

Software Caching using Dynamic Binary Rewriting for Embedded Devices

Chad Huneycutt, Kenneth Mackenzie
College of Computing Georgia Institute of Technology
Atlanta, GA 30332
{ chadh, kenmac }@cc.gatech.edu

Abstract

A software cache implements instruction and data caching entirely in software. Dynamic binary rewriting offers a means to specialize the software cache miss checks at cache miss time. We describe a software cache system implemented using dynamic binary rewriting and observe that the combination is particularly appropriate for the scenario of a simple embedded system connected to a more powerful server over a network. As two examples, consider a network of sensors with local processing or cell phones connected to cell towers.

Software caching with binary rewriting fits the embedded scenario in three ways. First, automatic caching enables the convenient programming model of a large address space at the embedded system. Second, an all-software implementation minimizes cost and (potentially) power over a hardware implementation while maximizing flexibility. Flexibility is crucial to achieving high, predictable hit rates which are necessary when cache miss time is high. Third, dynamic binary rewriting shifts some software caching overhead from the simple embedded system to the powerful server. The rewriter reduces the time and space of cache miss checks through specialization at the cost of extra overhead in servicing cache misses. The cache miss checks run in the embedded system while the misses are handled by the server.

We describe a software cache system for instruction caching only using dynamic binary rewriting and present preliminary results for the performance of instruction caching in that system. We measure time overheads of 19% compared to no caching. We also show that we can guarantee a 100% hit rate for codes that fit in the cache. For comparison, we estimate that a comparable hardware cache would have space overhead of 12-18% for its tag array and would offer no hit rate guarantee.

Keywords: Software Caching, Binary Translation, Dynamic Compilation, Low Power, Embedded

1 Introduction

Programmability is the primary motivation for automatic management of a memory hierarchy (automatic caching). Programmability enables new functionality, can reduce time to market and improve time in market by strengthening reusability of software in a product line. A memory hierarchy can be managed manually by a programmer but the effort is considerable. General-purpose computers universally employ automatic management of the memory hierarchy via hardware caching and virtual memory mechanisms, but many DSPs and microcontrollers used in embedded systems avoid caching to absolutely minimize its drawbacks.

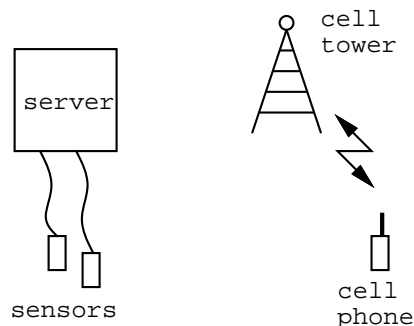


Figure 1: Two examples of networked embedded devices: distributed sensors and cell phones. Both include local processing but are continuously connected to more powerful servers.

The drawbacks of caching are added cost, added power and loss of predictable timing. A cache design targeting embedded systems must address these three issues.

A natural target class of embedded applications is one in which the embedded system is in constant communication with a server. In this scenario the server maintains the lower levels of the memory hierarchy for the embedded system. Two examples of this class (Figure 1) include a distributed network of low-cost sensors with embedded processing and distributed cell phones which communicate with cell towers. In each example the embedded device is nearly useless without the communication connection and thus can afford to depend partially on the server for basic functionality. Also in each example, the embedded device is heavily constrained by cost and/or power consumption while the server can be far more powerful.

A specific example of the benefits of caching is illustrated in Figure 2. Consider a sensor with local processing. The sensor operates in one of several modes (initialization, calibration, daytime, nighttime), but only two are performance-critical and the transitions between the modes are infrequent. Figure 2 depicts an idealized view of the memory address space in this system in which the code/data for each mode is disjoint. The key observation is that only the performance-critical modes need to fit entirely in memory and then only one at a time. The local memory can be sized to fit one mode, reducing cost and power over a larger memory.

Software Caching.

