Investigating a SoftCache via Dynamic Rewriting

Joshua B. Fryman, Chad M. Huneycutt, Kenneth M. Mackenzie
College of Computing
Georgia Institute of Technology
Atlanta, GA 30332-0280
{ fryman, chadh, kenmac }@cc.gatech.edu

Abstract

Software caching via binary rewriting enables networked embedded devices to have the benefits of a memory hierarchy without the hardware costs. A software cache replaces the hardware cache/MMU mechanisms of the embedded system with software management of on-chip RAM using a network server as the backing store. The bulk of the software complexity is placed on the server so that the embedded system contains only the application's current working set and a small runtime system invoked on cache misses. We present a design and implementation of instruction caching using an ARM-based embedded system and a separate server and detail the issues discovered. We show that the software cache succeeds at discovering the small working set of several test applications for a reduction of 7 to 14X of the original application code. Further, we show that our software overhead remains small for typical functions in embedded applications. Finally, we discuss the implications of software caching for dynamic optimization, for power savings and for hardware design.

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

1 Motivation

Ubiquitous computing environments imply the adaptation and usage of computational hierarchies with varying interconnect links. These hierarchies will have at the highest level large compute capacity devices, such as clusters or MPP systems, while at the lowest level will be a wide distribution of small devices with different characteristics. These leaf nodes of the hierarchy as suggested by Figure 1 may be anything from audio pickups woven into the fabric of wallpaper to larger objects such as tilt-pan-zoom cameras. Regardless of the type of sensory data the node interacts with, they will all have some type of CPU embedded in their apparatus as well as some type of link back into the compute environment. The communications link may vary in bandwidth and connection mechanism based on the type of the device.

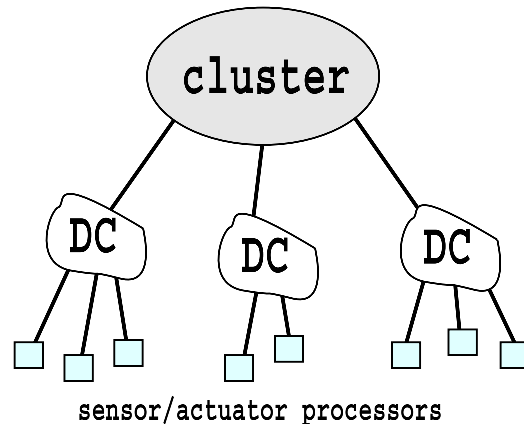

Figure 1: **A hierarchical model, with multiple leaf nodes connected to some type of data concentrator or more powerful backend cluster.**

Proliferation of these small devices in mass quantity motivates finding ways to reduce the cost and power requirements for each leaf node type. Such nodes may be powered by body heat [2] or low-grade solar cells; may have just the computational ability required for their designated task; and may be targeted for manufacture at a cost point of pennies. To advance reaching these goals we propose removing all extraneous and consuming resources such as cache, MMU, and TLB components from such embedded devices , while adding an on-chip SRAM. No external memories or storage would exist outside of this minimal single chip solution.

The drawback of this model lies in programmer perspective of the remote node. The application programmer does not want to be burdened with worrying about memory or other particulars of a limited capacity device. We therefore propose that application programmers write their code as though the remote node is a full workstation (or model equivalent) with effectively infinite resources for algorithm design and implementation. An important consideration here is that the application directed to the leaf node will gen-

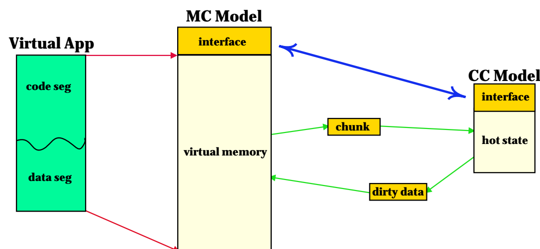

Figure 2: **The conceptual interaction model for the SoftCache. The virtual binary is loaded by the MC into memory; the MC sends pieces to the CC as needed. The CC sends dirty data back to the MC on evictions.**

erally contain some set of primary operating modes (e.g., initialize, calibrate, morning, afternoon, evening) and will spend a significant amount of execution time in each primary mode of operation.

