SoftCache Architecture

A Thesis
Presented to
The Academic Faculty

by

Joshua Bruce Fryman

In Partial Fulfillment
of the Requirements for the Degree
"Doctor of Philosophy"

College of Computing
Georgia Institute of Technology
August 2005

SoftCache Architecture

Approved by:

Hsien-Hsin S. Lee, Co-Advisor
Electrical and Computer Engineering
Georgia Tech, Advisor

Umakishore Ramachandran, Co-Advisor
College of Computing
Georgia Tech

Santosh Pande
College of Computing
Georgia Tech

Kenneth M. Mackenzie
College of Computing
Georgia Tech

Karsten Schwan
College of Computing
Georgia Tech

David E. Schimmel
Electrical and Computer Engineering
Georgia Tech

Date Approved: 18-July-2005

DEDICATION

It's the nature of questions to multiply.

Joshua Fryman, 1992

Everything's a production.

Tiago Stock, 1996

This work is dedicated to my wife, Hathai Sangsupan, who has been loving and supporting despite the incredible pressure this has brought to bear on our lives. This work is also dedicated to Brieana Fryman, our wonderful daughter who can always bring a smile to our faces, even during the roughest of times.

ACKNOWLEDGEMENTS

There are countless people I must thank for their support and advice during the years of my graduate studies, yet there are a few who deserve special mention for their extraordinary friendship and advice. In no particular order, they are:

- Ken Mackenzie, who took a chance on a high-risk student and inspired atypical thinking.
- Kishore Ramachandran, who always had friendly advice and encouragement to keep going.
- Sean Lee, who brought an encyclopedic knowledge of related work, implementation details, and insight into any discussion.
- Karsten Schwan, who managed to provide excellent contacts to other people and industry while also securing new equipment for the lab.
- Dave Schimmel and Santosh Pande, for always listening to random ideas and providing excellent suggestions.
- Chad Huneycutt, Peter Sassone, Ivan Ganev, Austen MacDonald, Craig Ulmer, Adam Johnson, and the rest of *Arch-Beer*, good friends who always had time for a serious discussion or just plain procrastination over dinner somewhere.
- Neil Bright, Dan Forsyth, and the rest of CNS, who kept the IT infrastructure running and put up with my intense interference and sometimes unwarranted animosity when things stopped working.
- Philip and Osnat Teitelbaum, for pushing me to go for a graduate degree at all.
- Hathai Sangsupan, who has supported me through thick and thin and even consented to marry me despite crazy hours and untold all-night hacking sessions.
- Brieana Fryman, who always can make her daddy smile.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER I

SUMMARY

Multiple trends in computer architecture are beginning to collide as process technology reaches ever smaller feature sizes. Problems with managing power, access times across a die, and increasing complexity to sustain growth are now blocking commercial products like the Pentium 4. These problems also occur in the embedded system space, albeit in a slightly different form. However, as process technology marches on, today's high-performance space is becoming tomorrow's embedded space. New techniques are needed to overcome these problems.

In this thesis, we propose a novel architecture called SoftCache to address these emerging issues for embedded systems. We reduce the on-die memory controller infrastructure which reduces both power and space requirements, using the ubiquitous network device arena as a proving ground of viability. In addition, the SoftCache achieves further power and area savings by converting on-die cache structures into directly addressable SRAM and reducing or eliminating the external DRAM.

To avoid the burden of programming complexity this approach presents to the application developer, we provide a transparent client-server dynamic binary translation system that runs arbitrary ELF executables on a stripped-down embedded target. The drawback to such a scheme lies in the overhead of hundreds of additional instructions required to effect cache behavior, particularly with respect to data caching. Another substantial drawback is the power use when fetching from remote memory over the network. The SoftCache comprises this dynamic client-server translation system on simplified hardware, targeted at Intel XScale (ARM) client devices controlled from Intel x86 servers over the network.

Reliance upon a network server as a "backing store" introduces new levels of complexity, yet also allows for more efficient use of local space. The explicitly software managed aspects create a cache of variable line size, full associativity, and high flexibility. This thesis explores these particular issues, while approaching everything from the perspective of feasibility and actual architectural changes.

Specifically, this thesis includes the macroblocks to make the SoftCache work, including an instruction-cache based client-server dynamic translator; a data-cache extension; a power, delay, and space analysis; and an analysis of different applications and their implications in this system.

The novel contributions of this thesis to the existing body of research are briefly summarized as:

- A distributed client-server dynamic binary translator/rewriter, a framework that is used to implement the SoftCache system in a ubiquitous network environment
- A novel memory characterization based on target address enumeration, a technique that facilitates solving the previously intractable problem of software emulation for data caching
- A novel energy-delay study comparing remote network accesses to local main memory, indicating that local memory is less than ideal

These techniques will partially relax the problems faced with current and next-generation designs. Reduction of the logic required for memory controllers and simplification of on-die memories will shrink die size and reduce latency effects. Application growth and corresponding complexity is transparently solved with dynamic binary translation, and manufacturing cost for the embedded domain is reduced with simpler device development models.

CHAPTER II

INTRODUCTION AND MOTIVATION

Embedded consumer systems continue to add more features while shrinking their physical device size. Current 2.5/3G cell phones incorporate 144kbps or better network links, offering customers not only phone services but also e-mail, web surfing, digital camera features, and video on demand. With feature expansion demanding additional storage and memory in all computing devices, densities of DRAM and Flash are increasing in an attempt to keep pace. This continuous storage expansion translates to a growing power dissipation, temperature, and battery drain.

To reduce energy effects and increase battery life, designers use the smallest parts and lowest part count possible. These design practices have the added benefit of keeping manufacturing costs down. This effort to minimize available resources works against application feature expansion and device flexibility for dynamic upgrades.

In an attempt to address some of these problems, companies such as 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 [63].

One avenue for power savings has not been fully considered, however. Many embedded devices, and all mobile devices, have a network link (GSM, Bluetooth, Ethernet, etc.) into a larger distributed environment. After designers incorporate sufficient power to support a network link, they then attempt to minimize use of the link due to its excessive energy needs during activity. Products therefore incorporate all the needed local storage in the device, buffering as much as possible to avoid retransmission. This ignores the fact that the remote server has a much less restricted power budget, and can be made arbitrarily fast to handle requests quickly.

For ubiquitous always-on devices like 3G cell phones, there is the potential to use the network link as a means for accessing applications remotely. Such network usage could reduce local storage space, thereby reducing energy demands on the mobile platform. This remote memory could lie in a remote server, or simply be cached within the network infrastructure.

Utilizing the network link to access remote memory can provide a more energy efficient solution than traditional local memory. Traditional designs assume that the additional cost of utilizing the network link for moving code and data will far outweigh any benefit of removing or reducing local storage. The common misconception assumes the network is in use constantly, and therefore is much more power consuming than local storage. This is not always the case, as we will demonstrate. The best low-power mobile DRAM available today is 100 times less expensive to access in terms of energy per bit than a very low-power Bluetooth network. But for these same parts, the sleep-mode current of the Bluetooth network module is 10 times less expensive than the DRAM part. Therefore, if sufficient time elapses between accesses, the network link is more power efficient than local DRAM.

2.1 Design Space

In the embedded space, there are two basic classes of motivating applications. The first class comprises ubiquitous sensor networks, where the price point of each sensor device must be minimal. The ability to dynamically load new code into such "motes" is critical to support changing needs in the environment. The SoftCache system we will propose provides a virtual workstation to the programmer, speeding development processes, and transparently running the virtual application on the restricted sensor device. The second class, cell phones and PDAs, constantly rolls out new features and service enhancements. 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 stripped-down approach allows for security patches, new applications, and a vehicle for pay-per-service on non-standard activities (*e.g.,* unlock the car you have been locked out of).

One step higher, the SoftCache technique may be applicable to more complex devices such as 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.

A much more aggressive area of application lies in next-generation high-performance microprocessors. Rather than have one large Pentium 4 or equivalent, the future is moving toward massive CMPs of simplified cores. With 32, 64, or even 128 cores on one die, it becomes irrelevant what instruction set the machine uses. Edge cores can run SoftCache like systems, and dynamically translate IA64, x86-64, ARM, MIPS, and SPARC all at the same time. This would 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 schemes like the SoftCache when compared to traditional caches. Inside cores are fed from edge cores, with specialized interrupt mechanisms passing chunks of code and data around as necessary.

2.2 Thesis Goals

In this thesis we explore an alternative mechanism for caching instructions and data near the central processing core to address these problems, without significantly impacting program performance. Such a solution hinges on the concept of removing the hardware control over cache storage, thereby exposing the cache memory directly to software management. Rather than burdening the programmer with the mechanics of manual memory management as attempted in the past (*e.g.,* with overlays), an automatic translation or rewriting system demonstrates transparent support for such a software managed cache.

The concept of a software managed cache fits within the design space of other memory management options when all are evaluated on a few key criteria, as suggested by Table 2.2. Of increasing importance is the issue of power dissipation, with modern caches consuming tens of Watts. Related to power slightly, yet closer to cost and speed of access, is the size of the memory storage space. Caches have substantial overhead in their hardware design, whereas RAM does not require tags, valid bits, etc. Regardless of design, a major concern is how easy it is for programmers to use. Any system that increases complexity for programming should be immediately rejected, particularly when software maintenance is approximately 90% of total lifetime software cost. A trade-off exists, however, when programmers use inflexible tools that disallow optimizations which could significantly impact performance, such as data pinning. Our qualitative perspective on these criteria

is shown in Table 2.2. While a software managed cache will address the topics sketched here, it also introduces new problems in the overall system which require additional evaluation.

Table 1: Placing software-managed caches in *the overall design space*.

criteria	**hardware cache**	**software cache**	**virtual memory**	**manual overlays**
power	bad	good	bad	better
size (space)	bad	better	bad	good
programming	better	good	good	bad
flexibility	bad	better	good	bad

In this thesis, we implement and evaluate a software managed cache system. Using the top three criteria from Table 2.2 – power, layout size, and programmer perspective or ease – we systematically examine a SoftCache implementation.

As a typical example of an embedded processor, the StrongARM-110 is a low-power application-flexible device. This processor, originally developed by DEC, is now part of Intel's intellectual property and the basis for their entire XScale product line. The evaluation of the SA-110 microprocessor by its design engineers showed that the entire cache subsystem (storage, tags, MMU, and write buffers) accounts for 62% of the total die power consumption[75]. This power consumption was spent in half of the chip area and roughly 65% of the 2.5 million transistor budget. These observations imply that, by Amdahl's Law[4], the most gain can come from significantly cutting the size (and power) of the cache related systems.

Based on the results from the ARM analysis, it is apparent that a significant amount of total power consumption is spent on the hardware support features of the cache. This suggests that removing the hardware control of caches by placing the cache storage area under software management may result in substantial power savings. Redesigning a cache system to support software management explicitly is a non-trivial task. Such a design is the primary focus of this work.

Caches and automated memory management systems (like virtual memory) evolved, however, to facilitate easier programming models. Many would consider it a step backward to remove the automated hardware support for caching, thereby placing the burden of memory management back on the programmer. We therefore propose to achieve software management over the cache memory space by an automated *binary translation* mechanism which will transparently manage the cache

storage space. Such a system provides the benefit of transparent operation to the programmer while also reducing power consumption through removal of hardware support for virtual memory and caches.

A potential drawback to such a solution is that software is generally slower than hardware. A major objective of this thesis discussion is to demonstrate that such a software managed cache system will not be significantly slower than hardware. A further obstacle is the use of network infrastructure for accessing remote memory. Conventional wisdom is that using a network is far more expensive than using local storage, and we propose to explore this area carefully.

2.3 Thesis Organization

This thesis is organized into five primary chapters, with a concluding chapter and two appendicies. The conceptual organization and behavior of our SoftCache system is contained in Chapter 3. These conceptual basics are extended by introducing a working instruction cache replacement in Chapter 4, followed by a working data cache replacement in Chapter 5. The implementation details of a working SoftCache present many technical challenges for proving the feasibility of our system. Chapter 6 explores the analytical underpinnings of our proposed system and uses typical hardware datasheets to derive initial answers. To translate the analytical results to real hardware, Chapter 7 explores an Intel PDA reference design platform for true power consumption and network delays. Finally, Chapter 8 concludes the primary thesis material and presents future research suggestions.

Appendix A is a detailed explanation of proper hardware measurement and analysis for real systems. This appendix also contains error estimations for different pitfalls and limitations of measurement tools available. It is a reference to aid in proper power studies of hardware. Appendix B is a detailed set of results for both the instruction and data caching behaviors of the SoftCache system for the MiBench embedded benchmark suite.

CHAPTER III

SOFTCACHE ARCHITECTURE

This chapter introduces the overall organization of the SoftCache system, demonstrating the various behaviors or modes of operation. Later chapters will present detailed examples of the actual dynamic binary translation system.

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 hardware simplification may reduce the power and area requirements, leading to longer operation hours and lower cost.

3.1 SoftCache System Overview

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. This thesis focuses only on the ARM/XScale implementation. 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. Dynamic data allocation, completely managed by the server, is discussed in Section 3.4.

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. On the client, 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.

3.2 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 annotated to support fast rewriting for the target client. To facilitate this translation process, the server also maintains a shadow copy of the client's local memory. As blocks are rewritten on demand, they are copied into the shadow copy and then *patches* are sent from the server to the client to update the client memory as needed. Additional details of the static analysis and partial evaluation are in Section 4.2.

3.3 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 3.3. As illustrated in Figure 1, the client begins by activating its interface block, that 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

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

`main()` and returns it to the client with a patch_memory command. This transaction is immediately followed by a resume_execution command. Once the initial block is loaded, execution begins.

Table 2: SoftCache Interface Commands

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

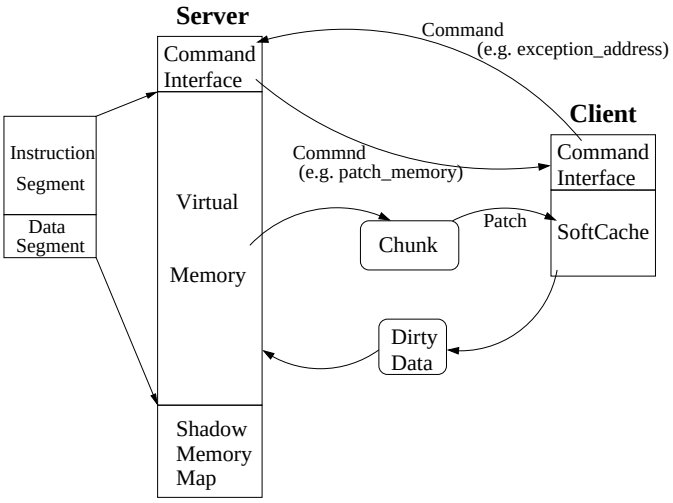


Figure 1: Execution Model of a SoftCache System.

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 path-resolving exception 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* that maintain a copy of the client memory

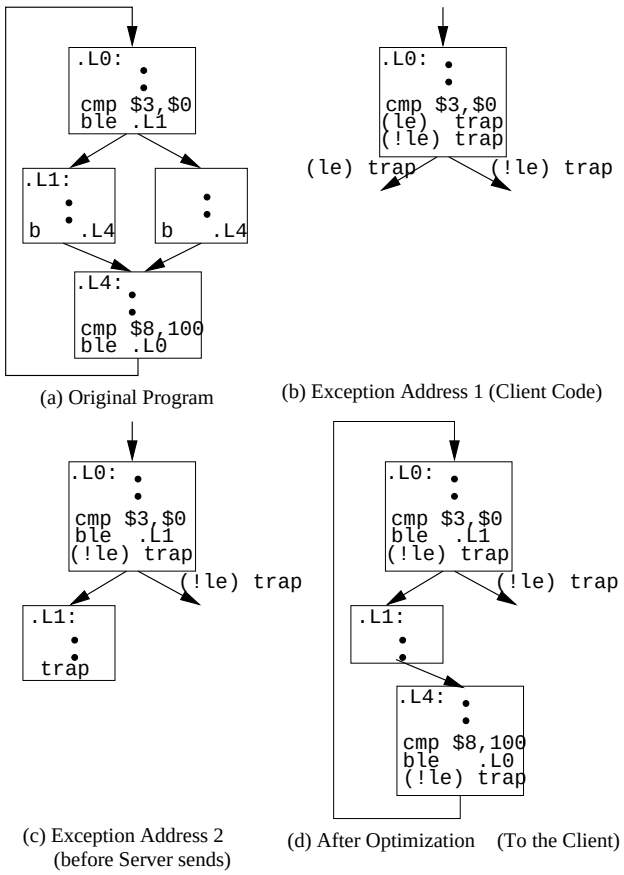


Figure 2: Dynamic Execution of a Client in SoftCache.

allocation map to determine the block to load for a 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 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 an optimization by eliminating the branch instruction during patching, a technique similar to trace construction using the fill unit in high performance processors [43]. 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. Additionally, based on the ARM instruction set limited range for immediate offsets with load/store instructions, it becomes possible to periodically eliminate some instructions as discussed in Section 4.2.

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, for those instruction chunks that are evicted, trap instructions need to be patched back into all predecessor basic blocks, replacing all branches to the now invalidated region. Similarly the stack has return addresses that require invalidation to prevent unwinding the call stack into an invalidated region.

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 that runs at maximum performance, with the drawback of extremely high miss latency.

3.4 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 that 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

[2]Our current implementation employs a FIFO invalidator. Other, more intelligent, algorithms could be used in future work to further improve locality.

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 large 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, the trap is executed which eventually reports to the server that the instruction is 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 up to 15 instructions that emulate the load/store operation. The end of the emulation resolves 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.

Complete details of data caching support are in Section 5.

3.5 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. The SoftCache provides a virtual workstation to the programmer, thus speeding development processes, and it 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, ARM, and AMD have unveiled their respective multi-core products. With a future aim of 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.

3.6 Related Work

This section explores how the SoftCache concept is related to prior and ongoing research work in many areas. While the SoftCache may use techniques from the areas discussed, in several areas it

does not directly use the concepts presented. Instead, such areas of related work are tools that need to be or would be built from the potential advantages the SoftCache presents to applications.

Given that caches are such pervasive units in modern computer architecture designs, substantially changing how the cache is implemented of necessity touches on a wide body of indirectly- and directly-related work.

While the techniques presented in this thesis for building the SoftCache are currently done in a post-compiler fashion, it is clear that they could be implemented inside a modern compiler. A compiler-based version of the SoftCache likely would be more impressive with its results due to more precise analysis of the source code and knowledge of the generated binaries.

3.6.1 Overlays, Paging, and Caching

These three topics – overlays, paging, and caches – all attempt to answer the same problem: running a large program in a limited memory space efficiently. The SoftCache, being a novel cache design, is another solution in this fundamental problem space.

Initial efforts to run large programs in memory-limited systems resulted in varied implementations, mostly in software, of program *overlaying* (also known as *segmentation* or *folding*). Here, the programmer is burdened with manual partitioning of program and data into regions that can be replaced or overlaid at critical points of program execution. As programmers were encouraged to use high-level languages to focus more on abstract problem solving, the mechanics of managing low-level details of the computer hardware were hidden in languages and compilers[112, 93, 28, 29]. The complexity of such management detracted from goals of simpler programming models and machine independence (*e.g.,* the ability to run a program on the same machine with differing memory configurations).

Virtual memory systems, a mechanism to support automatic paging of needed instructions and data, were an attempt to redress this issue. The first working virtual memory system with automatic paging can be traced back to the Atlas Computer in 1961-1962 [68]. While Denning [28] recalls arguments for either manual- or automatic-memory management, Sayre [93] demonstrated that automatic management of memory via paging was equal to (or superior than) manual management for inexperienced programmers and/or a significant class of applications. This observation resulted

in a move to support automatic memory management via virtual memory, away from programmer-managed overlays, even though proficient programmers could outperform automatic systems.

With the growing rise in speed differences between a large main memory, a secondary memory system such as disk or tape, and the central processing core, arguments were presented for a smaller, faster memory located near the core [104]. This memory could be automatically managed, as Wilkes described in depth [111]. What Wilkes called *slave memories* IBM adapted and renamed *cache memories* and the widespread adoption followed soon thereafter. With the continuing growth of the processor-memory performance gap [55], the utilization of cache memories as a tool in a memory hierarchy has become increasingly important. As the memory-processor gap has increased, some have begun calling for software management of L2 caches [48].

While the direct management of memory fell out of favor with the advent of virtual memory, a large class of systems were later built without virtual memory – early Intel processors such as the 8086 and 80286, or the Motorola 68000 and 68020. Such devices, lacking hardware mechanisms for automatic paging by an operating system, required programming languages and models that supported the classic model of the overlay. Even today, with DSPs such as the ASPS 812x, languages and compilers support directives for program overlays.

Typically such devices are used in embedded platforms or simpler systems such as DSP filter engines. Even though the programmer is using a high-level language, she must still concentrate on partitioning the application program and data into blocks that can be switched in and out of the primary access memory. However, Sayre [93] observed that manual management of memory by skilled programmers can be 20% more efficient than automatic systems.

The key to such savings comes from the idea of *working sets* as presented by Denning [27]. Denning and Sayre, as well as others, showed that by varying the page size, more efficient utilization of memory was possible (along with replacement policies and placement policies). Smaller page sizes yield a more precise dynamic footprint of an application to just those instructions and data that are necessary for steady-state program execution. As Denning [28] noted, "*small page sizes permit a greal deal of compression without loss of efficiency.*" The drawback to small pages is the cost overhead of managing page tables in computational effort and memory storage.

The SoftCache system presented in this work proposes to use the mechanism of binary rewriting

to achieve the same, or better, efficiency of paging in a hardware cache by software control. The translation mechanism in the rewriting may keep the ISA of the original application, but will rewrite the program to perform manual memory management in a small, on-chip SRAM region. Such a system provides a programmer-transparent paging policy with a highy flexible manual management system.

3.6.2 Binary Rewriting and Program Analysis

Binary rewriting or translation is one result of program analysis. Using the application source code as a basis, compilers can 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 [21, 103]. Such analysis, while beneficial, fails to capitalize on possible optimizations that can only be determined *dynamically* with feedback from how the application is behaving with live data.

Introducing dynamic feedback and translation provides a method of obtaining even more refined program optimizations. Dynamic compilers can generate optimal instruction traces, which may require sophisticated code cache management schemes [54, 53]. Based on input which cannot be predicted statically, further dead-code and -data path elimination becomes possible [6], as well as pointer disambiguation. Additional benefits can be found by rearranging the code or data layouts based on dynamic performance for more optimal accesses in cache or main memory [80, 89]. Repeated application of such procedures can lead to successive program reduction that is quite substantial.

However, the SoftCache operating by mechanism of dynamic binary rewriting does not have the benefit of full knowledge of program source code. Rather, 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 [57, 11, 2, 99, 91, 100, 79], less has been done directly on binary systems. Most “binary” analysis systems explored to date mandate certain restrictions, such as availability of higher-level assembly [18, 19, 20, 24, 67] code or compiler-generated intermediate-representation, while others place limitations on jump instruction types or pointer ambiguity. A few

toolkits have been released that support regeneration of this information if only the compilers and libraries of the toolkit are used [15].

The power of these static and dynamic techniques have been used in other projects similar to our SoftCache ideas. The Hot Pages system uses sophisticated pointer analysis with a compiler that supports such transformations [76]. Shasta is a shared memory system that uses static binary rewriting to share variables among multiprocessors [94]. While the SoftCache can yield equivalent results, it offers more potential by use of dynamic program behavior resolution.

Many simulators also use binary rewriting in varying forms to achieve faster results compared to strict interpretation simulators. Such systems as Talisman-2 [10], Shade [22], and Embra [113] simulators 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, such as those supporting Java, with a distributed model of a JVM [101] also have some commonality with the ideas behind the SoftCache. JIT 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 work with dynamic techniques that the SoftCache could capitalize on focus on removal of redundant computations [13, 74]. Such systems depend on dynamic patterns to emerge and feedback systems to isolate and replace redundant work with inlined results from earlier computations.

