SoftCache: A Technique for Power and Area Reduction in Embedded Systems

Joshua B. Fryman, Hsien-Hsin S. Lee, Chad M. Huneycutt,
Naila F. Farooqui, Kenneth M. Mackenzie, David E. Schimmel
Center for Experimental Research in Computer Systems (CERCS)
Georgia Institute of Technology
Atlanta, GA 30332-0280
{ fryman, leehs, chadh, naila, kenmac, schimmel }@cercs.gatech.edu

Abstract

Explicitly software managed cache systems are postulated as a solution for power considerations in computing devices. The savings expected in a SoftCache lies in the removal of tag storage, associativity logic, comparators, and other hardware dedicated to memory hierarchies. The penalty lies in high cache-miss cost and additional instructions required to effect a cache model. In this paper, we characterize SoftCaches by placing them in the overall computing landscape, analyzing the energy and space trade-offs. We present results that indicate a SoftCache saves power and space over hardware caches. Based on the TSMC 0.25μm process from MOSIS, we use schematic and layout representations of hardware and SoftCache models for comparison. Accounting for additional instructions executed and simplification of logic, we examine high SoftCache miss cost in relation to the overall system. For a 256KB "mode" change every 1.45 hours, the SoftCache exhibits 1% application slowdown for energy savings of 30% or more in a low-power device such as the SA-110 microprocessor used in PocketPC platforms.

1 Introduction

Embedded consumer systems continuously add more features while shrinking their physical size. Current 3G cell phones incorporate 144kbps or better network links [10], offering customers not only phone services but also e-mail, web surfing, digital camera features, and video on demand. With feature creep demanding additional storage and memory in all computing devices – not just cell phones – densities of DRAM and Flash increase trying to keep pace. Unfortunately, energy consumption does not reduce at a similar rate. The need for larger feature sets and local storage works counter to longer lifetime in battery powered operations.

While energy usage and battery life are constraining embedded devices, other problems are appearing. Rather than fixing a feature set into a cell phone, designers and consumers want to change operational behavior dynamically. Switching active audio or video codecs, upgrading web browsers, and downloading new or customized applications are all becoming common needs.

To reduce energy consumption and increase battery life, designers use the smallest parts and lowest part count possible. This has the added benefit of keeping manufacturing cost down. This effort at minimizing available resources works against feature creep and the need for device flexibility for dynamic upgrades.

Other examples of ubiquitous computing environments in the wild can be seen in the ZebraNet project [25] and similar animal or habitat monitoring systems [28, 17], houses with computer awareness integrated [33], urban traffic or location monitoring cameras, or similar widespread devices [48].

Trying to address some of these problems, companies like NTT Japan are investing time and research effort in solutions that allow for Mobile Computing – dynamically migrating application code between the remote device and other network connected systems [23]. Research efforts in academia are investigating alternate designs [55].

In this paper, we use the SoftCache, an explicitly software managed cache- like storage, to address these problems. The SoftCache converts the on-chip cache structures to generic SRAM, removing the additional transistors used for MMUs and write buffers. The cache storage area and tag space is kept as addressable SRAM, and cache behavior is effected by execution of additional instructions.

The SoftCache provides a model where local non-volatile (NV) storage is reduced or removed entirely, and pushed back across the network to remote storage servers. Local DRAM is also reduced or eliminated. By using the already-present network link, it becomes possible to construct a distributed client-server model for computing that uses stripped down client hardware but provides features to the user as though no physical reduction had occurred via the computing capacity of remote servers in the network.

The net result when applied to instruction caching is an explicitly software managed system. This provides benefits such as full associativity and flexible resource utilization. A key argument in the SoftCache model has been that it also provides for a substantially smaller energy consumption, which is increasingly important.

The most frequent criticism of this technique, however, has been that the additional cost of utilizing the network link for moving code and data will far outweigh the benefit of removing or reducing local storage. Other arguments have been presented that the SoftCache will not result in significant processor energy consumption savings.

This paper addresses these concerns and arguments. Our results are at times counter intuitive, and the final analysis surprised even us in the determination of how energy efficient a SoftCache can be. In this paper, we demonstrate these main points:

1. Network link energy consumption is much less than local storage.
2. There is less overhead for a software cache than a hardware cache.
3. Measurable space reduction can be achieved in processor die.
4. SoftCaches consume at least 10-20% less energy over traditional designs.

2 The SoftCache

2.1 Operation

The SoftCache system advocates using the existing network infrastructure to effect a more distributed computing model. Rather than adding additional hardware to the mobile embedded platform, the SoftCache model removes most if not all local DRAM and NV storage space. Capitalizing on the network link, it assumes there is a substantially faster, more capable, larger storage capacity server or cluster reachable via the built-in link. To achieve their goals, the remote server performs dynamic application analysis and optimization, sending only the necessary pieces to the embedded platform to run locally. Given sufficient storage for the working set of the application, no other storage is necessary in the embedded device.

To avoid burdening the programmer with the limitations imposed on the mobile device, the SoftCache uses a dynamic binary rewriting system that takes as an input a *virtual* application that assumes no limits on resources. The remote server or cluster breaks this application into pieces, and then sends each piece as required to the mobile platform. By carefully keeping track of what pieces are resident

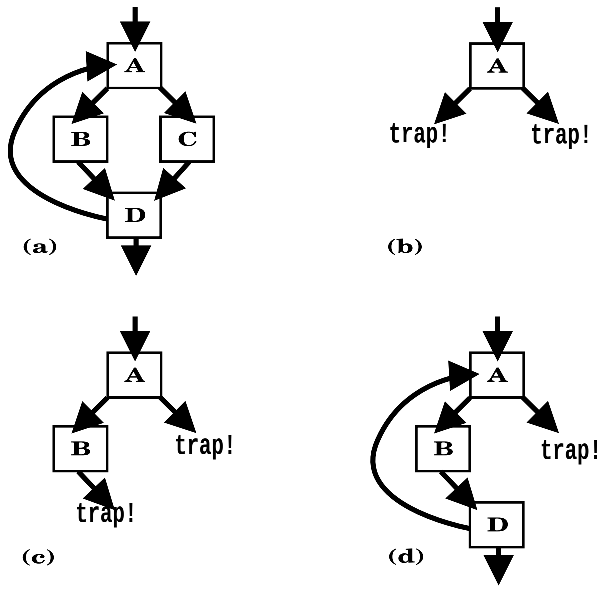

Figure 1: **Propagating new basic blocks. Part (a) represents the original program. Parts (b) through (d) show the gradual placement of basic blocks as the execution path is dynamically determined.**

in the mobile device, the server can rewrite the instructions and addresses of data to match the actual values needed for the current state in the limited platform. This process of remote server control, coupled with binary rewriting, can lead to eviction and placement policies that are much more sophisticated than traditional hardware models. It also provides substantial potential by using variable-size blocks to transfer to the client, as well as full associativity.