Implementing caching in *software* helps address the issues of cost, power and predictability. An all-software cache design addresses these issues as follows:

- **Cost:** Software caching requires no die area devoted to caching. A fraction of on-chip memory may be consumed by tags and other overheads but the fraction is an adjustable tradeoff. Software caching can be implemented today on the simplest of microcontrollers.

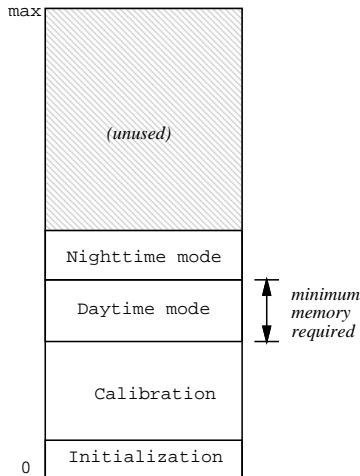


Figure 2: Example: code space for an embedded processor providing local processing to a sensor. The code includes modules for initialization, calibration and two modes of operation, but only one module is active at a given time. The device physical memory can be sized to fit one module.

- **Power:** A hardware cache pays the power penalty of a tag check on every access. The primary performance objective in a software caching design is to avoid cache tag checks. Even though a program using the software cache likely requires additional cycles it can avoid a larger fraction of tag checks for a net savings in memory system power.
- **Predictability:** Software caching offers flexibility to provide predictable timing (zero misses during a section of code) in two ways. At a minimum, one can define manually managed areas of on-chip memory and thus provide predictable modules within the application. More interestingly, a software cache can be fully associative so that a module can be guaranteed free of conflict misses provided the module fits in the cache (Figure 2).

The costs of an all-software cache design versus a hardware design are that, even when all cache accesses are hits, some extra instructions must be executed for cache tag checks, mapping or other overhead.

Software caching may be used to implement a particular level in a multilevel caching system. For instance, the L2 cache could be managed in software while the L1 caches are conventional. For our embedded scenario, we envision a single level of caching at the embedded system chip with a local memory in the range of 1s to 100s of kilobytes.

Dynamic Binary Rewriting.

Dynamic binary rewriting in the context of software caching means modifying the instructions of the program at cache miss time. The instructions are rewritten to encode part of the cache state in the instructions themselves. For instance, branches in code can be rewritten to point to the in-cache version of a branch target (if the target is in-cache) or to the miss handler (if the target is not in-cache). Similarly, (some) data accesses can be rewritten to point to in-cache locations.

The key contribution of dynamic rewriting to software caching is that it provides a means to make a fully associative cache with low overhead for hits. Rewritten instructions may be placed anywhere in the cache on a basic-block by basic-block basis. Fully associative hardware caches are impractical unless the cache block size is large.

Rewriting helps reduce the cache hit time and code size at the expense of increased cache miss time. In the target scenario of net-

worked embedded devices, rewriting shifts the cost of caching from the (constrained) embedded system to the (relatively unconstrained) server.

Relationship to Java Just-in-Time Compilation.

The idea of using dynamic rewriting to support software caching is orthogonal to the idea of using just-in-time compilation to accelerate interpretation in Java. The two concepts are natural to be combined, however. The pointer constraints of Java would help software caching by providing more opportunities to specialize data references than can be found in arbitrary machine code. In return, software caching as described here offers a means of organizing a “Distributed JVM” [7] with a tiny memory footprint on an embedded system: put the just-in-time compiler on the server and manage the embedded system’s memory as a cache.

This section has introduced software caching, its application to embedded systems connected by a network to a server and the use of dynamic binary rewriting to improve software caching. The rest of this document is organized as follows. In the interests of a concreteness, we present a detailed design first in two sections. Section 2 describes an instruction cache using dynamic binary rewriting in software. We describe a prototype and results showing 19% overhead in time. Section 3 describes a paper design for a data cache. After the design presentation, Section 4 expands on the range of tradeoffs in implementing a memory hierarchy in hardware or software, with or without rewriting and discusses related work. Finally, Section 5 concludes.