CHAPTER IV

SOFTCACHE SUPPORT FOR INSTRUCTION CACHING

Our work, documented in this thesis, centers around the ARM prototype implementation of the SoftCache system. In this chapter, we explore the challenges and solutions to support a robust instruction-caching based system that implements the SoftCache Architecture. This infrastructure is expanded in the next chapter to support data caching as well. This chapter concentrates solely on instruction caching, and particular problems that are encountered to support a SoftCache framework. Specific examples of the dynamic translation are presented along with benchmark results.

4.1 Design Issues

We found several limitations of the ARM architecture which hampered the conceptual design for the SoftCache client-server system. While we discovered these problems on the ARM platform, other embedded targets may have some or all of these issues in addition to their own peculiarities.

The first challenge lies in the method of accessing data variables. Intermixed within the instruction stream are the addresses of data or bss segment variables or constants. An example of this is shown in Listing **??**. This has two interesting side-effects: (1) the same cache line will appear in both the instruction and data caches, a less than optimal situation; and (2) without a complete control- and data-flow reconstruction, it is impossible to distinguish instructions from data. To work around this embedded data-in-text problem, there are two solutions: (a) make a code chunk constitute an entire function and any associated data constants within it, or (b) perform control- and data-flow reconstruction on the arbitrary ELF image. While our earlier work used solution (a), in this thesis we explore solution (b). Details of the reconstruction are in Section 4.2.

The second challenge comes from a complete lack of consistency between instruction and data caches, as well as the write-back buffer. Therefore, any time we need to modify the on-chip memory, we flush the caches and buffers with software routines. This results in a substantial penalty every time we must modify the client system. Some Intel XScale devices do have non-ARM instructions

Listing 4.1: A very simple C function and the ARM assembly output from gcc. To load register r0 with the string reference, it actually loads a constant embedded in the text segment which the instruction stream branches around.

```
1  int main( void ) {
2      printf("Hello World!\n");
3      return 42;
4  }
5
6
7  section   .rodata
8      .align    2
9  LC0:
10     .ascii "Hello World!\012\000"
11 section                .text
12     .align    2
13     .global   main
14     .type main,function
15 main:
16     stmfd sp!, {lr}
17     ldr    r0, .L3
18     bl printf
19     mov    r0, #42
20     b   .L2
21     .align    2
22 L3:
23     .word .LC0
24 L2:
25     ldmfd sp!,{pc}
```

for flushing the cache, but not all ARM devices support this type of operation. Ultimately this issue will be irrelevant as we will not have caches present in our hypothetical hardware.

The third challenge lies in the encoding format for all ARM instructions. Every 32-bit encoded instruction has a conditional leading nibble – that is, the first four bits indicate a conditional evaluation for *every* instruction. These conditions include the normal types (*equal, greater than, less than or equal*, etc.) as well as *never* and *always*. The end result is that from a control-flow perspective, *every* instruction is potentially a conditional branch to itself or the following instruction. This greatly expands the complexity of reconstructing control-flow information.

The fourth and last challenge is that in the ARM, the PC is a working register. Any instruction can read or write to the PC by specifying `r15` as the corresponding register to read/write. Therefore, every instruction must have its operands evaluated to ensure that the PC is not involved.

4.2 Static Analysis

There exists no prior work on reconstruction of control- and data-flow information from arbitrary ELF images. Prior work in this area has focused on two main strategies: dynamic compilation, or custom toolchains and binary images. The dynamic compilation group uses the source code directly to generate control- and/or data-flow information, emitting via dynamic compiler the actual machine code as needed. Other methods in this category use debugging symbols in the binary image to correlate back to the original program source, where the flow calculations are carried out. The other primary solution, using custom toolchains, requires the binaries to be constructed from modified compiler toolchains. These modifications generally embed within the binary image the flow information determined by the compiler, such that by using a provided (generally closed-source) library, the binary image can be evaluated during execution. Additional details for these types of prior work are in Section 4.5.

Therefore, by implementing a control- and data-flow reconstruction system for arbitrary ELF images for ARMv4 targets, we have a novel solution that fully supports existing programs without requiring recompilation. Moreover, our solution can easily be expanded to translate any ARM instruction set into some other ARM instruction set (*i.e.,* ARMv4 to ARMv3, or the converse). This provides our primary objective of compatability while also enabling the typical range of optimizations that dynamic compilers or dynamic translators offer.

The following sections detail how we reconstruct the control- and data-flow information from arbitrary ELF images.

4.2.1 Seeding and Simulating

The basic algorithm of flow reconstruction is trivially simple. However, as with all simple concepts, the devil lies in the details of implementation. A basic pseudo-code algorithm for the main flow generation is in Listing 4.2.

The analysis starts by reconstruction of the control-flow graph (CFG) information from the ELF image. The premise is to seed the initial link register (LR) with a bogus value that will be stored in the PC upon program completion. For the ARM, a bogus PC is any value ending with the least significant bit set, as no odd-address instruction is tolerated. In “thumb” mode, instructions

Listing 4.2: A basic algorithm for control-flow reconstruction.

```
1  Registers[ LR ] = 0xDEADBEEF                    // seed return PC
2  Registers[ PC ] = ExtractEntry( ELFimage )      // set initial PC
3  First = Registers[ PC ]                         // set basic block start
4  while (Registers[PC] != 0xDEADBEEF)             // until the bogus PC..
5  {
6     // first, control-flow
7     Evaluate( Registers[PC], &Next, &Alt )       // get possible outcomes
8     Mark( Registers[PC], FLAG, HIT )             // flag this PC as seen
9     Mark( Registers[PC], CFG1, Next )            // indicate control flow
10    if (Alt)                                     // ... Alt is only set
11    {
12       PushQueue( Alt )                          // ... if we see branches
13       Mark( Registers[PC], CFG2, Alt )          // ... note secondary
14    }
15    if (Next != PC+4)                            // exiting basic block?
16    {
17       SetBasicBlock( First, Registers[PC] )     // ... store basic block
18       First = Next                              // ... reset start point
19    }
20
21    // next, data-flow
22    if (LoadStore( Registers[PC] )               // Is this a DFG op?
23    {
24       Addr = EvalLoadStore( Registers[PC] )     // ... extract target
25       Mark( Registers[PC], DFG, Addr )          // ... store target
26    }
27
28    // last, set the PC and move on
29    if (Next != 0xDEADBEEF)                      // if not set to exit...
30       Registers[ PC ] = Next                    // ... keep going
31    else                                         // otherwise
32       do
33       {                                         // ... check Q; empty Q
34          Next = PopQueue()                      // ... returns 0xDEADBEEF
35       } while (MarkTest( Next, Flag, HIT ))     // have we seen this PC?
36 }
```

are two bytes, and in normal ARM mode, instructions are four bytes. Here, the value used is $0xDEADBEEF$. This value also makes it easy to track the LR as it occurs in the stack.

Once the LR is set to a seed, the initial PC is set to the entry point of the ELF image which is defined in the ELF header. Our SoftCache system ignores the typical setup of the *bss* and *data* segments contained in the C Run-Time (CRT) support routines, and instead emulates these actions internally.

As each instruction is in turn evaluated to see what the next PC should be set to, it becomes necessary to capture two possible next PC choices. In the presence of conditional branches, the next PC could be the taken *or* the not-taken case. Both eventualities need to be compensated for,

so the path that leads to a next PC other than PC+4 is pushed into a queue of pending evaluations. Also during this stage, the conditional nibble of every instruction is checked to evaluate whether this instruction should have an impact. While this conditional nibble is something the SoftCache framework supports, in common practice only branches tend to use the leading conditional nibble.

To ensure that we skip already visited instructions, each instruction is decorated to indicate a visited state. Later, when dequeuing pending PC targets from prior conditional instructions, each is tested to ensure that the queued PC was not previously handled during normal execution.

The end result of this iterative process is a visit to every *instruction* in the text stream that can be reached. This automatically performs the equivalent of dead-code elimination, and reduces statically-linked multi-megabyte images to a small fraction of their initial size.

However, this technique fails under certain conditions. In specific, with object-oriented programs and the *vtable* configuration, or with rewritable arrays of function pointers, it becomes impossible to know exactly where control can move to. With the ARM design of the PC treated as a working register, the ease of changing the PC becomes a secondary cause of unknown control-flow change and is similarly unpredictable. For those cases where it is not possible to disambiguate the change occuring, a special value is pushed into the decoration indicating that an unknown shift is occuring.

During the translation stage, when an instruction decorated with an unknown control shift is encountered, it is replaced with a trap instruction matching the same conditional nibble. The client then *emulates* the control shift to determine what target is being accessed, before passing this information back to the server for processing. The server has enough knowledge to determine whether this shift is a one-time determination of a fixed path, or truly variable such that every transition must be checked. In either case, the server will update the client as necessary before signalling for execution to resume.

The construction of the data-flow graph (DFG) decorations is similar in nature to the control-flow graph construction. As an iterative process, the DFG reconstruction simply looks for load/store operations and then decorates each such operation with the data target if it is possible to predetermine. For ambiguous targets, a special decoration is attached to cause the translation system to emit emulation instructions as opposed to allowing the load/store to operate. This is the corresponding

action for an unknown control shift applied to the data flow behavior.

4.2.2 Tracking the Stack

In addition to the DFG reconstruction, the stack in use will grow with the function call graph. As the function calls grow, it becomes possible to exhaust the on-die memory reserved for the stack. Therefore, each basic block is annotated with how many bytes of stack usage are required. The worst-case requirements of each function are then pushed back to a decoration on that function in the internal state tables of the server. During translation, all function calls – typically effected with the branch-and-link instruction, *bl <offset>* – are instrumented to verify that sufficient stack storage remains.

The reserved on-die stack space is split into two pages, top and bottom. The stack pointer is initially set to one of the pages, and program execution proceeds normally. When a worst-case stack usage may trigger an exhaustion of the current stack page, then a page swap occurs as follows.

- Copy the alternate page to the server
- Construct a false function frame with a return address that restores the alternate page
- Set the LR to the false function frame
- Reset the alternate page to all *0x0000*
- Reset the stack pointer to the alternate page
- Proceed with execution

Along with the stack monitoring, a further complication occurs in that programs compiled with high optimization levels tend to drop the frame pointer (*i.e., -fomit-frame-pointer* in gcc). This frees up one additional register for general use, while making use of a debugger nearly impossible. In order to support arbitrary ELF images, the SoftCache must recreate a stack profile by watching the stack pointer (SP) during the course of each basic block.

By further adding logic to watch the LR as it is stored or manipulated in registers, each basic block is annotated with an offset into the stack where a copy of the LR is stored. When evictions take place, by walking the chain of annotations it becomes possible to fix the stack without a frame pointer.

However, this level of annotation requires *partial evaluation* during program analysis. That is, where-ever possible instructions are evaluated when the result can be computed deterministically.

Ambiguous results are annotated as unknown, but all possible known values are decorated on each operation. This annotation process for the LR allows the determination of *where* the LR may exist in the stack. To date, no instances of LR storage in the stack have failed to be deterministically resolved although hand-written test cases can be constructed that break our system. Empirical evidence suggests that compilers do not generate code with a variable offset for storage of the LR into the stack.

4.2.3 Trampolines

As previously discussed, many of the techniques we use resolve in a worst-case (ambiguous) situation to instrumenting load/store or control flow changes. These instrumentations can be quite expensive and lead to large code growth, similar to the Code Cache system [53]. An alternative is to use a trampoline system between control transfers and for ambiguous load/store operations. These trampolines mimic the behavior of stack page swapping, where a false function frame is constructed that will point the LR to a special handler.

The advantage of a trampoline is that it is very simple to use and debug. The drawback is that hand-tuned assembly code to implement a robust trampoline can amount to a worst-case execution path of 54 instructions, or a best-case execution path of 15 instructions. At the basic block level of granularity, trampolines are ineffective. At a complete function level of granularity, trampolines are effective so long as the function bodies are sufficiently large. However, increasing the granularity of the trampoline management leads to inefficient use of the on-die memory, hence our decision to use basic blocks and instrumentation as necessary.

4.3 Simulation Framework

The SoftCache system runs in two environments: a prototype implementation on an Intel XScale based PDA, and a simulation environment derived from SimpleScalar/ARM as modified by Gilberto Contreras at Princeton in his work on XTREM [23]. Contreras modified the ARM backend to more closely model the actual XScale, and this forms the core of the XTREM project. Contreras also generously shared his source code, facilitating a more rapid simulation framework for the SoftCache.

The simulation environment uses gcc to generate ELF images. These images are then fed into an analyzer program which reconstructs the CFG and DFG, along with stack annotations. The results of this analysis are written out to a separate file to facilitate debugging and verification. The ELF image and the annotations are then loaded into the actual server program, which opens a socket on the local x86 platform host. The client runs as an XTREM process, which opens a socket to the x86 host server process. Execution then proceeds as outlined previously. During the course of execution, the modified XTREM engine keeps track of various statistics, and generates both an instruction trace and a load/store behavior trace. These traces are used for later verification and rapid prototype approximations to memory behavior. The replacement policy for managing the instruction cache is strictly FIFO.

4.4 MiBench Results

To evaluate the SoftCache system which we propose as a solution for the small embedded space, we use the MiBench [47] embedded benchmark suite. This suite is comprised of six primary categories of applications: automotive/industrial, consumer, office, network, security, and telecommunications. For each benchmark, we use the *large* input data set where possible, and run each application from beginning to end, omitting no instructions or data references.

The following subsections discuss the results for each of the 24 applications we evaluate. These results include the miss rate during dynamic execution, as well as statistics on how many blocks are executed, server-client bytes transferred, etc. Each benchmark also contains a reported result for *Subset Collisions*, which represents a change in basic block structure for an already cached basic block. As a micro-optimization, when a code hammock is evaluated it is rewritten into a straight-line code sequence with an exception to catch a change in evaluation, allowing for contiguous code placement. Each graph represents misses on the *y*-axis and the dynamic instruction number on the *x*-axis. Each plot point represents the number of misses accumulated over the prior 250 instruction blocks in summation.

Given that the SoftCache implementation under evaluation is aimed for devices that use Intel XScale processors, our on-die storage is modeled after the XScale cache structures. The XScale has dual 32KB storage regions, one for instructions and one for data. Therefore, our on-die SRAM for

instructions with the SoftCache is limited to 32KB. In order to avoid negative performance impacts, the steady state or hot path in each benchmark must fit in this 32KB. Results based on data caching support in the SoftCache are presented later in this thesis.

One drawback to using any benchmark is that no benchmark can accurately capture all realistic behaviors. For our class of device target, the ubiquitous embedded device or sensor mote, each unit can be expected to do one task well and not much else. Therefore, the MiBench suite represents a realistic set of simple tasks for such devices. Unfortunately, like all benchmarks, the MiBench suite is contrived and uses large arrays or input files. In our simulation framework, these arrays or input files act as large memory regions that are essentially reached from the server. In reality, this is not the case since such *input* would reside on the embedded platform. Therefore, while our results are somewhat skewed with an inherent benchmark bias, this bias is universal to all benchmarks for our target platform. All benchmarks are compiled with *gcc -O2*.

Of the 24 benchmarks in the MiBench suite, the performance for each application can be loosely grouped into one of four categories: well-behaved, adjustable, minor capacity conflict, and major capacity conflict. Table 4.4 presents the key characteristics of the MiBench suite for the instruction cache behavior of the SoftCache system. Individual benchmark results and plots for the instruction cache behavior are in Appendix B.1.

4.4.1 Well Behaved Applications

Many of the MiBench applications are well-behaved in the SoftCache framework. The inherent design bias of the SoftCache requires that an application exhibit long periods of stability between short periods of data transfer through the client-server interface. One representative example is the application *sha*, with the miss rate plot shown in Figure 3.

This benchmark is an excellent application of the SoftCache. Once the initial setup of the application is made, no misses occur until the very end when the result is being reported. This benchmark also fits easily within the 32KB limitations of the on-die SRAM. The long stability between misses – approximately 250 million instructions – is on the order of one second of execution time.

Other benchmarks that are well-behaved include: crc, madplay, math, pgp, say, search, susan, and tiffbw. Each of these benchmarks could have their performance improved by employing the

Table 3: Instruction caching results summary for the MiBench suite.

benchmark	Unique PCs	Executed Blocks	Bytes Transferred	Evictions	Subset Collisions
bf	2,588	104,141,294	18,036	0	35
bitcnts	3,818	171,578,140	17,528	0	66
cjpeg	8,399	14,095,128	40,944	366	148
crc	2,811	292,927,673	12,536	0	41
dijkstra	3,497	45,957,936	15,724	0	57
fft	3,864	51,389,101	18,804	0	75
gs	8,522	81,790	37,400	218	64
ispell	3,618	110,879	16,080	0	60
lame	21,332	50,883,969	55,520,712	1,651,653	85,301
lout	16,279	139,206	82,816	2,276	188
madplay	10,167	22,572,523	48,644	910	135
math	5,221	488,846,352	26,952	0	130
patricia	5,635	142,654,643	26,304	0	114
pgp	3,202	66,374	14,732	0	60
qsort	3,490	167,108,825	16,008	0	57
rawcaudio	1,820	2,135	8,128	0	28
rijndael	2,279	2,926	10,064	0	33
say	6,621	5,860,106	31,584	0	115
search	2,293	1,067,366	10,956	0	42
sha	2,916	9,811,056	13,136	0	34
susan	3,873	3,904,530	17,556	0	54
tiff2bw	5,568	11,386,109	24,404	0	65
tiffdither	7,564	200,054,165	35,736	0	211
toast	3,603	4,044	15,268	0	20

same adjusting techniques discussed in the next section.

4.4.2 Adjustable Applications

The bulk of the remaining MiBench applications can be made well-behaved in the SoftCache framework by adjusting the fetch-on-demand system. The inherent design bias of the SoftCache requires that an application exhibit long periods of stability between short periods of data transfer through the client-server interface. However, these applications have sporadic misses that are cold-start misses, not capacity misses. The adjustment required is to aggressively pre-fetch the next N misses. The actual material to be pre-fetched can be determined either from static analysis or from dynamic feedback. One representative example is the application *tiffdither*, with the miss rate plot shown in Figure 4.

By pre-fetching the next N misses, the sporadic miss rate will can be converted to just one or two misses throughout the entire program. Other benchmarks that fall into this category are: bf, bitcnts, dijkstra, fft, ispell, patricia, qsort, rawcaudio, rijndael, tiffdither, and toast.

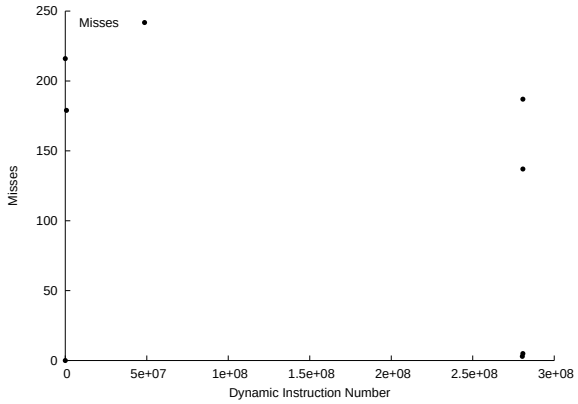


Figure 3: The miss rate over the full dynamic execution of the MiBench benchmark *sha* is representative of well-behaved SoftCache applications.

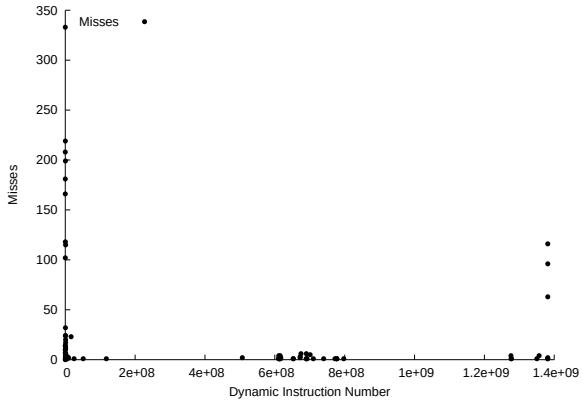


Figure 4: The miss rate over the full dynamic execution of the MiBench benchmark *tiffdither*.

4.4.3 Minor Capacity Conflicts

A few of the MiBench applications require slightly more than the (arbitrary) 32KB limit enforced for code storage in this evaluation. Such applications will exhibit the latency of client-server communications, which may be sufficiently large to make a solution like the SoftCache unattractive. One representative example is the application *gs*, with the miss rate plot shown in Figure 5.

The first half of the program is continually missing due to cold start *and* capacity problems. Eventually the program reaches a steady state interpreting the PostScript input file, but the delay penalties incurred from the initial swapping are likely to be too great for the SoftCache to compensate for. This category of problem includes the cjpeg and lout benchmarks, both of which also require slightly larger on-die storage to avoid excessive swapping.

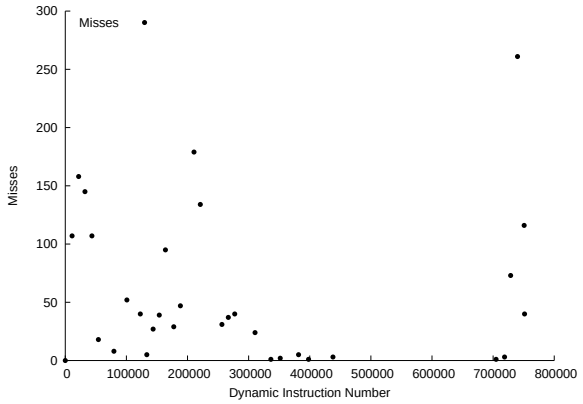


Figure 5: The miss rate over the full dynamic execution of the MiBench benchmark *gs*.

4.4.4 Major Capacity Conflicts

The sole remaining MiBench application, lame, demonstrates the problem with major capacity miss pressure. The miss rate plot shown in Figure 6.

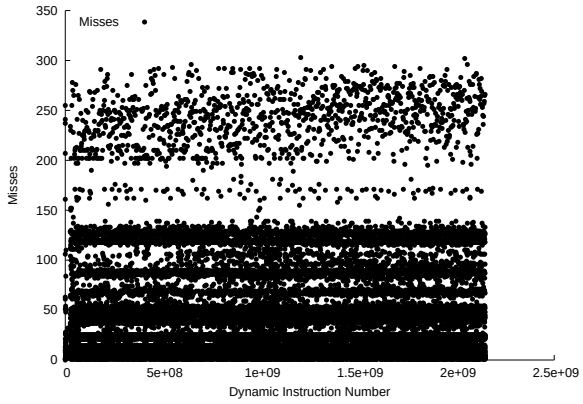


Figure 6: The miss rate over the full dynamic execution of the MiBench benchmark *lame*.

This benchmark is far too large to run in the SoftCache framework under any conditions. With over 1.5 million evictions and 55MB of code transferred, this is a perfect representation of the wrong class of application to run on a SoftCache framework. However, the excessive misses that the SoftCache incurs will *also* be encountered by traditional hardware caches. The reason that hardware cache solutions maintain an acceptable level of performance is the lower miss penalty to the backing store – local DRAM in this case. This is purely an energy-delay trade off, which is explored in more detail later.

4.5 Related Work

This section expands on the general related work outlined previous in Section 3.6. This chapter focused on an implementation for instruction cache behavior in the SoftCache system, therefore the related work presented here is a discussion on other cache implementations.

4.5.1 Alternate Caches

The eXtended Block Cache [66] proposed by Intel corporation adds an element of redundant instruction supression 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 [12]. The key idea of the block-based design is to cache sequences of traces and then store a pointer to the translated sequence. Several traces can then be buffered with only minor penalties due to the indirection encountered. While similar techniques could be applied in the SoftCache, these types of designs still rely on hardware to solve the performance problems. More complicated hardware results in a greater power dissipation.

Of more direct comparison to SoftCache are other schemes for placing the cache storage area under software control, whether completely or partially. While different proposals have been made with varying degrees of success in microbenchmarks, the common themes have focused on either disabling unused sections of the cache or achieving better results through refinements of trace caches.

The Span Cache [114, 115] explores a model of direct-addressing regions of the cache. It exposes the cache storage area as directly addressable through additional registers, but has a fallback 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.

Intel has also begun providing with their new X-Scale processor, a derivative of the StrongARM, the ability to convert associative-way regions of cache storage into addressable RAM [25]. This has been offered as a solution to not only hard real-time guarantees, but also for situations where the

programmer knows more than the hardware can deduce via normal cache algorithms.

