Software Caching using Dynamic Binary Rewriting for Embedded Devices

Chad M. Huneycutt, Joshua B. Fryman, Kenneth M. Mackenzie
College of Computing Georgia Institute of Technology
Atlanta, GA 30332
{ chadh, fryman, 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. We describe two software cache systems for instruction caching only using dynamic binary rewriting and present results for the performance of instruction caching in these systems. 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.

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.

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.

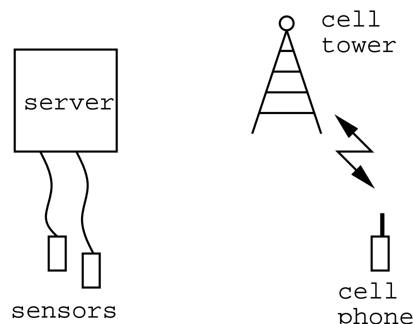

Figure 1. Two examples of networked embedded devices: distributed sensors and cell phones. Both include local processing but are continuously connected to more powerful servers.

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.

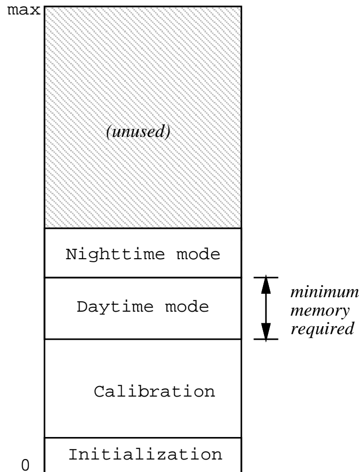

Figure 2. Example: code space for an embedded processor providing local processing to a sensor. The code includes modules for initialization, calibration and two modes of operation, but only one module is active at a given time. The device physical memory can be sized to fit one module.

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.

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

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. Section 2 describes an instruction cache implementation using dynamic binary rewriting in software. We present two prototypes that were developed and results. Section 3 describes a paper design for a data cache. Section 4 summarizes the range of tradeoffs in implementing a memory hierarchy in software as compared to hardware and discusses related work. Finally, section 5 concludes.

2 Software I-Cache

This section describes a software cache for instructions using binary rewriting and a client-server model. Similar to a hardware cache, a software cache consists of two controller interfaces around the local memories. A software cache controller (CC) sits on the client and handles hits to the remote translation cache, and the memory controller (MC) resides on the server and services misses.

Rewriting

Figure 3 illustrates the basic rewriting mechanism. On

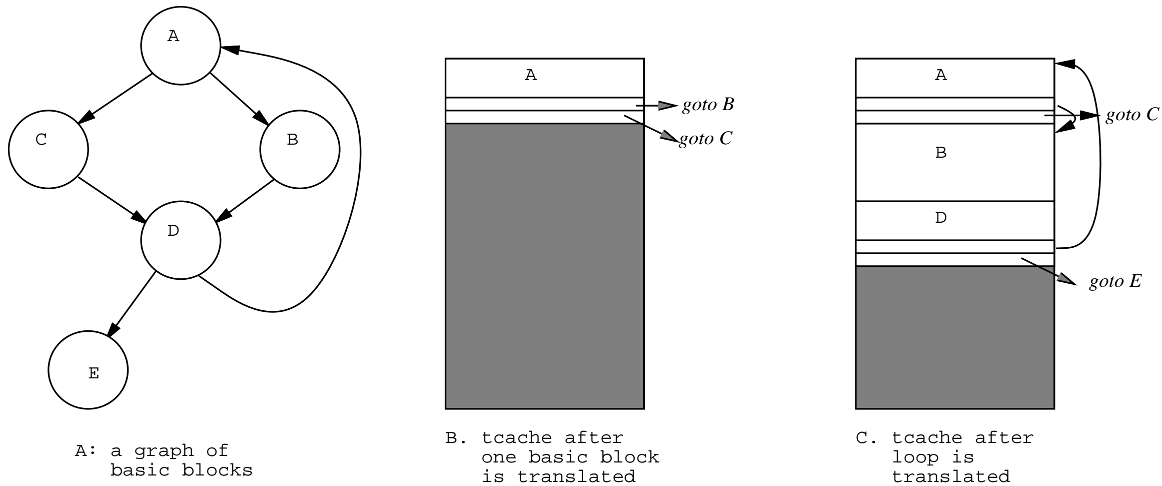

Figure 3. Example of Translation Cache (*tcache*) operation. Basic blocks in the original program (A), are copied on demand to the translation cache. As they are copied, branches exiting the blocks are initially rewritten to point to cache miss handlers (B) and eventually, if used, again rewritten to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