2 Software Caching of Instructions

This section describes a software cache for instructions implemented using dynamic binary rewriting to minimize tag checks. The rewriting approach eliminates most tag checks and permits the instruction cache to be fully associative. The cost is, of course, time and complexity of handling cache misses. The section discusses the basic principles of rewriting, the issues that arise from cache invalidations and from ambiguous pointers, describes our prototype instruction cache on a SPARC system and presents performance results for that prototype.

Rewriting Mechanism.

Figure 3 illustrates the basic rewriting mechanism. Instructions from the original program are copied into a section of on-chip memory called the *tcache* (translation cache), basic block by basic block. At copy time, the branch at the end of the basic block is rewritten to point to a cache miss handler for the following block. If the branch is subsequently taken, then the branch is rewritten again to point to the in-cache copy of the target basic block.

The example illustrates both advantages of rewriting. First, after all the basic blocks used in the loop have been discovered, copied and rewritten in the cache, the loop runs at full speed with no cache tag checks (possibly extra branch instructions). Second, the instruction cache is effectively fully associative. Instructions in the source program may be relocated anywhere in the *tcache*.

Interpreted in conventional caching terms, the rewritten instructions encode cache tags. When an object (e.g. a basic block of instructions) is in the cache, relevant jumps and branches in the cache are adjusted to point to that object. When the object is not in-cache, relevant instructions are adjusted to point to the cache miss handler instead. Instead of tags or a mapping table, the state of the cache is implicit in the branch instructions which, for the most part, are required in the program anyway. Not all tags can be represented this way, but many can be.

Alternatively, interpreted in dynamic compilation lingo, we use binary rewriting to specialize mapping and tag-checking code for the current state of the cache. The specialization succeeds in

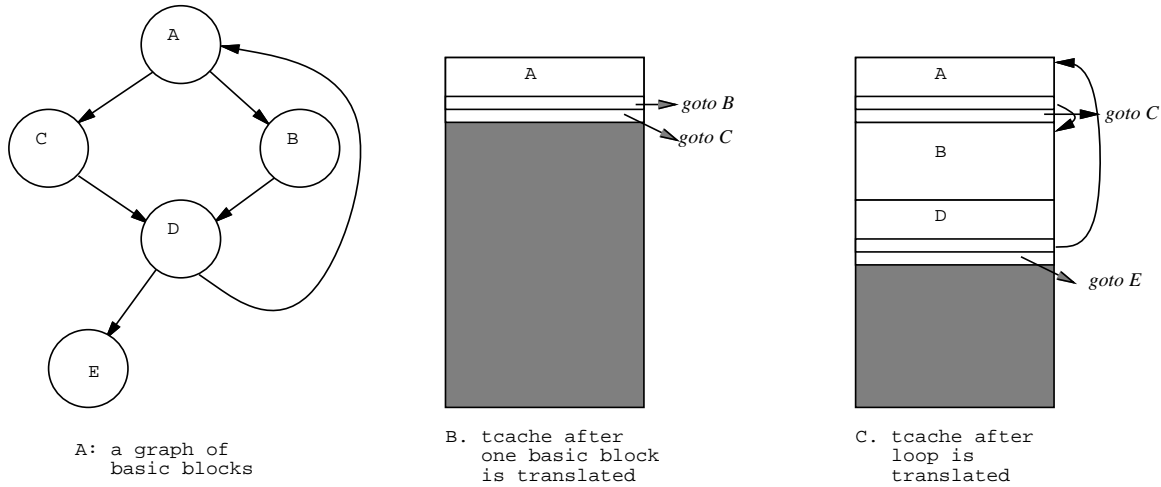


Figure 3: Example of Translation Cache (*tcache*) operation. Basic blocks in the original program (A), are copied on demand to the translation cache. As they are copied, branches exiting the blocks are initially rewritten to point to cache miss handlers (B) and eventually, if used, again rewritten to point to other blocks in the *tcache* (C).