Conceptually, this can be envisioned as the client executing one basic block of code at a time. At the end of each basic block are two choices: taken or not-taken. Each choice leads to an exception event. Once the exception occurs, the client can request from the server the appropriate target path. As the server builds up a map of what blocks reside within the client's actively changing state, it can rewrite new blocks being sent. Rather than inserting exceptions for paths where the target is already resident, a branch is directly inserted as appropriate. When it becomes necessary to evict from the client storage, the server can also rewrite the branches of already resident blocks to either point to relocated objects, or to become exceptions again as target objects are removed entirely. The server also pre-emptively patches away already resident unknown target exceptions to local branches as new blocks are loaded into the client. This block-by-block determination is shown in Figure 1.

As can be expected, with this model running over time with sufficiently large client storage, the working-set of the application will become all-resident and no exceptions will

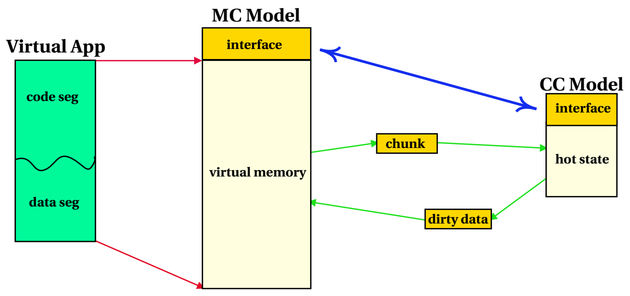

Figure 2: **Conceptual SoftCache interaction model.**

occur. During this time, the application is in steady state, no network communications are necessary, and a lower energy consumption can be observed – from simpler processor die logic due to removal of external parts such as DRAM and NV storage.

The complete SoftCache model of operation is depicted graphically in Figure 2. The SoftCache implements a server, which is dubbed the Memory Controller (MC), and the client, which is dubbed the Cache Controller (CC). (That these names parallel the typical hardware implementation of cache systems is intentional.) It is assumed that applications for such a system exhibit mode behavior, where a steady-state is reached in a subset of program code (such as an image processing core algorithm). Any application may have multiple steady-state regions within it. Based on studies characterizing the working set of applications, when the cache storage is of sufficient size, mode changes occur infrequently. The sizing of the cache storage is application dependent, but has been demonstrated to be on the order of tens of kilobytes for common embedded algorithms [1, 6, 19].

Thus SoftCache offers the benefit of hardware cache implementations (programmer perspective, application speed) without the penalties typical of hardware solutions (power, space). The drawback of such a cache lies in handling a miss scenario, where code or data must be fetched to (and possibly evicted from) the limited local storage. This handling must be done via software, which is considerably slower than custom hardware controllers. Complete details and algorithmic models are presented in prior publications [19, 22].

While the SoftCache addresses the problems briefly sketched here, it also poses new challenges in the overall system which are evaluated and addressed in the following sections.

2.2 Characterization

The SoftCache is offered as a replacement cache mechanism in the general classes of computing devices. Some effort has been spent to narrow the domain of these devices to only embedded networked systems, but a more complete understanding is attained when portraying it as fitting into *any* type of device.

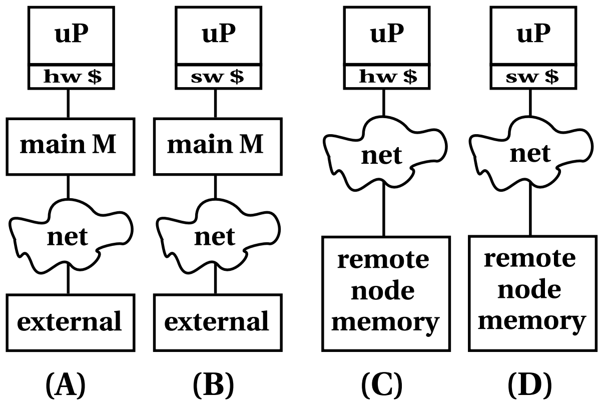

Figure 3: **The basic computing device models.**

Figure 3 depicts the basic types of computing device configurations. Models (A) and (C) correspond to typical scenarios with hardware caches. Models (B) and (D) are the obvious counterparts for SoftCache systems. When comparing the models, we recognize different types of problems.

Comparing models (A) and (B), consider these as workstation-class machines. We note that there is no apparent benefit to using the SoftCache method. While SoftCache techniques will reduce main CPU power consumption, the CPU is not the entire power consumption of the environment. These models represent a full computing station, with DRAM main memory, NV storage (disk, flash, or other), as well as possible I/O devices. What the SoftCache does bring is a long penalty to replace cache contents. The SoftCache is not a viable solution by itself when considering workstation-class machines.

Comparing models (C) and (D) indicates a possible SoftCache win. These models are envisioned as the mobile embedded computing nodes in an ubiquitous computing environment. Here, the entire processing unit can be placed by itself. So long as some type of interconnection link exists to other devices, it becomes possible to relocate large backing store facilities elsewhere. This provides an opportunity for a small, low-power device to represent the processor core and associated cache. The primary issues to resolve for this to be a "win" for SoftCaches focus on the penalties a SoftCache incurs that a traditional cache will not.

Aside from the issue of demonstrating energy reduction in the processor die, consideration must be given to the additional instructions SoftCache systems execute to emulate caches. These additional instructions require storage within the SoftCache itself, thereby reducing the space available to the target application. They also translate to an increase in energy needed for the SoftCache. Given these constraints, we demonstrate that the SoftCache can be a substantial energy reduction when compared to traditional designs.

A more interesting comparison is to reconsider model (A) as an embedded device (such as a cell phone) and compare this to model (D) implementing the same functions. This illuminates a key point of debate about the viability of a SoftCache – "is the removal of the main memory (and other storage such as Flash) across a network a power-saving design?" We demonstrate the answer is yes. This is counter-intuitive – accessing local memory should be less power consuming than accessing remote memory over a network. This implies that SoftCaching can be a win in more complex domains than simply limited feature embedded devices.

3 Experimental Results

We now systematically address each of the components in arguments raised against the SoftCache model by using a variety of techniques. Obtaining and analyzing existing low-power mobile DRAM data sheets for power consumption and comparing these to existing "link" products [42, 51] will provide insight into the issue of local v. remote storage. The issue of cache overhead is addressed by comparing common hardware cache systems to working implementations of a SoftCache [19, 22]. To address the issue of energy consumption, we have constructed detailed power simulations of both typical hardware caches and a reference design for the SoftCache. Using the TSMC 0.25μm 5-Metal layer process from MOSIS, coupled with the NCSU CDK, Cadence schematics and layouts are analyzed for power consumption information. The net-lists are fed into a combination of tools to obtain power, including Synopsys' HSpice and NanoSim-PowerMill tools.

Using the results from these methods of analysis, we derive key energy delay equations for the embedded system as a whole. By analysis of these equations, the SoftCache is more effective than traditional cache designs for power consumption.

