SoftCache: Dynamic Optimizations for Power and Area Reduction in Embedded Systems

Joshua B. Fryman, Hsien-Hsin S. Lee, Chad M. Huneycutt,
Center for Experimental Research in Computer Systems (CERCS)
Georgia Institute of Technology
Atlanta, GA 30332-0280
{ fryman, leehs, chadh }@cercs.gatech.edu

ABSTRACT

We propose a SoftCache for low-power and reduced die area while providing application flexibility. Our implementations demonstrate that the network is a power efficient means for accessing remote memory. The impact of this work suggests that SoftCache systems may be useful in future consumer electronics. Our results show that die power is reduced by 20%, die area is reduced by 10%, and transferring applications over the network is more energy-delay effective than local DRAM.

1. INTRODUCTION

Embedded consumer electronics continuously add more features while shrinking their physical size. With feature creep demanding more memory in all computing devices, densities of DRAM and Flash memory increase trying to keep pace. Unfortunately, energy consumption does not decrease 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, there are other looming issues. For example, the demands of feature flexibility and reconfigurability lead to a frequently changing operational behavior including switching active codecs, upgrading browsers, downloading new applications, etc.

To address these issues, companies like NTT Japan are investigating solutions that perform dynamic application code migration between the remote device and other network systems [6]. Intel has designed the PXA 27x family of processors with integrated SRAM, NVM, and network support. In this paper, we propose the SoftCache, an explicitly software managed cache-like storage with versions built on both the Intel XScale and Sun UltraSPARC, to enable low power yet versatile ubiquitous computing. The SoftCache converts the on-chip cache structures to generic SRAM, removing the die space 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 (off-die) non-volatile memory (NVM) requirements are reduced or eliminated, with additional storage needs coming across the network to remote storage servers. Using the already-present network link and a distributed client-server model, a stripped down client provides features to the user as though no physical reduction had occurred. This is accomplished via the computing capacity of remote network servers. The net result when applied to caching is an explicitly software managed system. This provides benefits such as dynamic instrumentation and feedback, full associativity, variable data block sizing, and flexible resource utilization. It also demonstrates a substantial energy reduction.

The rest of this paper is organized as follows. Section 2 discusses the SoftCache architecture and its operations. We then analyze the area and power aspects of a SoftCache in Section 3. We discuss related work in Section 4, and Section 5 concludes.

2. SOFTCACHE ARCHITECTURE

The basic idea behind a SoftCache system is to use remote servers as virtual memory, which effectively contain infinite resources while leaving the "indispensable" components on the capability-limited embedded devices. For an always-connected environment, information including code and data can be retrieved on demand instead of storing all of it in the embedded device. As such, the hardware features on these embedded devices, in particular both volatile and non-volatile memory, can be kept at a minimal level. This may reduce the power and area requirements, leading to longer operation hours and lower cost.

2.1 Basics of a SoftCache System

The SoftCache system is based on a client-server computing model. We implemented and tested two working SoftCache prototypes, one based on the Intel XScale platform and the other on Sun UltraSPARC. During startup, a server loads the invoked application and prepares it for translation to the embedded device; the client dynamically communicates with the server to load necessary code and data on demand for execution. The atomic granularity of each request made by the client is called a "chunk." The size of each code fragment in a chunk, a design option, can be a basic block, a hyperblock, a function, or even an arbitrary program partition. The data allocation, completely managed by the server, is detailed in Section 2.4.

For each target application, an arbitrary ELF file image is provided to the server and is broken into chunks of code and data for future on-demand transfer. An exception is triggered to acquire the demand chunk from the remote server whenever the client attempts to fetch and execute non-resident code targets or data variables. Once acquired, these chunks will then stay inside the local on-chip memory until they are eagerly evicted or de-allocated when the application is terminated. As long as the embedded device contains just enough on-chip RAM to hold the "hot code" and associated data, a steady state will eventually be reached. Henceforth there will be no more remote transfers until the execution shifts program phases into a different code or data working set.

Our existing SoftCache design focuses on small embedded processors and ignores 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, such as a hardware L1 and SoftCache L2.