In order to run this virtual application on the target node, we have developed a system, which we call a SoftCache, to solve this task in a manner transparent to the programmer. The SoftCache consists of two components: a server, which we dub the Memory Controller (MC); and the client, which we dub the Cache Controller (CC). The MC runs on a non-leaf node, communicating to the remote CC which is running on an assigned leaf node. 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., function 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. Each primary mode of operation will contain some subset of code and data which constitutes the essence for that mode. Our objective is to put just this essence into the leaf node, freeing it from the burden of unused memory and wasted NVM storage. Figure 2 illustrates the basic model of the SoftCache. The CC can be seen as containing the local subset of code and data that a cache would supply to the CPU, but the backing store is located far away with interaction occurring through software control, forming our software cache system *SoftCache*.

Two interesting examples of where the SoftCache might be applied come from current technology situations. Consider a hypothetical urban traffic camera, monitoring Interstate highways or a particular traffic intersection: this has two primary modes of operation – daytime and nighttime. On average the device will spend 12 hours in daytime mode and 12 hours in nighttime mode. If it spends 5 minutes moving from the daytime to the nighttime then a minor execution time tradeoff has been made against simplified hardware. A link back into the hierarchy here may be as slow as 1200bps. Another example lies in 3G phones coming into use today. These phones have more functionality than "phone" mode (web browser, video on demand, etc.) with current carriers targeting data rates of 144Kbps or higher [5]. By reducing the complexity of the phone and total part count, the consumer price may drop significantly with a tradeoff of scant seconds delay changing steady state modes.

The remainder of this document is organized as follows. Section 2 describes the system design and implementation, Section 3 presents results, Section 4 describes related work and Section 5 discusses the implications of caching in software. Finally, Section 6 concludes.

2 Design and Implementation

In this section we will address three different topics: conceptual design of the MC system, conceptual design of the CC system, and the actual implementation performed to date.

2.1 MC Design

Identified as static parse time or dynamic run time tasks, the tasks of the MC are:

- break code into "chunks" (static)
- replace function call sites with a call to CC interceptor (static)
- data dependency annotation (static & dynamic)
- non-ambiguous data address fixups (dynamic)
- ambiguous data instrumentation to CC interceptor (dynamic)
- book keeping on state of CC and MC images (dynamic)

The MC will break the code up into variable-sized chunks, depending on selection criteria. Regardless of chunk size, the MC must append the appropriate instructions (if necessary) to move from one chunk to another. The primary "trick" of the MC operation in conjunction with the CC is to replace all function call sites with a call site to a CC-resident interceptor. This interceptor will be entered where the original program would have made the function call. When triggered, the CC interceptor requests from the MC the missing function, and eventually "patches" the application call site. The new target of the call site is the real function, retargeted to account for the new address of the

function, This rewriting removes the penalty of calling the CC interceptor.

Once each chunk is extracted, it must be further annotated with what data variables are referenced from it so that when this code chunk is downloaded, the necessary data values are transferred with it. This annotation is just a data dependence graph. As the data are transferred, all referring instructions must have their addresses fixed to match the location where data will reside in the CC. Conversely, when this code chunk is evicted, we need to recover any modifications to these data variables and therefore need to request the CC pass these variables back to the MC (dirty data).

The MC must also consider the special cases of pointers – either function or data. Data dependence analysis may disambiguate the target involved in the pointer operation, in which case the pointer operation can be replaced with inlined code to achieve the same result. The alternative case is when the pointer is never resolved and must be considered ambiguous. In these cases, a special interceptor call in the CC must be used just to handle these instructions; the MC simply replaces the instruction that would assign or dereference the pointer with a call to the proper pointer interceptor. The unfortunate effects here are twofold: this interceptor can never be patched out like the call site interceptor, and this interceptor must do more work by actually calculating the addresses involved and simulating the instruction replaced.

When invalidating regions in memory, the MC must also provide the appropriate replacement patches for any code chunk that had calls into or out of each invalidated chunk. It can achieve this by replacing the invalid call site with a call to the CC call interceptor, at which point the whole process starts over. For proper correction inside nested function calls, the MC can exploit the regularity of the function frame calling convention to "fix" return pointers such that instead of returning to a function, they will point to another special interceptor which will in turn reload the appropriate code chunk. Similar consideration must be made when evicting items from the data cache.