3.1 Cache Overhead

Cache overhead is a comparison of the hidden costs in a hardware cache and the hidden costs of the SoftCache. Hardware caches must store tags, control bits, and other state information for each cache line. The SoftCache has no tags to store, but does carry an overhead for the miss-handler instructions, communications interface, and extra program instructions. SoftCache techniques currently run only on instruction caches, and this analysis considers just instruction caches.

To understand the overhead for hardware cache memory management, we examined the cache structures of several current processors. The overhead calculation is only the extra bits stored with each cache line, without calculating impact in other locations such as locking bits in a TLB. We deliberately *excluded* parity and ECC bits from our calculations, for if these are needed in the hardware cache region they will likely be needed in the generic SRAM replacement that resides in the same hardware. The processors used for comparison are contemporary embedded system low-power devices, including: the Intel XScale which uses the same cache line structure as the DEC (now Intel) SA-110 [16, 43]; the Motorola PowerPC850 [32]; and the MIPS R4Kp [29].

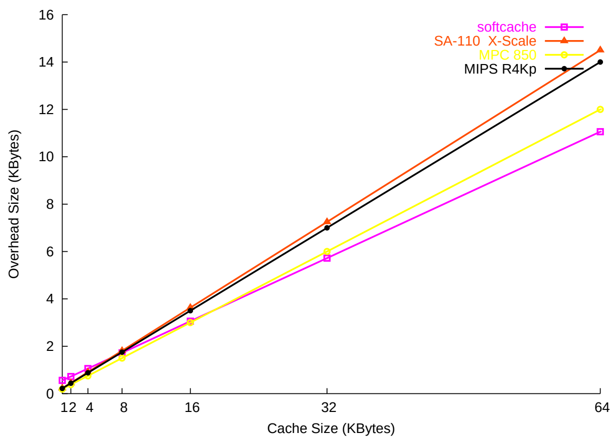

Figure 4: **Overhead storage costs by cache size.**

To compute the overhead of the SoftCache, we used details of the existing implementations [19, 22]. In the non-optimal ARM version, a best-case miss-handler will execute 54 instructions, and worst-case 73. There is also a small primitive communications interface written in C. We therefore allocate 100 instructions for an optimized miss handler and network interface, assuming 32-bit instructions. We use basic blocks as a unit size of instructions in the SoftCache. This translates to a branch occurring every 5-7 instructions. We assume the basic block is five working instructions followed by one branch. The SoftCache carries an extra storage penalty of one branch, given that there is no guarantee of contiguous basic block alignments. Both the taken and not-taken paths must be stored as branch or exception instructions, since no fall-through case may be permissible. This indicates for every six program instructions, one additional branch instruction must be inserted.

Figure 4 shows the results of this comparison. The SoftCache pays a measurable penalty for its 100-instruction miss-handlers that make it a losing proposition for caches below 16KB in size. At the 16KB size, the SoftCache beats all but the MIPS processor and in this case is worse by a few bytes. Beyond a 16KB cache, the SoftCache is clearly a more efficient solution. If the SoftCache were to move from a basic block unit to a larger hyper-block or super-block size, it would attain even better competitive performance. Also note that the SoftCache penalty is not constant due to the extra (*worst*-case) assumption of storing an additional branch with every basic block. While storing these additional branch instructions with every basic block consumes resources, it takes substantially less than the extra storage used by hardware caches.

To date, existing SoftCache designs have focused on small embedded processors and have ignored issues that arise with multiple cache levels. There is potential for treating both L1 and L2 as SoftCaches, or constructing a SoftCache/hardware hybrid for performance reasons.

3.2 Area

The assumptions of the SoftCache makes the area reduction argument plausible. Given that the SoftCache exhibits storage overhead usage that is only slightly better than the small hardware caches we are comparing it to, as shown in Section 3.1, we ignore any arguable area savings in the physical storage within banks. For larger cache sizes (above 32KB), the storage overhead savings can become significant. For smaller embedded processors, the obvious benefits of area savings come from removal of other logic, such as MMU, write buffers, cache control logic, and similar circuits. Based on the published technical data of the SA-110 from DEC [30], we measure the overall area reduction. While these measurements may not be equivalent with respect to other microprocessors, given the complex cache of the SA-110 it is indicative that a quantifiable area savings would occur.

Based on the results presented by DEC in their work, it is estimated that the MMUs and write buffer consume 11% of the total die area. These units also consume 19% of the total die power when running a computationally intensive program such as Dhrystone [30]. Further arguments could be made that the tag structure in the SA-110 is using fully associative CAMs, which contain higher transistor counts than SRAMs, but without knowing how these are implemented (9T/ 10T/11T, sizing, process parameters) it is not possible to establish the potential degree of savings.

Saving 19% of the total die power by removing 11% of the used area is a significant reduction by itself. The SoftCache argument for area reduction is valid.

3.3 Local v. Remote Store

The SoftCache model is frequently brought under suspicion for recommending the reduction or removal of local storage (DRAM, NV space) and utilization of the network link for remote storage. The underlying issue is how the energy consumption of local storage compares to that of using a network. Intuitively we expect local DRAM to be much more energy efficient than any network. According to our analysis described as follows, we find that to be true only in a limited way.

Using data sheets available from vendors including Elpida, Fujitsu, Micron, and Samsung, we selected low-power or mobile DRAM parts representing typical market product performance. We calculate the energy consumption in terms of pJ per bit by computing the *best*-case power consumption listed in the electrical characteristics of each product. This gives us a relative measure of how much energy is used in a *best*- case situation to read or write to the memory. During sleep mode, these devices consume very low current but still require some power for refresh functions. For analysis arguments, we use the Fujitsu FCRAM model MB82D01171A, a 2MB part with the lowest power consumption of all devices measured.

Similarly, we obtained energy information from the data sheets published by several network links vendors. In a similar manner as for the DRAM, we calculate the *worst*-case power per bit consumed, and the standby or sleep-mode power. In this situation, the transmit (TX) and receive (RX) currents are considered separately, as some links display different needs by operating state. For our analysis, we chose the AMI Semiconductor ASTRX1 as a model network link with average characteristics. Details of the analysis of DRAM and network links are available in [18].

As expected, using a network link is 10,000+ times more expensive in power than DRAM – in an extreme-case scenario! (We are comparing the *best*-case DRAM performance to the *worst*-case link performance.) But this does not tell the entire story. One of the key arguments behind the SoftCache design is that there are several modes of operation, and changing modes is an infrequent event. It can also be seen that the sleep-mode energy consumption for Mobile DRAMs designed for low power is 1,000 times more expensive than for a network link. This indicates that if the time between mode switching is sufficiently long, the aggregate consumption of active- and sleep-mode energy by DRAM will exceed the active- and sleep-mode energy consumption of the network link. Finding the amount of time that must be spent in computation (hence leaving the DRAM or network link in sleep mode) before switching modes is an exercise in the energy-delay benefit, with the answer in given after analysis in section 3.6. Before this answer can be determined, we explore additional aspects of the problem.