2.2 Static Analysis by the Server

As previously mentioned, the server loads an arbitrary ELF image. Our implementations require the image to be statically[1] linked. The server constructs, as per the ELF header information, a virtual memory space and seeds the `bss` and `data` segments to appropriate values. As the program is loaded, extensive static analysis is performed to isolate blocks of code (e.g., into basic blocks or functions), beginning with the ELF-specified entry vector. Each block is

[1] It can be easily extended to support dynamically linked images.

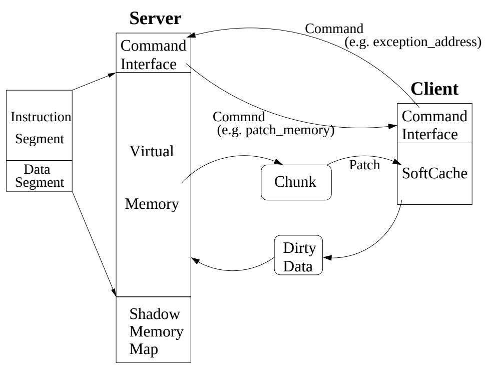

Figure 1: Execution Model of a SoftCache System.

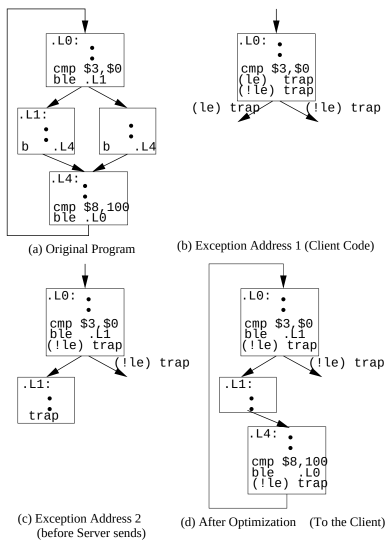
.L0:
cmp $3,$0
ble .L1
.L1:
b .L4
.L4:
cmp $8,100
ble .L0
(le) trap
(!le) trap
trap
(a) Original Program
(b) Exception Address 1 (Client Code)
(c) Exception Address 2 (before Server sends)
(d) After Optimization (To the Client)

Figure 2: Dynamic Execution of a Client in SoftCache.

annotated to support fast rewriting for the target client. To facilitate this, the server also maintains a shadow copy of the client's local memory. As blocks are rewritten on demand, they are stuffed into the shadow copy and then *patches* are sent from the server to the client to update the client memory as needed.

2.3 Dynamic Execution

When an application is running on the client, the client and server communicate via five major SoftCache interface commands as listed in Table 1. As illustrated in Figure 1, the client begins by activating its interface block, which will connect to a remote server and request the first chunk of code and data by sending an initial_start command. The server in turn translates the first block of `main()` and returns it to the client with a patch_memory command. This is immediately followed by a resume_execution command. Once the initial block is loaded, execution begins.

The command patch_memory is the key technique to support and enable an effective SoftCache system. We use Figure 2 to demonstrate the patch memory operation based on instruction chunks transferred at basic-block granularity. Figure 2(a) shows the control flow graph of our example code. The server translates one basic block at a time and sends it back to the client for execution. Each branch at the end of a basic block will be replaced with exception traps. For example, the exit of a conditional branch with two possible exit conditions, taken or not-taken, will be guarded by two exception trap instructions as shown in Figure 2(b). As the client reaches the end of the block (i.e. a trap execution), one path is resolved. This invokes the interface wrapper to again query the server for the missing chunks of code or data, with the client passing back the address that generated the fault with the exception_address command.

The server checks the *shadow memory map tables* which maintain a copy of the client memory allocation map to determine which block to load for the given exception address. The server then translates, shadow-updates, and patches the remote client with the new chunks using the patch_memory command followed by a new resume_execution command. Figure 2(c) shows that the new code on the client side with a newly patched basic block followed by the taken path. The taken-path guard trap instruction of its predecessor basic block is now replaced with a translated branch instruction.