One interesting benefit of the SoftCache system implemented as a direct SRAM access lies in the area of real-time computing. Systems that need hard real-time guarantees on program performance can have serious problems with hardware caches, which can variable access times. With the SoftCache, once an application is folded into the memory space and reaches steady-state, it runs with exactly the same access times and performance with every loop through the program.

Exploring the possible usage of on-chip memories like the ScratchPad [25], Panda began a series of experiments on the concept of Scratch-Pad usage for optimizing data accesses. This was later expanded on by the efforts of many [83, 109, 7, 82, 8, 102] and hinged 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 [108]. 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.

Samsung recently published work similar to the SoftCache for instructions only [84]. Their technique did not use a client-server system *per se*, but rather a transitional on-die memory manager that was called on every function entry or exit. They reduced code size by an average of 33% with moderate performance impact and power increases. Our system, by offloading the memory management, is capable of more advanced analysis and higher performance.

Other strategies have hidden the cost of incorrect hardware guesses by providing a mechanism for the programmer or compiler to generate *prefetch* instructions. Such instructions, as supported by the UltraSPARC-II, Pentium-III, and other devices, allow via manual control a way to populate instruction or data caches with what will soon be needed. The growing implementation of such features in hardware suggests that the memory-processor gap has become sufficiently critical that average-case “good” algorithms for automated memory management may no longer be acceptable.

CHAPTER V

SOFTCACHE SUPPORT FOR DATA CACHING

The simplistic technique of the prior chapter to support instruction caching behavior within the SoftCache will not work for data caching support. The problem lies in the successful determination of where, exactly, any data reference will go in memory. The presence of pointers, arrays, heaps, and so forth greatly complicate the determination of possible memory locations for any given load/store operation. The common sentiment[1] is that data caching is too impractical to work for any system like the SoftCache, and only traditional hardware mechanisms or complex programmer models are viable.

The reason for this sentiment lies in the perception of pathological examples. One classic example can be expressed in two short C lines:

```
scanf("%x",    &funcptr);
*funcptr();
```

The reality of the situation, however, is that such code is seldom encountered. While this example demonstrates a pathological control-flow analysis problem, the equivalent data-flow analysis problem is dereferencing an input address. While these types of behaviors are useful for applications like debuggers, this type of behavior is rarely useful in a general computing framework since it can lead to unpredictable results. This increases the burden of maintenance, which is already the most expensive part of any real software project.

By carefully constructing the data-flow graph for input programs, most memory references can be narrowed down to one of a few possible addresses. Some of these addresses are relative to a register, such as the stack pointer. Other references are relative to an array base. Those addresses that truly cannot be resolved to a small window of possibilities include heap accesses and pointer castings.

[1] This sentiment is frequently expressed during paper reviews, discussions with colleagues, etc.

We propose to change this sentiment of impossibility by solving the underlying complexity of tracking memory reference locations. We achieve this by combining two partial solutions into a set of architectural enhancements. By using multiple strategies, we are able to cover all of the problems that can be encountered by the SoftCache. Moreover, our system is flexible enough that future strategies can easily be incorporated to make a more efficient and practical SoftCache.

First, we present a novel classification for distinguishing between stable and deterministic memory references, and unstable references. This classification is exploited to partially support data caching in Section 5.1. The key idea is to examine the stability of load/store operations by enumeration of the targets for each load/store instruction. Those operations that fit into the *stable* category can easily be optimized into high-performance data caching similar to how the instruction caching mechanism works from the prior chapter. However, all of the *unstable* operations require special handling, which is a major performance impact.

Therefore, we then add as a second technique a specialized support mechanism to improve performance for those memory references that are unstable and therefore cannot be optimized. This support is detailed in Section 5.2. The fundamental principle of the support mechanism is to mimic at a very primitive level the traditional hardware notion of cache tags. This support requires no tag logic, however, and therefore does not incur the power and performance problems of traditional caches.

5.1 UTI-MTI Classification

We propose to classify memory operations into two primary categories, Uni-Targeted Instructions and Multi-Targeted Instructions, based on their reference behavior. A Uni-Targeted Instruction (UTI) is a memory operation that only accesses one unique memory address over a dynamic trace of instructions. A Multi-Targeted Instruction (MTI) accesses multiple memory addresses over the the same trace.

Both UTI and MTI occurrences are identified by the PC of the instruction, as illustrated in Figure 7. In the figure, the LD instruction at PC `0x1234` reads from the target address in `[r1]`, which is constant regardless of where the instruction is executed dynamically. This unchanging target address is an example of UTI behavior. The LD instruction at PC `0x1244` reads from `[r3]`,

yet the value in this register changes as the dynamic execution progresses. This changing target address is an instance of MTI behavior. Conceptually the UTI may be using a global variable, whereas the MTI may be traversing an array or chasing pointers.

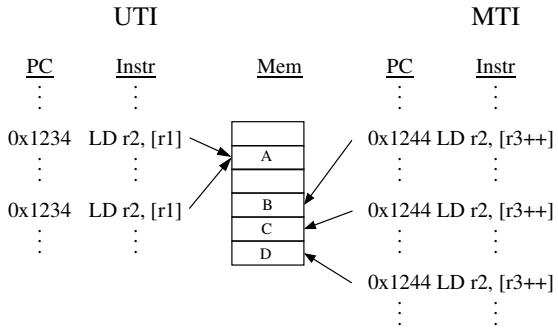


Figure 7: Basic concept of Uni-Targeted Instructions and Multi-Targeted-Instructions.

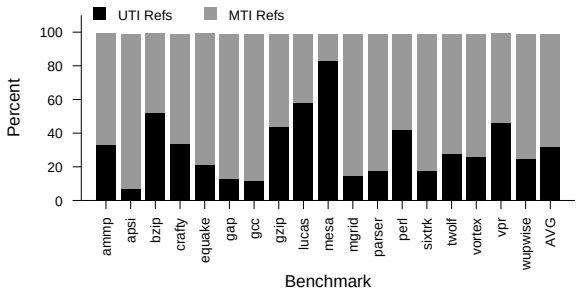
In a code examination, uni-targeted instructions have been traced back to operations that refer to global variables such as data constants and semaphores, as well as some stack variables where only one function call path chain can trigger the target instruction. With only one possible call path for a function, the stack variables are essentially fixed with respect to the function address in the stack, even though the stack itself is continuously changing. Multi-targeted instructions have been traced back to stack variables reached from multiple function call paths, as well as more typical array operations and linked list code. With benchmarks like gcc, MTI data can also be traced back to garbage collection routines.

5.1.1 UTI/MTI Dynamic Distribution

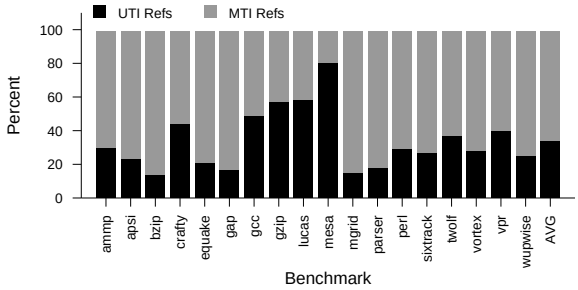
To quantify the distribution of UTI and MTI targets in applications, Figure 8 (a) shows the breakdown for the SPEC2000 benchmark suite over the *entire* application. Figure 8 (b) shows the breakdown on 100M instruction traces from the interval chosen by SimPoint [85].

While these results have some particularly large individual variations (*e.g.,* bzip, gcc), the average results are similar (31% dynamic UTI for the full run, 29% for the SimPoint version). To accelerate our simulations, we use SimPoint with the expectation that individual IPC gains may vary as suggested by these results, yet the geometric mean should be indicative of the result were full benchmark runs used.

The differences between full application and SimPoint runs for the ratio of UTI presence is



(a) Full application runs on reference inputs



(b) 100M instructions from SimPoint analysis

Figure 8: Distribution of UTI/MTI dynamic instances.

shown in Figure 9. An interesting implication is that SimPoint does not always accurately reflect this type classification, which suggests future exploration to ensure that SimPoint considers the right statistics to capture this type of behavior.

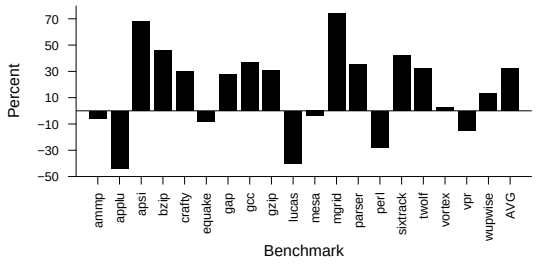


Figure 9: The difference between full application UTI dynamic instruction percentage and the SimPoint based 100M instruction benchmark subset.

5.1.2 UTI/MTI Static Distribution

Building on this separation at the instruction level of UTI and MTI traffic, a detailed simulation demonstrates that the actual number of UTI targets is small despite the large dynamic instruction

percentage. Table 5.1.2 shows the results over the SPEC2000 benchmarks in classification of the UTI-MTI behavior. The geometric mean unique UTI targets for the full run is merely 1031, and can be as few as 301. For the SimPoint execution, this average drops to 270 (not shown in Table 5.1.2).

Table 4: The breakdown of MTI and UTI information over complete runs of SPEC2000 benchmarks. The dynamic instruction count only reflects LD/ST traffic. The static instruction counts are those actual PCs which correspond to either UTI or MTI over the program lifetime. The memory addresses are those locations over the lifetime of all dynamic instructions that are classified as UTI or MTI.

SPEC	Dynamic Instructions			Static Instructions			Memory Addresses		
2000	UTI	MTI	UTI %	UTI	MTI	UTI %	UTI	MTI	UTI %
ammp	56.0 B	115 B	32%	2.57 K	43.2 K	5.6%	638	2.63 M	0.0%
applu	74.6 B	94.3 B	44%	7.80 K	111 K	6.5%	1262	22.7 M	0.0%
apsi	12.8 B	168 B	7.1%	11.0 K	139 K	7.3%	1256	25.0 M	0.0%
bzip2	27.2 B	25.6 B	51%	2.27 K	28.2 K	7.4%	301	78.5 M	0.0%
crafty	28.1 B	55.4 B	34%	6.49 K	102 K	5.9%	1095	408 K	0.0%
eon	12.7 B	24.9 B	34%	15.2 K	127 K	10%	5637	101 K	0.1%
equake	12.1 B	46.5 B	21%	1.92 K	29.4 K	6.1%	490	6.88 M	0.0%
facerec	17.1 B	53.2 B	24%	3.76 K	72.6 K	4.9%	1032	4.08 M	0.0%
fma3d	58.0 B	77.6 B	43%	11.1 K	143 K	7.2%	3016	15.1 M	0.0%
galgel	2.19 B	169 B	1.3%	8.54 K	126 K	6.3%	1478	4.71 M	0.0%
gap	13.5 B	89.9 B	13%	4.08 K	77.3 K	5.0%	1551	25.0 M	0.0%
gcc	2.21 B	16.1 B	12%	19.2 K	599 K	3.1%	3312	41.1 M	0.0%
gzip	19.0 B	24.0 B	44%	2.04 K	29.5 K	6.5%	429	15.5 M	0.0%
lucas	30.6 B	22.4 B	58%	3.90 K	50.0 K	7.2%	722	20.8 M	0.0%
mcf	889 M	19.4 B	4.3%	1.33 K	20.9 K	5.9%	369	24.9 M	0.0%
mesa	113 B	22.5 B	83%	5.77 K	58.1 K	9.0%	1478	4.97 M	0.0%
mgrid	73.6 B	403 B	15%	2.63 K	48.0 K	5.2%	608	7.27 M	0.0%
parser	39.9 B	187 B	18%	37.7 K	79.9 K	32%	508	14.7 M	0.0%
perl	4.66 B	6.42 B	42%	6.82 K	89.4 K	7.1%	1436	173 K	0.0%
sixtrk	626 M	2.77 B	18%	12.0 K	159 K	7.0%	2548	2.79 M	0.0%
swim	58.4 B	92.6 B	39%	3.41 K	51.4 K	6.2%	770	25.0 M	0.0%
twolf	41.8 B	107 B	28%	8.10 K	111 K	6.8%	1004	1.00 M	0.0%
vortex	13.3 B	37.0 B	26%	12.4 K	216 K	5.4%	1573	17.2 M	0.0%
vpr	533 M	623 M	46%	3.27 K	47.6 K	6.4%	720	602 K	0.0%
wupw	26.8 B	81.2 B	25%	2.92 K	46.7 K	5.9%	589	23.1 M	0.0%
GEOMEAN	14.4 B	40.7 B	26%	5.56 K	76.3 K	6.8%	1031	8.01 M	0.0%

Of the actual program LD/ST instructions, UTIs are less than 7% as indicated by the static instruction data in Table 5.1.2. However, these 7% of instructions comprise 26% of the dynamic LD/ST references. This trend of very few actual PCs generating a substantial amount of memory

traffic should be readily identifiable. Since those same PCs only access a few hundred unique memory locations, the data for these operations will fit into very small caches. Less than 4200 bytes (1031 targets $\times$ 4 bytes) are required to hold the entire UTI data set for the full benchmark runs on average. The MTI data set, however, comprises at least tens of megabytes.

Since approximately 4200 bytes can capture 26% of the dynamic LD/ST traffic, which is generated by relatively few instructions, a mechanism to capture this information in the memory hierarchy may lead to performance gains through higher cache hit rates. As a minimum, isolation of such UTI information will eliminate pollution in or by this 26% of memory traffic. Since the actual unique targets represent less than 0.01% of all memory targets, prediction mechanisms can be used to capture this behavior.

5.1.3 Phasing

The consideration of MTI targets as a series of stable UTI targets may capture call-path locality in the reference stream. Repeated chains of function calls may experience periods where local variables are actually at constant addresses in the stack. We characterize such behavior as *phasing*, since during dynamic execution the variables treated as UTI may change relative to an absolute static division. We examine a few full benchmark runs to determine whether this concept of phasing is valid. The observed MTI behavior in the stack suggests that discarding history (beyond a certain age) may reveal that MTI memory operations may be temporally reclassified as UTI. Therefore an age-based decay of history may reveal other trends inside of the UTI-MTI landscape. To illustrate the concept of phasing, consider the *function call graph* of Figure 10.

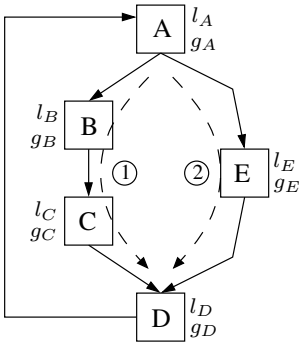


Figure 10: Example program structure to illustrate the concept of phasing and ratio changes.

Assume that all global variables, g_i, are accessed via UTI memory operations and that all local

variables, l_i, are accessed via MTI. If the program control flow at the function level is a repeating sequence alternating between the paths $\{1, 2\}$, such that $ABCD \rightarrow AED \rightarrow ABCD \rightarrow \ldots$, then the local variables on the stack for the functions B, C, D, and E may change locations. Using a shorthand of ΣXi to represent the total unique addresses of type X, we can define the UTI unique target ratio R_{UTI} of this call graph as:

$$R_{UTI} = \frac{\Sigma g_i}{(\Sigma g_i + \Sigma l_i)} \tag{1}$$

The continuous oscillation between the execution paths $\{1, 2\}$ prevents any local variable outside of A from acting as UTI. However, if the execution path were to continuously be *only* one of the paths $\{1, 2\}$, such as $ABCD \rightarrow ABCD \rightarrow \ldots$, then every pass through the loop will access local variables at the same address on the stack. Even though we assume for this example that local variables are MTI, if this single path executes sufficiently long we can treat these variables as UTI.

For this trivial example, the phasing behavior reduces R_{UTI} to the constant 1 – indicating that *all* memory references are effectively UTI during the window of observation. In reality, the actual fluctuation based on array accesses, function call paths, and other variables will cause the R_{UTI} to fluctuate during any particular window of dynamic execution.

However, we can calculate based on Table 5.1.2 the *expected* ratio R_{UTI} over the lifetime of each benchmark. This ratio is what we expect to find if we pick a window of dynamic instructions from that benchmark and re-compute the ratio over the memory access stream in that window. By comparing the per-window R_{UTI} to the full benchmark calculation, it becomes possible to determine the relative increase or decrease of the unique UTI targets with respect to the total unique targets in that window. If the window ratio is increasing, there are more UTI opportunities available for our system to work with. If the window ratio is decreasing, there are fewer UTI for our system. Therefore, the ratio R_{UTI} can be used as an approximation to the phasing behavior in a window of dynamic instructions.

By resetting the captured UTI-MTI state information every billion instructions, we analyze a few benchmarks to determine how their UTI percentage changes compared to a full application classification. Figure 11, using a log-scale Y axis, shows the results for gcc, mcf, and vpr, skipping

the first two billion instructions to avoid warm-up behavior. A value of 100% corresponds to the dynamic window exactly matching the UTI expectations based on the full application run. Substantial phasing behavior appears with changes between 210-1200% in gcc, and a nearly constant 225% in mcf and 4400% in vpr. The gcc oscillations are due to intermittent garbage collection. These few benchmarks show the trends that are observed over all the benchmarks we run.

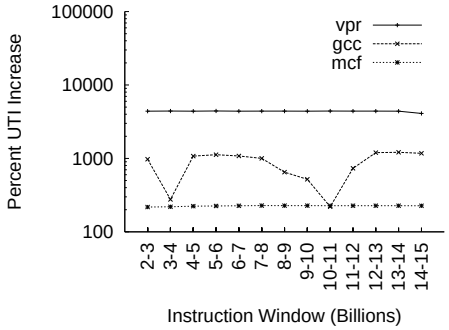


Figure 11: By resetting all UTI-MTI state every billion instructions, the relative percentage of UTI targets increases indicating phasing behavior.

As the UTI occurrence increases, any scheme we design for capturing UTI behavior should have more opportunities for performance improvement. However, any decay of history which is too aggressive may result in over pressuring the isolated cache, reducing performance. To obtain the best performance possible, a support of phasing information may be useful in any solution to capture the UTI-MTI behavior.

5.1.4 Capturing UTI-MTI Divisions

Given the insight that approximately 30% of the dynamic memory traffic for the entire SPEC2000 benchmark suite can be contained in 2-4KB of storage, this suggests that reserving a 4KB region of the on-chip storage for storing UTI data is a performance effective solution. The drawback is that the remaining 70% of dynamic memory traffic cannot be so easily captured, and requires some form of instrumentation or other emulation to ensure proper dereferencing.

5.2 Unstable and Ambiguous References

While the UTI-MTI division provides a convenient mechanism for directly handling approximately 30% of dynamic memory, the remainder of load/store references must be emulated to ensure correct

Listing 5.1: Examples of standard load/store, expected value test load/store, and emulation load/-store program fragments.

```
1  ;
2  ; Default load/store behavior
3  ;
4
5     ldr      r3, 24[r1]              ; perform load/store
6
7  ;
8  ; Expected address verfied load/store
9  ;
10
11    mov      r3, #expected_value     ; expected r1 value
12    eor      r3, r3, r1              ; test expected
13    swine    #17                     ; trap if not equal
14    ldr      r3, 24[r1]              ; perform load/store
15
16 ;
17 ; Full emulation via trap
18 ;
19
20    swi      #17                     ; invoke software interrupt
21    ldr      r3, 24[r1]              ; now perform load/store op
```

operation of programs within the SoftCache framework. This emulation can be achieved in one of two ways: (1) expected value test, or (2) full emulation. The basic options are represented in Listing 5.1.

The drawback to expected address checking is the instrumentation of the load/store operation inflating code size by three additional instructions for every load/store operation. It also requires extensive book-keeping and updates of the embedded constants on the part of the server, which will cause additional network traffic impacting overall performance and power efficiency. Considering that load/store operations occur very frequently, and typically 70% of these will be exploding to a best-case situation of 3 additional instructions, this is a significant performance impact. In the case where the expected value fails the check, it will invoke the same emulation system as skipping the check entirely.

The emulation of load/store operations is very heavy, regardless of whether it occurs from an expected value check failing or by design. Directly calling an emulation interface is much less pressure on the code size, but the trap emulation is costly in performance – it requires a switch to supervisor mode, preserving registers, and an additional 10-15 instructions depending on context. This exception overhead in turn must be followed by a round-trip communications sequence with

Listing 5.2: An example of the guard instruction for an extended ARM ISA.

```
1  ;
2  ; Default load/store behavior
3  ;
4
5      ldr     r3, 24[r1]              ; perform load/store
6
7  ;
8  ; Guard evaluation
9  ;
10
11     guard   #239, #17               ; ensure load/store is right
12     ldr     r3, 24[r1]              ; now perform load/store op
```

the server to invoke additional resources to resolve the target and determine whether the target exists locally. In essence, the expected value check is an optimization over pure emulation trading code space for execution time.

The explosive code growth and application delays these systems force upon dynamic translation systems like the SoftCache are the primary reasons why data caching is classically held as impractical. We propose to change this by adding a new instruction to the ARM ISA to *guard* the following load/store operation. In specific, the idea is to approximate the tag arrays of traditional hardware caches in the on-die SRAM. Unlike traditional hardware caches, however, these are pure SRAM cells and not CAM cells, eliminating the power and delay penalties of traditional tag arrays. Moreover, by using the on-die SRAM storage space to contain the pseudo-tag array no additional storage overhead is required.

The proposed new instruction is the *guard* instruction, followed by *two* immediate data fields. The representative assembly code is written as `guard 501, 34`. The first immediate data field is the index of the load/store effective address being guarded. The second immediate data field is the interrupt value to trigger if the guard check fails. An example usage is in Listing 5.2.

The premise is that when a guard instruction is encountered, the next load/store operation checks that the effective address falls within the guard base address range given an assumed line size. After each load/store instruction, the internal microprocessor flag for guard checking is cleared until the next guard operation sets it again.

For a 64-byte line size, the least significant 5 bits are dropped to generate the base address. The specified guard index is used as a pointer into an array of 4-byte values stored at a fixed location in

on-chip SRAM. For a 32KB data cache with a 64B line size, 512 entries are required for 2KB of guard storage overhead. In principle, the array would be structured similarly to Figure 12.

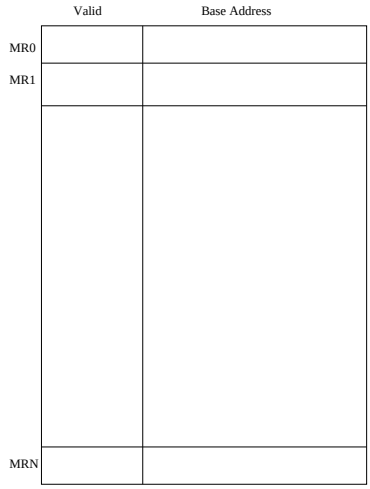


Figure 12: The *guard* address table to approximate tags.

Since the bottom bits of the base address are not used, they can store control information such as a valid bit or special annotations indicating mutex/lock, dirty data, etc.

5.2.1 Capturing Unstable References

In order to handle ambiguous memory references and MTI situations, the server transfers the equivalent of MTI cache lines to the embedded client. As the MTI cache fills to capacity, conflicts occur and old data is discarded while new data is loaded from the server. Each load/store operation that cannot be determined as stable is preceded by a guard instruction using the appropriate index into the guard array. Since the actual guard array is just in on-chip SRAM, the server's standard memory patch operation is sufficient to update the indices. During a datacache eviction, the server already has sufficient book-keeping details that guard instructions dependent upon the just-evicted data can be replaced with a generic trap instruction to re-fetch the data on demand.

5.3 Simulation Framework

The SoftCache system runs in two environments: a prototype implementation on an Intel XScale based PDA, and a simulation environment derived from SimpleScalar/ARM as modified by Gilberto Contreras at Princeton in his work on XTREM [23]. Contreras modified the ARM backend to more

closely model the actual XScale during an internship at Intel, and this forms the core of the XTREM project. Contreras also generously shared his source code, facilitating a more rapid simulation framework for the SoftCache.