3.4 Bank Power

To understand how the SoftCache model alters the energy used within the cache, we modeled the individual cache banks used in both traditional processors and a SoftCache equivalent. By removing the MMU and write buffers, the only "extraneous" logic left lies in the CAM tag structures used by hardware caches. We constructed a simulation to

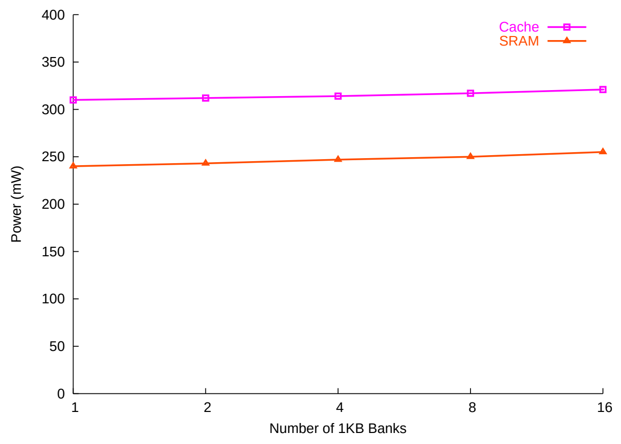

Figure 5: **Hardware cache line compared to SRAM line power.**

analyze how the tag energy would be reduced by a simpler SoftCache design.

To compare power models between the hardware cache banks and SRAM memory banks as proposed for the SoftCache, it is necessary to do transistor level simulations. Using the technical documentation available for the SA-110 [30, 43], we designed a 1KB hardware cache bank with a line size of 32 bytes. This design uses the same 32-way associative CAM structure as the SA-110 and XScale processors. Using the same 32-byte data line size, we also implemented an SRAM bank. Each bank (cache or SRAM) stored 1KB of data. After moderate effort in sizing and verification of the circuit designs, Synopsys PowerMill generated an approximate power measure for designs where the number of banks was varied. The stimulus to the PowerMill simulation was a modified trace gathered from SimpleScalar-ARM. To reduce complexity, we fixed the trace such that all references were hits in either the cache or SRAM (with the cache and SRAM being pre-populated to the same initial data contents). The SRAM core cell for the CAM as well as data storage used minimum sized FETs in the inverters with the β-ratio of 2.0. Additionally, the CAM cell is a 9T implementation with a shared match-line across all 23 bits. Circuits not modeled are control logic for cache invalidates, cache-miss handling, and so forth. This presents the hardware cache in a more energy efficient model than would actually exist.

The results of this comparison can be seen in Figure 5. While the power savings of the SRAM may not be exactly the same as would this design implemented in an SA-110 base, it is indicative of the power reduction that can be expected. The bulk of the difference in power between the cache model and the SRAM model is attributed to the extra power consumed by the tag arrays implemented with CAMs.

3.5 SoftCache Penalties

This section continues the comparison of a *best*-case DRAM solution against a *worst*-case network link solution. Consideration of the hidden overhead involved in SoftCache is ignored, as is the hidden overhead of a hardware cache. The two models being compared are (a) μP with hardware cache and local DRAM storage, and (b) μP with SoftCache and a network link.

It is clear that additional instructions must be executed in the SoftCache to effect a hardware cache equivalent. These instructions come in two flavors: miss handlers, and penalty branches. We can compare these penalties to the actual work being done during any given mode of computation to understand the penalty that each model incurs.

The total amount of time spent in a given computational mode is the arbitrary amount of time doing actual work, as opposed to moving data around in order to perform work. This time for the computation itself is denoted T_C. Assuming our worst-case expectation of executing some 100 instructions in a miss handler every time we need to fetch another basic block, the SoftCache performance and power penalty could be substantial.

Hardware caches use integrated controllers that fetch cache lines from memory at high speed. Since the SoftCache uses the basic block size for transfer, it transfers instructions in 6-instruction blocks on average. This requires accessing the network to send a request to the server, waiting for the server to process the request, and then the time and network required to receive the correct response. However, the transfer rate for the network is substantially slower than for DRAM. The additional time the CPU is “idle” and waiting for the network activity to change must be factored in a well.

Using the SA-110 as the hardware baseline model, we can assume a reduction to idle-mode during these times at 20mW for idle power [30]. The time spent in different states of transfer can be represented as a function of the network link rate. The SA-110 core consumes 0.5W during CPU intensive programs that run primarily from on-chip cache, such as Dhrystone. The same core in a SoftCache model – where MMU and write buffers have been discarded – would consume 0.4W. After factoring in the energy consumption in the cache banks, this number could be as low as 0.25W. For our analysis we have used the reduced value of 0.4W.

The “penalty” branch instructions occur when a basic block is brought into the client, and one branch path is resolved. At a later point, the alternate branch path may be resolved as well, but the target address for the hot path may not be the sequentially next instruction as it was in the original program. Therefore, some form of extra branch is required to move to the correct location. In an extreme case, we would have to execute every penalty branch instruction, which would cause the CPU to consume extra energy.

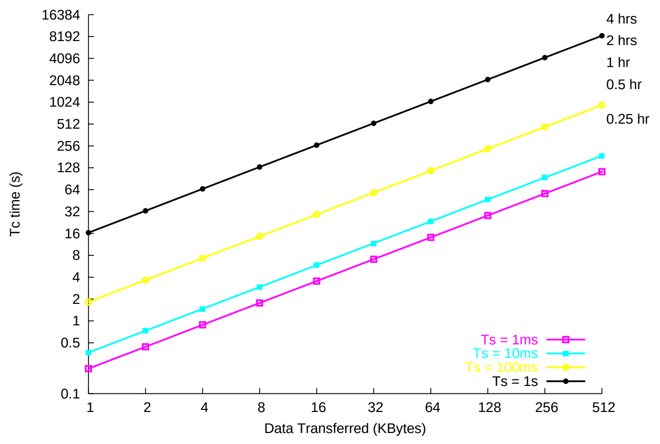

Figure 6: **Duration of computation T_C that must pass for the network link to be more energy efficient, where T_S and B_N vary.**

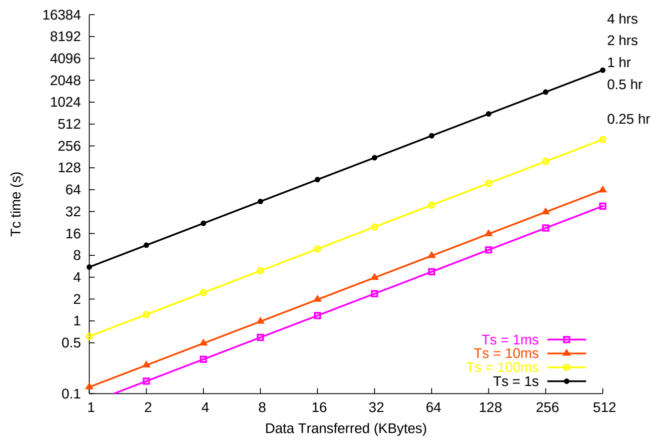

Figure 7: **Duration of computation T_C that must pass for the network link to be more energy efficient, where T_S and B_N vary and branch penalty is removed.**