For unconditional branches, the server performs optimization by eliminating the branch instruction during patching, a technique similar to trace construction using the fill unit in high performance processors [4]. This optimization is illustrated in the transition from Figure 2(c) to Figure 2(d). The whole process continues iteratively until the hot-state of the program is resident in the client memory, at which point begins a full-speed execution. This execution may be even faster than in a hardware cache model, given the faster access times SRAMs can sustain when the cache overhead (e.g. tag look-up and compare) is removed.

Given sufficient time and a large application, the server will realize that insufficient space is remaining in the SoftCache to load new chunks. In this case, the system performs memory invalidation eagerly[2] to free up sufficient memory to continue execution. For dirty data that must be extracted, the server sends the command return_memory with a starting address and a size. Multiple commands are used to return non-contiguous information, such as functions, stack data, etc. Also note that for those instruction chunks that are evicted, trap instructions need to be patched back into all predecessor basic blocks.

One immediate benefit of the SoftCache strategy is the fully-associative nature of the SoftCache system. Under explicit software management, truly variable chunk sizes (basic block, hyperblock, function, etc.) can be employed for optimizing performance dynamically. The result is a highly flexible virtualized caching system.

2.4 Handling Data in SoftCache

The difficulty to safely handling data caching in such a scheme is that data addresses are not necessarily known *a priori* by the server. Two basic categories of data operations exist – those with static targets and those with dynamic targets. Static targets are those locations which never change, regardless of how they are accessed, such as global variables. Dynamic targets may be pointers used to traverse arrays, or heap-based memory objects.

By careful analysis the server can identify static targets. Similar to how chunks are loaded and resolved on demand, load/store operations in the original program are replaced with trap instructions. When the client executes the trap, the server looks up the actual target of the load/store. If the target is resident on the client, only a patch is sent to replace the generating trap instruction with a correct load/store operation. If the target is missing, it is first loaded into the client and then the patch follows.

Dynamic targets come in two flavors, stable and unstable. Stable dynamic targets are not known in advance, but the analysis of program data and control flow may indicate that the load/store target is unchanging over windows of computation. The server replaces a stable dynamic load/store operation with a trap. Unlike static data references, these traps are not replaced with an updated load/store instruction once the server discovers where the instruction is pointing in memory. Instead, a test is inserted to determine if the target matches the expected value. If the comparison is true, the load/store proceeds as expected. If the comparison fails, however, a trap is executed which eventually reports to the server that the instruction is

[2]Our current implementation employs a random invalidator. Other history-based algorithms can be used to further improve locality.

Command	Sent by	Client Function	Server Function
initial_start	Client	Request the first data chunk	Translate the first block of `main()`
patch_memory	Server	Receive data chunks	Allocate code and data chunks
resume_execution	Server	Continue the instruction execution	Wake up the client to continue execution
exception_address	Client	Request missing data chunks	Translates address and updates the shadow map table
return_memory	Server	Return dirty data	Send data address and size for update in server

Table 1: SoftCache Interface Commands

transitioning from one stable address to another. Therefore, the original program load/store instruction is replaced with three instructions: compare; trap on not-equal; and load/store.

Unstable dynamic targets can only be handled in a high overhead manner. Since it is not possible to exploit temporary stability, the server must instrument every load/store instruction with 15-20 instructions that emulate the load/store operation. The end of the emulation resolves the target of the target of the memory operation, and local tables may be consulted to determine if the target is present. If it is, the calculated address is modified to match the real location, and the load/store proceeds. If the target is missing, a trap is triggered and the server once again helps the client resolve the problem.

In addition, dynamic memory allocation (*i.e.*, `malloc`) is also translated into a trap operation. In brief, the server is fully responsible for managing all aspects of memory on the client to keep the entire system working correctly.

2.5 Motivating Applications

In the embedded space, there are two primary classes of applications that motivate this work: ubiquitous sensor networks and 3G cellular phone services. Although we propose the SoftCache as an enabling technique for embedded devices, the concept could be applied to other domains. We therefore describe several scenarios for application.

Ubiquitous sensor network. For such systems, the price point of each sensor node must be minimal. The ability to dynamically load new code into *motes* is critical to support changing needs in the environment. SoftCache provides a virtual workstation to the programmer, speeding development processes, and transparently runs the virtual application on the restricted sensor device.