the MC instructions from the original program are broken into "chunks" (for our purposes, a chunk is a basic block, although it could certainly be a larger sequence of instructions, such as a trace or hyperblock). These chunks are then sent to the CC, which places them in the translation cache (*tcache*). At translation time, branches are rewritten to point to a cache miss handler. If a branch is subsequently taken, then the target of the original branch is translated, and the branch is rewritten again to point to the now 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, transferred, and rewritten in the cache, the loop runs at full speed on the client with no cache tag checks (but possibly with 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 removing the tag check entirely for the common case of branch instructions whose destinations are known at rewriting time.

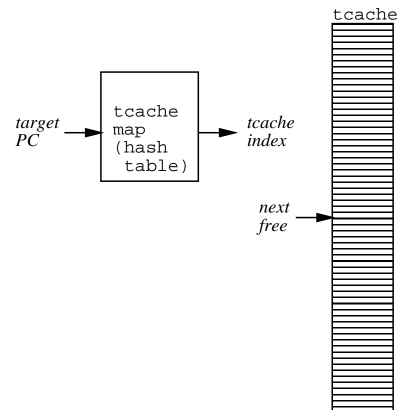

Figure 4. Data structures used for instruction caching. The *tcache* holds the rewritten instructions. The *tcache map* holds mappings from the original address space to indices in the *tcache*.

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

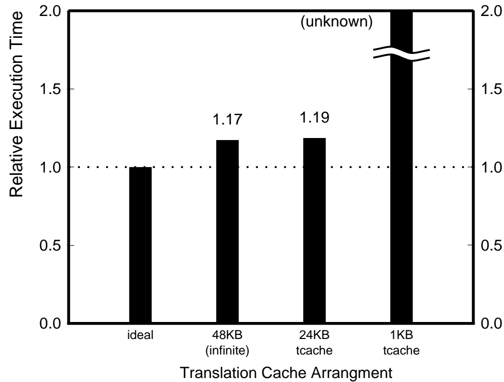

Figure 5. Relative execution time for the software instruction cache. The times are for `129.compress` from SPEC95 and are normalized to the "ideal" execution time with no software cache.

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 [11] or an ISA, such as the Java Virtual Machine, which is more amenable to analysis.

2.1 SPARC Prototype

We have implemented a prototype of a software instruction cache. 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 encompasses both the MC and CC. The MC copies and rewrites the embedded code into a *tcache* in memory and then executes that code directly. Communication between the CC and the MC is accomplished by jumping back and forth in places where a real embedded system would have to perform an RPC to the MC. Since only instruction caching is modeled, the rewritten 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.

App.	Dynamic `.text`	Static `.text`
`129.compress`	21KB	193KB
`adpcmenc`	1KB	139B
`hextobdd`	23KB	205KB
`mpeg2enc`	135KB	590KB

Table 1. Application dynamically- and statically-linked text segment sizes. `129.compress` **is from the SPEC CPU95 suite,** `adpcmenc` **is from Media-Bench, 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 SPARC I-Cache Results

We ran experiments using benchmarks that might appear in an embedded context. 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. In our current implementation, we add two new instructions per translated basic block. These extra instructions could be optimized away to provide a performance closer to that of the native binary.

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.

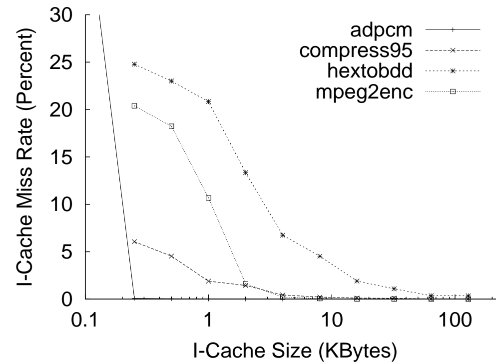

Figure 6. Hardware cache miss rate versus cache size. The cache is a direct-mapped L1 instruction cache with 16-byte blocks. The cache size is the size of data only – tags for 32-bit addresses would add an extra 11-18%.