3.6 Energy and Delay

While the prior discussion explains penalty instructions and network link usage, they don't portray the energy trade-offs with respect to overall performance. Instead we introduce a set of equations to show how energy is impacted by the primary variables network link speed, R_L, time for the server to process a request (not counting TX/RX times), T_S, and total bits transferred for a mode change, B_N. *(A complete derivation and analysis is presented in [18].)*

With respect to the network link and the power savings in the SoftCache model shown in section 3.2, we can derive equations to represent the total energy spent as well as the total time for a typical mode. These are dependent on which model is being used – DRAM or link. The total energy for DRAM, E_D, and total time for DRAM, T_D, corresponds to the total energy and time for the link version, E_L and T_L. Note that prefetching, mispredictions, and other pressures that increase memory traffic are not considered – that is, we consider a perfect access model to memory for best-case performance of memory, with perfect CPU utilization.

We find the equilibrium point for the total computation time, T_C, by equating the energy of DRAM and network link. This equilibrium point is the minimum time span that T_C must encompass for the two models (local DRAM and hardware cache v. SoftCache and network link) to be equivalent. Beyond this equilibrium point, the SoftCache is more energy efficient due to the differences in sleep energy. That is, solving

$$E_D = E_L \tag{1}$$

gives the average amount of time that must be spent in any given mode before changing. (Each term of equation 1 uses T_C to compute the total energy consumed during the mode.)

Evaluating this result for various values of bits required for the mode change, B_N, we obtain a plot of B_N vs. T_C as shown in Figure 6. The result is sensitive to variances of T_S, the server processing time. While the server can be made powerful enough to keep the T_S response time low, it will be non-zero. With one server controlling multiple clients, it can also be expected that some contention may exist for the server attention. This figure indicates how the penalty changes with increasing contention. Moreover, this equilibrium equation includes the *worst*-case branch penalty behavior (every penalty instruction executed). The same graph with the branch penalty removed can be seen in Figure 7.

The surprising result is not that the SoftCache does become an energy win given sufficient time, but that it can do so in seconds! In reality, the V_{DD} supply for the network link could be passed through a cutoff-transistor to completely disconnect the link devices, thereby reducing their sleep current to 0A [40]. This is possible since only the client ever initiates a connection to the server – the server cannot spuriously send commands to the client. This would make the link power model more quickly a win in net energy.

Given that the network link can be more energy effective in seconds, the rationale for mode changes being infrequent (on the order of tens of minutes) does not initially seem correct. Closer examination reveals why finding the *equilibrium* point is not sufficient to understand the problem. Ideally, the additional time spent in the slow network link (40kbps for the ASTRX1) to switch modes should not adversely affect application performance. The goal is to fix the application slowdown due to network traffic to a maximum of 1% penalty. At this point, we exclude the branch penalty worst case from the evaluation.

Factoring in the rate of the network link, R_L, to the create an energy-delay equation, we realize that the relatively slow

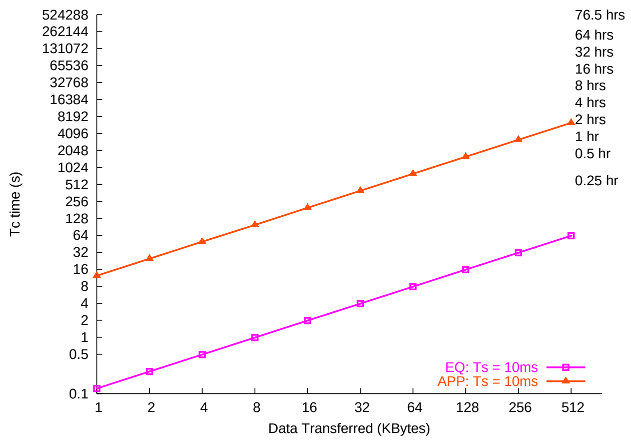

Figure 8: **Amount of time to elapse for network transfer delays to have a 1% impact on application performance. T_S = 10ms.**

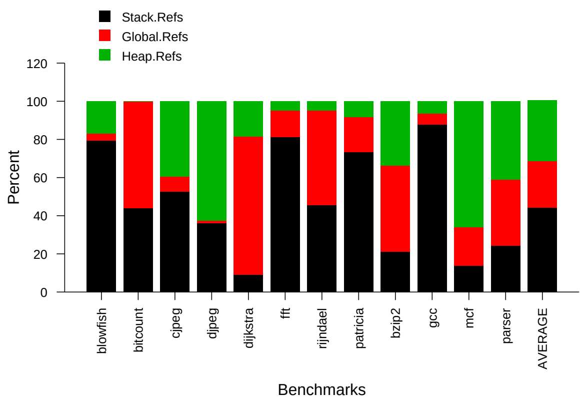

Figure 9: **Results of a modified SimpleScalar 4 (ARM) analysis on benchmarks to classify memory access patterns.**

speed of the network can force a tremendous impact on application performance. Figure 8 shows the effect of these additional considerations. This figure includes the original equilibrium values, marked as ”EQ”, and the consideration for slowdown in the network affecting application run-time versus the original equilibrium point, marked ”APP”.

This application impact ignores the consideration of the branch penalty overhead. To include the branch penalty in our equation, where the penalty is the worst-case, gives an equation with no solutions. This is a logical result since we desire an application impact of less than 1%, but we specify the worst-case case of branch impact of 16.7% penalty on the application. Even the reduced energy consumption in the SoftCache processor die can not compensate for this behavior. In a real situation, the true slowdown will correspond directly to the overall branch penalty. Some effort can be spent on the server-side to rewrite branches to be penalty-free, but some branches will not be susceptible to corrections. These additional branches will be the primary slowdown of the application, with a lesser slowdown due to network traffic.

In the best-case scenario, the SoftCache is clearly an energy-delay benefit given sufficient time. For the worst-case, however, no positive solution can exist. This follows the logical expectation that given a sufficient performance penalty from branches, the extended run-time of the mode iterations costs more than the power saved by using the SoftCache model. From our analysis, a tolerance of approximately 25% of the worst-case behavior can be supported and the SoftCache remains an energy-delay win scenario.

4 Future Directions

The SoftCache concept is still in prototyping stage, with many issues in need of a solution. While working on the infrastructure for our system, however, we obtained insight into other dimensions of the problem that are on the near horizon. Another line of research is starting now in optimization of the SoftCache to prevent excessive fault handling when changing working sets.

4.1 Data Caching

To fully support data caching in the SoftCache framework requires a complete control-flow and data-flow analysis of random binary images with no source code information. This is a known hard problem. To gain some insight into how data caching might perform, however, characterization of memory access patterns are useful.