Cellular phones. New features and service enhancements are constantly rolled out for cell phones. Rather than requiring re-flashing of entire applications — an inherently risky process — the SoftCache allows a micro-bootloader to be resident and load any application on demand. This not only allows for security patches, new applications, and similar features – it also provides a vehicle for pay-per-service on non-standard activities.

Network processors. When large corporations like Cisco write their switch software for a blade with 16 ports, they would like that same software to run seamlessly on 4 ports or 8 ports. Rather than spend precious developer hours revising applications and debugging one-offs, replete with the maintenance headaches, systems like the SoftCache can seamlessly handle the change in underlying hardware if coupled with domain specific knowledge.

Chip Multiprocessors. One emerging architecture for high performance systems is Chip Multiprocessors (CMPs). Processor powerhouses such as Intel, IBM, Broadcom, and AMD have unveiled their respective multi-core products. With 32, 64, or even 128 cores on a die, instruction sets become less relevant. Edge cores of a CMP can run SoftCache-like systems, and dynamically translate IA64, x86-64, ARM, MIPS, and SPARC all at the same time. This will enable a new generation of incredibly flexible systems, able to run any program from any platform with such dynamic translation systems. In the CMP model, the extremely high-speed on-board interconnect at a fraction of the power for network links makes an immediate advantage for SoftCache when compared to traditional caches. Internal cores are fed from edge cores, with specialized interrupt mechanisms passing chunks of code and data around as necessary.

3. EXPERIMENTAL RESULTS

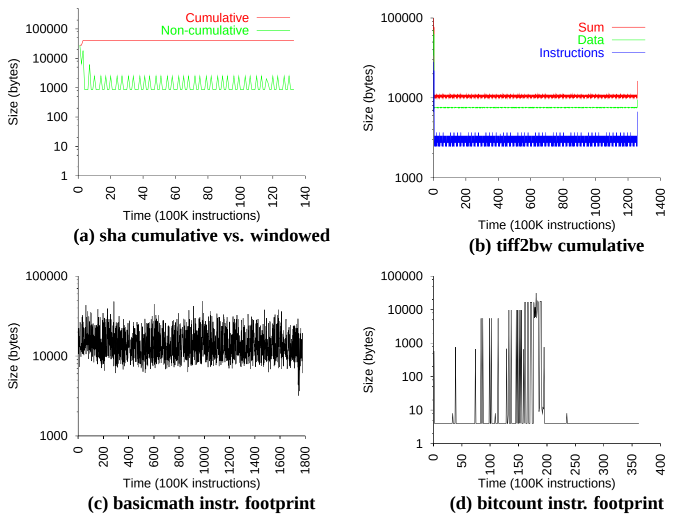

Figure 3: Instruction and Data Profiles.

With the motivation of industrial need for systems like the SoftCache, we now fully work out the power and delay issues associated with this design to determine the practicality of the system. There will be substantial impact from the additional instructions, delays and power due to network traffic, etc., that must be examined to validate the feasibility of SoftCaches.

We now systematically address each of the components in arguments raised against the SoftCache model by using a variety of techniques. Recent studies of network interface devices and DRAM options in energy-delay performance [3] provide insight into the issue of local v. remote storage with the SoftCache. The issue of cache overhead is addressed by comparing common hardware cache systems to our working implementation of a SoftCache described in section 2. To address the issue of energy consumption, we have constructed simulations of both typical hardware caches and a SoftCache. Using the results from these methods of analysis, we derive key energy delay results for the embedded system as a whole. Our analysis of these results show the SoftCache may be more effective than traditional cache designs for power consumption, while facilitating a variety of dynamic optimization and feedback mechanisms. Our SoftCache techniques currently run reliably only on instruction caches, and this analysis considers just instruction caches. Data caching is an ongoing effort.

3.1 Instruction and Data Profiles

The SoftCache is targeted for a size of approximately 64KB – the size required to replace the on-die caches of processors such as the SA-110 or XScale series, with combined I+D caches of approximately 64KB. By running with only 64KB of total on-die SRAM under the SoftCache, we can determine whether the working set of “real” applications can fit within this space constraint.