The simulation environment uses gcc to generate ELF images. These images are then fed into an analyzer program which reconstructs the CFG and DFG, along with stack annotations. The results of this analysis are written out to a separate file to facilitate debugging and verification. The ELF image and the annotations are then loaded into the actual server program, which opens a socket on the local x86 platform host. The client runs as an XTREM process, which opens a socket to the x86 host server process. Execution then proceeds as outlined previously. During the course of execution, the modified XTREM engine keeps track of various statistics, and generates both an instruction trace and a load/store behavior trace. These traces are used for later verification and rapid prototype approximations to memory behavior. The replacement policy for the data cache is strictly FIFO with respect to the guard table and the UTI storage pool.

5.4 MiBench Results

We again evaluate the SoftCache system using the MiBench [47] embedded benchmark suite. As with the instruction caching evaluation, we use the *large* input data set where possible, and run each application from beginning to end, omitting no instructions or data references.

The following subsections report the results for each of the 24 applications we evaluate. These results include the miss rate during dynamic execution, as well as statistics on how many unique load/store instructions were encountered, quantities and classes of load/store operations, etc. Each graph represents Misses on the *y*-axis and the dynamic load/store instruction number on the *x*-axis. The *x*-axis does not represent every dynamic instruction, only load/store instructions. Each plot point represents the number of misses accumulated over the prior 250 load/store instructions in summation.

Table 5.4 presents the key characteristics of the MiBench suite for the data cache behavior of the SoftCache system. Individual benchmark results and plots for the data cache behavior are in Appendix B.2.

Given that the SoftCache implementation under evaluation is aimed for devices that use Intel

Table 5: Data caching results summary for the MiBench suite.

benchmark	MTI PCs	UTI PCs	MTI Refs	UTI Refs	Misses	Evictions
bf	421	476	334,302,552	54,224,742	378	0
bitcnts	366	999	164,265,629	19,133,795	187	0
cjpeg	1,221	1,857	23,084,552	16,138,543	122,477	121,965
crc	335	702	825,250,413	159,960,596	228	0
dijkstra	484	799	42,979,220	74,326,910	237,372	236,860
fft	438	902	89,430,352	41,886,360	141,542	141,030
gs	1,689	2,139	173,734	29,601	3,576	3,064
ispell	535	783	152,808	49,774	6,830	6,318
lame	5,045	3,486	599,509,141	31,865,406	3,270,672	3,270,160
lout	1,767	6,588	224,988	48,990	2,212	1,700
madplay	1,472	2,213	76,344,468	35,120,382	32,550	32,038
math	610	1,225	569,474,694	101,613,946	205	0
patricia	649	1,299	183,565,348	96,640,079	131,405	130,893
pgp	482	575	30,717	11,267	216	0
qsort	413	820	114,137,996	63,425,373	430,952	430,440
rawcaudio	182	465	3,306	676	119	0
rijndael	237	574	4,277	911	141	0
say	837	1,795	8,205,453	36,735,235	20,127	19,615
search	273	525	1,265,998	400,836	28	0
sha	322	723	32,563,258	4,020,084	266	0
susan	507	816	3,351,885	6,431,816	7,814	7,302
tiff2bw	851	1,299	57,981,267	313,557	7,010	6,498
tiffdither	1,096	1,523	166,737,501	91,260,043	26,247	25,735
toast	431	971	7,534	1,640	206	0

XScale processors, our on-die storage is modeled after the XScale cache structures. The XScale has dual 32KB storage regions, one for instructions and one for data. Therefore, our on-die SRAM for data with the SoftCache is limited to 32KB. The additional 2KB for the guard virtual tag array, as well as the 2KB for the UTI memory region and the 4KB for the two stack pages comes from the mini data cache, mini instruction cache, and fill buffer SRAM regions.

The MiBench suite represents a realistic set of simple tasks for such devices. Unfortunately, like all benchmarks, the MiBench suite is contrived and uses large arrays or input files. In the simulation framework we have, these arrays or input files act as large memory regions that are essentially reached from the server. In reality, this is not the case since such *input* would reside on the embedded platform. Therefore, while our results are somewhat skewed with an inherent benchmark bias, this bias is universal to all benchmarks for our target platform.

With respect to data caching, all benchmark suites will exhause the limited on-die data storage facilities since they assume reading and writing files. Real ubiquitous embedded systems will read and write from local memory on peripheral support circuitry, such as a digital camera ASIC. All benchmarks are compiled with *gcc -O2*.

Of the 24 benchmarks in the MiBench suite, the performance for each application can be loosely grouped into one of four categories: well-behaved, adjustable, minor capacity conflict, and major capacity conflict.

5.4.1 Well Behaved Applications

Several of the MiBench applications are well-behaved in the SoftCache framework with respect to data caching behavior. The inherent design bias of the SoftCache requires that an application exhibit long periods of stability between short periods of data transfer through the client-server interface. One representative example is the application *search*, with the miss rate plot shown in Figure 13.

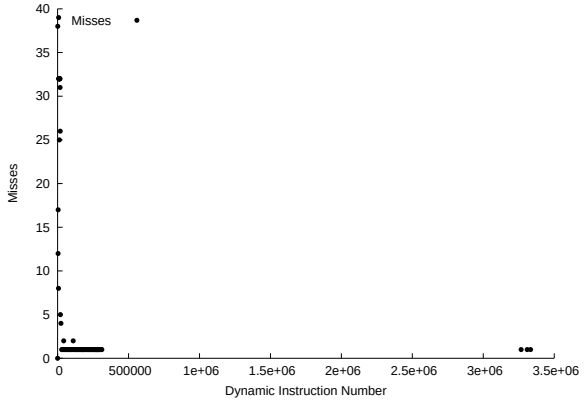


Figure 13: The miss rate over the full dynamic execution of the MiBench benchmark *search*.

This benchmark is an excellent application of the SoftCache. Once the initial setup of the application is made, no misses occur until the very end when the result is being reported. This benchmark also fits easily within the 32KB limitations of the on-die SRAM. The long stability between misses – approximately 3 million load/store operations, or 18 million instructions – is on the order of 100ms of execution time.

Other benchmarks that are well-behaved include: bf, crc, and sha. Each of these benchmarks could have their performance improved by employing the same adjusting techniques discussed in the next section.

5.4.2 Adjustable Applications

Many of the remaining MiBench applications can be made well-behaved in the SoftCache framework by adjusting the fetch-on-demand system. The inherent design bias of the SoftCache requires

that an application exhibit long periods of stability between short periods of data transfer through the client-server interface. However, these applications have sporadic misses that are cold-start misses, not capacity misses. The adjustment required is to aggressively pre-fetch the next N misses. The actual material to be pre-fetched can be determined either from static analysis or from dynamic feedback. One representative example is the application *math*, with the miss rate plot shown in Figure 14.

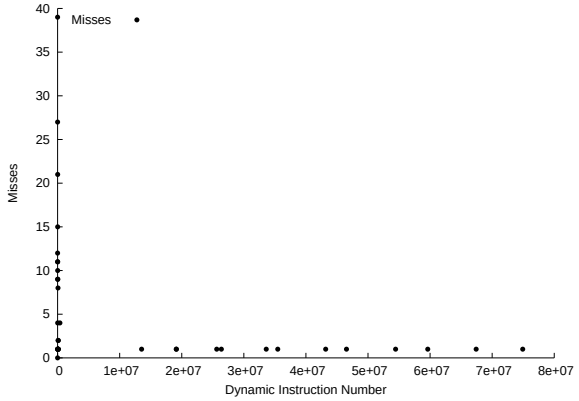


Figure 14: The miss rate over the full dynamic execution of the MiBench benchmark *math*.

By pre-fetching the next N misses, the sporadic miss rate will can be converted to just one or two misses throughout the entire program. Other benchmarks that fall into this category are: bitcnts, pgp, rawcaudio, rijndael, and search.

5.4.3 Minor Capacity Conflicts

A few of the MiBench applications require slightly more than the (arbitrary) 32KB limit enforced for data storage in this evaluation. Such applications will exhibit the latency of client-server communications, which may be sufficiently large to make a solution like the SoftCache unattractive. One representative example is the application *susan*, with the miss rate plot shown in Figure 15.

Throughout the program, the application is regularly missing due to cold start *and* capacity problems. The delay penalties incurred from the initial swapping are likely to be too great for the SoftCache to compensate for. This category of problem includes the other benchmarks fft, gs, lout, say, tiff2bw, and toast.

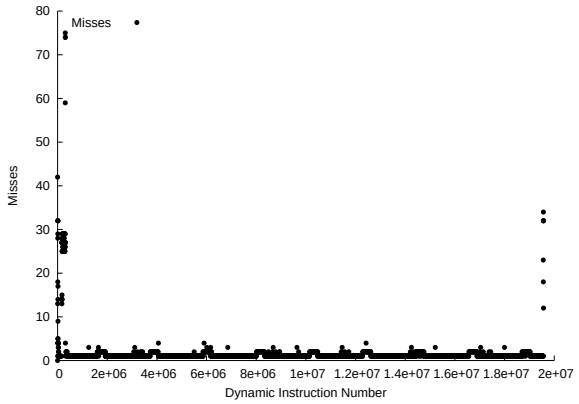


Figure 15: The miss rate over the full dynamic execution of the MiBench benchmark *susan*.

5.4.4 Major Capacity Conflicts

The remaining MiBench applications demonstrates the problem with major capacity miss pressure. One representative example is the benchmark *ispell*, with the miss rate plot shown in Figure 16.

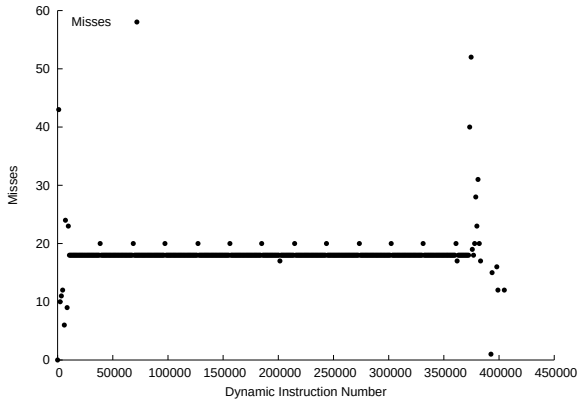


Figure 16: The miss rate over the full dynamic execution of the MiBench benchmark *ispell*.

This benchmark is too large to run in the SoftCache framework under any conditions. With a continuous stream of cache misses requiring the lengthy client-server communications process, this is a perfect representation of the wrong class of application to run on a SoftCache framework. The other benchmarks that fit this category are cjpeg, dijkstra, lame, madplay, patricia, qsort, and tiffdither.

5.5 Related Work

The architecture community has invested much effort into reducing the memory bottleneck. More outstanding achievements are branch predictors, value prediction, and non-blocking caches. However, we are far from the first group to consider the information content of the reference stream to memory directly. In addition to those works cited earlier in this thesis, there are several other groups which have covered some aspects of this work.

Perhaps one of the earliest studies on memory reference behavior, by Hammerstrom and Davidson [50], considered the theoretical amount of information that could be gleaned by data-dependent behavior in reference streams. Using the ideas of entropy and statistical analysis, they found that the addressing overhead is much higher than the actual data content, a result that still holds today as various address compression techniques are now used to reduce power consumption.

Farkas and Jouppi [39] considered the benefits of non-blocking loads, which is the baseline for non-blocking caches. Using a variety of different designs, they were able to reduce miss stalls by up to a factor of 2 for integer applications. Other numeric applications had more substantial gains.

These earlier works are the foundation behind the classification of delinquent load operations – those operations which cause a miss in such a way that performance is dramatically impacted. Even today, the identification and elimination of delinquent loads is a pressing issue [81]. Industry is also exploring mechanisms to avoid these penalties, including Intel's Virtual Multithreading [110] to automatically begin prefetching during delinquent stalls hoping to avoid future stalls.

Tyson et al [107] considered the reference pattern from LD/ST operations, and found that by controlling cache line allocation, memory traffic could be reduced up to 60%. However, Tyson et al were unable to turn this memory pressure reduction into measurable performance improvement. Tyson and Austin [106] later considered memory renaming, which uses a similar concept to our UTI/MTI division. They predicted, based on the PC, an index to speculative values to accelerate memory operations. Their idea of a *load-store cache* is similar to our isolation of the UTI information, yet our technique is complementary such that combining both methods should attain better results than either alone. Other work attempted to capitalize on similar ideas to reduce energy signatures in caches or otherwise alleviate critical load/store misses [90, 65, 71].

Moshovos and Sohi [77] designed a system to predict and capitalize on dependent memory operations with memory cloaking and bypassing. Their system of reducing the memory latency attained between 3.2 - 4.3% IPC improvements.

All of these techniques focus on reducing the memory bottleneck from modern processors. Our methods offer a new avenue for exploration, by highlighting the potential exploitation of the dynamic behavior in LD/ST operations. Our methods also appear to be complementary to existing techniques, such that additional gains are possible when our system is applied on top of other methods.

One study by Driesen and Hölzle [30] used a similar approach to enumerating the actual targets of instructions, but they only consider branch instructions. Their scheme relied on preventing overall branch-predictor pollution. This is similar to our desire to not mix UTI and MTI data, since UTI is stable.

CHAPTER VI

ANALYTICAL PERFORMANCE MODEL

The prior chapters introduced and explored the implementation details and empirical results of the SoftCache for both instructions and data. What we propose, however, is a fairly radical departure from traditional designs and has necessarily garnered critical peer reviews questioning the feasibility of such a system. The next two chapters represent the study of different aspects which we propose to change. First, a series of analytical models that represent bounds on performance impacts are constructed in Chapter 6. Once the theoretical exploration is complete, real hardware data based on the Intel PXA255 processor is covered in depth in Chapter 7.

Prior to designing an experimental setup for measuring power, performance, and delays on real hardware, it is necessary to gain insight into the underlying issues. This insight will focus the actual measurements of real systems to only those components strictly necessary for a feasibility evaluation of the SoftCache framework. To study the limits and problems, we examine several aspects in turn. First, we consider the network impact in Chapter 6.1. Next, changes necessary in the core processor architecture are explored in Chapter 6.2. With approximations for the network and processor changes, Chapter 6.3 explores the penalties of the SoftCache system overhead.

6.1 Network Impact

This section explores the energy and delay trade-offs that occur when some or all of the local storage is moved out of the embedded device, and into a remote network server. Contrary to designer intuition, we demonstrate that this can be more power efficient than local storage.

6.1.1 Device Models

To investigate the possible performance effect of using the network as a mechanism for accessing remote storage, we consider different device models and characteristics. There are three fundamental models of embedded computing devices that we examine: Legacy, Pull, and Push. Each model

is characterized by the type of network link and communications model incorporated. We assume that any applications exhibit sufficient locality such that there are well-defined "working sets" that change infrequently [1, 9].

Each model we consider independently. While general comparisons can be made across models, each has different design-time characteristics making direct comparison difficult. The underlying hardware design behind each model is the same, however, and an example baseline is shown in Figure 17 (a). The classical mode of operation in such a device is that the program and data values are copied from Flash to local DRAM for performance reasons. This copying requires sufficient DRAM for holding all or part of the Flash contents, in addition to all the data.

We suggest that by utilizing the network link to access the equivalent contents of Flash from a remote server, a more energy efficient model can be constructed at a lower cost. This efficiency is achieved by reducing the Flash component to just a boot-block sized unit, and removing some part of DRAM from the local storage. The DRAM we remove normally contains the contents of Flash copied on boot-up or during application mode change. Instead, we propose that a space large enough to hold the worst-case working set of code and data be reserved in the local DRAM. This concept for reduction is shown in Figure 17 (b).

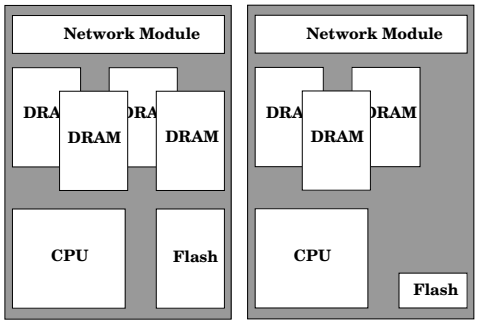


Figure 17: Basic 3G cell phone or other ubiquitous networked device. On the left is part (a) a typical mobile embedded device. Part (b) on the right shows the small reduction proposed in this work.

While we suggest removal or resizing of the DRAM chip(s) and the reduction of the Flash storage, these actions are not strictly necessary. By carefully using V_{DD} gating, each DRAM and Flash unit could be disabled when not needed. This V_{DD} gating would result in power trade-offs similar to our results, but would not intrinsically provide the increased flexibility for future

application insertion and patching. Moreover, the total manufacturing cost of our design decreases, whereas V_{DD} gated units do not (and may even increase).

Next, we introduce each of the three embedded system models and the notation we use to analyze the energy and delay issues inherent in each. Full analysis of each model is in Section 6.1.5. Additional details on the equations are available in [44].

6.1.2 Legacy

The Legacy device was originally conceived and constructed without expectation for ever needing communication to other systems. We examine the issues of energy and delay in this model by assuming a network link is added and local storage is reduced. The legacy application remains unchanged, but the code and data now come from network memory via a SoftCache mechanism.

The original design expected a certain amount of normal energy consumption during computation and sleep or idle times. To see how adding a network link impacts this, we model the extra energy incurred by using the network link to fetch new code and data, as well as the energy consumed by the network link when in a sleep or idle state. We assume the network link is only used for fetching new code and data, and that the legacy application itself is not attempting to communicate to other devices. We also model the extra time the CPU now spends waiting for network transactions to complete.

In order to "request" new code or data, a message must be generated and sent to the remote server. This transmission time (T_{TX}) will consume energy as determined by the type of network link (E_{TX}). Once the request is received at the remote server, there is some interval of time spent processing the request (T_{Srv}), during which there will be additional energy consumption on the local device monitoring the network (E_{Srv}). Once processed, the server will reply with the necessary information, which takes time to receive (T_{RX}), consuming more energy (E_{RX}). The "payload" of the transmission will consist of some number of bits, which consume power in proportion to the rate of the network communications.

In comparison, local storage only incurs a very minor time to access (T_{DRAM}), with a correspondingly small energy use (E_{DRAM}). Whether transferred by network or from local storage, the

CPU will be idle[1] during these transfers, consuming some amount of energy.

Regardless of which method is used – network or local storage – after transferring the payload, time is spent in computation (T_C) before the next request is generated. During this time, the CPU will consume a different amount of energy while busy (E_C), and the backing store can be put into a powered-down or sleep mode. Thus, during the work period, the network link and local storage will consume their respective sleep or idle power.

The network link energy components in the legacy model are defined by the individual power terms in use during each system mode. These individual terms are expressed in equations 2 through 5. The total energy consumed by the network link (E_N) in the Legacy model is shown in equation 6.

$$E_{TX} = T_{TX}\left(P_{TX} + P_{CpuIdle}\right) \quad (2)$$

$$E_S = T_{Srv}\left(P_{RX} + P_{CpuIdle}\right) \quad (3)$$

$$E_{RX} = T_{RX}\left(P_{RX} + P_{CpuIdle}\right) \quad (4)$$

$$E_{C_{NET}} = T_C\left(P_{NetIdle} + P_{CpuBusy}\right) \quad (5)$$

$$E_N = E_{TX} + E_S + E_{RX} + E_{C_{NET}} \quad (6)$$

The local storage energy terms are shown in equations 7 through 8. The total energy for the local storage (E_L) is shown in equation 9.

$$E_{DRAM} = T_{DRAM}\left(P_{DRAMbusy} + P_{CpuIdle}\right) \quad (7)$$

$$E_{C_{DRAM}} = T_C\left(P_{DRAMidle} + P_{CpuBusy}\right) \quad (8)$$

$$E_L = E_{DRAM} + E_{C_{DRAM}} \quad (9)$$

The two terms $E_{C_{NET}}$ and $E_{C_{DRAM}}$ both use the same value for computation time between memory accesses, T_C. The principal idea is that regardless of *where* the necessary code or data is coming from, there is some constant amount of time T_C spent in computation between accesses to

[1]The CPU could be working on other tasks during this time, thus having a different energy signature. This is addressed later in section 7.4.

off-chip code or data. The difference in these terms stems from the difference between the idle or sleep energies of the network link compared to the DRAM.

In terms of energy, the network model is equivalent to the local storage model when $E_N = E_L$, but to consider the delay impact on application performance, we construct the energy-delay product[2] in equation 10.

$$E_N \cdot (T_{TX} + T_{Srv} + T_{RX} + T_C) = E_L \cdot (T_{DRAM} + T_C) \tag{10}$$

Solving equation 10 for T_C provides the energy-delay equilibrium point where using a network backing store is equivalent to using local DRAM. When T_C is greater than this equilibrium value, the network link is more energy-delay efficient from a total system perspective. That is, so long as the next application page fetched from the network occurs *after* computation for a time period of T_C, the network memory model is more energy-delay efficient.

6.1.3 Pull

Unlike the isolated Legacy model, the Pull model assumes a network link has been incorporated in the embedded device since creation. The critical point is that in the Pull model, the original design engineers already budgeted power for a network link to be present and at least in sleep state. The characterization Pull comes from how the network is used: the local device, on its own initiative, pulls information from the network. External network devices cannot arbitrarily send information to a device operating in a Pull mode.

Using the same basic notation as the Legacy model, there are only minor differences in the energy analysis. The link is classically designed to be in a power-down sleep or idle state during normal operation, except when the running application requests remote activity. Therefore, we only calculate the impact of new behavior (our additional network traffic) over the original expected behavior (sleep state). We need only consider the *difference* between the network link being in sleep state as opposed to actively sending and receiving messages.

[2]When to use which variant of the ED^k metric is an issue of contention; in general, for pure circuit modification techniques, the value to use is $k = 2$ which will indicate whether fabrication process improvements will outweigh any circuit improvement. For our analysis of system-level changes, the more appropriate value is $k = 1$, since no process shrink will address the entire system uniformly nor the significant power necessary for activities like wireless network communications.

The result is that the original Legacy model assumption (all network link energy is a new burden) is invalidated. Instead, we subtract the energy required for sleep-mode from the energy required to transmit and receive information. This change in the additional energy needed represents the new burden on the power source. The modifications to the original equations 2 through 5 are the new equations 11 through 14, which we substitute into the total network energy model of equation 6.

$$E_{TX} = T_{TX}\left(P_{TX} + P_{CpuIdle} - P_{NetSleep}\right) \quad (11)$$

$$E_{S} = T_{Srv}\left(P_{RX} + P_{CpuIdle} - P_{NetSleep}\right) \quad (12)$$

$$E_{RX} = T_{RX}\left(P_{RX} + P_{CpuIdle} - P_{NetSleep}\right) \quad (13)$$

$$E_{C_{NET}} = T_{C}\left(P_{NetIdle} + P_{CpuBusy} - P_{NetSleep}\right) \quad (14)$$

The equations for total local storage energy (9) and the energy-delay product (10) are otherwise the same, given these substitutions.

6.1.4 Push

Similar to the Pull model, the Push model also assumes a network link was built-in originally. In contrast with the Pull model, the network link is always on so that if a device is not actively transmitting, it is in receive-listen mode. Thus external network services can immediately push information to the local device, such as e-mail notices, software patches, etc.

As the Pull model reduces the energy drain to store information in the network compared to the Legacy model, the Push model reduces the drain further. Since the device was designed assuming an always active receive mode network, the original design allotted sufficient power for this purpose. Therefore, we subtract the power term for normal receive-mode network links, rather than the smaller power term for a sleep-mode link as in the Pull model. That is, we only account for the additional energy of both sending extra messages out and idling the CPU during responses.

We again replace the original equations 2 through 5 with our new equations 15 through 18, which we substitute into the total network energy model of equation 6.

$$E_{TX} = T_{TX}\left(P_{TX} + P_{CpuIdle} - P_{RX}\right) \quad (15)$$

$$E_{S} = T_{Srv}\left(P_{RX} + P_{CpuIdle} - P_{RX}\right) \quad (16)$$

$$E_{RX} = T_{RX}\left(P_{RX} + P_{CpuIdle} - P_{RX}\right) \quad (17)$$

$$E_{C_{NET}} = T_{C}\left(P_{NetIdle} + P_{CpuBusy} - P_{RX}\right) \quad (18)$$

As with the Pull model, the equations for total local storage energy (9) and the energy-delay product (10) are otherwise the same, given these substitutions.