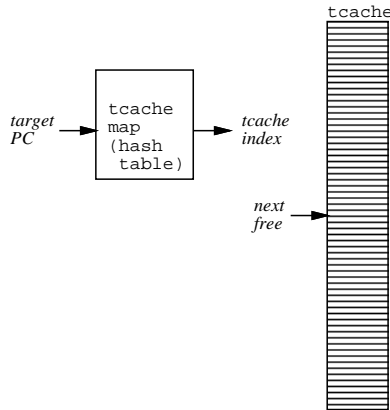


Figure 4: Data structures used for instruction caching. The *tcache* (translation cache) holds the rewritten instructions. The *tcache map* holds mappings from the original address space to indices in the *tcache*.

removing the tag check entirely for the common case of branch instructions whose destinations are known at rewriting time.

We address three more issues beyond the basic rewriting mechanism in order to complete the cache design. First, we must provide a means to invalidate cache entries. Second, some pointers to instructions are not known at rewriting time (e.g. return addresses). These “ambiguous” pointers require other caching and invalidation mechanisms and/or (modest) programming model limitations. Third, invalidations raise an atomicity issue: cache mapping and tag checking code must be atomic with respect to invalidations.

Invalidation.

One invalidates a conventional cache entry by changing the tag or the mapping. With rewriting, we need to find and change any and all pointers that implicitly mark a basic block as valid.

There are two sources of such pointers: pointers embedded in the instructions (the branches) of other basic blocks in the *tcache* and pointers to code stored in data, e.g in return addresses on the stack or in thread control blocks. Pointers in the *tcache* are easy to record at the time they are created. To find pointers in data, the runtime system must know the layout of all such data.

Ambiguous Pointers.

Rewriting is limited by pointers that cannot be determined at rewriting time (ambiguous pointers). For instance, computed jumps and returns from subroutines involve pointers to instructions where the pointers are unknown at rewriting time.

There are three approaches to handling ambiguous pointers:

- Use code analysis to determine constant values for pointers (if possible) or to generate useful assertions about pointers. For instance, a computed jump may in fact be provably constant at rewriting time.
- Decree limitations to the programming model in order to provide useful invariants about pointers. For instance, limit the call/return idiom so that the location of return addresses is always known.
- Perform a cache lookup in software at runtime. A runtime cache lookup is always possible as a fallback strategy.

We use the strategy of applying limitations to the programming model combined with cache lookup as a last-ditch approach. The limitations are modest and correspond essentially to code produced by a compiler. A more sophisticated rewriter could employ elaborate analysis [4] or an ISA, such as the Java Virtual Machine, which is more amenable to analysis.

Atomicity.

Software caching combined with interrupts introduces an atomicity problem with respect to the state of the cache. The problem is that the time from the translation or validation of an address to the use of that address must appear atomic with respect to changes in the translation or validity of the address. For example, since a software cache tag check adds multiple instructions to a load operation, an interrupt can arrive after the tag check succeeds but before the load is executed. There is a problem if the interrupt then invalidates the cache line that is about to be loaded.

There are several standard solutions to an atomicity problem of this type:

- Disable interrupts during the critical section from cache check to use.
- Test for the PC in the critical section at interrupt time and, if so, back up the PC to abort the critical section.

- Test for PCs in critical sections only at invalidation time, but test *any* possible saved PC.

Identifying a cache check critical section requires that the checks use a recognizable idiom. There is then a tradeoff between the flexibility in writing checks and making them identifiable. For instance, one would like to rewrite computed jumps to replace the constant jump addresses in the original code, but then those constants must be found and invalidated. This may be impossible, due to the inability to identify (or to abort) the instructions used to read the constant and compute the jump.

We do not yet address the atomicity problem because we do not yet support interrupts. We advocate having the server perform an arbitrarily elaborate check at invalidation time.