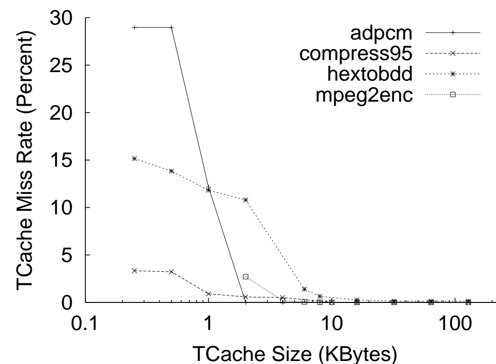

Figure 7. Software cache miss rate versus cache size. The software cache size is the size of the *tcache* in bytes. The software miss rate is the number of basic blocks translated divided by the number of instructions executed.

2.3 ARM Prototype

We have also begun implementation of a softcache for a real embedded system. It operates as follows: when the MC is run, it is provided with an executable file image (the application written for the remote target) and breaks it into small pieces of code and data which can each be sent to the CC as needed. The CC will execute these "chunks" of code, generating exceptions as it tries to use non-resident code targets (*e.g.,* , procedure calls to code which has not been received) or data targets (variables not resident). While the cost of transfer for these chunks between the MC and CC will depend on the interconnect system, as long as the remote node contains enough on-chip RAM to hold the "hot code" and associated data, it will eventually reach steady state in local CC memory. For more details of our implementation,

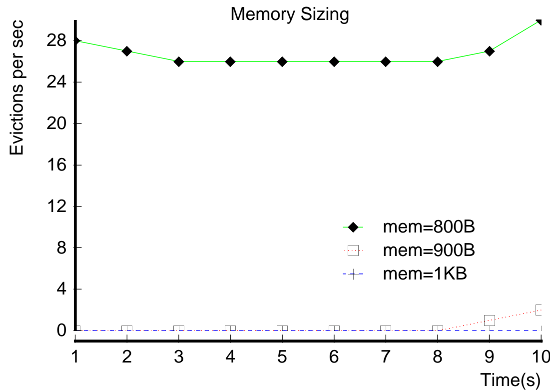

Figure 8. When memory is insufficient to hold the steady state of the application, paging occurs. This can be seen in the topmost line corresponding to 800 bytes of CC memory. When memory is increased to 900 bytes in the CC, the paging falls to zero during steady state, but at the end minor paging occurs to load the terminal statistics routines. This is seen in the light, dotted line on the graph. When the CC memory exceeds the steady state needs, the paging falls off even further as shown in the bottom line for 1024 bytes of CC memory.

see [5].

The MC and CC were written for the Compaq Research Laboratories' Skiff boards. These units contain a 200MHz Intel SA-110, 32MB of RAM, and run the Linux kernel with patches (kernel 2.4.0-test1-ac7-rmk1-crl2). In a real embedded device, the ARM is a popular choice and the Skiff board is well-equipped for testing a variety of embedded programs. For rapid prototyping and debugging, we use the resources of the Skiff boards and Linux kernels. The Skiff boards remotely mount filesystems via NFS and have a 10Mbps network connectivity.

Using cross-hosted gcc 2.95.2 compilers, all development work was done on x86 workstations but run on the Skiff boards. One Skiff board was set up as MC and the other as CC. Debugging was by native ARM gdb running directly on the Skiff boards. For communication between MC and CC we used TCP/IP sockets with standard operations over a 10Mbps network, where all network traffic is accounted for by either the MC, CC, or x86 host telnet sessions to the target Skiff boards. The MC was given a gcc-generated ELF format binary image for input.

The ARM prototype implementation succeeds at splitting the softcache functionality into MC and CC components but is limited in the ways listed below compared to the SPARC prototype from the previous section. These are not fundamental limitations, merely the current state of the

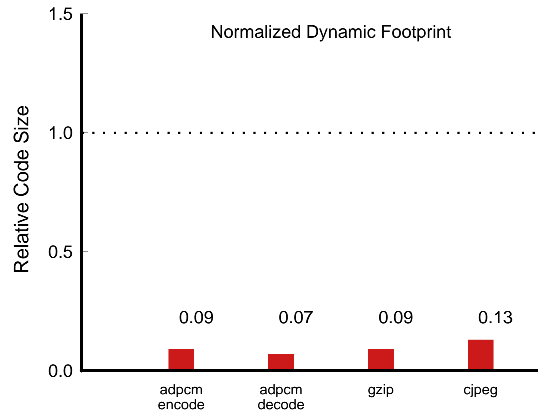

Figure 9. The dynamic footprints above have been normalized so that the static footprint of each benchmark is 1. As can be seen, the hot code is a 7-14X reduction compared to the full program size.

implementation given its different development focus:

- Code is chunked by procedures rather than by basic blocks.
- Procedure call sites use a "redirector" stub as a permanent landing pad for procedure returns to avoid having to walk the ARM's stack at invalidation time.
- Indirect jumps are not supported.