6.1.5 Analysis

We now analyze in detail both the energy equilibrium point as well as the energy-delay product for each of the three modes discussed in Section 6.1.1. In order to have a quantitative analysis, we use technical data on current market products for both DRAM and Flash memory.

Current data sheets available from vendors (including Elpida, Fujitsu, Intel, Micron, NEC, and Samsung) for low-power or "mobile" parts represent typical market 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 local storage device. During sleep mode, these devices consume very low current but still require some power for refresh functions. These calculations are shown for DRAM in Table 6, and for Flash in Table 7.

Similarly, we calculate energy information from the data sheets published by several network link vendors. Unlike the DRAM, we determine the *worst*-case power per bit consumed, and the standby or sleep-mode power. In this situation, the transmit (TX) and receive (RX) modes are considered separately, as some links display different profiles by operating state. We restrict our search to monolithic, fully-integrated network modules to ensure valid power measurements. Using multiple chip solutions requires external components and glue logic which make analytical power calculation difficult if not impossible. The components we consider and their power calculations are shown in Table 8.

For our analysis, we demonstrate a conservative extreme: *best*-case local storage vs. *worst*-case network links for remote storage. While neither of these models is generally realistic, they demonstrate the *conservative* bounds where network storage is more effective than local storage. Thus in actual application, network links will be more efficient than we demonstrate here.

Next we define our memory, network, and platform CPU models and specify exact characteristics. Then we observe the basic trade-off between local storage and network storage of information. Finally, we examine whether these trends hold across alternate network choices and the implications for system designers.

6.1.6 Best-Case Memory

To construct the best-case memory power model, we carefully choose to ignore certain effects in the CPU-to-memory interaction.

Since Flash is substantially slower than DRAM, the application is copied from Flash to DRAM for faster execution, and then Flash is placed in deep-sleep mode or V_{DD} gated off. Therefore, we *ignore* the contribution of Flash to the total energy. We also ignore the effects of initiating and waiting for memory access, and assume all accesses begin instantaneously at the maximum supported rate of the DRAM device.

Moreover, we define the transition from idle or sleep mode to active mode as instantaneous. We choose minimal V_{DD} and current consumption at all times, and ignore refresh operations. We also define that any code or data accessed is in the DRAM, and does not load from Flash.

This selection constitutes a *best*-case memory model. For analysis arguments, we use for the DRAM device the Fujitsu FCRAM model MB82D01171A, a 2MB part with the lowest power consumption of all devices measured in pJ/bit.

6.1.7 Worst-Case Network

For this analysis we restrict the additional traffic needed to support the network memory model to unalterable content such as programs, static global data, etc.

We further model the request for code or data to a remote server as fully encapsulated in a 64-byte packet. This packet size could be reduced or expanded based on the network topology and error handling needs, but has sufficient storage space for a range requests. The response packet,

being a variable-payload version of the request packet, consists of 20 bytes for control information followed by the actual payload of variable size. These values are based on our implementations of such a client-server SoftCache system [58].

We assume that for the total count of DRAM chips, at least one is for mirroring part or all of Flash. Based on the working set principle, only a small fraction of this space is actually needed at any given moment. Rather than store a large mirror image, only sufficient space for the worst-case working set should be reserved in local DRAM, with excess DRAM then removed. By using the network link to access applications, we could also shrink the Flash such that it contains only a boot image, and not all applications that could ever be run. This non-volatile memory reduction also reduces the burden of pushing massive code patches out to all systems in the network. Since steady-state mode changes occur relatively infrequently [1, 9], the need to load new code and data from the network will also occur infrequently.

The *worst*-case network model uses typical V_{DD} with worst-case current consumption in all cases. With slower transfer rates, higher current consumption, and a long duration of remote server processing T_{Srv}, the network appears unattractive for energy savings at first glance. We will now demonstrate that this is not the case.

The analysis that follows will implicitly use the concept of one computational task running on the mobile device. Since the CPU would normally be busy during the additional network transactions to receive new code (using the best-case zero-overhead local memory access), we model the CPU as completely idle during these periods. If multiple tasks were present, the CPU could simply switch to the next task and continue processing. This switch would not add the energy overhead of sitting idle and delaying all work, and thus is not the worst-case scenario for network impact.

Our analysis uses the idea that *one* DRAM chip is removed, although it could include reducing Flash as well as multiple DRAM chips. Our network link model is the CSR BC2-Ea, a fully integrated Bluetooth module. This module exhibits a starting time of 10μs and a settling time of 5μs in the internal ADC for gain control. A transition from Active to Sleep mode in the network module is sub-1ms. As we will demonstrate later, the server processing time for requests is set to 10ms. With such a large time for server processing, we ignore these second and third-order effects of the network module design.

6.1.8 Mobile CPU

Our CPU model for the mobile device is the DEC SA-110, a 0.5W processor during high computation and 0.02W during idle periods when operating at 160MHz [75]. While not the most contemporary processor in the Intel ARM-based line, this model has the most exhaustive published power data. Chapter 7 explores using more modern devices, such as the Intel PXA255 processor, as a basis.

Several interesting factors arise from using the SA-110 processor as our representative model. The SA-110 can transition between Idle and Active mode with effectively no delay. This transition is accomplished by the two separate clock domains within the SA-110 – in Idle mode, the internal bus clock and clock grid stop signaling. The actual steps to enable Idle mode are toggling a register, loading an uncachable address, and waiting for an interrupt – a few instructions. The recovery is achieved after receiving an interrupt, and restoring the original register values. Since the transition between Active and Idle mode is nearly instantaneous, we do not model the time necessary for such transitions in the CPU.

6.1.9 Initial Impact

Given that network transmission speeds lag substantially behind the bandwidth of local memories, the bounds on T_C will be dependent on the network speed. With increasing payload in transfers, the remote server processing time becomes less important than the overall network performance. Figure 18 illustrates the boundaries as T_{Srv} varies from zero to one second.

It is unreasonable to assume zero processing overhead on the remote storage system. The incoming network request has to be received, interrupt handlers invoked, memory searched, etc. Using our already existing client-server system as a basis [58], an Intel PIII 800MHz system running RedHat Linux 8.0 is capable of processing and responding to requests in sub-10ms times. During this time the server is also running a fully interactive X desktop with multiple applications running. Therefore, we use 10ms as an approximate remote server processing time. While the remote server could be optimized and made arbitrarily powerful, it will still be serving multiple targets and similar response times may be realistic.

To compare the Legacy, Pull, and Push models using our established T_{Srv} of 10ms, we again plot

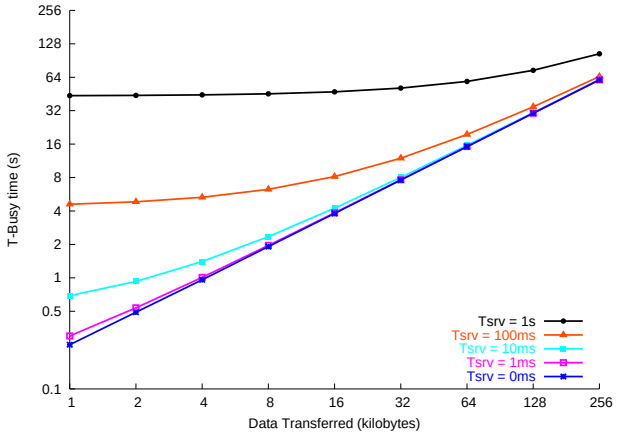


Figure 18: Each line represents the busy-computation time T_C equilibrium point for different remote server processing times T_{Srv}. The network transmission speed is the limiting factor during payload transfers, shown as the asymptote when T_{Srv} = 0ms.

the necessary T_C to reach energy-delay equilibrium compared to local storage accesses. Figure 19 demonstrates the trade-offs between the three models. Any value of T_C beyond the times shown in this figure are a "win" for using remote storage instead of local storage.

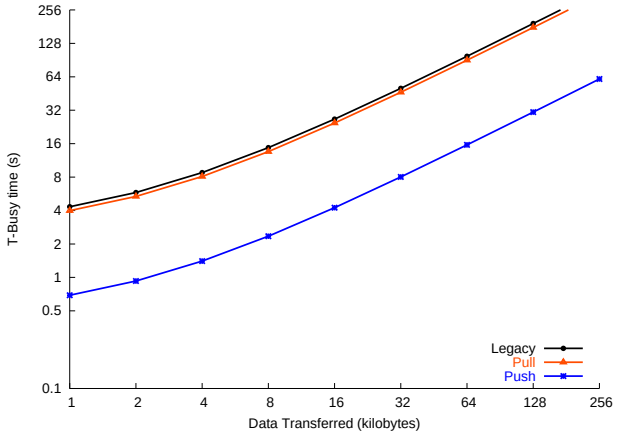


Figure 19: Comparing the energy-delay equilibrium characteristics of Legacy, Pull, and Push models when T_{Srv} = 10ms.

The Legacy model presents the worst energy-delay product result. We have added a network link to a design that did not expect it. For the network link to be more efficient than local DRAM, it requires a significant amount of time spent in computation, T_C.

The Pull model provides better energy-delay results than the Legacy model, as can be expected from subtracting the sleep mode power. The improvement turns out to be small compared to the

energy costs associated with transferring the data as well as the remote server processing time T_{Srv}. The actual difference between the Legacy and Pull models is slightly less than 8%. This result does indicate that adding a network link to a legacy system when using a pull-based communications model will have a small impact when compared to the energy consumed by local storage devices.

The Push model uses the least additional energy and thereby benefits most from using remote storage. Since the network link is already expected to be in a receive-mode state at all times, the only extra energy used to access remote storage is the energy of the transmit operations.

The energy-delay benefit for a 1KB "page" change with a T_{Srv} of 10ms in the Legacy model requires 4.33s as a minimum change interval. With the Pull model, the required busy time falls to 3.99s. When considering the Push model, the time is reduced to 0.69s. In relative comparison, this same 1KB "page" of code loaded across the network with $T_{Srv} = 10ms$ will present a total application delay of 16ms to the user while accessing the network. This result includes the sending, processing, and return of payload through the network.

A larger "page" size to transfer may be more realistic to consider, however. For a 16KB change with $T_{Srv} = 10ms$, the Legacy model requires 26.6s between transfers, and the Pull model requires 24.5s. The Push model reduces this time to a mere 4.2s. The delay the user experiences while the network transfer occurs is 185ms.

6.1.10 Portability

In order to compare these results based on a very low-power Bluetooth integrated module to other network types, we now consider two alternate network models. Neither of these alternatives come in complete monolithic solutions, but instead comprise two or three highly integrated chips with some minimal external glue logic. The estimates for the Cypress Wireless USB chipset and the Bermai Integrated 802.11a chipset include only the main chip components. Power consumption of the glue logic is not considered, and therefore these numbers are slightly smaller than they should be in a worst-case scenario. In particular the 802.11a chipset has particularly large currents in *any* mode of operation before considering the glue components.

Using the Push model as a baseline, we compare the CSR BC2-Ea solution to both the Cypress and Bermai solutions. Figure 20 displays the results of this comparison. The surprising result from

this figure is that the very power-hungry 802.11a network is a much better selection than low-power Bluetooth or similar modules. The substantially higher data rate causes the limiting factor not to be the network link speed, but rather the remote server processing time.

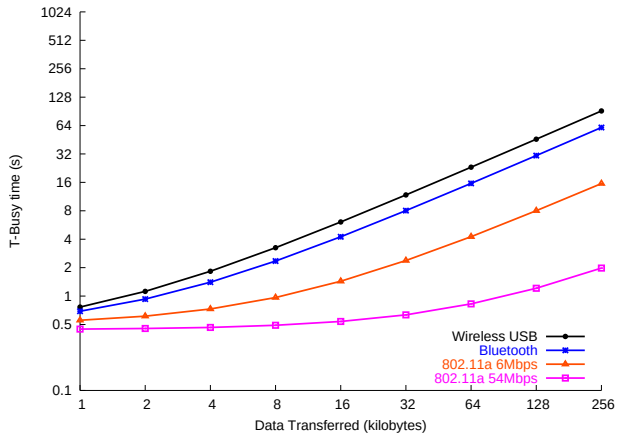


Figure 20: Comparing the Push model energy-delay equilibrium characteristics with Bluetooth, Wireless USB, and 802.11a network modules when T_{Srv} = 10ms.

This comparison is against a Push model, where sufficient power was built-in to support the network in constant-receive mode. A more illustrative example of the substantial power drain involved in 802.11 chipsets can be seen by comparing to the Pull model, shown in Figure 21. Note that the initial energy cost of the 802.11 network far exceeds other options, but that if the typical payload transferred in the network is $\geq$ 24 kilobytes then the 802.11 network is a better design choice.

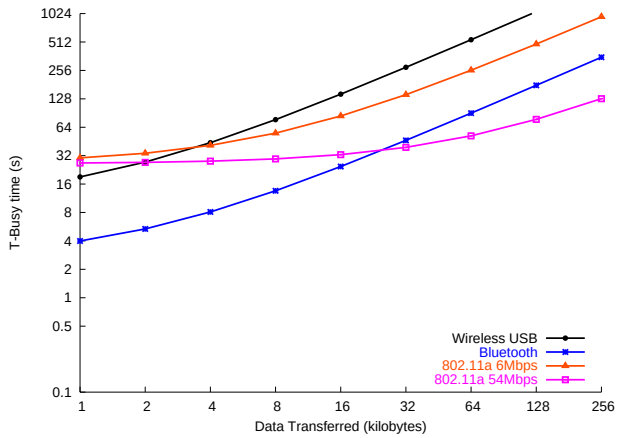


Figure 21: Comparing the Pull model energy-delay equilibrium characteristics with Bluetooth, Wireless USB, and 802.11a network modules when T_{Srv} = 10ms.

While these results do not specifically tie to any estimated average transfer size, they show interesting trends. Ultimately the typical payload size will be entirely dependent upon the applications and network support by commercial providers. This work shows that by performing careful analysis on what types of applications and data will be used in the network, and the characteristics of those applications, increasing local storage may be the wrong approach to longer battery life.

6.2 ARM Processor

Considering the power numbers previously presented for the SA-110 processor, a SoftCache implementation on an SA-110 core would result in immediate and obvious power savings. Given the specifications and implementation details of the SA-110 [75], we can observe the following using typical memory cell layouts [51]:

- The SA-110 uses a fully-associative 32-way tag system
- A typical FA tag bit is implemented as a CAM cell
- Modern CAM cells typically use a 9 or 10T design
- There are 23 address bits in a SA-110 tag, plus a valid bit
- Each SA-110 cache line contains 32 data bytes + 4 physical address bytes + 4 control bits
- Typical uni-ported SRAM cells use a 6T layout, with each additional port using at least 2T
- The SA-110 cache tags consume 12.2% of the total cache T count

Recall that the I+D caches alone consume 43% of the total die power in the SA-110. Using even a rough 50% switching activity model and trying these observations for crude estimation, which are unrealistic assumptions, it is apparent that a drop of 5.2% of the total die power would be achieved by removing the tag storage alone. Further, the SoftCache would remove entirely the I+D MMU systems, resulting in additional power saving of 17%, for a total power saving of 22.2%. In return, a small additional requirement for the extra control logic and necessary additional instructions would be present, reducing this power savings by some amount. This extra power spending we believe is

an order of magnitude less than the power saved. Note that such crude estimations fail to account for logic required to implement comparators, multi-hit resolution in the tag CAMs, etc.

Additional power savings can be achieved by using a detailed static analysis of applications and/or dynamic feedback mechanisms of program performance. By using such analysis information, it is possible to convert the multi-banked SRAM storage on die to permit "drowsy" or "sleep" modes that consume less energy [117, 3, 40]. Different layouts and implementation details for supporting these constructs could present a novel feature such that control over bank power is explicitly exposed to applications for self-optimization – either high performance or low power.

This section investigates these issues of power, keeping focus not only on power but also on performance such that overall application speed is comparable to an unmodified design. Different design approaches for the SoftCache achieve different balances in power and performance.

6.2.1 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.

To understand the overhead for hardware cache memory management, we consider 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 *exclude* parity and ECC bits from our calculations, for if these are needed in the hardware cache they will likely be needed in the generic SRAM replacement the SoftCache uses in the same process. The primary drawback to studying actual processors is that each one implements cache control in a different manner. The XScale, for example, stores the physical address on every cache line since it is a virtually-indexed, virtually-addressed cache. Therefore we use a "baseline" for comparison assuming a 32-way associative cache, of 32-byte line size, with a 32-bit address to memory.

To compute the overhead of the SoftCache, we use details of our real system [46, 58]. In the unoptimized ARM SoftCache, 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 require just under 100 instructions for the miss handler and raw UDP network interface, using 32-bit instructions. We use basic blocks as a unit size of instructions in both the SPARC and ARM SoftCache implementations. This approximates to a branch occurring every 5-7 instructions. For conservative analysis, we define the basic block as five working instructions followed by one branch. The SoftCache carries an extra storage penalty of one additional branch, given that there is no guarantee of contiguous basic block alignments when compared to the original program continuity. 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.

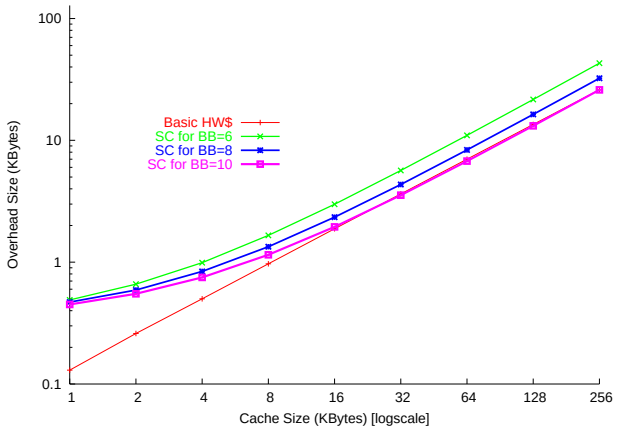


Figure 22: Overhead storage costs by cache size.

Figure 22 shows the results of this comparison. The SoftCache pays a measurable penalty for its 100-instruction miss-handler that make it a losing proposition for caches below 16KB in size. At the 16KB size, the SoftCache is a lower overhead solution. These trends, however, are not typical of real processors. To make a more direct comparison, we consider real cache implementations, as shown in Figure 23.

The microprocessors 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 [25]; the Motorola PowerPC850 [78, 95] and the MIPS R4Kp [73]. The XScale runs with an overhead of 24%. The MIPS has 22% overhead, and the PowerPC has 19%. Our basic hardware cache model is only using 12% overhead, clearly a generous comparison. The SoftCache, however, has overhead directly proportional to the size of the block it operates with. Assuming a basic block

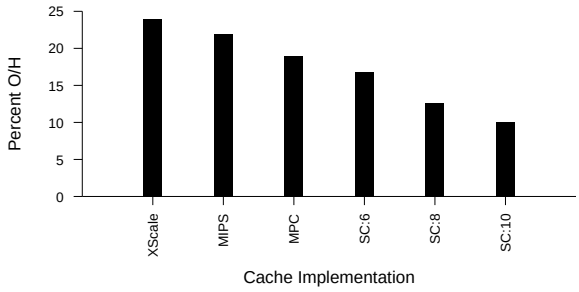


Figure 23: Different real cache overhead costs.

of 6 instructions, the overhead is 16.7%. For basic blocks of 8 or 10 instructions, the overhead drops to 12.5% and 10% respectively.

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. The SoftCache penalty is not constant due to the extra (*worst*-case) assumption of storing an additional branch with every basic block *coupled with* the baseline miss handler. While storing these additional branch instructions with every basic block consumes resources, it takes substantially less than the extra storage used by hardware caches.

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.

6.2.2 Area

Given that the SoftCache exhibits storage overhead usage that is variably better than the small hardware caches we are comparing it to, as shown in Section 6.2.1, we ignore any arguable area savings in the physical storage within banks. For larger cache sizes (above 32KB), the storage overhead savings may 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 [75], the MMUs and write buffer consume approximately 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 [75]. While the tag structure in the SA-110 is using fully associative CAMs, which

contain higher transistor counts than SRAMs, we lack implementation details (9/10/11-T, sizing, process parameters) to establish the potential degree of savings from this structure.

Saving 19% of the total die power by removing 11% of the used area is a significant reduction by itself. The SoftCache design will save area when compared to a traditional hardware cache. 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.

6.2.3 Bank Power

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 [96], the de facto standard for evaluating 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 adds onto the SRAM information the tag CAM cell matching and resolution logic. Typically, the SoftCache could operate faster without the additional hardware of tags, which slow 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 24 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 10% energy is saved. For 64KB, the combined instruction and data cache storage capacity of the XScale, the savings are approximately 6%. This estimated 6% saving tracks well with our earlier gross approximation of a 5.2% reduction (based on switching activity, section 6.2).

For the MMU and write buffers, it is more difficult to measure without a complete real processor implementation, thus we use the published power data of SA-110 as a reference. As shown by

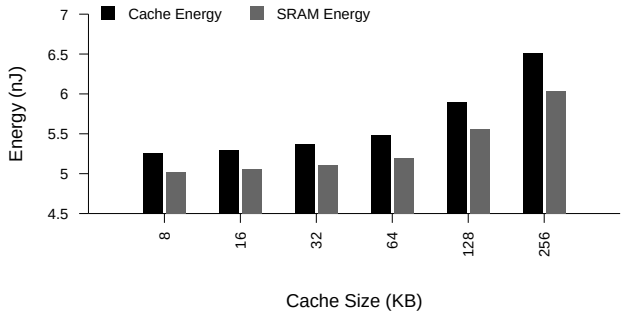


Figure 24: CACTI Power Comparison of SoftCache vs. Conventional Cache

Montanero et al [75], these units consume 19% of the total die power when running a computationally intensive program such as Dhrystone. In other words, by removing these components in a SoftCache system, 20% potential energy savings can be achieved. The study of real hardware power savings is in section 7.

6.2.4 Local vs. Remote Store

The SoftCache model seems counterintuitive since it recommends the reduction (or removal) of local storage (DRAM, NVM) 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.

Our work [45] demonstrates that this is not the case from a purely energy-delay standpoint. Network links are 10 to 100 times more expensive in power than accessing local memories, a fact in agreement with common beliefs. Surprising, keeping DRAMs active without accesses are 10 to 100 times more expensive in power than idling or sleeping network interfaces. We use our prior analytical model for network impact (section 6.1) and incorporate additional terms to track the extra instructions executed in time and energy, as well as the payload transfers during SoftCache 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. This suggests 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 given after analysis in section 6.3.1. Before this answer can be determined, we explore additional aspects of the problem.

6.3 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 transfer 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 [75]. 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 [0.25,0.35]W. For our analysis we have used the reduced 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.

6.3.1 Energy and Delay

While the prior discussion explains penalty instructions 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.

With respect to the network link and the power savings in the SoftCache model shown in section 6.2.2, 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 $E_D = E_L$ gives the average amount of time that must be spent in any given mode before changing. (Each term uses T_C to compute the total energy consumed during the mode.)

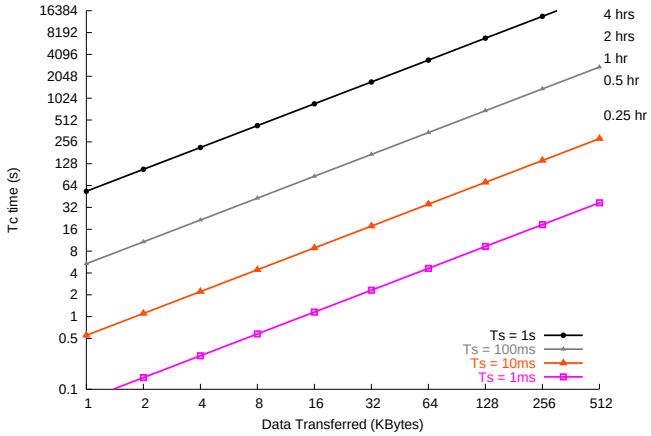


Figure 25: Duration of computation T_C that must pass for the network link to be more energy efficient, where T_S and B_N vary.

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 25. 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 26.