2.1 I-Cache Design

We have implemented a prototype of a software instruction cache on a Sun SPARC-based workstation. Our I-cache design uses rewriting for the majority of control transfers, relies on modest restrictions to the programming model to rule out several forms of pointer ambiguity, and resorts to a mapping through a hash table for the remaining ambiguous pointers.

There is no embedded system per se, but instead one program that includes both “server” and “embedded” code. The server copies and rewrites the embedded code into a *tcache* in memory and then executes that code directly. Communication between the embedded code and the server is accomplished by jumping back and forth in places where a real embedded system would have to perform an RPC to the server. Data caching is not modeled – the rewritten embedded code accesses data objects in the same memory locations as it would have if it had not been rewritten.

We define the following limitations to the programming model to rule out some forms of ambiguous pointers. The limitations are modest in that they correspond to idioms that a compiler would likely produce anyway.

- Procedure return addresses must be identifiable to the runtime system at all times. Specifically, procedure call and return use unique instructions, the current return address is stored in a particular register and a particular place in the stack frame, the stack layout must be known to the runtime system and any non-stack storage (e.g. thread control blocks) must be registered with the runtime system. The instruction and register requirements are met naturally by the compiler. The stack layout is already defined in SPARC because of SPARC’s use of trap-managed register windows. The interface to the thread system is the only new requirement (and we have not yet implemented it).
- Self-modifying programs must explicitly invalidate newly-written instructions before they can be used. For instance, dynamically linked libraries typically rewrite a jump table to the library. SPARC already requires this invalidation as its instruction and data caches are not coherent.

Finally, we do not limit pointers arising from computed jumps:

- For computed jumps, we fall back to a lookup through a hash table. The hash table need only contain the subset of destination addresses that could actually occur at the computed jump – a subset of the *tcache map* in Figure 3.

The implementation runs on UltraSPARC workstations under Solaris 2.7 and Linux 2.2.14. The next subsection describes results.

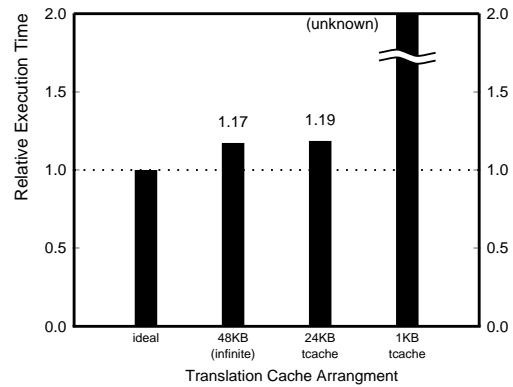


Figure 5: Relative execution time for the software instruction cache. The times are for 129 .compress from SPEC95 and are normalized to the “ideal” execution time with no software cache.

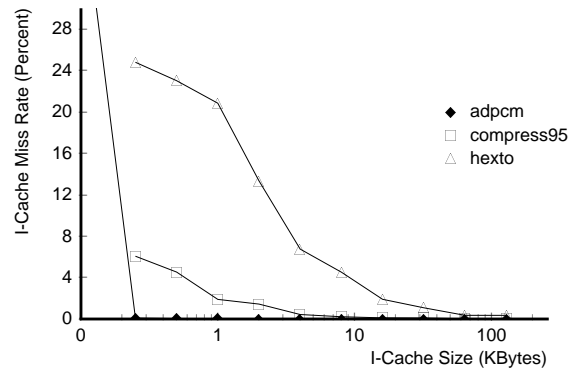


Figure 6: Hardware cache miss rate versus cache size. The cache is a direct-mapped L1 instruction cache with 16-byte blocks. The cache size is the size of data only – tags for 32-bit addresses would add an extra 11-18%.