2.4 ARM Results

Using our implementation we were able to determine the network overhead for each code chunk downloaded to be 60 application bytes (not counting Ethernet framing overhead and intermediate protocol overhead) exchanged between CC and MC. This has future implications for the minimum code chunk size we will want to consider. The actual time the MC spends looking up data in tables and preparing the code chunk to be sent will vary with MC host, and could easily be reduced to near zero by more powerful MC systems.

Space Reduction

We then used a combination of static analysis and dynamic execution to find the minimum amount of memory required to maintain steady state in the CC system, which we expected to be a small subset of the entire application code size. Our benchmark applications included gzip, and pieces of MediaBench [8] including ADPCM encode, ADPCM decode, and cjpeg. As expected, we were able to verify the hot code as having a much smaller footprint than the primary application, although we do note that our choice

of benchmarks are universally compression-based. We feel that the primary task of the remote devices in the hierarchical context will be to reduce the data set generated and send only reduced amounts to higher systems.

The hot code was initially identified by using gprof to determine which functions constituted at least 90% of the application run time. We then set the CC cache space to be equal to the size of these functions. We observed that varying this size down caused the SoftCache to page more in steady state code, while varying the size up did not increase performance during steady state. Figure 8 shows how the page rate is an indicator to proper sizing of the memory for the CC.

This confirmed that only the gprof identified functions need be resident in the CC cache in steady state. The results from our analysis are shown in Figure 9, which indicates the size of hot code relative to app size. The original code size is not statically linked, and therefore the overhead of libc, crt0, and similar routines are not accounted for. In the real limited hardware system without Linux underneath, libc and similar routines would be considered part of the application image to be cached, and the effective "hot" sizes would be much smaller.

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 use rewriting to specialize accesses

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

```
ld [%r2], %r1 -->           ld [guess(ldaddr)], %r1
                            ld [tagbase + %r1], %r3
                            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
```
the variable address is not known but we have a prediction about which line in cache

Figure 10. 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).

with constant addresses (scalar global variables).

Additional tag check reductions are possible with additional effort or limitations. For instance, the HotPages system [11] 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 10 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. We briefly present issues raised by software caching and novel capabilities enabled by software caching followed by a discussion of 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 [12] 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.

Novel Capabilities

The software caching system presented here provides a framework for exploring dynamic optimization of memory. To illustrate the power of this approach, here are three novel capabilities of a software cache using rewriting: (1) we could dynamically deduce the working set and shut down unneeded memory banks to reduce power consumption, (2) we can integrate "cache" with fixed local memory at arbitrary boundaries and (3) we could direct memory accesses to multiple parallel memory banks to improve performance.

Since the software cache is fully associative, we can size or resize it arbitrarily in order to shut down portions of memory. In low-power StrongARM devices, the total power in use by the components of the chip we wish to remove are: I-cache 27%, D-cache 16%, Write Buffer 2%. This shows that 45% of the total power consumption lies in the cache alone, before considering other components [10]. By converting the on-chip cache data space to multi-bank SRAM, we can find an optimization for power based on memory footprint. By isolating each piece of code together with its associated variables, it becomes possible to power-down all banks not relevant to the currently executing application subset. By adding architectural support of "power-down" and "power-up" instructions, we can dynamically choose which banks to power at any given moment, leading to a significant potential power savings. Other efforts at enabling partial sleep-mode in on-chip RAM [1, 9, 16] focus on getting the steady-state of the program into just part of the total cache space. We suggest using the entire space and actively selecting which region is currently being powered. This depends on reliable SRAM cells that can be put in sleep mode without data loss [16].

Another facility that the SoftCache enables is a more flexible version of data pinning. With the SoftCache using arbitrarily sized blocks of memory as a "cache-line" model,

we can pin or fix pages in memory and prevent their eviction without wasting space. Additionally, we do not lose space by converting part of our cache to pinnable RAM – rather, our entire RAM space can be pinnable or not on arbitrary page sizes. We can also compact regions of pinned pages to be contiguous rather than disjoint, to allow for more efficient memory usage. Whether pinning code (interrupt handlers) or data (cycling buffers), this flexibility allows us to further optimize the memory footprint and usage characteristics of an application in a dynamic manner.