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 [92]. This reduction is possible only if the client initiates connections to the server – 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

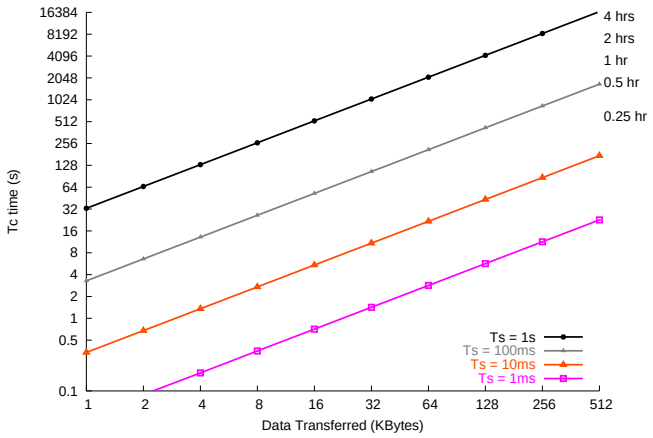


Figure 26: Duration of computation T_C that must pass for the network link to be more energy efficient, where T_S and B_N vary and branch penalty is removed.

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 27 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".

As Figure 27 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 result suggests that loading a "new application" remotely, such as Mario Bros (approximately 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, a transfer of 16KB pages from the network is more energy efficient than local DRAM after four seconds of use, and is more energy-delay efficient after 16 seconds.

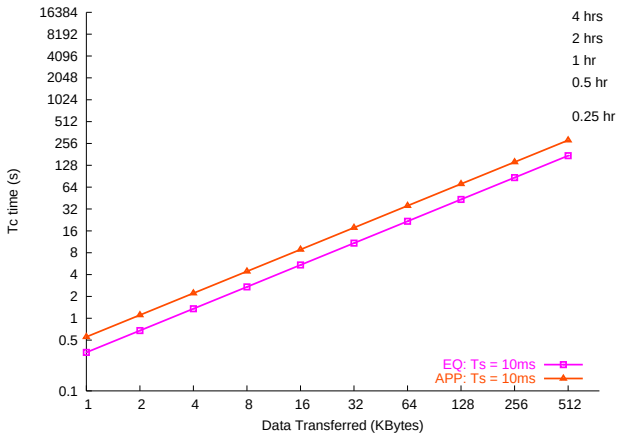


Figure 27: Comparison of time for equilibrium energy win ("EQ") and energy-delay product win ("APP") on application performance. T_S = 10ms.

Vendor	Model	MB	width	MHz	V_{DD}	access mA	sleep mA	pJ/bit	pJ/bit/MB
Elpida	EDL1216AASA	16	16	133	2.3-2.7	80	1.5	86.5	5.4
Fujitsu	MB82D01171A-80	2	16	125	2.3-2.7	20	0.2	23.0	11.5
Micron	MT48V4M32-10	16	32	100	2.3-2.7	100	0.35	71.9	4.5
Micron	MT48V16M16-10	32	16	100	2.3-2.7	80	0.35	115.0	3.6
NEC	μPD4664312	8	16	150	2.7-3.3	45	0.1	49.6	6.2
Samsung	K4S643233-75	8	32	100	2.3-2.7	85	5	61.1	7.6
Samsung	K4S283233-75	16	32	100	2.7-3.6	220	6	185.6	11.6
Samsung	K4S561633-1H	32	16	100	2.7-3.6	130	6	219.4	6.9

Table 6: Mobile DRAM characteristics: size, bit-width, speed, voltage, best-case access current, best-case sleep-mode, current use, pJ per bit in accessing, and a normalized pJ per bit per megabyte of memory. Refresh impact not included.

Vendor	Model	MB	width	MHz	V_{DD}	access mA	sleep mA	pJ/bit	pJ/bit/MB
Fujitsu	MBM29LV320xE	32	16	12.5	2	7	0.005	70.0	2.19
Fujitsu	MBM29DS163xE	16	16	10	1.8	8	0.005	90.0	5.63
Intel	28F256K3	32	16	75	2.7	24	0.030	54.0	1.69
Intel	28F64OW30	8	16	40	1.7	8	0.007	21.3	2.66
Intel	28F32OW18	4	16	66	1.7	7	0.008	11.3	2.82
Micron	MT28S2M32B1LC	8	32	133	3	130	0.300	91.6	11.45
Micron	MT28F642D18	8	16	54	1.7	10	0.025	19.7	2.46
NEC	UPD29F032203AL-X	4	16	12.5	2.7	16	0.005	216.0	54
NEC	uPD29F064115-X	8	16	12.5	1.8	15	0.025	135.0	16.88
Samsung	K9F2816x0C	16	16	20	1.65	8	0.010	41.3	2.58
Samsung	K9F2808x0B	16	8	20	1.7	5	0.010	53.1	3.32
Samsung	K9F6408x0C	8	8	20	1.65	5	0.010	51.6	6.45

Table 7: Low-power Flash characteristics: size, bit-width, speed, voltage, best-case access current, best-case sleep-mode, current use, pJ per bit in accessing, and a normalized pJ per bit per megabyte of memory.

Vendor	Type	Range	Model	kbps	V_{DD}	TX mA	RX mA	sleep uA	TX μJ/bit	RX μJ/bit
AMI Semi	SpreadS	300m	ASTRX1	40	3.3	14	25.0	10.0	1.155	2.063
AMI Semi	Modem	n/a	A519HRT	1.2	5.0	0.6	0.6	n/a	2.500	2.500
CSR	Bluetooth	100m	BC2-Ea	1500	1.8	53	53.0	20.0	0.064	0.064
MuRata	Bluetooth	100m	LMBTB027	1000	1.8	60	58.0	30.0	0.108	0.104
NovaTel	Wireless	n/a	Expedite	38.4	3.3	175	130.0	5.0	15.039	11.172
OKI Semi	Bluetooth	100m	MK70	921.6	3.3	115	72.0	n/a	0.412	0.258
Option	GSM	n/a	GlobeTrotter	116	3.3	550	50.0	50.0	15.647	1.422
Radiometrix	UHF	30m	BiM-UHF	40	5.0	21	16.0	1.0	2.625	2.000
Siemens	Bluetooth	20m	SieMo S50037	1500	3.3	120	120.0	120.0	0.264	0.264
UTMC	Bus	n/a	UT63M1xx	1000	5.0	190	40.0	n/a	0.950	0.200
Vishay	IrDA	Varies	TFBS560x	1152	5.0	120	0.9	1.0	0.521	0.004
Wireless Futures	Bluetooth	100m	BlueWAVE 1	115.2	3.3	60.9	60.9	50.0	1.745	1.745
Cypress	W-USB	10m	CYWUSB6941,2	1000	3.3	120	135	20.0	0.396	0.446
Bermai	802.11a	50m	BER7000	54000	3.3	454	364	3030	0.028	0.022

Table 8: Network link characteristics: speed, voltage, current draw in various states, and worst-case μJ per bit power consumption for TX and RX. The last two entires (Cypress and Bermai) are only approximations.

CHAPTER VII

REAL HARDWARE EVALUATION

Using the results and insights gleaned from Chapter 6, a second study for more contemporary hardware power information is necessary. The prior estimations and component characterizations lack any correlation to modern real hardware, which makes for a weakness in the overall argument. In this section, we use an Intel PXA255 Reference PDA design, the Sitsang-400, to evaluate different characteristics of performance and power consumption by careful application of microbenchmarks. However, this necessarily requires an evaluation of a proper experimental setup in order to have valid power and performance data. This section will explore both proper technique as well as the actual hardware measurements.

Appendix A contains an overview of the proper methods for power measurement, including common fallacies and pitfalls. This appendix also presents sample circuits and error estimations when expensive facilities are unavailable. Chapter A.3 also details the hardware modifications made to facilitate real hardware power measurements. Chapter 7.1 explores the internal architecture of the Sitsang's PXA255 processor, as well as the state of the system board layout. Finally, Chapter 7.3 provides a record of the results from measuring power and performance on the Sitsang platform.

7.1 Sitsang Architecture

The basic power distribution system, simplified for discussion in this thesis, is illustrated in Figure 28. The four tap points for collecting I_{load} and V_{supply} previously mentioned are marked in this figure.

With this power distribution logic, disabling the LCD, Backlight, Audio, and USB eliminates the extraneous peripheral drains from "SYS_PWR". Thus, the system power is driving only the other regulators for the PXA255 processor, DRAM, etc. Similarly, by disabling all peripherals except the Ethernet interface unloads the 4.2V supply except when network activity is taking place. As previously mentioned, the CPLD on the Sitsang motherboard provides a board-level power control

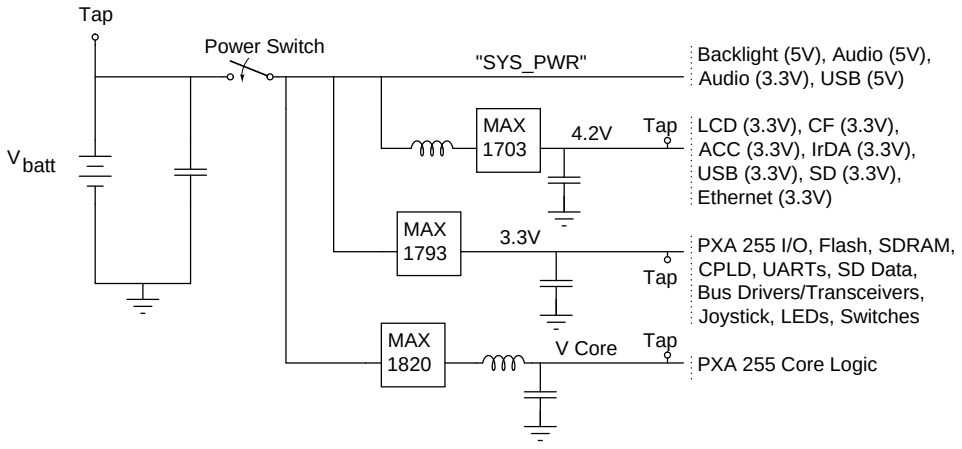


Figure 28: The simplified Sitsang power distribution logic.

register. This control register includes a bit for toggling power to the Ethernet interface, an SmSC LAN91C96 module. By careful isolation of a quiescent system, to a system with the network chip enabled, to a system in active transmit or receive mode, isolation of the various power terms involved in supporting a 10Mbps twisted pair LAN is possible.

With the 3.3V supply, disabling the LEDs, switches, and joystick minimize the drain. The bus drivers/transceivers cannot be disabled, nor can the CPLD. The primary problem then becomes isolation of the power components on the 3.3V supply for the SDRAM, Flash, and CPLD. The $V\ Core$ voltage is easily measured independent of the rest of the system. Not shown in Figure 28 is the ability to alter the core voltage with on-board software controlled selectors. At this time, experiments in DVS/DFS/DVFS are ignored, and this area is left for future exploration.

To isolate the individual power terms for the different primary components, it becomes necessary to run a series of empirical tests and perform linear regression analysis [17] as well as model fitting [17] to extrapolate each device's power profile. The mechanisms for obtaining the empirical data are dependent upon the internal architecture of the PXA255. A simplified view of the PXA255 SoC internals [32] is shown in Figure 29.

Of particular importance in the internal block digram of the PXA255 is the system bus that sits between the XScale microarchitecture core and the rest of the system. While exact specifications for the system bus are not published, this diagram shows that the XScale core running from caches touches nothing else in the SoC. However, any time the XScale core goes to memory, it travels the system bus to the memory controller prior to going off-chip (assuming no DMA involvement).

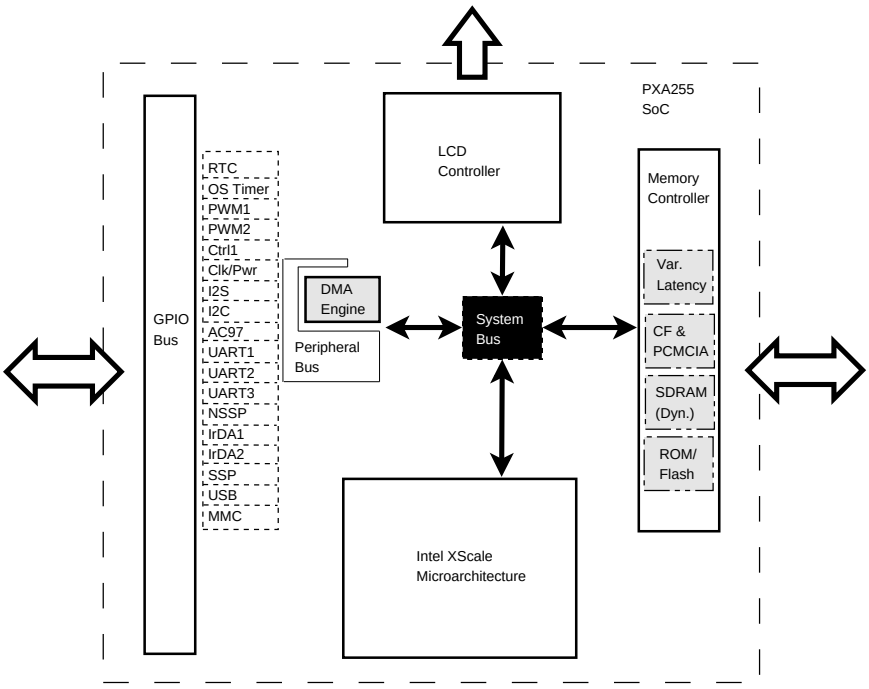


Figure 29: A simplified internal diagram of the PXA255 processor.

Similarly, to read or write the network, the XScale core goes through the system bus to the peripheral bus to the SSP controller and then off-chip. To better illustrate the internal workings of the XScale core, Figure 30 shows the major functional blocks [34, 32, 33].

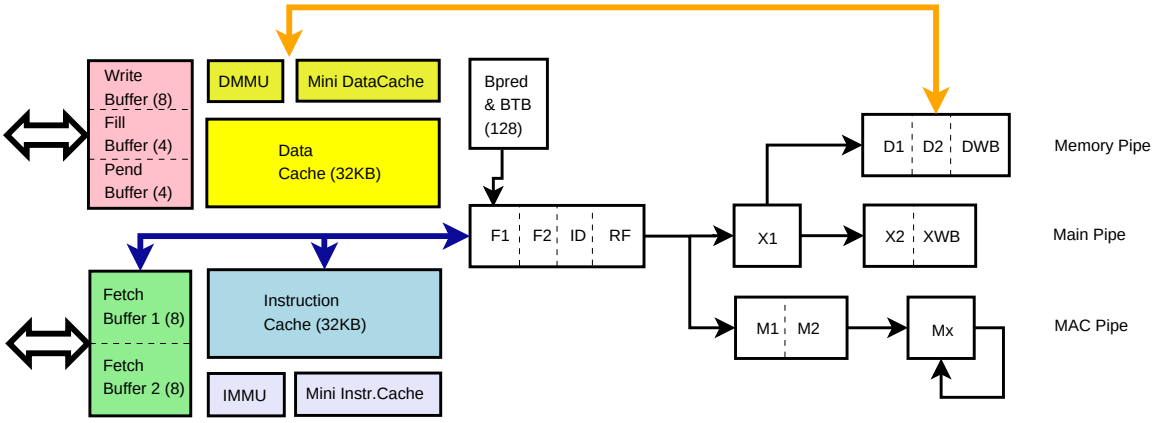


Figure 30: A simplified internal diagram of the XScale microarchitecture inside the PXA255 processor.

The XScale microarchitecture allows for a substantial amount of internal state configuration, including the enable/disable of caches, MMUs, the BTB, and other minor elements. However, it is not possible to disable the fetch buffers. Each fetch buffer is a sequential 32-byte block from a

"missed" instruction target. Therefore, even with the caches disabled, the XScale core will continue to execute any instructions that it can which exist in the fetch buffers. During this period, any miss will cause the next fetch buffer in a rotating basis to be evicted and the new target instruction sequence loaded.

With the fetch buffers always active, it is possible to have a quiescent system that is continuously executing up to 16 instructions from the two fetch buffers while never touching caches or main memory. This is a useful quirk in the system design that will enable us to perform microbenchmark studies to achieve our goal of linear regression with model fitting to accurately construct a power model to examine the SoftCache system.

The primary components we must characterize in power to evaluate the SoftCache are: (1) data and instruction caches; (2) data and instruction MMUs; (3) PXA255 busy and idle states; (4) SDRAM access and idle states; (5) ethernet receive, transmit, and idle states. In the next section, we sketch our microbenchmark approaches for each of these tasks.

7.2 Tuned microbenchmarks

To design a microbenchmark that will behave exactly as expected requires a safe operating environment. Running microbenchmarks under complex operating systems or interrupt-driven embedded environments may allow non-deterministic behavior which inhibits repeatibility of results. To solve this problem, we ported the U-Boot[1] boot loader framework to the Sitsang platform. U-Boot is an entirely polling-based bootloader environment, with the added feature of allowing programs linked against the U-Boot binaries to be loaded and run from the command line as extensions to U-Boot. Therefore, after successfully porting U-Boot and enabling the minimal necessary features for the PXA255, SDRAM, and ethernet, microbenchmarks are loaded as U-Boot extensions.

To ensure that the microbenchmarks run as intended, each microbenchmark is hand-coded in ARM assembly language and then compiled with *gcc -O0* to prevent compiler optimizations. These microbenchmark binaries are then loaded via Motorola S-record format over the serial port, and executed. Upon completion of the microbenchmark, execution returns to the U-Boot environment.

[1]The open-source U-Boot project is located at http://u-boot.sourceforge.net. Our patches are in revision for acceptance to the mainline release.

While U-Boot does not support interrupts directly, this lack of support is a feature where all the hooks are in place but are not activated. For any microbenchmark that needs to use interrupt-based behavior, that microbenchmark can enable interrupts on startup and disable them on exit. Therefore, there is no impediment *per se* to working with interrupts, but the microbenchmark must do all the required work of managing the interrupts.

We now sketch the pseudo-code for each of the microbenchmarks used. The default U-Boot status is to disable all MMUs, caches, etc. Similarly, all peripherals are disabled except the serial port for host communications. Each microbenchmark will activate only those resources of interest.

7.2.1 Caches

The first problem is to classify the baseline power consumption of each cache in an idle state, captured by the microbenchmark in Listing 7.1. This short program simply alternates the caches which are enabled, but is careful to avoid actually running from the caches. A particular problem with the XScale implementation from Intel is that even when the caches are disabled internally, they are still accessed for every miss in instruction or data reference. Therefore, enabling or disabling a cache will have zero net power impact. This first microbenchmark is to confirm this detail of implementation.

Listing 7.1: Idle cache power classification

```
1  for (i = 0; i <= COUNT; i++)
2      ; // busy idle, caches off
3  cacheOn( ICACHE )
4  for (i = 0; i <= COUNT; i++)
5      ; // busy idle, Icache only
6  cacheOff( ICACHE )
7  cacheOn( DCACHE )
8  for (i = 0; i <= COUNT; i++)
9      ; // busy idle, Dcache only
10 cacheOn( ICACHE )
11 for (i = 0; i <= COUNT; i++)
12     ; // busy idle, I+Dcaches on
13 cacheOff( ICACHE )
14 cacheOff( DCACHE )
```

Understanding the cache miss event is clouded by the behavior of the various buffers. For the instruction cache, any miss will allocate one of the two instruction fetch buffers. The target address is loaded as part of the 32-byte cache line that would occur for the target. Once the fetch buffer has been loaded, if the instruction cache is enabled, the fetch buffer will be written into the cache.

Therefore, there are two components to instruction cache miss power – fetch buffer loading, and fetch buffer to cache line transfer. The microbenchmark to capture this behavior is shown in Listing 7.2.

Listing 7.2: Instruction cache miss power classification

```
1  //
2  // classify just the fetch buffer load cost
3  //
4  for (j = 0; j <= COUNT; j++)
5  {
6      for (i = 0; i <= COUNT; i++)
7          ;                                       // busy idle, caches off
8      goto miss1
9          __asm("nop, nop, nop, ...., nop")  // 32 bytes of no-op (8 nop's)
10     miss1:
11     for (i = 0; i <= COUNT; i++)
12         ;                                       // busy idle, caches off
13     goto miss2
14         __asm("nop, nop, nop, ...., nop")  // 32 bytes of no-op (8 nop's)
15     miss2:
16     for (i = 0; i <= COUNT; i++)
17         ;                                       // busy idle, caches off
18     goto miss3
19         __asm("nop, nop, nop, ...., nop")  // 32 bytes of no-op (8 nop's)
20     miss3:
21     for (i = 0; i <= COUNT; i++)
22         ;                                       // busy idle, caches off
23     continue;
24         __asm("nop, nop, nop, ...., nop")  // 32 bytes of no-op (8 nop's)
25 }
26 //
27 // now, classify fetch buffer to cache line transfer
28 //
29 cacheOn( ICACHE )
30 for (j = 0; j <= 1024; j++)
31 {
32     for (i = 0; i <= COUNT; i++)
33         ;                                       // busy idle, Icaches on
34     goto miss1
35         __asm("nop, nop, nop, ...., nop")  // 32 bytes of no-op (8 nop's)
36     miss1:
37     for (i = 0; i <= COUNT; i++)
38         ;                                       // busy idle, Icaches on
39     goto miss2
40         __asm("nop, nop, nop, ...., nop")  // 32 bytes of no-op (8 nop's)
41     [...]
42     miss128:
43     for (i = 0; i <= COUNT; i++)
44         ;                                       // busy idle, Icaches on
45     invalidateCache( ICACHE )
46 }
47 cacheOff( ICACHE )
48 invalidateCache( ICACHE )
```

The first part of this listing has three short loops that each occupy one fetch buffer, forcing

a continual collision and buffer refetch. The second part of the listing models the transfer from the fetch buffer to the cache line, with a periodic invalidate with the goal of capturing the power signature of just one transfer on the oscilloscope.

Instruction cache hit behavior is easily established by repeating the miss characterization and eliminating line 48 – the cache invalidation. The end result is that after one iteration through the outter loop, the cache will be warm with the code. All cache accesses after that point are hits.

The data cache is easier to characterize than the instruction cache. While the data cache is always accessed regardless of the enabled status, it has no fetch buffers to interfere with miss measurements. However, evictions do travel through the write buffer, which has 8 entries of 16 bytes each. Each cacheline has two dirty bits, each corresponding to 16 bytes on the line, to reduce write-back memory pressure.

To capture the states of the datacache, the benchmark of listing 7.3 is sufficient. One drawback to the XScale implementation is that the datacache cannot be activated with the MMU disabled, otherwise undefined behavior occurs. To correct for the MMU overhead, we determine the MMU power signature later in this section.

The use of these microbenchmarks will isolate power terms to distinguish between various cache hits and cache misses, as well as supporting buffers. However, the inability to full deactivate the cache lookups suggests it is not possible to determine the exact cache energies.

Listing 7.3: Data cache power classification

```
1  //
2  // run a loop to do cache misses
3  //
4  int x, y, * p = 0x0;
5  cacheOn( ICACHE )
6  cacheOn( DCACHE )
7  for (j = 0; j <= COUNT; j++)
8  {
9     for (i = 0; i <= COUNT; i++)
10       ;                                  // busy idle, caches off
11    x = *(p + j*32);                      // load next cache line (force miss)
12 }
13 //
14 // run a loop for cache hits
15 //
16 p = &x;
17 for (j = 0; j <= COUNT; j++)
18 {
19    for (i = 0; i <= COUNT; i++)
20       ;                                  // busy idle, Icaches on
21    y = *p;                               // load-hit
22 }
23 //
24 // run a loop for write-backs
25 //
26 for (j = 0; j <= COUNT; j++)
27 {
28    for (i = 0; i <= COUNT; i++)
29       ;                                  // busy idle, Icaches on
30    y++;                                  // write a new value
31    invalidateCache( DCACHE );            // force writeback
32 }
33 cacheOff( DCACHE )
34 invalidateCache( DCACHE )
35 cacheOff( ICACHE )
36 invalidateCache( ICACHE )
```

7.2.2 MMUs

The primary goal of interest is to establish how much power each of the two MMUs consume. For the SoftCache implementation, these units will be simply removed. Therefore, we are less interested in the possible energy consumption in exceptional cases, and focus only on establishing their consumption during steady state. With the XScale/StrongARM design, it is important to realize that the virtual-index, virtual-tag system makes TLB operations off the critical path. Instead, the TLBs are checked in parallel with the cache access. On each cache line, the physical address is stored that represents where the cacheline was loaded from. After the TLB physical address is

generated, it is then compared to the cacheline copy of the physical address. The primary goal is to establish how much overhead the TLB lookups on every cache access – instruction or data – add to the basic cache access.