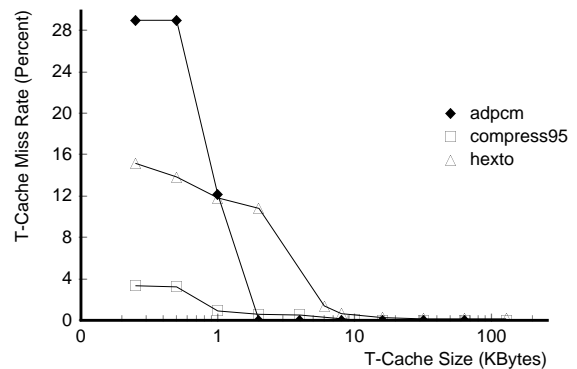


Figure 7: Software cache miss rate versus cache size. The software cache size is the size of the *tcache* in bytes. The software miss rate is the number of basic blocks translated divided by the number of instructions executed.

App.	Dynamic .text	Static .text
129.compress	21KB	193KB
adpcmenc	1KB	139B
hextobdd	23KB	205KB

Table 1: Application dynamically- and statically- linked text segment sizes. 129.compress is from the SPEC CPU95 suite, adpcmenc is from MediaBench, and hextobdd is a local graph manipulation application. All were compiled with gcc -O4. The dynamic .text segment size is an underestimate while the static .text size is an overestimate.

2.2 I-Cache Results

We ran experiments using a few small benchmarks. The benchmark characteristics are summarized in Table 1. We compare the performance of the software instruction cache with that of a simple hardware cache: a direct-mapped cache with 16-byte blocks.

The main point to take away is that the software cache comes close to the performance of the hardware cache in time and space costs, yet the software cache requires no hardware support.

Figure 5 shows the time performance of the software cache. We compare the wall times of compress95 running under the software cache system with that of compress95 running natively on the UltraSPARC. The input to compress95 is large enough that the initial startup time of the cache is insignificant. The result shows that the software cache operates with a slowdown of 19% provided the working set of the code fits entirely in the cache. Of course, if the working set does not fit, performance is awful (the rightmost bar) but the system continues to operate.

Figures 6 and 7 show the space requirements of software and hardware caches for the benchmarks by looking for the “knee” of the miss rate curve in each case. The result is that, for this coarse test, the cache size required to capture the working set appears similar for the software cache as for a hardware cache.

Finally, note that the working sets observed in the Figures 6 and 7 are a tiny fraction of the program code sizes listed in Table 1. This ratio is an illustration of the benefits of caching depicted earlier in Figure 2: only a fraction of the address space needs to be physical.

3 Software Caching of Data

This section sketches a paper design for a software data cache to complement the instruction cache presented in the previous section. A data cache can benefit from rewriting but the issues of pointer disambiguation are much more serious than for instructions.

We present one design that has two main features. First, it again exploits some modest limitations to the programming model to eliminate some forms of pointer ambiguity. Second, we define a “slow hit” as a cached data item found on-chip, but with extra effort beyond the case of an ordinary hit. We then can build a fully associative data cache where at least slow hits can be guaranteed provided the data fit in cache.

Basic Caching Mechanism.

The basic mechanism for data caching in software is to emulate the action of a hardware cache by implementing mapping, tags or both. Load and store instructions are rewritten as a sequence of instructions to perform the mapping or tag check. The key goal for a software data cache is to reduce the number of tag checks.

Reducing Tag Checks.

We reduce tag checks in two ways. First, we rely on limitations to the programming model to treat scalar objects on the stack differently from the rest of data. The stack may thus be treated specially and most tag checks on the stack eliminated. Second, we

```
ld [caddr], %r1 --> ld [dataadr(caddr)], %r1
    the constant address is
    known to be in-cache

ld [%r2], %r1 --> ld [guess(ldaddr)], %r1
    the variable address is
    not known but we have
    a prediction about which
    line in cache
    srl %r2, NOFFSETBITS, %r4
    cmp %r3, %r4
    bne lmiss
    sll %r1, NBLOCKBITS - 2
    and %r2, NBLOCKMASK, %r3
    add %r1, %r3, %r1
    ld [database + %r1], %r1
```

Figure 8: Instruction sequences used to implement data caching. If the address is constant, the translated, in-cache address can be rewritten in the code (top). Otherwise, the address is looked up by searching the cache starting with a predicted entry in the cache (bottom).

use rewriting to specialize accesses with constant addresses (scalar global variables).