2.2 CC Design

The CC is conceptually a simple piece of software that interfaces in a hidden layer between the running application and the MC. The essential components of the CC are:

- interceptors (call site, ambiguous data, stack return, IRQ)
- local system call interfaces
- communications system for talking to MC

On startup, the CC will pass control to the target application's local `main()` routine, and it only gets control back when an interceptor is entered. At this point, the interceptor follows the appropriate algorithm and resumes execution of the target application. For situations that generate exceptions which the CC is incapable of dealing with, the MC provides assistance.

Application system calls can be handled by implementing a very basic subset on the remote node and then considering anything more complex to be part of the target application, cached appropriately. Bottom level system calls can either be hooks into equivalents on the client, or messages that are passed to the MC for processing. Two conditions are placed on interrupt-based code: the first requires that all interrupt vectors be assigned to CC vector wrappers that will load the appropriate IRQ handling code if necessary. The second requires that all IRQs be of less priority than the MC/CC communications system.

2.3 Real Implementation

An implementation of the MC and CC were done on 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 equipped for testing 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 the x86 host telnet sessions to the target Skiff boards. The MC was given a gcc-generated ELF format binary image for input.

The implementation does not presently support data caching; we only cache instructions. Therefore, all data variables and data pointers are handled without penalty. At boot-time the MC loads the entire data segment to the CC, and also specifies the size of the bss segment which the CC zeros appropriately. We also do not permit function pointers in this first implementation on real target hardware. We found several limitations of the ARM architecture which hampered the following conceptual designs for the MC and CC. It it important to note that while we discovered these problems on the ARM platform, other embedded targets may have some or all of these issues as well.

The first challenge lies in the method of accessing data variables. Intermixed within the instruction stream are the

```
#include <stdio.h>
int main( void )
{
    printf("Hello World!\n");
    return 42;
}
```

```
section  .rodata
         .align   2
LC0:
         .ascii  "Hello World!\012\000"
text
         .align   2
         .global  main
         .type    main, function
main:
         stmfd    sp!, {lr}
         ldr      r0, .L3
         bl       printf
         mov      r0, #42
         b        .L2
         .align   2
L3:
         .word    .LC0
L2:
         ldmfd    sp!,{pc}
```

Listing 1: **A very simple C function and the ARM assembly output from `gcc`. To load register `r0` with the string reference, it actually loads a constant embedded in the text segment which the instruction stream branches around.**

addresses of data or bss segment variables or constants. An example of this is shown in Listing 1. To work around this problem, we made a code chunk constitute an entire function and any associated data constants with it in our initial implementation. To subdivide a function requires the instruction analysis to propagate and carry these constants with each chunk. We speculate that optimum code chunk size will be either a basic block or a collection of basic blocks, but determining this will require further work.

The second challenge comes from a complete lack of consistency between instruction and data caches, as well as the write-back buffer. Therefore, any time we need to modify memory we must flush the caches and buffers with software routines. This results in a substantial penalty every time we must modify the CC status. Some Intel SA devices do have non-ARM instructions for flushing the cache, and we are investigating using this mechanism. Ultimately this issue will be irrelevant as we will not have caches present in our final hardware.

The third challenge comes from the lack of a usable function frame to "walk" when invalidating code regions. This lack can come as a side effect of optimized application code. This forces us to not patch a call site from the interceptor to a real function address, but rather to a "redirector" which will handle the call site and manage enough state information that we can return to where we came from, even if we have invalidated the code we need to return to. By using the saved state information, we are able to re-fetch the missing

```
int s3( int x ) {
    int i;

    for (i = 0; i < IN_COUNT; i++)
        ;
    return x + s3(x-1);
}
void main( void )
{
    for (i = 0; i < LOOPS; i++)
    {
        for (j = 0; j < OUT_COUNT; j++)
            ;
        s3( 500 );
    }
}
```

Listing 2: **Source code sample for varying the the effective workload of cycles per function call. As the cycles per function call decrease, the expected penalty cost of the redirector system increases.**

code before returning from a function. This redirector consists of 56 assembly instructions in the best-case path, and constitutes a substantial penalty on function calls. When we implement the instruction and dataflow analysis required for data caching, we can also track the stack usage in each chunk and back-trace in the stack to locate return-target addresses and redirect them to reload invalidated code. This optimization will have no penalty in steady state.