While an application may require more memory than is available in the SoftCache over its dynamic lifetime, for a given *window* of instructions, it only touches a small subset. Figure 3a illustrates this for the *sha* benchmark from MiBench. Even though the application quickly ramps up to a working set of about 40KB, for any dynamic 100K instruction window size, the application only accesses between 800 and 2400 bytes.

We observe that data accesses also exhibit this locality over a dynamic window of instructions. Figure 3b demonstrates that for the *tiff2bw* benchmark of MiBench, for any dynamic window of 100K instructions 12KB is sufficient to hold both instructions and data. The entirety of the instruction working set in *tiff2bw* is contained in 1.5-2.5KB of storage. The data used for the benchmark is contained in approximately 7.5KB, although reading input from large files may not be typical of real embedded applications. The trailing spike on this benchmark is the reporting of results at the end of the application.

Returning to instruction caching, Figure 3c and 3d show typical dynamic instruction activity patterns. For part (c), the benchmark *basicmath* of MiBench is clearly oscillating between 8KB and 30KB of storage. This is an artifact of the window size being 100K instructions. Larger window sizes reduce this oscillation, where the working set size settles to nearly 30KB. The key point of part (c) is that even aggressive code can be captured in as little as half of our SoftCache space, leaving the remaining for data. Part (d), the benchmark *bitcount* from MiBench, shows an unusual working set footprint. This application varies between a few bytes worth of instructions and 20KB, depending on the phase of the application. The bulk of execution, however, is dominated by very small instruction working sets.

These results suggest that having large external memories, such as low-power DRAM, may be wasteful with respect to power budgets and manufacturing costs. The SoftCache will dynamically maintain just the working set in on-die SRAM, avoiding this budget burden.

3.2 Power Analysis of SoftCache

Area savings also come from removal of other logic, such as the MMU (primarily TLBs), write buffers, cache control logic, and similar circuits. According to the publicly available data on the DEC/Intel SA-110 [8], the MMUs and write buffer occupy approximately 11% of the total die area. In addition, the cache tag array in the SA-110 was implemented using fully associative CAMs, which contain higher transistor counts (9/10/11T) than conventional SRAM implementations (6T). Using these transistors as local SoftCache memory would be more efficient.

In addition to the area reduction, the SoftCache system also has the advantage of lower power dissipation due to the removal of these hardware components. First, to understand how the SoftCache model alters the energy used within the cache, we consider the hardware cache structure and a corresponding SRAM used in both traditional processors and a SoftCache equivalent.

CACTI 3.2 [12], the de facto standard for cache models, generates energy and timing information for all components in a cache structure. The SRAM power generated by CACTI is based on just those components needed for SRAM operation: address decoding, wordline and bitline driving, senseamp output, and output driver. The CACTI cache structure report also includes the tag CAM cell matching and resolution logic. Typically, the SoftCache could operate faster without the additional hardware of tags, which slows down the timing.

With CACTI, we model varying cache sizes with 32-byte line sizes and 32-way associativity in 180nm, which is the XScale cache structure. Figure 4 presents our results. It clearly demonstrates the energy advantage of SoftCache due to the removal of the tag arrays and control bits in a conventional cache. The trend in this figure is that for small caches, approximately 5% of the power can be saved by removing tag logic. As the caches increase to 256KB, up to 8% energy is saved. For 64KB, the combined instruction and data caches of the XScale, the savings are approximately 6%.

For the MMU and write buffers, it is more difficult to measure without a real processor implementation, thus we use the published power data of SA-110 as a reference. As shown in [8], these units consume 19% of the total die power when running a computationally intensive program such as Dhrystone. In other words, by re-

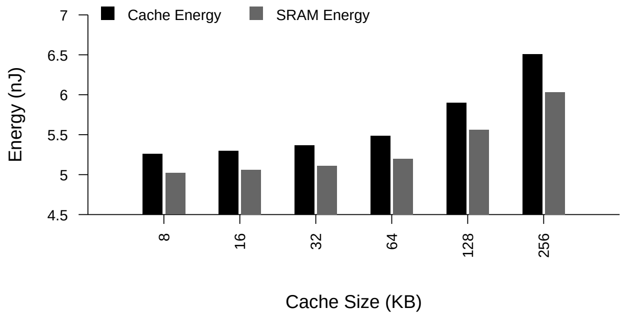

Figure 4: Power Comparison of SoftCache vs. Conventional Cache

moving these components and simplifying cache structure in a SoftCache system, a total of approximately 20% in energy savings can be achieved.