Across the MediaBench and SPEC2000 benchmarks, we find that the typical program contains 40% memory references. Of these references, we can break them down into various categories: 44% are stack, 32% are heap, and the remaining 24% are global-static or embedded text constants [4]. The stack pointer is updated approximately 0.7% for the total memory references. These numbers come from a modified SimpleScalar-4/ARM analysis the benchmarks, as shown in Figure 9.

The implications from this suggest that the primary memory reference consumer, the stack, is prone to SoftCache methods. Stack accesses are generally unaliased, and tend to be short-lived parameter passing values and local variables. The heap accesses are most likely to be aliased, and will represent a pointer interpretation problem due to the constant SoftCache address rewriting mechanism. This has the potential to have a substantial negative impact on application performance unless dataflow analysis reveals that aliasing is a rare occurrance. Only those accesses which can be aliased will need special protection against invalid pointer values. The global static data and embedded text stream constants can be broken up and moved on an as-

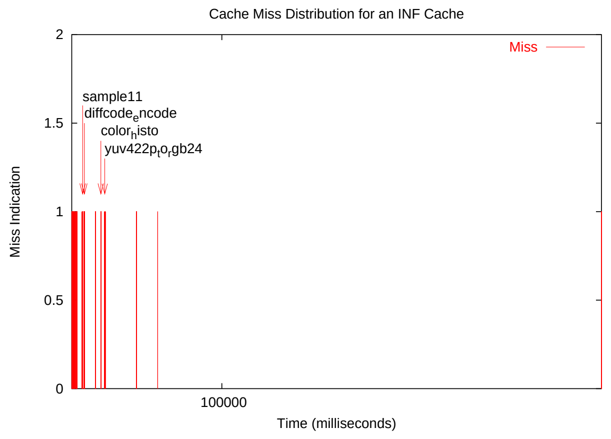

Figure 10: **The cache miss and invalidate patterns for an infinite SoftCache on-die storage.**

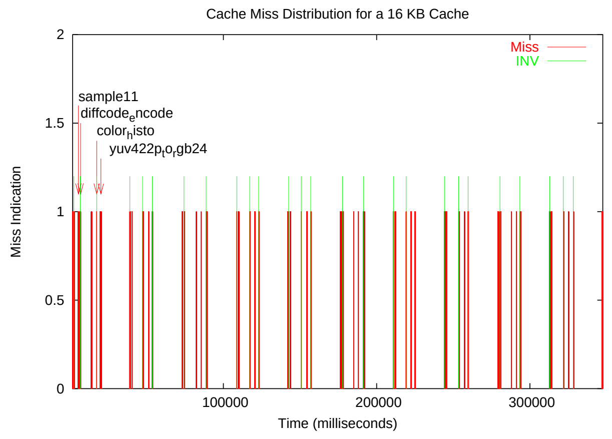

Figure 11: **The cache miss and invalidate patterns for a 16KB on-die SoftCache storage.**

needed basis for the basic blocks of a program. This can also remove the unwanted trampoline policy of compiler generated code, such as gcc output, when accessing a datum beyond the limited load/store offset defined in the ISA. Until our data caching work is complete, exact performance characteristics and usage patterns are unknown.

4.2 Mode Switching

The SoftCache incurs a large penalty on the repeated faults when switching from one mode to another. This naturally leads to the question of whether modes can be predicted and thus preloaded, thereby avoiding the repeated miss scenario. While this may seem a minor optimization at first glance, the reduction in network traffic will reduce collisions, protocol overheads, and related aspects that can lead to saturation of the infrastructure.

We explore the problem using a custom application which runs multiple image processing algorithms (processing camera feeds with different convolutions, for example). To establish a base-line, we run the application under a SoftCache model that simulates infinite on-die SRAM for the application. The resulting "cache misses" occur only when loading application code for the first time, as shown in Figure 10. There are four different μ-kernels within the processing application: sampling, encoding, color histogram, and color downsampling. The starting point for each kernel is shown in the figure.

To observe the SoftCache interactions for misses and invalidates, we reduce the cache space from infinite to 16KB. The visual display of the access pattern is quite apparent for the mode-switching during the lifetime of the program, as shown in Figure 11. This strongly implies that by careful analysis of the original program to the translated program, the entire working-set can be captured and buffered on the remote server. When a mode change is detected, the server can save the client repeated requests by preloading the next working set on the first miss occurance.

Our initial analysis is based on extracting all symbols for each "mode" from the program image. By pattern matching from the original symbol set and original PC addresses, modes can be found in the translated address space for the core working set of the current μ-kernel. In order to preload the next working set, careful balancing must be performed between the transitional code that must execute between working sets as well as the forthcoming working set kernel. We expect to find that most applications can have their working sets characterized in such a fashion.

5 Related Work

5.1 Binary Rewriting and Program Analysis

Binary rewriting or translation is one result of program analysis. Using the application source code as a basis, compilers perform a wide variety of *static* analysis techniques in code generation. These can lead to dead-code and -data elimination, instruction block sequencing, and overall more efficient binaries [13, 47]. Such analysis fails to capitalize on optimizations that can only be determined *dynamically* with how the application is behaving with live data.

Dynamic compilers can generate optimal instruction traces, which can require sophisticated code cache management schemes [20]. Based on input which cannot be predicted statically, further dead-code and -data path elimination becomes possible [3], as well as pointer disambiguation. Additional benefits can be found by rearranging the code or data layouts using profiling [35, 38].

The SoftCache must reconstruct knowledge of the program behavior from the binary alone – rebuilding control- and data-flow graphs with no *a priori* knowledge. While a large body of work has been created around program analysis for data-flow or control-flow [21, 8, 2, 39, 44, 34], less has been done directly on binary systems. Most binary analysis systems explored to date mandate certain restrictions [12, 15, 26].

The Hot Pages system uses sophisticated pointer analysis with a compiler that supports transformations [31]. Shasta is a shared memory system that uses static binary rewriting to share variables among multiprocessors [41]. While the SoftCache could yield similar results, it offers more potential by use of dynamic program behavior.

Many simulators also use binary rewriting in varying forms to achieve faster results compared to strict interpretation simulators. Such systems as Talisman-2 [7], Shade [14], and Embra [54] use this technique. These simulators have further burdens of modeling additional resources and behaviors rather than a goal of pure execution as in the SoftCache.

Just-In-Time compilers, like those supporting Java, with a distributed model of a JVM [45] also have some shared ideas with SoftCache. These systems generate unoptimized byte-code for programs, and when a "hot" trace is found, it is highly optimized and rewritten into native platform instructions rather than JVM byte-code. Other efforts have focused on tuning the Java garbage collector systems to increase memory efficiency [11].

