, December 1997.
- [16] Daniel Lenoski, James Laudon, Kourosh Gharachorloo, Wolf-Dietrich Weber, Anoop Gupta, John Hennessy, Mark Horowitz, and Monica Lam. The Stanford DASH Multiprocessor. *IEEE Computer*, 25(3):63–79, March 1992.
- [17] Scott McFarling. Cache Replacement with Dynamic Exclusion. In *Proceedings of the 19th Annual International Symposium on Computer Architecture*, pages 191–200, May 1992.
- [18] Dionisios N. Pnevmatikatos and Gurindar S. Sohi. Guarded Execution and Branch Prediction in Dynamic ILP Processors. *Proceedings of the 21st Annual International Symposium on Computer Architecture*, pages 120–129, April 1994.
- [19] David R. Raymond. Annotated Memory References: A Mechanism for Informed Cache Management, Masters Thesis. Technical report, Duke University Computer Science Department, May 1998.
- [20] J. Rivers and E. S. Davidson. Reducing Conflicts in Direct-Mapped Caches with Temporality-Based Design. In *International Conference on Parallel Processing*, 1996.
- [21] Eric Schnarr and James R. Larus. Instruction Scheduling and Executable Editing. In *Proceedings of the 29th Annual International Symposium on Microarchitecture*, December 1996.
- [22] Alan J. Smith. Cache Memories. *ACM Computing Surveys*, 14(3):473–530, 1982.
- [23] Gurindar Sohi. Instruction Issue Logic for High Performance, Interruptable, Multiple Functional Unit, Pipelined Computers. *IEEE Transactions on Computers*, 39(3):349–359, March 1990.
- [24] Amitabh Srivastava and Alan 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.
- [25] O. Temam and N. Drach. Software Assistance for Data Caches. In *Proceedings of the 1st International Symposium on High Performance Computer Architecture (HPCA)*, pages 154–163, January 1995.
- [26] Marc Temblay, Michael O'Connor, Venkatesh Narayanan, and Liang He. VIS Speeds New Media Processing. *IEEE Micro*, 16(4):10 – 20, August 1996.

- [27] Gary Tyson, Matthew Farrens, John Matthews, and Andrew R. Pleszkun. A Modified Approach to Data Cache Management. In *Proceedings of the 28th Annual International Symposium on Microarchitecture*, December 1995.
- [28] Elliot Waingold, Michael Taylor, Devabhaktuni Srikrishna, Vivek Sarkar, Walter Lee, Victor Lee, Jang Kim, Matthew Frank, Peter Finch, Rajeev Barua, Jonathan Babb, Saman Amarasinghe, , and Anant Agarwal. Baring it all to Software: Raw Machines. *IEEE Computer*, pages 86–93, September 1997.
- [29] Kenneth C. Yeager. The MIPS R10000 Superscalar Microprocessor. *IEEE Micro*, 16(2):28–40, April 1996.