Prior to presenting our tests and results, it is relevant we state the following issues with our implementation. We did not optimize our code, except for the hand-writing of the redirector system in assembly. We are aware that we generate extra network traffic and the messages are not packed or compressed in any way. It was our concern after discovering the implementation issues, and using our unoptimized code as a baseline, that we investigate whether the ARM is a suitable architecture for continuing our work on the platform. All the local data tables on the CC are implemented as hash lookups to arrays, with the MC consisting of a mixture of linked lists and regular unordered arrays. Interrupt-based code is not allowed in this version of the MC/CC system. For the tests that follow, the MC was run on a separate Skiff board of exactly the same features as the CC. The CC took advantage of the underlying Linux OS and passed application system calls directly to the CC host platform for handling.

3 Tests and Results

Given the initial problems, we were concerned that the ARM might be an unsuitable target. We have therefore performed a performance analysis to see how the aforementioned penalties translate into real application run-time penalties.

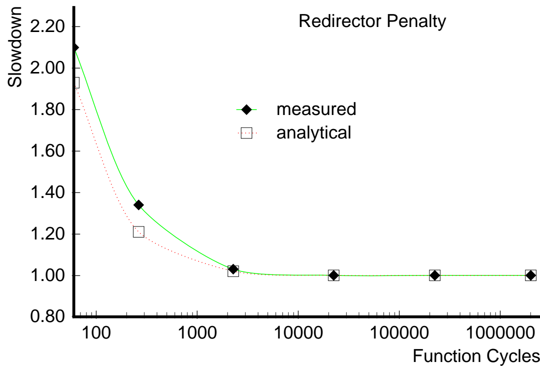

Figure 3: **As the function does more work the overhead of the redirector falls to zero. The 10% slowdown roughly corresponds to a 600-cycle average function size on this platform.**

First Experiment: Time Overhead

Our first concern is the best-case 56 instruction penalty on every function call. To evaluate this concern, we constructed a synthetic benchmark consisting of a deeply recursive function containing a variably-sized `for` loop. Changing the limit of the `for` loop allows us to control the amount of effective calculation inside each function invocation, as shown in Listing 2.

We supplied the CC with effectively infinite resources (enough RAM to hold all functions) so that the interaction with the MC would not affect the results. We then ran the benchmark for several runs, where in each run we changed the computation inside the recursive calls to vary the number of effective compute cycles per function call. Running each of these trials for several seconds shows the penalty incurred in cache-hit code with no misses, essentially eliminating the error from initial page-in time from the MC.

Figure 3 shows how the number of cycles per function call can be directly translated to slowdown of execution due to redirector penalty on the ARM. This is slowdown in relation to the real application running on the real hardware (a slowdown of 1.0 shows the SoftCache running as fast as the real binary).

While functions could be created as small as one instruction, we limited the minimum size to be 25 instruction cycles as the minimum for "useful" work computation. Within Figure 3 we present the analytical penalty we expect to pay in a perfect system as the broken line. This is in comparison to the measured penalty we pay in the solid line. We account for the error in that the analytical model does not account for variable OS interaction, imperfect resource utilization, instructions that take more than one cycle, and similar conditions.

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.

Second Experiment: 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 [10] 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 4 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 5, 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.

4 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 Hot Pages system uses transformations tightly integrated with a compiler that features sophisticated pointer analysis [12]. In contrast, we focus on the complementary problem of what we can do

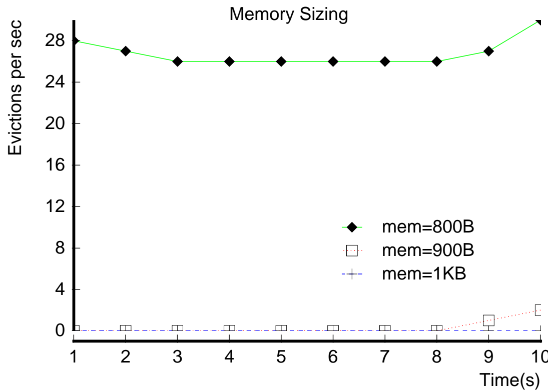

Figure 4: **When memory is insufficient to hold the steady state of the application, paging occurs. This can be seen in the topmost line corresponding to 800 bytes of CC memory. When memory is increased to 900 bytes in the CC, the paging falls to zero during steady state, but at the end minor paging 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.**