3.3 Local vs. Remote Storage

The SoftCache model seems counterintuitive since it suggests the reduction (or removal) of local storage (e.g. 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 local DRAMs are expected to be more energy efficient than any network.

Fryman et al. [3] demonstrated that this is not the case from a purely energy-delay standpoint. They compared low-power DRAM parts against a variety of network interfaces including Bluetooth, 802.11, etc. based on best-case DRAM parts compared to worst-case network parts. Network links are 10 to 100 times more power-consuming than accessing local memories. However, as the study indicated, keeping DRAMs active without accesses are 10 to 100 times more expensive in power than idling or sleeping network interfaces. We use their analytical model, and incorporate additional terms to track extra instructions executed in time and energy, as well as model the payload transfers during chunk loading/rewriting.

One of the key principles behind the SoftCache design is that there are several modes of operation, and changing modes is an infrequent event. Hence, if the time between mode switching is sufficiently long, the aggregate energy consumption of active- and sleep-mode by DRAM will exceed the active- and sleep-mode energy consumed by 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 analyzing the energy-delay benefit, with the answer given in section 3.5. Before this answer can be determined, we explore additional aspects of the problem.

3.4 SoftCache Overheads

Here we continue the comparison of a *best*-case DRAM solution against a *worst*-case network link solution. Consideration of the additional instruction overhead involved in SoftCache is ignored, as is the overhead of a hardware cache (tags, state bits, etc.). The two models being compared are (a) processor with hardware cache and local DRAM storage, and (b) processor with SoftCache and a network link.

The SoftCache needs to execute additional instructions to effect a hardware cache equivalent. These instructions come in two flavors: miss handlers, and penalty branches. We can compare these overheads 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. The time for the computation itself is denoted T_C. Assuming the worst-case scenario, the client executes roughly 100 instructions in a miss handler (in our

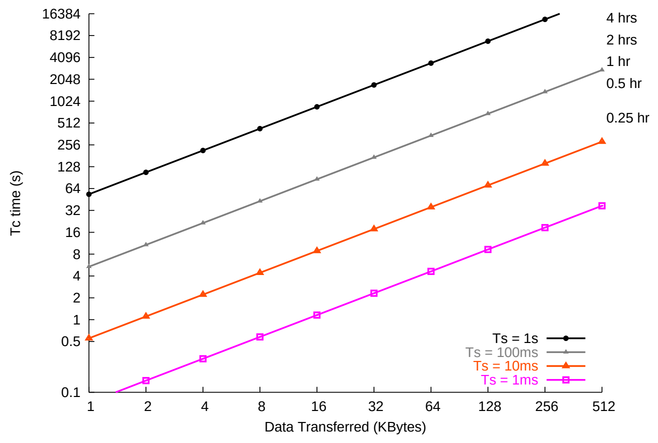

Figure 5: Duration of computation T_C that must pass for the network link to be more energy efficient, where T_S and B_N vary.

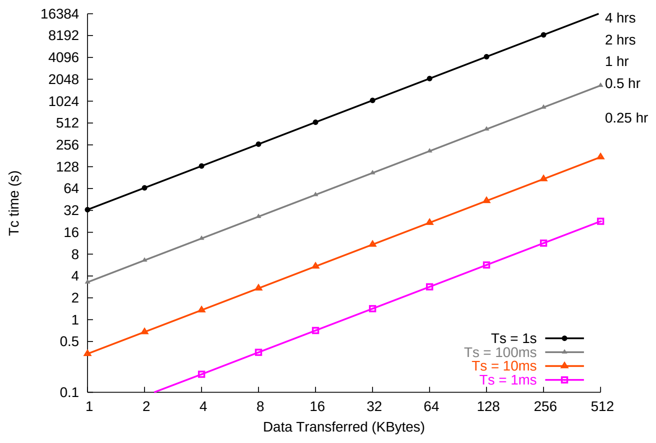

Figure 6: Duration of computation T_C that must pass for the network link to be more energy efficient, where T_S and B_N vary and branch penalty is removed.