Both the instruction and data MMUs are identical in nature and implementation. Isolation of just one such MMU will give the consumption for both units. Since the XScale implementation allows for independent manipulation of the instruction cache and instruction MMU, we study the IMMU interaction behavior to determine the power overhead. The microbenchmark to capture this is shown in Listing 7.4.

Listing 7.4: Isolation of the IMMU power consumption

```
1  //
2  // test the IMMU power overhead
3  //
4  cacheOn( ICACHE )
5  for (j = 0; j <= COUNT; j++)
6  {
7      for (i = 0; i <= COUNT; i++)
8          ;                                   // busy idle, Icaches on
9      MMUsetup( IMMU, AllMem, CACHABLE )
10     MMUenable( IMMU )
11     for (i = 0; i <= COUNT; i++)
12         ;                                   // busy idle, Icaches on
13     MMUdisable( IMMU )
14     MMUinvalidate( IMMU )
15 }
16 cacheOff( ICACHE )
17 invalidateCache( ICACHE )
```

7.2.3 PXA255 States

We are primarily interested in the different states of the PXA255 for busy and idle. This is somewhat complicated since the PXA255 can run in one of three modes – turbo, run, and bus. Typically, the PXA255 has clock multipliers set up such that the turbo speed is 400MHz, run speed is 200MHz, and bus speed is 100MHz. A third mode exists beyond busy and idle, the sleep mode, but we ignore this as unrelated to the SoftCache design. Any time the CPU would be in sleep mode, the user application has requested it. The SoftCache switches from busy to idle mode while waiting for the remote server to respond to queries, and thus these two modes are sufficient. To establish an upper bound of the busy power, we need to hand-craft an assembly code microbenchmark to fully utilize each pipeline in the PXA255. Based on Figure 30, this should balance the multiply-accumulate

(MAC) execution with load/store operations and traditional instructions. The microbenchmark we use to approximate maximum busy power behavior is shown in Listing 7.5 based on a *run mode*, not the turbo mode.

Listing 7.5: Approximating maximum CPU utilization for the PXA255

```
1  //
2  // Load up the CPU to be very, very busy
3  //
4  cacheOn( ICACHE )
5  __asm{
6       ldr       r0, #1
7       ldr       r1, #2
8       ldr       r2, #3
9       ldr       r3, #4
10      ldr       r4, #COUNT
11 L1:
12      mla       r6, r2, r3, r1
13      add       r7, r0, r1
14      ldr       r9, Ld
15      mul       r8, r2, r3
16      subs      r4, #1
17      ble       L1
18      b         L2
19 Ld:
20      .word     #0
21 L2:
22 }
23 cacheOff( ICACHE )
24 invalidateCache( ICACHE )
```

Switching to the idle mode on the PXA255 is quite trivial. The behavior in idle mode is that the core CPU logic is suspending, pending an external event including reset, interrupt, or peripheral requests. During the idle mode, all peripherals include the DMA controller, LCD engine, etc., work as though the CPU were in ones of the standard run modes. Therefore, the only task for moving to idle mode is to write a co-processor register. The CPU core will suspend execution at that point. The next instruction after the co-processor write will not take effect until the idle mode is deactivated by an external request. Listing 7.6 illustrates this simple test.

Listing 7.6: Putting the PXA255 into idle mode

```
1  //
2  // Switch the CPU to idle mode, and wait for the
3  // co-processor write to finish
4  //
5  setMode( IDLE );
6  __asm{"CPWAIT");
```

7.2.4 Ethernet

The nature of the SmSC LAN91C96 ethernet unit's connection to the power subsystem makes it remarkably easy to isolate the power consumption of the network module. The ethernet chip and associated logic draws its power from the 4.2V supply of Figure 28. To observe the power consumption of the ethernet in the three states idle, receive, and transmit is simply a matter of driving the power control register in the CPLD as well as carefully writing or polling the ethernet device. While this unit is not a wireless connection as dicussed in Section 6.1.7, the same types of applications exist for wired networks as wireless. Of particular interest in the power study here is the ability to isolate the I/O driving power for the PXA255 based on the three separate supply lines involved. On the PXA255, the core logic power comes from one supply line, whereas the I/O pad power comes from the 3.3V regulated supply. The fluctuations in the 3.3V supply will correspond to the activation of the I/O pins during transactions to the network interface, which is exclusively driven from the 4.2V supply. The microbenchmark in Listing 7.7 shows the isolation of the three ethernet states, which as a side effect offers an estimate of the PXA255 I/O power use.

Listing 7.7: Isolating the Ethernet power states

```
1  //
2  // isolate the network power terms for the 10Mbps TP LAN chip
3  //
4  cacheOn( ICACHE )
5  for (j = 0; j <= COUNT; j++)
6  {
7      powerOn( ETHERNET )
8      for (i = 0; i <= COUNT; i++)
9          ;                                   // busy idle, Icaches on
10     for (k = 0; k <= COUNT; k++)
11     {
12         ethernetRead( DATAREGISTER )
13         for (i = 0; i <= COUNT; i++)
14             ;                               // busy idle, Icaches on
15     }
16     for (k = 0; k <= COUNT; k++)
17     {
18         ethernetWrite( DATAREGISTER )
19         for (i = 0; i <= COUNT; i++)
20             ;                               // busy idle, Icaches on
21     }
22     powerOff( ETHERNET )
23     for (i = 0; i <= COUNT; i++)
24         ;                                   // busy idle, Icaches on
25 }
26 cacheOff( ICACHE )
27 invalidateCache( ICACHE )
```

7.2.5 SDRAM

The problem with isolating the SDRAM access energy is that the SDRAM chips themselves are powered from the same supply line as the PXA255 I/O pads. Therefore, in order to determine how much power is being consumed from a fetch buffer as opposed to the SDRAM, it is necessary to eliminate the I/O pad power. By establishing the I/O pad power from the network microbenchmark, we can approximate the actual power consumption of the SDRAM alone.

There are four primary modes of interest with the SDRAM on the Sitsang – idle, short burst fetch with a row buffer hit, a row buffer miss, and a sustained burst DMA style transaction. The first three of these are easily obtained by using the same microbenchmarks as the cache hit/miss tests. The last of these is obtained by setting up the DMA engine to copy a short region of memory from to an adjacent location. There is, essentially, no real microbenchmark necessary as the DMA engine configuration is simply a linked list construction and insertion.

7.3 Sitsang Measurements

Combining the actual instruction and data caching overheads, along with the results of the microbenchmark studies, yields a comparison of the Sitsang as a test platform for all three scenarios – legacy, pull, and push. The plot of results is shown in Figure 31.

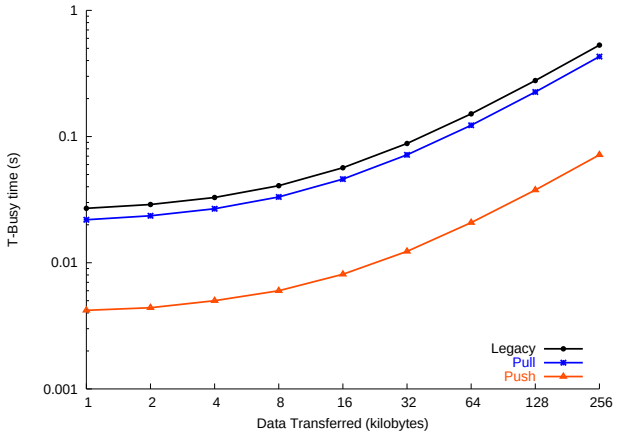


Figure 31: Comparison of the three possible evaluations – Legacy, Pull, and Push – on the actual Sitsang hardware, when T_{Srv} = 10ms.

The exceptionally low busy-time required for computation between transfers is a function of the actual network link in use. The wired ethernet port draws substantial power during transmission, but uses very little power during reception. The idle state is equal to the receive state, and V_{DD} gating is not considered at this time.

The end result is that for a Sitsang board with a 10Mbps wired network connection many applications are quite viable, assuming all DRAM is removed and running a projected set of modifications to the PXA255 processor as previously outlined. This also assumes that the LCD panel and other peripherals are not in use.

Based on these power numbers, the following MiBench applications are all energy-delay wins on the Sitsang platform: bf, bitcnts, crc, math, pgp, rawcaudio, rijndael, search, and sha. These 9 benchmarks represent *definitive* wins for the SoftCache. Two additional benchmarks – toast and lout – *may* be wins depending on the actual data set used and network delays.

If the only area under consideration was the instruction cache, then all but the MP3 application lame would be definitive wins for the SoftCache solution. Due to the design of the benchmarks, however, and the very large input files, the data cache is easily overrun with misses and evictions.

7.4 Multi-tasking

In a SoftCache system where all virtual memory support is eliminated, an issue of how to handle multi-tasking arises. Fundamentally, there are three basic approaches to handle multiple applications, but the simplest answer is to simply not allow it. The version of the SoftCache in this thesis is a first-pass mechanism to support the idea of flexible low-power ubiquitous devices. Our initial target has no requirement for multi-tasking. The next generation of SoftCache system may need to provide this feature, however, and we sketch three solutions briefly.

The first option is to design applications such that the need for context switching is infrequent. If the rate of switching is sufficiently low, then moving from one application to another is no different from the SoftCache perspective of a single application loading additional code and data. When the timer interrupt occurs, the entire state of the current SoftCache system is swapped out in one large transaction, and the new process is swapped in. The primary implication from the data in this section is that with current *wireless* network technologies, the power penalty of such a swap would necessitate a context switch time measured in tens of seconds.

The second option is to increase the SoftCache storage space locally to hold up to N process working sets. The clear drawback is that on-die SRAM is expensive in terms of real estate and power, with leakage power increasingly a problem as processes plunge below 180nm. A more effective variant is to use multi-threshold logic or different oxide thicknesses for on-die SRAM as opposed to the core logic of the microprocessor iteself.

The third option strikes a compromise for the second option, namely the increase of local storage. With current generation XScale parts like the PXA270-family [26], each processor package is actually a stacked set of three wafers: an XScale layer similar to the PXA255, a moderate capacity DRAM up to 32MB, and a moderate capacity StrataFlash up to 32MB. The implementation detail of accessing the stacked package is that the on-die XScale cache is as fast as always. This would be the "working set" of the currently running application. The in-package DRAM or Flash is a minimum of 10 cycles away, across several intervening busses. While this distance makes the access of on-package storage access fast compared to a main memory or network memory, it is far too slow for a per-instruction or per-load/store operation basis. Therefore, when a timer interrupt

occurs: (1) the local client copies the active working set into the on-chip storage, then (2) sends a message to the server indicating what the final state of the just-switch application was, and (3) loads the next application from on-package memory, and finally (4) notifies the server which application was reloaded for the next increment of execution. This will be a very high-performance context switch, providing a balance of local and remote storage uses. The remote server will manage the on-package memories and juggle the migration of tasks in and out of remote devices.

These three solutions are far from the only solutions possible. Hybrid approaches with a mixture of SoftCache and traditional cache or virtual memory support are also possible, as well as mixtures of our own solution sketches. The primary point is that context switches present no additional complexity to support than supporting the SoftCache itself – the question lies in how to handle the energy-delay side effects, which will be application specific.

7.5 Related Work

Utilization of the network for accessing backing store as a low-power mechanism is novel. Prior work concentrated on using remote memories for high-performance reasons, avoiding accesses to slow disks or to expand memory for working sets of code or data [59, 60, 31, 41, 72, 86]. Other work examining the network in power-limited devices has concentrated and minimizing the usage [52] and optimizing protocols.

A significant amount of effort is being spent to find ways of improving the overall energy efficiency of networks. WLANs can improve their efficiency by using ad-hoc relaying [70], while others look at tying battery level with ad-hoc routing methods to increase network robustness as well as node run-times [69].

Using the availability of low-power short-range devices such as Bluetooth, researchers are building larger networks in an energy-efficient manner. These new systems compete with more traditional network options [5]. Such prototypes strengthen the viability of using limited embedded hardware for larger projects.

With each generation of network technology, data rates increase and power consumption decreases. Next-generation technology such as Ultra-Wideband is anticipated to be higher data rate and lower power than current Bluetooth devices, with similar if not better range. As network links

approach the performance characteristics in bandwidth and power of local DRAM, the argument for moving to network-based storage becomes more compelling.

A large body of research [23, 61, 62, 16, 87, 88, 49, 105, 98, 116, 14, 97] has explored power simulations, measurements, and simulators. Different groups take different approaches, ranging from cycle-accurate simulators to trend-predicting approximations. Regardless of the methodology and scale of most studies, few groups examine the error terms or implicit assumptions about how their underlying models are constructed. While no one group can cover every detail, more effort should be put into constructing valid accurate and precise power models and simulators, where results are consistent with a small variance across many runs.

CHAPTER VIII

CONCLUSIONS AND FUTURE WORK

8.1 Conclusions

In this thesis, we presented a novel architecture called SoftCache to address the issues of power, cost, and complexity for embedded systems. We reduced the on-die memory controller infrastructure which reduces both power and space requirements, using the ubiquitous network device arena as a proving ground of viability. In addition, the SoftCache achieved further power and area savings by converting on-die cache structures into directly addressable SRAM and reducing or eliminating the external DRAM.

To avoid the burden of programming complexity this approach presents to the application developer, we provided a transparent client-server dynamic binary translation system that runs arbitrary ELF executables on a stripped-down embedded target. The drawback to such a scheme lay in the overhead of hundreds of additional instructions required to effect cache behavior, particularly with respect to data caching. Another substantial drawback is the power use when fetching from remote memory over the network. The SoftCache was built upon this dynamic client-server translation system on simplified hardware, targeted at Intel XScale (ARM) client devices controlled from Intel x86 servers over the network.

Reliance upon a network server as a "backing store" introduced new levels of complexity, yet also allowed for more efficient use of local space. The explicitly software managed aspects created a cache of variable line size, full associativity, and high flexibility. This thesis explored these particular issues, while approaching everything from the perspective of feasibility and actual architectural changes.

The novel contributions of this thesis to the existing body of research are briefly summarized as:

- A distributed client-server dynamic binary translator/rewriter, a framework that is used to implement the SoftCache system in a ubiquitous network environment

- A novel memory characterization based on target address enumeration, a technique that facilitates solving the previously intractable problem of software emulation for data caching
- A novel energy-delay study comparing remote network accesses to local main memory, indicating that local memory is less than ideal

These techniques partially relaxed the problems faced with current and next-generation designs. Reduction of the logic required for memory controllers and simplification of on-die memories shrinks die size and reduces latency effects. Application growth and corresponding complexity is transparently solved with dynamic binary translation, and manufacturing cost for the embedded domain is reduced with simpler device development models.

8.2 Future Work

Several of the results in this thesis are early results dependent on simulation and analytical models. While working prototypes exist, they are not sufficiently robust to run truly *any* arbitrary ELF binary. There are several avenues for future exploration and growth in the SoftCache framework.

8.2.1 Program Analysis

One major issue is that the current program analysis is overly conservative. There may be other opportunities in the CFG and/or DFG reconstruction to reduce ambiguity and unknown branching. Further studies in particular should focus on the semantic division of data references, and attempt to find patterns similar to the UTI-MTI division to further break up data segments into simpler units.

8.2.2 Cache Implementations

The SoftCache support for both instruction and data regions uses a FIFO replacement model, along with the basic block subset collision problem solely in instruction cache regions. The next steps for research should investigate the ability to use larger block sizes effectively, such as hyperblocks or superblocks. Another avenue for exploration is whether entire phase changes can be predicted, such that the full working set can be swapped between the client and server to avoid the on-demand gradual replacement that presently transpires. Lastly, algorithms other than FIFO replacement should be considered.

8.2.3 Analytical Hardware Models

While the power and space data presented in this section shows a trend, it lacks calibration with modern processors and modern fabrication techniques. Recent studies have been launched to develop full power models of processors like the XScale [23]. However, the results to date have marginal validity, given the methods employed. A precise, complete-system with detailed processor power model which is a cycle-accurate simulator is missing. Therefore, we propose to first develop such a simulator with as much precision as possible based on the reference Intel Sitsang platform, with a PXA-255 processor. The resulting model should be a cycle-accurate simulator that incorporates power data for every component in all modes of operation – DRAM, CPU, caches, MMU, etc.

Once such a simulator exists that has been calibrated exhaustively to real hardware, the SoftCache may be run inside of the simulator to observe its true power signature. Running the SoftCache on existing hardware will always result in misleading information, since it is not possible to fully disable the logic we propose to remove. While real performance approximations can come from real hardware, expected power consumption can only come from such detailed simulations.

The other area which should be studied is the network utilization and timing impact. The power signature of the network traffic will be contained in the power model. However, different methods and types of network transmission will impact the practicality of the SoftCache.

8.2.4 Real Hardware Feasibility

While the analytical model is accurate to the standards of datasheets from manufacturers, most datasheets are actually analytical model results themselves. This introduces a source of compounding error, as none of the analytical models use data that are necessarily calibrated to real hardware. However, no existing chips exist to actually enable a full SoftCache hardware implementation for empirical study. Future research should try to experiment with other designs or even custom microprocessors in order to get a more accurate picture of actual power consumption. Reports in the literature for power consumption in microprocessors is typically derived from Synopsys PowerMill simulations, with extremely loose correlation to the actual hardware.

8.3 Future Vision

The techniques and ideas developed in the course of our SoftCache system have novel properties that are not constrained to just use within the embedded ubiquitous network space. The concept of a client-server communications layer to emulate caches is equally useful in large-scale chip multiprocessors (CMPs), where not all cores will have uniform access to memory and other resources. This suggests the idea of *edge translation*, where cores that sit near memories process applications into smaller pieces that are then run on distant cores. The on-die interconnects possible in large-scale CMPs, with 64 or 128 cores on one die, will act as extremely high bandwidth and low-latency networks.

Moreover, the idea of the UTI/MTI memory characterization is highly applicable to high performance hardware caches as well. The idea that 30% of dynamic memory traffic can be encapsulated in a very small 1-2KB storage SRAM suggests that new hardware cache strategies could improve hit rates by protecting UTI data from the mass pollution by MTI data. This same concept can also be applied to *any* caching medium, from Internet routing tables to directory tables in shared memory computers.

Most importantly, we challenge the inherent assumption that using a network is *detrimental* to power consumption. As a general heuristic, it is entirely context driven based on applications and application data characteristics. By demonstrating a large class of applications where this common belief is inaccurate, other common belief laws also become prone to challenges. In a field where most of the major contributions were developed during the 1960's, it is necessarily time to reconsider whether the assumptions from that era still hold true today.

APPENDIX A

PROPER POWER MEASUREMENT METHODS

In any scientific publication, there are two explicit objectives: (1) to explain the research, and (2) to provide sufficient detail that others may reproduce the exact experiment and verify the published results. A key problem lies in reproducibility, since it carries implicit requirements that the setup used by the original research group is both *accurate* and *precise*.

Accuracy is a measure of correctness in the experimental setup, such that if the real value is 92.3mV, the measured value is as close to 92.3mV as possible. *Precision* is the consistency of the apparatus, such that if the experiment were performed N times (where N is a large value), the variance in the measured value is very small. Having just one of accuracy or precision is useless for research efforts.

How to obtain both accuracy and precision at the same time is a context-sensitive problem. While each scientific field has unique problems and solutions, we restrict our discussion to just power studies of real computer hardware – in specific, we will examine real embedded system hardware in a PDA platform.

Our objective is to provide a rigorous method for accurate and precise power measurements in real hardware. While we explore the problems and our proposed solution using a PDA reference design, these techniques apply equally well to all domains that require a power study.

In prior work by many groups, experimental analysis has been used to evaluate new methods for saving power in embedded systems. However, each group has used different methods, different equipment, and different equations resulting in uncertainty when trying to compare ideas. Our system for power analysis will account for differences in equipment and scope of study, providing error estimates that can be used to ensure that results are meaningful when substitutions are used.

In this work, we base our study on the Intel Sitsang-400 PDA Reference Design Platform version B1-1-3 (hereafter, *Sitsang*). This reference design is an advanced PDA model, with a wide variety of expansion options. The base unit consists of the Intel PXA255 XScale processor (model

ABC400), 64MB of Flash as four Intel E28F128J3A159 chips, and 64MB of SDRAM as two Samsung K4S561632D-TC75 chips. The PXA255 is also connected to a Toshiba LTM04C380K 640x480x18bpp LCD panel. The Toshiba LCD has a resistive touchscreen connected to a Burr-Brown ADS7846. The Sitsang battery is a Panasonic CGP345010G prismatic lithium ion rechargable pack rated for a nominal 3.7V at 1500mAh, with an operating output between 3.5V and 4.2V.

The Sitsang motherboard includes an SMSC LAN91C96 10Mbps Ethernet chip along with the twisted pair physical interface and magnetics module. Other features include three USB ports, a CF socket, a SD memory stick socket, UARTs, JTAG header, AC'97 codec, speaker and microphone. An expansion header that straddles the address and data buses provides the necessary features for custom daughterboard interfacing. While there are many features and possibilities, this work will concentrate on the core PXA255 processor, SDRAM, Flash, and Ethernet. Some discussion of the LCD panel will also be presented. Therefore, we omit the exact specifications of the other features for brevity.

The Sitsang is an ideal platform for power research due to the unique design of the motherboard with respect to power supply controls. The PXA255 address and data buses are intercepted by a Xilinx XCR3384XL CPLD. This CPLD implements per-block V_{DD} gating at the chip level. It is possible to individually enable or disable the power to: the CF slot, UART, USB, Bluetooth UART, SD socket, accelerometer, LCD, backlight, IrDA, AC'97 codec, and 10Mbps ethernet. Additional settings enable self-wakeup after changing the core voltage, as well as disabling *all* peripheral power. Such fine-grained power control is a result of individual power regulators with dedicated enable lines out of the CPLD. Actual supply voltage and current consumption can be measured either at the battery interface, or at the output of of any regulator to a logic area (such as the ethernet interface) with minor circuit modifications.

Prior to exploring the exact methods we use in our study, it is necessary to discuss issues with precise power measurements in general. While we necessarily relate our discussion to other publications [23, 61, 62, 16, 87, 88], this is not indicative of errors in other research efforts. Instead, our comments should be understood within the framework of prior research such that on a case-by-case basis, future researchers will use the most appropriate methods for their studies.

A.1 V_{DC} *is not a constant*

Many power studies on real hardware tend to treat the supply voltage under consideration as constant. While this simplifies the calculations and reduces the equipment overhead necessary for detailed power measurements, treating the voltage as constant may lead to significant error.

As an illustration of the error that may result, Figure 32 is a snapshot of power analysis on the Sitsang platform. The screen capture from our Tektronix TDS5104B (1GHz, 5GS/s) shows three traces. The bottom (orange) trace shows the real power consumption of a Sitsang unit sitting idle in our modified U-Boot bootloader with the backlight set at 75% intensity. The scale is in 1.0Ws/div steps[1], with 0Ws below the displayed region. The power calculation internal to the TDS5104B with its Advanced Math option comes directly from monitoring the battery supply. We use a Tektronix P6139A voltage probe to monitor the actual battery supply voltage, and a Tektronix TCP202 current probe to capture the real current use by the platform directly from the battery output. The internal power calculation is the product of the V_{batt} supply and the I_{batt} consumption.

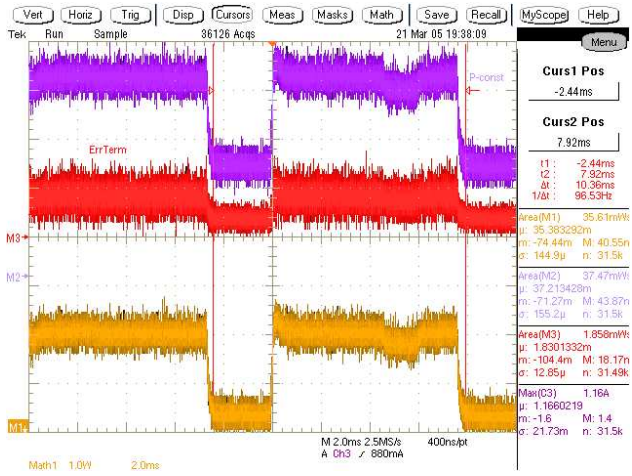


Figure 32: Error (middle red trace) introduced in an idle XScale PDA by treating V_{DC} from the battery as a constant.