5.2 Alternate Caches

The eXtended Block Cache [24] proposed by Intel corporation adds an element of redundant instruction suppression while lowering fragmentation. The XBC goals were to provide a method of comparable performance to trace caches while being more efficient in design. Another modification to the idea of a trace cache is the Block-Based Trace Cache [9]. The key idea of the block-based design is to cache sequences of traces and then store a pointer to the translated sequence. More complicated hardware designs like this result in a greater energy consumption.

Other techniques being pursued for reducing cache energy usage lie in putting regions of the cache storage in "sleep" mode or using subdividing techniques. Some of the recent work in this area [27, 56] concentrates on reorganizing the layout of cache regions either into sub-blocks for lower access energy, or for placing data banks into sleep mode while keeping tags fully powered.

The Span Cache [52, 53] explores a model of direct-addressing regions of the cache. It exposes the cache area as directly addressable through additional registers, but has a fall-back case of behaving exactly like a normal hardware cache if an entry is not found within the direct-addressed system. A benefit of this system is that it allows variable cache block sizes with only a minor penalty when compared to a traditional hardware cache design. The SoftCache also provides variable block sizes, with full associativity, but involves a hardware reduction rather than addition.

Exploring the possible usage of on-chip memories like the ScratchPad [16], Panda began a series of experiments on the concept of Scratch-Pad usage for optimizing data accesses [36]. This work was later expanded on by the efforts of many [37, 5, 46, 50], hinging on the same fundamental idea – adding a small on-chip RAM in addition to the hardware cache system. Similarly the Cool-Cache project advocated using a scratchpad for scalars [49]. To manage this on-chip memory, efforts have focused on modifying a C compiler to statically (or with profiling feedback) determine the most-executed blocks of code or referenced data, and then generating in the program stream the necessary load-to-scratchpad and evict-from-scratchpad instructions and controls. While relevant in one sense for the usage of on-chip memory, the SoftCache uses a truly dynamic method for placing code or data into on-chip memory, and removes the cache hardware.

6 Conclusion

We explored the claims and assumptions behind the recent proposals for explicitly software managed cache systems. We characterized how a SoftCache design fits into the overall landscape of computing devices to better understand the mechanisms required to prove or disprove the assumptions behind such a system. Evaluating the SoftCache on the criteria of overhead storage cost, link vs. DRAM energy and speed costs, space, and total energy consumption, we found that the SoftCache was clearly a viable solution for some devices. A natural class of such a device is proposed as an embedded device within an ubiquitous computing framework, but we have shown that under certain application characteristics (low miss rate, infrequent mode changes) the SoftCache is a more generally applicable mechanism. We have shown that for a 256KB data transfer to switch modes, 1.45 hours must elapse to have a 1% application performance degradation before switching to the next mode. During this time, roughly 700J can be conserved in a processor core such as the SA-110. These energy savings represent a 30% reduction over traditional designs. Obtaining these savings also require replacing a 2MB DRAM chip with a network link similar to the 40kbps ASTRX1 fully integrated transceiver. Modern embedded devices tend to use faster network links and more DRAM parts than analyzed here, which makes the SoftCache even more energy efficient. Not addressed are the issues of protocol and security for the network transmission, which are separate engineering problems.

Further work needs to be spent on examining the cache structures not simulated in this study, as well as investigating the problems associated with data caching. Other areas that are under investigation include isolating "modes" and swapping directly from mode-to-mode, without using a gradual block replacement system; characterization of medium- to large-scale applications and benchmarks for performance behaviors; and refinement of the transistor models and reference designs for both the SoftCache and the SA-110 hardware comparison.

7 Acknowledgements

This work was funded in part by the National Science Foundation under grants CCR-98-76180, CCR-01-21638, and EIA-99-72872.

References

[1] Gheith A. Abandah and Edward S. Davidson. Configuration Independent Analysis for Characterizing Shared-Memory Applications. Technical report, EECS Department, Univerisity of Michigan, CSE-TR-357-98 1998.

[2] Hiralal Agrawal. On Slicing Programs with Jump Statements. In *Proceedings of the ACM SIGPLAN Conference on PLDI*, pages 302–312, June 1994.

[3] Vasanth Bala, Evelyn Duesterwald, and Sanjeev Banerjia. Dynamo: A Transparent Dynamic Optimization System. In *Proceedings of PLDI*, Vancouver, Canada, 2000.

[4] Chinnakrishnan Ballapuram and Hsien-Hsin S. Lee. Energy Efficient d-TLB and Data Cache using Semantics-Aware Multilateral Partitioning. In *International Symposium on Low Power Electronics and Design*, 2003.

[5] Rajeshwari Banakar, Stefan Steinke, Bo-Sik Lee, M. Balakrishnan, and Peter Marwedel. Scratchpad Memory: A Design Alternative for Cache On-chip memory in Embedded Systems. In *Proceedings of the 10th International Workshop on Hardware/Software Codesign*, May 2002.

[6] Ravi Batchu, Saul Levy, and Miles Murdocca. A Study of Program Behavior to Establish Temporal Locality at the Function Level. Technical report, Rutgers University, DCS TR-475 2001.

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

[8] David Binkley. Slicing in the presence of parameter aliasing. In *Software Engineering Research Forum*, pages 261–268, Orlando, Florida, November 1993.

[9] B. Black, B. Rychlik, and J. Shen. The Block-Based Trace Cache. In *Proceedings of the 26th ISCA*, May 1999.

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

[11] G. Chen, M. Kandemir, N. Vijaykrishnan, M.J. Irwin, and M.Wolczko. Adaptive Garbage Collection for Battery-Operated Environments. In *Proceedings of USENIX JVM02 Symposium*, August 2002.

[12] Cristina Cifuentes, Doug Simon, and Antoine Fraboulet. Assembly to High-Level Language Translation. Technical Report 439, University of Queensland, August 1998.

[13] Andrea G.M. Cilio and Henk Corporaal. A linker for effective whole-program optimizations. In *Proceedings of HPCN*, pages 643–652, Amsterdam, The Netherlands, April 1999.

[14] 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.

[15] Keith D. Cooper, Timothy J. Harvey, and Todd Waterman. Building a Control-flow Graph from Scheduled Assembly Code. Technical Report TR02-399, Rice University, June 2002.

[16] Intel Corporation. Intel XScale Microarchitecture Technical Summary. Technical report, Intel WWW Site, 2000.

[17] Deborah Estrin, David Culler, Kris Pister, and Gaurav Sukhatme. Connecting the Physical World with Pervasive Networks. In *Pervasive Computing*, Jan 2002.

[18] Fryman et al. Energy Analysis in SoftCache Systems. Technical report, Georgia Institute of Technology, Tech Report Draft, 2003. `http://www.cc.gatech.edu/-~fryman`.

[19] Joshua B. Fryman, Chad M. Huneycutt, and Kenneth M. Mackenzie. Investigating a SoftCache using Dynamic Rewriting. In *Feedback Directed and Dynamic Optimization Workshop 4*, November 2001.