Additional tag check reductions are possible with additional effort or limitations. For instance, the HotPages system [4] uses a static compiler with sophisticated pointer analysis to selectively create deeper loop nests such that array accesses within the innermost loop all access the same cache line and can thus be checked with one tag check at the beginning of the innermost loop. Alternatively, Java restricts pointers at the language level and passes considerable information down to instruction level in the form of Java Virtual Machine in bytecodes.

Full Associativity.

We propose to make a data cache that is fully associative yet includes a fast, predicted common-case path. A fully associative software cache for data will be slow because we cannot get rid of as many tag checks as we can for instructions. We propose to implement the fully associative cache with *predictions* to speed accesses.

3.1 D-Cache Design

We propose to implement data caching in two pieces: a specialized stack cache (*scache*) and a general-purpose data cache (*dcache*). Local memory is thus statically divided into three regions: *tcache*, *scache* and *dcache*.

The stack cache holds stack frames in a circular buffer managed as a linked list. A presence check is made at procedure entrance and exit time. The stack cache is assumed to hold at least two frames so leaf procedures can avoid the exit check.

The data cache is fully associative using fixed-size blocks with tags. The blocks and corresponding tags are kept in sorted order. A memory access has three stages. First, the load/store instruction is expanded in-line into a tag load and check (aided by a block index prediction, described below). A tag match indicates a hit. On a mismatch, a subroutine performs a binary search of the entire *dcache* for the indicated tag. A match at this point is termed a “slow hit”. If there is no match in the *dcache*, the miss handler communicates with the server to perform a replacement.

The cache check code must guess the index of the block (and tag) in the fully-associative *dcache*. We use additional variables outside the *dcache* to maintain predictions. The variable predicts that the next access will hit the same cache location. As a variation, the hit code could implement a stride prediction (since the *dcache* array is in sorted order) or a “second-chance” prediction of index $i + 1$ on a miss to index i before searching the whole *dcache*. The prediction variable does not need to be updated when the *dcache* is reorganized although an update could help.

Figure 8 shows the SPARC assembly for the case of a specialized load/store for a global scalar versus the code for a predicted lookup

in the sorted *dcache*.

In summary, the data caching design has two features. First, it uses ISA restrictions and rewriting to specialize a subset of data accesses. Second, it provides for a fully associative data cache with prediction for faster access. The guaranteed memory latency is the speed of a slow hit: the time to find data on-chip without consulting the server.

4 Discussion

Software caching opens up the design space for caching. Software caching appears particularly appropriate for the scenario of a low-cost/low-power embedded system connected to a server where the server actively participates in managing the memory hierarchy. This section lists the issues of software caching and discusses related work.

Caching Issues.

- *Manual vs. automatic management.* Manual management of the memory hierarchy, like assembly language programming, offers the highest performance but the most difficult programming model.
- *Hardware vs. software.* Viewed in terms of time, hardware mechanisms can afford to check every single memory access while software mechanisms will depend on avoiding such checks. Viewed in terms of power consumption, avoiding checks is always desirable.
- *Binary Rewriting.* Rewriting (applied to software caching) specializes instruction sequences for the current cache state. Trace caching [5] or in-cache branch prediction can be considered hardware versions of rewriting. Software rewriting eliminates tag checks through specialization.
- *Value of Limitations.* Some very useful specializations are illegal under the contract of the instruction set. These specializations may be used either (a) speculatively, (b) when the illegal situation can be proven not to arise or (c) if additional limitations are imposed on the instruction set. A small set of limitations to the instruction set corresponding essentially to the natural idioms of compiler-produced machine code enable a number of important specializations.
- *Language-level help, e.g. via Java.* Language-level support is orthogonal but can considerably simplify either manual or automatic memory management. With a manual approach, one can use objects to help delimit the code and data belonging to an operating “mode”. With an automatic approach, language-level restrictions on pointers considerably simplify pointer analysis and enable better specialization of data caching.