UltraSPARC implementation) every time it needs to fetch another basic block.

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 energy required to receive the correct response. However, the transfer rate for the network is substantially slower than for DRAM. The additional time when 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, idle-mode reduces power during these times to 20mW [8]. 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 will be in the range of 0.25-0.35W. For our analysis we use the worst-case value of 0.35W.

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. In the following section, we combine all of these issues for penalties and power to demonstrate the viability of our SoftCache system.

3.5 Energy and Delay Implication

While prior discussion explains instruction overheads and network link usage, they do not 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 of this network model is in [3].

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 vs. SoftCache and network link) to be equivalent. Beyond this equilibrium point, the SoftCache is more energy efficient due to the differences in idle and sleep energy. That is, solving the equation $E_D = E_L$ (with T_C used to compute the total energy consumed during the mode) gives the average amount of time that must be spent in any given mode before changing.

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 5. 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 6.

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 device, thereby reducing sleep current to 0A [13]. This is possible only if the client initiates connections to the server, *i.e.*, the server cannot spuriously send commands to the client. This restriction 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 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.

Factoring in the rate of the network link, R_L, to create an energy-delay equation, we realize that the relatively slow speed of the network can force a tremendous impact on application performance. Figure 7 shows the effect of these additional considerations. This figure includes the original equilibrium values, marked as "EQ", and

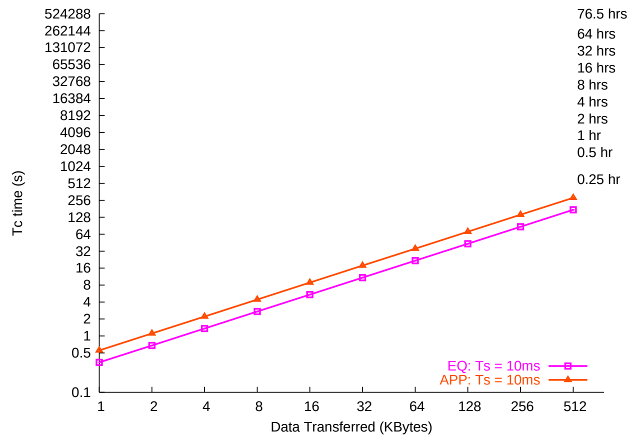

Figure 7: Comparison of time for equilibrium energy win (”EQ”) and energy-delay product win (”APP”) on application performance. T_S = 10ms.

the consideration for slowdown in the network affecting application run-time versus the original equilibrium point, marked ”APP”.

As Figure 7 illustrates, over slow network links applications can transfer 32KB every minute, and be more efficient than traditional designs. The high power and slow speeds of the network are the limiting factors in this analysis. For the proposed application of massive CMP-binary translation, both of these terms would improve significantly. For the embedded cell phone system, this suggests that loading a “new application” remotely, such as Mario Bros (∼128KB), is a better solution than loading it from local memory as long as the average time the game is played exceeds two minutes. On a smaller scale, transfer of 16KB pages from the network are more energy efficient than local DRAM after four seconds of use, and are more energy-delay efficient after 16 seconds.

4. RELATED WORK

Dynamic compilation can generate optimal instruction traces, yet requires sophisticated code cache management schemes [5]. Based on input that cannot be predicted statically, further dead-code path elimination becomes possible as done in Dynamo [1], as well as pointer disambiguation. Our techniques are equally applicable to both data and code, offering additional flexibility.

The Hot Pages system uses sophisticated pointer analysis with a compiler that supports transformations [9]. Shasta is a shared memory system that uses static binary rewriting to share variables among multiprocessors [14]. While the SoftCache could yield similar results, it offers more potential by exploiting dynamic program behavior.