Third, given multiple banks of on-chip memory, software caching can be used to execute multiple load/store operations in parallel. By knowing the dynamic behavior of the system, we can rearrange during runtime where data is located to optimize accesses to different banks and increase overall memory utilization. Additional footprint improvements can be made by finding just the relevant data regions of structures that will be accessed and providing only these items in the CC's local memory.

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 [11]. 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 [13]. Such techniques could be used to implement shared memory between an embedded system and its server.

Dynamic binary rewriting systems have been developed to re-optimize code for improved program performance [7, 2]. In these systems the optimized code is stored in an area that is managed as an instruction cache. For instance, Dynamo [2] identifies hot traces through interpretation, optimizes them and stores them as "fragments" in a fragment cache. We differ from these dynamic optimizers in that caching is our object rather than code optimizations: our goal is to extract the best performance given limited space. Further, we enable separating the rewriting (in the MC) from the cache control (CC) for embedded scenarios.

Fast simulators have implemented forms of software caching using dynamic rewriting. The Talisman-2 [3], Shade [4] and Embra [15] 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 [6]. We propose to take an additional step.

A Distributed JVM [14] 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. A 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. We describe two implementations, one on SPARC with caching on basic-block boundaries, support for indirect jumps and no overhead for procedure calls, the other on ARM that separates the system into memory- and cache-controller pieces. Both systems show that the software cache succeeds at finding the active working set of the test applications.

Acknowledgements

This material is based upon work supported by the National Science Foundation under Grant No. 9876180.

References

[1] D. Albonesi. Selective Cache Ways: On-Demand Cache Resource Allocation. In *Journal of Instruction Level Parallelism*, pages 248–261, 2000.

[2] V. Bala, E. Duesterwakd, and S. Banerjia. Dynamo: A Transparent Dynamic Optimization System. In *PLDI*, 2000.

[3] R. Bedichek. Talisman-2 — A Fugu System Simulator. `http://bedichek.org/robert/talisman2/`, August 1999.

[4] R. F. Cmelik and D. Keppel. Shade: A Fast Instruction-Set Simulator for Execution Profiling. CSE 93-06-06, University of Washington, 1993.

[5] J. B. Fryman, C. M. Huneycutt, and K. M. Mackenzie. Investigating a SoftCache via Dynamic Rewriting. In *4th Workshop on Feedback-Directed and Dynamic Optimization*, December 2001.

[6] E. G. Hallnor and S. K. Reinhardt. A Fully Associative Software-Managed Cache Design. In *Proceedings of the 27th Annual International Symposium on Computer Architecture*, 2000.

[7] R. J. Hookway and M. A. Herdeg. DIGITAL FX!32: Combining Emulation and Binary Translation. *Digital Technical Journal*, 9(1), 1997.

[8] C. Lee, M. Potkonjak, and W. H. Mangione-Smith. MediaBench: A Tool for Evaluating and Synthesizing Multimedia and Communications Systems. In *MICRO-30*, 1997.

[9] M. Margala. Low-Power SRAM Circuit Design. In *Proceedings of IEEE International Workshop on MemoryTechnology, Design and Testing*, pages 115–122, 1999.

[10] J. Montanaro, R. T. Witek, K. Anne, A. J. Black, E. M. Cooper, D. W. Dobberpuhl, P. M. Donahue, J. Eno, G. W. Hoeppner, D. Kruckmyer, T. H. Lee, P. C. M. Lin, L. Madden, D. Murray, M. H. Pearce, S. Santhanam, K. J. Synder, R. Stephany, and S. C. Thierauf. A 160-MHz, 32-b, 0.5-W CMOS RISC Microprocessor. In *IEEE Journal of Solid-State Circuits, Vol. 31, No. 11*, November 1996.

[11] C. A. Moritz, M. Frank, W. Lee, and S. Amarasinghe. Hot Pages: Software Caching for Raw Microprocessors. MIT-LCS-TM 599, Massachusetts Institute of Technology, 1999.

[12] 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.

[13] D. J. Scales, K. Gharachorloo, and C. 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.

[14] E. G. Sirer, R. Grimm, A. J. Gregory, and B. 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.

[15] E. Witchel and M. Rosenblum. Embra: Fast and Flexible Machine Simulation. In *Proceedings of ACM SIGMETRICS '96: Conference on Measurement and Modeling of Computer Systems*, 1996.

[16] H. Zhou, M. Toburen, E. Rotenberg, and T. Conte. Adaptive Mode Control: A Static-Power-Efficient Cache Design. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, 2001.