Related Work.

There is related work in all-software caching, fast cache simulation, software-*managed* caches where software handles refills only, dynamic binary rewriting and just-in-time compilation.

Software caching has been implemented using static rewriting mechanisms. The HotPages system uses transformations tightly integrated with a compiler that features sophisticated pointer analysis [4]. In contrast, we focus on the complementary problem of what we can do with dynamic rewriting and without deep analysis. The Shasta shared memory system used static binary rewriting to implement software caching for shared variables in a multiprocessor [6]. Such techniques could be used to implement shared memory between an embedded system and its server.

Fast simulators have implemented forms of software caching using dynamic rewriting. The Talisman-2 [1], Shade [2] and Embra [8] simulators use this technique. Simulators must deal with additional simulation detail which limits their speed. Also, the dynamic rewriters deal with very large *tcache* sizes and avoid the problem of invalidating individual entries by invalidating the *tcache* in its entirety and infrequently.

As the ratio of processor to memory speed increases, software management of the memory hierarchy creeps upward from the traditional domain of virtual memory management. Software has been proposed to implement replacement for a fully-associative L2 cache [3]. We propose to take an additional step.

A Distributed JVM [7] splits the implementation of a Java just-in-time compiler system between servers and clients to achieve benefits of consistency and performance. We use a similar notion of distributed functionality between a server and an embedded system to minimize memory footprint, system power and cost on the embedded system.

5 Conclusion

This paper described software caching and a particular way of implementing software caching using dynamic binary rewriting. We observe that this system is particularly natural for a class of embedded applications: one where a low-cost, low-power embedded device is continuously connected to a more powerful server. The caching system provides the programmability of a memory hierarchy while the software implementation minimizes system cost and potentially reduces power consumption over a hardware caching implementation. The binary rewriting approach shifts some caching overhead from the embedded device to the more powerful server, matching the characteristics of the scenario.

The paper contributes a prototype instruction caching system with measurements of its performance and a discussion of design issues for a full-blown software caching system with data caching.

References

- [1] Robert Bedichek. Talisman-2 — A Fugu System Simulator. <http://bedichek.org/robert/talisman2/>, August 1999.
- [2] Robert F. Cmelik and David Keppel. Shade: A Fast Instruction-Set Simulator for Execution Profiling. CSE 93-06-06, University of Washington, 1993.
- [3] Erik G. Hallnor and Steven K. Reinhardt. A Fully Associative Software-Managed Cache Design. In *Proceedings of the 27th Annual International Symposium on Computer Architecture*, 2000.
- [4] Csaba Andras Moritz, Matthew Frank, Walter Lee, and Saman Amarasinghe. Hot Pages: Software Caching for Raw Microprocessors. MIT-LCS-TM 599, Massachusetts Institute of Technology, 1999.
- [5] E. Rotenberg, S. Bennett, and J. Smith. Trace Cache: A Low Latency Approach to High Bandwidth Instruction Fetching. In *Proceedings of the 29th Annual International Symposium on Microarchitecture*, pages 24–34, 1996.
- [6] Daniel J. Scales, Kourosh Gharachorloo, and Chandramohan A. Thekkath. Shasta: A Low Overhead, Software-Only Approach for Supporting Fine-Grain Shared Memory. In *Seventh International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 174–185, October 1996.
- [7] Emin Gün Sirer, Robert Grimm, Arthur J. Gregory, and Brian N. Bershad. Design and Implementation of a Distributed Virtual Machine for Networked Computers. In *Proceedings of the 17th ACM Symposium on Operating Systems Principles*, pages 202–216, 1999.
- [8] Emmett Witchel and Mendel Rosenblum. Embra: Fast and Flexible Machine Simulation. In *Proceedings of ACM SIGMETRICS '96: Conference on Measurement and Modeling of Computer Systems*, 1996.