[20] Kim Hazelwood and Michael D. Smith. Code Cache Management Schemes for Dynamic Optimizers. In *Proceedings of the Sixth Annual Workshop on Interaction between Compilers and Computer Architectures*, 2002.

[21] Susan Horowitz, Thomas Reps, and David Binkley. Interprocedural slicing using dependence graphs. In *ACM TOPLAS*, volume 12, No. 1, January 1990.

[22] Chad M. Huneycutt, Joshua B. Fryman, and Kenneth M. Mackenzie. Software Caching using Dynamic Binary Rewriting for Embedded Devices. In *International Conference on Parallel Processing*, 2002.

[23] NTT Japan. BLUEBIRD Project. 2003. `http://www.-ntts.co.jp/java/bluegrid/en/`.

[24] Stephen Jourdan, Lihu Rappoport, Yoav Almog, Mattan Erez, Adi Yoaz, and Ronny Ronen. eXtended Block Cache. In *Proceedings of the Sixth International Symposium on HPCA*, January 2000.

[25] Philo Juang, Hidekazu Oki, Yong Wang, Margaret Martonosi, Li-Shiuan Peh, and Daniel Rubenstein. Energy-Efficient Computing for Wildlife Tracking: Design Tradeoffs and Early Experiments with ZebraNet. In *Proceesings of ASPLOS-X*, October 2002.

[26] Daniel Kästner and Stephan Wilhelm. Generic Control Flow Reconstruction From Assembly Code. In *Proceedings on LCTES*, pages 46–55, 2002.

[27] Soontae Kim, N. Vijaykrishnan, Mahmut Kandermir, Anand Sivasubramaniam, and Mary Jane Irwin. Partitioned Instruction Cache Architecture for Energy Efficiency. In *ACM Transactions on Embedded Computing Systems*, June 2002.

[28] Alan Mainwaring, Joseph Polastre, Robert Szewczyk, David Culler, and John Anderson. Wireless Sensor Networks for Habitat Monitoring. In *ACM International Workshop on Wireless Sensor Networks and Applications*, September 2002.

[29] MIPS. MIPS32 R4Kp Core Datasheet, Rev. 01.07. Technical report, MIPS Technologies, 2002.

[30] James Montanaro and et al. A 160-MHz, 32-b, 0.5-W CMOS RISC Microprocessor. In *IEEE Journal of Solid-State Circuits*, volume 31, No. 11, pages 1703–1714, November 1996.

[31] 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.

[32] Motorola. MPC850 Family User's Manual, Rev. 1. Technical report, Document MPC850UM/D, 2001.

[33] Georgia Institute of Technology. The Aware Home Project. 1999-2003. `http://www.cc.gatech.edu/fce/-ahri/publications/index.html`.

[34] Alessandro Orso, Saurabh Sinha, and Mary Jean Harrold. Effects of Pointers on Data Dependences. In *Proceedings of the 9th International Workshop on Program Comprehension*, 2001.

[35] Krishna V. Palem and Rodric M. Rabbah. Bridging Processor and Memory Performance in ILP Processors via Data-Remapping. Technical Report GIT-CC-01-014, Georgia Institute of Technology, June 2001.

[36] Preeti Ranjan Panda, Nikil D. Dutt, and Alexandru Nicolau. Effi cient Utilization of Scratch-Pad Memory in Embedded Processor Applications. In *Proceedings of European Design and Test Conference*, March 1997.

[37] Preeti Ranjan Panda, Nikil D. Dutt, and Alexandru Nicolau. On-chip vs. Off-chip Memory: The Data Partitioning Problem in Embedded Processor-Based Systems. In *ACM Transactions on Design Automation of Electronic Systems*, pages 682–704, July 2000.

[38] Rodric M. Rabbah and Krishna V. Palem. Design Space Optimization of Embedded Memory Systems via Data Remapping. Technical Report GIT-CC-02-011, Georgia Institute of Technology, March 2002.

[39] John L. Ross and Mooly Sagiv. Building a Bridge between Pointer Aliases and Program Dependences, 1998.

[40] Kaushik Roy and Sharat Prasad. *Low-Power CMOS VLSI Circuit Design*. Wiley-Interscience, USA, 2000.

[41] 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.

[42] AMI Semiconductor. ASTRX1 - Single Chip Transceiver. Technical report, Datasheet, Model ASTRX1 2001.

[43] Digital Semiconductor. SA-110 Microprocessor Technical Reference Manual, Rev. C. Technical report, Order No. EC-QPWLC-TE, 1996.

[44] Saurabh Sinha, Mary Jean Harrold, and Gregg Rothermel. Systems-Dependence-Graph-Based Slicing of Programs with Arbitrary Interprocedural Control Flow. In *International Conference on Software Engineering*, pages 432–441, 1999.

[45] 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.

[46] Stefan Steinke, Nils Grunwald, Lars Wehmeyer, Rajeshwari Banakar, M. Balakrishnan, and Peter Marwedel. Reducing Energy Consumption by Dynamic Copying of Instructions onto Onchip Memory. In *Proceedings of the International Symposium on System Synthesis*, October 2002.

[47] Bjorn De Sutter, Bruno De Bus, Koen De Bosschere, and Saumya Debray. Combining global code and data compaction. In *Proceedings of ACM SIGPLAN Workshop on Languages, Compilers, and Techniques for Embedded Systems*, 2001.

[48] Sameer Tilak, Nael Abu-Ghazaleh, and Wendi Heinzelman. A Taxonomy of Wireless Micro-Sensor Network Models. In *Mobile Computing and Communication Review*, April 2002.

[49] Osman S. Unsal, Raksit Ashok, Israel Koren, C. Mani Krishna, and Csaba Andras Moritz. Cool-Cache for Hot Multimedia. In *MICRO-34*, 2001.

[50] Manish Verma, Stefan Steinke, and Peter Marwedel. Data Partitioning for Maximal Scratchpad Usage. In *Proceedings of Asia South Pacific Design Automated Conference*, January 2003.

[51] VISHAY. Fast Infrared Transceiver Module. Technical report, Datasheet, Model TFDU6102 2001.

[52] Emmett Witchel and Krste Asanovìc. The Span Cache: Software Controlled Tag Checks and Cache Line Size. In *Workshop on Complexity-Effective Design at the 28th ISCA*, June 2001.

[53] Emmett Witchel, Sam Larsen, C. Scott Ananian, and Krste Asanovìc. Direct Addressed Caches for Reduced Power Consumption. In *MICRO-34*, pages 124–133, December 2001.

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

[55] Dong Zhou, Santosh Pande, and Karsten Schwan. Method Partitioning - Runtime Customization of Pervasive Programs without Design-time Application Knowledge. In *International Conference on Distributed Computing Systems*, May 2003.

[56] Huiyang Zhou, Mark C. Toburen, Eric Rotenberg, and Thomas M. Conte. Adaptive Mode Control: A Static-Power-Effi cient Cache Design. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, September 2001.