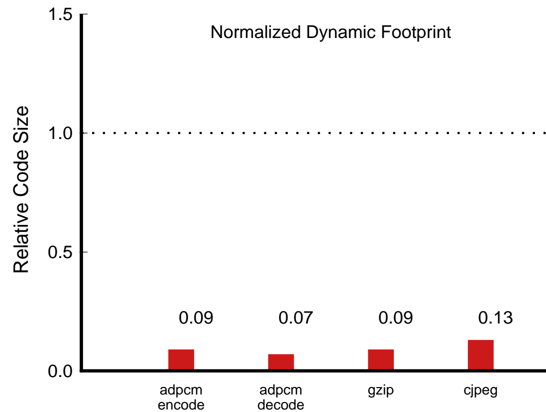

Figure 5: **The dynamic footprints above have been normalized so that the static footprint of each benchmark is 1. As can be seen, the hot code is a 7-14X reduction compared to the full program size.**

with dynamic rewriting 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.

Fast simulators have implemented forms of software caching using dynamic rewriting. The Talisman-2 [4], Shade [6], and Embra [16] simulators use this technique. Simulators must deal with additional simulation detail which limits their speed. Also, the dynamic rewriters deal with very large CC cache sizes and avoid the problem of invalidating entries by invalidating the CC cache frequently and/or in entirety. 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 full-associative L2 cache [8]. 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.

The idea of optimizing memory footprint has been presented in the context of dead code and data elimination [7, 15]. Repeated analysis after removal of dead code leads to dead data elimination, which in turn leads to more dead code elimination, and so forth. This type of compile- and link-time analysis can reduce application size in a static manner. The SoftCache can go further and use dynamic feedback to reduce application size beyond what static analysis can resolve.

In previous work on our project [9] an entirely software-based system was developed for the SPARC architecture. This system did not use a client-server interaction model, instead jumping through code between those routines which would lie on the client and server. This system performed instruction caching but not data caching, and placed restrictions on the instructions that could be cached. In this work, we presented a new implementation of the system with separation between client and server on separate devices, with less restrictions.

5 Discussion

The SoftCache system presented here provides a framework or vessel for exploring options in dynamic feedback situations. These options include the typical scenarios of critical-path code optimization, speculation, and profiling. We believe that the SoftCache provides truly novel opportunities for looking at new situations and new types of optimizations, which we will present in brief here. We believe that this type of enabling technology will lead to revolutionary ways of considering optimizations and feedback.

In a very basic sense, the SoftCache is dynamically optimizing the memory footprint of an application. Not only are we reducing the code that is resident, we also minimize the data that is resident. This provides new opportunities for memory interface issues. If we construct our on-chip memory to be multiple banks, we can 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 speed 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 memory.

This concept of multi-bank memory has other interesting benefits as well. 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 [3]. 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 piece. 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, 11, 17] 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 [17].

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.

The SoftCache system also enables new ideas of protection models to be implemented. With dynamic rewriting, we can insert protection checks around memory accesses to "sandbox" applications such that a buggy system will not disable a non-buggy platform base. Other features could be to change the protection model of calls such that no overhead of an operating system or interrupt sequence need be required.

In many ways, the SoftCache system is replicating some of the work done by advanced compilers that perform data analysis and code optimization. A key question lies in whether the SoftCache system would be better targeted to an object-level system as it is now, or if it would perform better as a source- level system that uses a compiler for a front-end. This is an argument of trade-offs that we are unable to answer at this time. The usage of a compiler front-end to perform dataflow analysis and call graph calculations would be helpful to our system so that we do not have to replicate this body of work. There are drawbacks to requiring source trees and multiple instances of compilers running simultaneously, however. The ideal situation would be for a compiler to generate an intermediate representation (IR) and annotate that IR with the data flow and control flow information. The SoftCache could then read this IR with annotations, and render appropriate target output code. This remains an area for further investigation.