Just-In-Time compilers with a distributed model of a JVM 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. Other efforts have focused on tuning the Java garbage collector systems to increase memory efficiency [2].

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 [7] 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 [16] explores a model of direct-addressing regions of the cache. It exposes the cache as directly addressable through additional registers. One benefit is that it allows variable line sizes with a minor penalty. The SoftCache also provides variable line sizes but involves a hardware reduction rather than addition. The ScratchPad [10, 11, 15] proposed a series of optimizations for data accesses, by adding a small on-chip RAM in addition to the hardware cache. To manage this on-chip memory, profiling compilers were used to determine the most-executed code and data, and then generating the necessary load-to-scratchpad and evict-from-scratchpad instructions. While relevant in one sense for the usage of on-chip memory, the SoftCache employs a truly dynamic method for using the on-chip SRAM and eliminates hardware caches.

5. CONCLUSION

We explored the goals and assumptions behind the explicitly software managed cache systems. Evaluating the SoftCache on the criteria of overhead storage cost, link vs. DRAM energy and speed costs, space, and total energy consumption, we find the SoftCache is a viable solution. A natural application class of such a system is an embedded device within an ubiquitous computing framework, but we have shown that the SoftCache may be a more generally applicable mechanism.

We demonstrated that the SoftCache system is energy efficient when compared to local storage options, and after considering the delays associated with network transfers, is still an energy-delay product winning solution for some applications. We reduce die power by approximately 20%, while cutting die area by approximately 10%. Transfer of 16KB pages from the network are more energy efficient than local DRAM after four seconds of use, and are more energy-delay efficient after 16 seconds. Modern embedded devices tend to use faster network links and more DRAM parts than analyzed here, which makes the SoftCache even more energy efficient.

6. REFERENCES

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

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

[3] J.B. Fryman et al. Energy Effi cient Network Memory for Ubiquitous Devices. In *IEEE MICRO*, Sep/Oct 2003.

[4] D. H. Friendly, S. J. Patel, and Y. N. Patt. Putting the Fill Unit to Work: Dynamic Optimizations for Trace Cache Microprocessors. In *MICRO-31*, 1998.

[5] K. Hazelwood and M.D. Smith. Code Cache Management Schemes for Dynamic Optimizers. In *Sixth Annual Workshop on Interaction between Compilers and Computer Architectures*, 2002.

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

[7] S. Kim, N. Vijaykrishnan, M. Kandermir, A. Sivasubramaniam, and M.J. Irwin. Partitioned Instruction Cache Architecture for Energy Effi ciency. In *ACM Transactions on Embedded Computing Systems*, June 2002.

[8] J. Montanaro and et al. A 160-MHz, 32-b, 0.5-W CMOS RISC Microprocessor. In *IEEE JSSC*, volume 31, No. 11, pages 1703–1714, November 1996.

[9] 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.

[10] P.R. Panda and N.D. Dutt. Memory Architectures for Embedded Systems-On-Chip. In *High Performance Computing*, December 2002.

[11] P.R. Panda, N.D. Dutt, and A. Nicolau. Effi cient Utilization of Scratch-Pad Memory in Embedded Processor Applications. In *European Design and Test Conference*, March 1997.

[12] P.Shivakumar and N.P. Jouppi. CACTI 3.0: An Integrated Cache Timing, Power, and Area Model. Technical report, Compaq WRL, August 2001.

[13] K. Roy and S. Prasad. *Low-Power CMOS VLSI Circuit Design*. Wiley-Interscience, USA, 2000.

[14] D. J. Scales, K. Gharachorloo, and C. A. Thekkath. Shasta: A Low Overhead, Software-Only Approach for Supporting Fine-grain Shared Memory. In *ASPLOS-7*, pages 174–185, 1996.

[15] M. Verma, S. Steinke, and P. Marwedel. Data Partitioning for Maximal Scratchpad Usage. In *ASP-DAC*, 2003.

[16] E. Witchel, S. Larsen, C.S. Ananian, and K. Asanovic̀. Direct Addressed Caches for Reduced Power Consumption. In *MICRO-34*, 2001.