We also observe that certain architectural limitations of the ARM (and many computing devices in general) aggravate the behaviors encountered in implementing the SoftCache. Foremost is a lack of large immediate-data instructions to load addresses, values, and so forth. The workaround as discussed in this paper involves intermingling of data constants with the instruction stream. This has unfortunate side effects in complicating program analysis. (Even 'gdb' and 'objdump' fail to distinguish between code and data constants.)

Another limitation of some platforms is a lack of a long branch or branch-and-link type of instruction. The ARM supports a large address range of 24-bit signed offset, but many platforms support much less. The availability of the PC as a user-programmable register also obscures the possible control flow path of a program. While such a design has some advantages for flexible programming, it can also make analysis more difficult. Finally, the lack of a strictly user-mode software interrupt mechanism hampers the ease in implementing systems such as the SoftCache. Such a mechanism would trigger an interrupt handler provided in the user code when activated, without causing flushes or a context switch or operating-system intervention. Such a mechanism would be quite useful for catching exception conditions in programs, without paying a large penalty in execution performance.

It is also worth noting that the SoftCache system can provide deterministic behavior for applications with realtime requirements. Given that the on- chip memory is sufficient for holding the steady state code and data, the lack of hardware cache interactions guarantees that the system will run at a constant rate. No hidden effects of multi-level caches will be observed.

6 Conclusion and Future Work

Software caching via binary rewriting enables networked embedded devices to have the benefits of a memory hierarchy without the hardware costs. The bulk of the software

complexity is placed on the server so that the embedded system contains only the application's current working set and a small runtime system invoked on cache misses. We presented a design and implementation of instruction caching using an ARM-based embedded system. We showed that the software cache succeeds at discovering the small working set of several test applications for a 7 to 14X reduction over the original application code. Further, we show that our software overhead remains small for typical functions in embedded applications.

Our future goals are to experiment with sub-function sized chunks of code to find optimal chunk size. In the case that we observe the optimal size to be application-dependent, we can pursue dynamic feedback integrated into the MC and CC systems, or require the programmer to specify size to the MC at load time. After these goals are reached, data caching will be implemented. We are investigating putting an ARM core in a FPGA target, possibly adding instructions to facilitate solutions for some problems. This would enable observation of power reduction after removal of unwanted sections and insertion of a small memory. We also speculate in this work about the many novel views of optimization that are made possible by our system. When all components are operational, we will pursue solving the questions raised in power, footprint, pinning, and related topics.

Acknowledgements

We would like to thank the anonymous reviewers for their constructive feedback and suggestions. Their suggestions have been incorporated with other changes to make this work clearer and with a stronger emphasis.

This work has been funded in part by an NSF CAREER award to Kenneth Mackenzie and in part by a grant from the Georgia Tech Broadband Institute (GTBI).

References

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

[2] Applied Digital Solutions, Inc. A Miniaturized Thermoelectric Generator Powered By Body Heat. `http://biz.yahoo.com/bw/011001/10436_2.html`, October 2001.

[3] K. Asanovic. Vector Microprocessors. In *PhD Thesis, UC-Berkeley, Department of Electrical and Computer Engineering*, 1998.

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

[5] J. Blecher. Cell Phone Carrier Technology Chart. CNet Wireless Watch (`http://www.cnet.com/wireless`), September 2001.

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

[7] S. Debray, W. Evans, R. Muth, and B. De Sutter. Compiler Techniques for Code Compaction. In *ACM Transactions on Programming Languages and Systems*, vol. 22 no. 2, pp. 378-415, March 2000.

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

[9] Chad Huneycutt and Ken Mackenzie. Software Caching using Dynamic Binary Rewriting for Embedded Devices. Technical Report GIT-CC-01-26, College of Computing, Georgia Institute of Technology, 2001.

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

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

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

[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, 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 *17th ACM Symposium on Operating Systems Principles*, pages 202–216, 1996.

[15] B. De Sutter, B. De Bus, K. De Bosschere, and S. Debray. Combining Global Code and Data Compaction. In *Proceedings of ACM SIGPLAN Workshop on Languages, Compilers, and Tools for Embedded Systems*, 2001.

[16] W. Witchel and M. Rosenblum. Embra: Fast and Flexible Machine Simulation. In *Conference on Measurement and Modeling of Computer Systems (SIGMETRICS)*, 1996.

[17] 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.