Similarly, the top (purple) trace shows the power consumption if we instead treat V_{batt} as a constant value, set from a Tektronix PS503A variable DC power supply. For the Sitsang platform, with the unit idle (under load) in U-boot, the "constant" voltage of a fully charged battery is measured at

[1]The Tektronix calculates power in $Watt \cdot second$, shown as Ws, the normal unit of which is $Joules$. To keep the text of this discussion consistent with the figures, we also refer to Ws when discussing the figures.

approximately 4.0V. This top trace is the product of that unchanging voltage and the actual current, as measured for the bottom trace. The scale of the top trace is also 1.0Ws/div.

The difference or error between these traces is shown in the middle (red) trace, with a scale of 0.25Ws/div. A difference of zero is perfect agreement. A difference that is positive indicates that treating the V_{batt} as constant results in a higher power measurement than using the real V_{batt}. The converse is that a negative error indicates that the real power consumption exceeds the expected if the battery is considered at a constant voltage. The zero-axis for the middle trace is indicated by the "*M3*→" on the left edge of the displayed grid.

The implication of Figure 32 is that if discussions about power savings or expenditures are less than some minimum threshold, treating V_{DC} as a constant may be a substantial source of error. To calculate the error term introduced in the idle state of the Sitsang, we use the measurements provided from the power calculations and the maximum current drawn as shown in Figure 32. The area calculations are limited to the cursor range, such that at 96.53Hz an error of 1.329mWs is introduced *per event*. Thus the error can be calculated as shown in Equation 19.

$$(Error\ Area) \cdot (Frequency) = (1.329\ mWs)\left(96.53\ \frac{cycles}{s}\right) = 128.3\ mW \qquad (19)$$

For the idle state in U-boot with a 75% backlight intensity, any modification resulting in a new power value, K, should be expressed as $K \pm 128.3\ mW$. While we have illustrated the possible error with measurements from V_{batt}, the same holds true for *any* supply line on the system board, from V_{core} to V_{uart}. Moreover, it also holds true when a power supply other than a battery is used. For example, Figure 33 shows the error where a precision HP 3610A power supply is used to mimic the battery.

Measurement devices of less capability than the Tektronix 5104B, such as Fluke DMMs or inexpensive oscilloscopes, are incapable of showing the variance induced into V_{DC} as the current changes in such detail. To provide a mechanism for error estimation when treating V_{DC} as a constant, we use a test circuit to simulate V_{batt} and I_{batt} as shown in Figure 34. The circuit stimulus is a Tektronix FG5010 function generator running through a voltage-follower circuit on a National Semiconductor LF-353N op-amp to protect the generator. The voltage follower output is then run

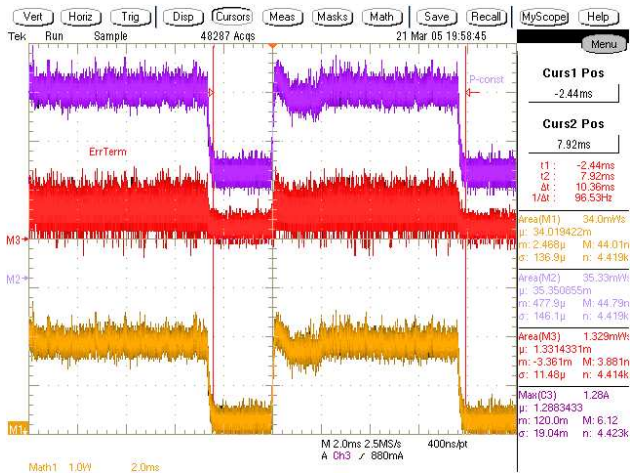


Figure 33: Error (middle red trace) introduced in an idle XScale PDA by treating V_{DC} from an HP precision power supply as a constant.

through two gain amplifiers, where the first stage gain is 10 and the second stage is variable as controlled by the 100kΩ potentiometer. For our study, we adjust the potentiometer until we can obtain the maximum current draw I_{load} without inducing clipping of the stimulus waveform after the gain stages. We use two gain stages to reduce amplifcation errors.

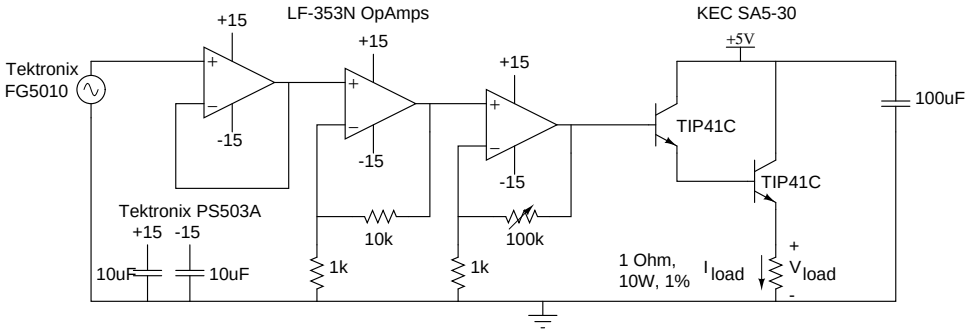


Figure 34: Test circuit to calibrate error terms in power measurements.

The output from the final op-amp is connected to the base of a Fairchild TIP41C transistor. This transistor acts as a buffer to prevent burning out of the final op-amp stage, and it in turn drives the base of a second and final TIP41C. The TIP41C collectors connect to a KEC Electronics SA5-30 power supply which is capable of providing 30A at 5V DC. The current sinking TIP41C emitter connects to a 1Ω 10W high precision resistor. The large capacity transistors and resistor permit high current load profiling without inducing significant element breakdown effects. The op-amps are all driven from the same Tektronix PS503A power supply, set to generate dual outputs of +15V

and -15V concurrently.

By varying the frequency and type of stimulus (square-, triangle-, or sine-wave) from the FG5010, oscillations of current load are induced. Modifying the driving amplification from the FG5010 changes the magnitude of the current load. Changing the frequency and amplitude of the wave stimulus also provides a guide to approximate the error based on what any research project is observing. Sweeping the current load from 0.5 to 2A, with a per-load frequency sweep from 100Hz to 100kHz, generates the resulting error terms shown in Table 9.

Due to temperature and ambient noise, these error terms carry their own approximate 10% error on the values shown, as observed over multiple trials. At frequencies above 25kHz, the sine and triangle waveforms become more similar as the power supply cannot match the load demand. Above the 50kHz frequency target, the square wave also degenerates to a more sinusoid shape. These error terms of Table 9 are not directly applicable to measured circuits. Rather, they suggest issues that careful researchers must check to ensure accurate results.

For example, the Linux kernel with pre-emption patches applied and the context switching time increased to 1kHz generates a complex square-wave pattern at 1kHz when the scheduler is activated. If two tasks are running, such as an MPEG decoder and a network streaming application, the square wave will oscillate by up to 0.5-1.0A on the Sitsang platform. Studies that attempt to show energy reduction by mechanisms that also treat the power supply of the battery constant are introducing between 180-565 mW of error.

These error estimation results are also particular to the precision HP power supply we use. Substitution for a different power supply will result in different characteristics, but the same general effect will be directly observable – that is, voltage lines are not constant.

This is slightly more problematic since real systems do not fluctuate in trivial patterns such as the ones we demonstrate. Real systems have *complex* fluctuations, as shown in Figure 32. While the overall shape is a stretch square wave, the representative regions of “high” and “low” are actually compound signals themselves.

A closer view of the complex behavior is shown in Figure 35. In this figure, the top blue line is the battery supply voltage at 500mV/div. The middle yellow line is the current consumption from the battery at 500mA/div. The bottom gold line is the power consumption at 1.0W/div. This waveform

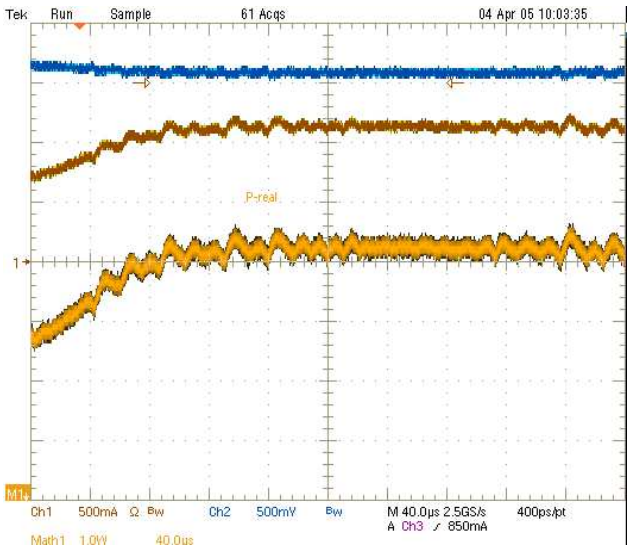


Figure 35: A close view of the complex component of real system power use.

corresponds to the first "high" time of the full waveform from Figure 32. The high frequency component shows power fluctuations of nearly 800mW with current fluctuations of approximately 225mA.

Since Table 9 represents simple functions, real system error estimates are of necessity built up from these values. A complex waveform such as Figure 32 can be approximated by a square wave error term for the measured frequency, coupled with the high-frequency error term that can be based on observations like Figure 35. Lacking the equipment for accurate and precise measurements directly, such error terms are useful approximations to evaluate the significance of experimental or simulation results.

A.2 Sample rate and repeatability constraints

The basic mechanics and versatility of oscilloscopes are well known and explained in published literature [36, 38, 37, 35, 64, 56], yet some aspects bear closer inspection in our discussion of power studies. We briefly explore the more salient points, and then characterize additional error terms introduced by failing to comply with the heuristics presented.

A.2.1 Bandwidth

The bandwidth of an oscilloscope corresponds to the attenuation of the input waveform. For a 100MHz bandwidth, an input sinusoid of 100 MHz will display at approximately 70% of the true waveform amplitude. In general, higher bandwidth for an oscilloscope provides a more accurate representation of the input stimulus. Tektronix recommends the *5 Times Rule* [36], where the bandwidth should be no less than five times the highest expected frequency component. This ensures a $\pm 2\%$ error term in measurements. A secondary consideration is that aggressive rise or fall times may require higher bandwidth. The rise time rule of thumb divides 0.35 by the rise time [35, 36]. With a rise time of 1ns, the oscilloscope requires 350MHz of bandwidth.

With respect to the Sitsang, we can easily induce major oscillations like the square-wave LCD backlight or task-switching among different resource bound jobs to reach 1kHz variations. The higher frequency oscillations, as shown in Figure 35, can be induced up to approximately 250kHz – but this is not a real maximum. Using the *5 Times Rule* suggests a minimum bandwidth of 1.25MHz on any oscilloscope is essential for power studies of such platforms. Power studies regarding modern workstations or laptops would, of course, have even higher bandwidth requirements. By comparison, considering the rise time we can observe in Figure 35, a measured rise time of 110μs for the square-wave portion translates to a bandwidth of 3.2kHz. However, the high frequency components exhibit a rise time of as little as 250ns, resulting in a bandwidth of 1.4MHz. This is comparable to the *5 Times Rule* result.

A.2.2 Sample Rate and Waveform Buffer

Many modern oscilloscopes of all price ranges use digital sampling from an analog-digital converter (ADC) device. The premise is that at a regular clock interval, the ADC value is copied into a waveform memory buffer. Once the memory buffer is full, the screen is redrawn to reflect the most recent data, and if necessary the data is exported to other devices or computers. There are two key hazards to accurate and precise measurements with digital oscilloscopes – analog units have equivalent problems which we do not discuss.

The primary difficulty with digital sampling is the sample rate clock. Between clock transisitons, the oscilloscope is effectively blind to any changes in the input signal. Regardless of how much

the signal changes, nothing will be observed until the next rising clock edge. By the well-known Nyquist-Shannon sampling rate, to reconstruct a signal the sampling rate must exceed two times the highest frequency expected. Since the highest frequency expected is at least 250kHz, the sampling rate must exceed 500kSa/s. Any lower value will result in aliasing artifacts. In reality, however, modern oscilloscopes use different methods to interpolate the data between two sample points. Depending on whether the interpolation is linear or a function [36] such as $\frac{sin(x)}{x}$, the sampling rate multiplier is heuristically in the range of 2.5-10. This translates to a minimum sampling rate of 625kSa/s to 2.5MSa/s.

These high sample rates result in the problem of managing the waveform buffer size. While some oscilloscopes can be configured with multiple megabytes of sample buffer per channel, many economical scopes can not. Instead, these may employ fixed buffers of a few hundred to a few thousand samples per channel. With most power studies, however, the time of interest is measured in seconds or minutes. With an 8-bit ADC resolution, a buffer of 32KB for 2.5MSa/s would last 0.013s. Clearly, higher-resolution ADCs which provide better signal sensitivity exacerbate this problem.

A second impact from the waveform buffer is that once the buffer has been collected, *all collection is disabled* while the screen is in update. Similarly, if the data is exported to other devices or computers, the collection is stopped until all updates finish. Only then, if the oscilloscope is set to continuous-run mode, will collection resume. Much like with the sample clock, in this period of time the oscilloscope is blind to any changes in the input signal.

A.2.3 Measuring Comparison

For a brief comparison, several measurement devices including oscilloscopes and advanced digital multi-meters (DMMs) commonly used in active research literature have their primary characteristics summarized in Table 10. Some vendors, such as Picotech, emphasize their repeatable-run sampling rates and tolerances, but the values presented here are single-shot. Single-shot values represent a fair comparison to the actual hardware support inside the oscilloscope, as repeated-run requires features described below.

All of the oscilloscopes, which cover a spectrum of pricing from $400 to $20,000, meet the

prior estimated needs of 2.5 MSa/s and a bandwidth of at least 1.25MHz. The DMMs shown are insufficient in resolution to have reliable results. However, not all researchers have the luxury of high-end oscilloscopes and current probes. This can be compensated for by using low-end oscilloscopes, such as the Bitscope 300N, coupled to a custom circuit for current sensing. We present an inexpensive circuit for this type of work later in this section.

A.2.4 Repeatibility

Many power studies are concerned with long-running events, particularly those that look at dynamic voltage scaling (DVS) or dynamic frequency scaling (DFS) or both (DVFS). With the need to collect data for multiple seconds or even minutes, there is no waveform sample buffer commercially available that can capture sufficient data. At a minimum 2.5MSa/s, capturing 60 seconds of information would require a 150 MSa buffer. For the lower-end oscilloscopes in Table 10, even capturing one second is beyond the local buffer capacity.

This problem can be handled in essentially one of two ways: (1) continuous looping of experiments coupled with low-sampling, or (2) high-sampling short periods in succession over continuous looping of experiments. In either case, the key is continuous looping of experiments, or repeatibility. In private conversation with several researchers, approximately half admit that their research is based on non-repetition of experiments for the reason, "it's too hard to make consistent." Sources to their predicament are the variability of OS schedulers, precise control over network parameters, and so forth. The point of this section is that failure to construct environments that allow for exact repeatibility will result in meaningless data.

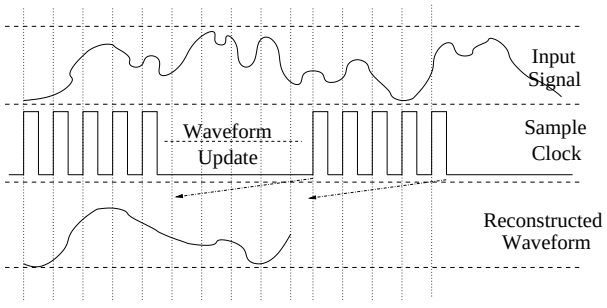


Figure 36: An illustration of the sampling error and need for repeatibility.

To understand the basic problem visually, Figure 36 illustrates the problem of sampling and

waveform updates in oscilloscopes. Without a sufficient sample rate, much of the input waveform shape will be lost during sampling. During waveform updates, no acquisition occurs, resulting in a loss of the entire input signal as shown by the arrow extensions. Both of these issues must be accounted for.

The principle of aggregate collection is that the sample window is set to be the duration of the experiment of interest, such as one minute. During this period, the data is collected as quickly as possible at different time offsets. By repeating the experiment many times, and using either a random or gradually increasing start offset to capture data, eventually sufficient detail is captured to reconstruct the proper input signal. Each reptition adds a small amount of measurements to the reconstructed waveform. Proper construction of such an apparatus will overcome any limitations on both sample rate and the waveform update blackout in any setup.

As a practical example, consider the Bitscope 300N in Table 10. To capture at 2.5MSa/s for 60 seconds requires 150MSa. With a limited buffer of only 32KSa/chan in the Bitscope, and by transferring data back to the PC at 115.2kbps, each "capture" takes 4.56 seconds to upload each waveform to the host PC. Assuming that each capture can be perfectly started such that no overlap with previous captures occurs, it requires a *minimum* of 4,688 repetitions of the experiment with *perfect synchronization* – at 4.56 seconds per repetition, this amounts to 5.93 hours of continuous collection. In the worst-case of a *random* collection start-point, it requires approximately $13,952,178$ repetitions[2] or approximately 2.02 years if the experiments could be run perfectly back-to-back. This random average does not assure perfect coverage, but it does assure at least 99.99% coverage. A conservative upper bound to collect N unique data points in 32k blocks is $2N\,ln(2N)$, based on the Coupon Collector's Problem [42], which for our example resolves to 5.86×10^9 iterations at 4.56 seconds per iteration – roughly 847 years. A mechanism to avoid this lengthy problem is presented in the next section.

[2]Based on a short C program to simulate this behavior, run for 10 iterations, with an average of the results. The *random()* linux function is used with the *srandom()* seed set from */dev/urandom*.

A.3 Equipment and Circuits

To alleviate the $2,000 cost of current probes like the TCP202, many researchers use sense resistors in the voltage supply line to obtain current measurements. However, any use of a sense resistor in a supply line that draws $\geq 500mA$ is likely to corrupt the circuit under study. Using the basic law $V = IR$, a sense resistor of 1Ω will drop the supply line by 1V if the current drawn is 1A. This 1V drop may cause a brown-out or outright failure of the test circuit. Alternately, knowing a system such as the Sitsang can draw up to 1.5A, and knowing that the maximum voltage drop sustainable is 0.2V to prevent brown-out, the ideal sense resistor value can be found as 0.13Ω. The drawback to using such a small value for R_{sense}, however, is that small fluctuations in current draw are not visible with high precision by most low-end oscilloscopes. For example, a fluctuation of 50mA for a 3.8V supply (190mW) will result in a voltage drop of only 6.5mV. This can be compensated for with a series of op-amps to differentially amplify the voltage across R_{sense}, as shown in Figure 37. For those situations that require higher current capacity, additional reduction in the R_{sense} value below 0.1Ω is risky for the typical low voltages involved in embedded digital systems. The effects of wire impedance and sensitivity may cause too much error in measurements.

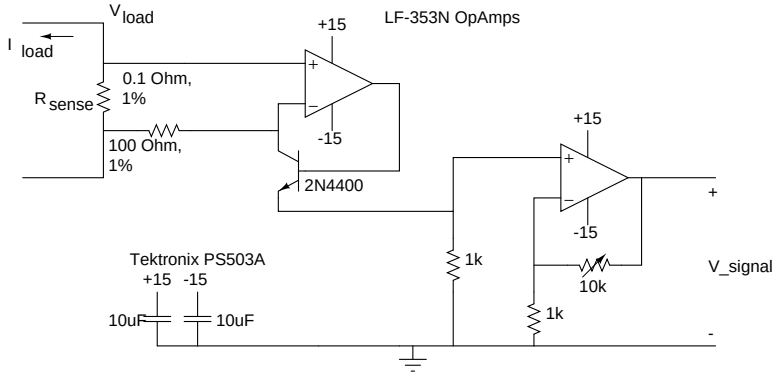


Figure 37: Measuring current with a sense resistor, R_{sense}.

Instead of using sense resistors and worrying about voltage drops, a better solution is to use an integrated circuit with a current sensor, such as the CUI, Inc., SCDxxPUR series. These devices generate an output voltage in linear proportion to the current load between two points. For example, the SCD10PUR generates a signal of 0 to 5V for current values between -10 to +10A. The minor drawback of these devices is their very low current capabilities on the linear output, requiring a

voltage-follower op-amp circuit. Moreover, for platforms like the Sitsang, there will never be a discernable negative current in the supply lines, so the output could be biased such that 0A corresponds to 0V. By adding a voltage-subtractor op-amp circuit based on potentiometers, an adjustable offset is easily added. In addition, the window of interest can similarly be gained with an amplifier op-amp circuit again serviced with potentiometers. The result of these three stages – follower, subtractor, and positive amplifier – present a linear signal with no perturbation of the supply line. Moreover, an entire circuit can be built for under $20, avoiding the high costs of current probes. Our design of such a circuit is shown in Figure 38. We over-bias the subtractor to obtain the full resolution of the oscilloscope by setting 0A to match -5V, and 1.5A to match +5V. This yields a dynamic range of 10V for a maximum of 1.5A fluctuation.

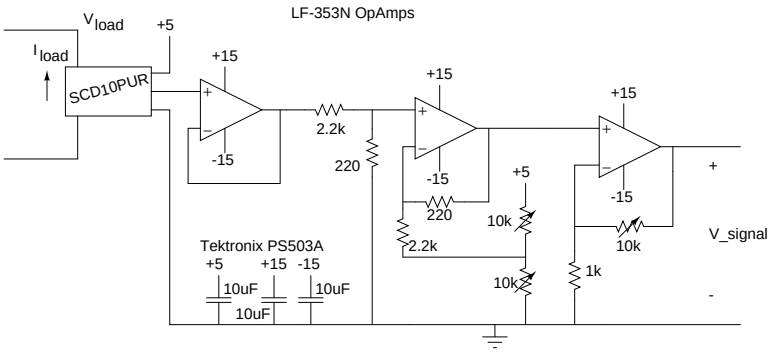


Figure 38: Measuring current with a current sensor like the SCD10PUR.

The problem of repeatibility, as discussed in Section A.2.4, suggests the obvious result that detailed sampling to capture precision power measurements is only effective for short events, typically of less than one second in duration. For long-running events, where instantaneous fluctuations are less interesting than aggregate power consumption, a better solution is to use an integrator circuit. The basic premise of an integrator is to construct a current mirror that generates a very small output current in proportion to the actual load current. This minor current is stored in a capacitor with a carefully selected time constant. At regular intervals, an ADC samples the capacitor charge and then resets the integrator circuit. Such a conceptual circuit [56] is illustrated in Figure 39. This type of design can be used to take periodic samples during long-running experiments, or just to take an end-of-experiment reading on the total current consumption. The paired FETs provide a leakage-free reset path to the integrator, a problem with single-FET reset designs. The selection of the τ

constant for the circuit is dependent on the expected sampling interval for the experiment.

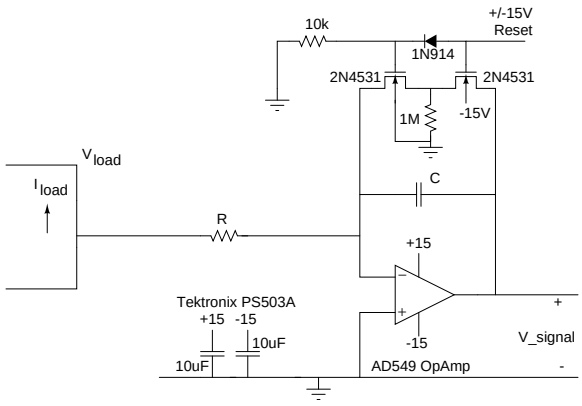


Figure 39: An example integrator circuit for long-running experiments.

A simpler design for integrator circuits can remove the op-amp and sense resistor from Figure 39. Devices like the MAXIM-IC MAX471 are current mirrors built around a precision internal R_{sense} on the order of 35 $m\Omega$. The primary drawback of these integrator-based designs is the loss of voltage fluctuation information, which requires careful error estimation based on results such as Table 9. Alternatively, adaptations on the integrator circuits here could be used to integrate voltage as well as current, thereby avoiding some of the error. However, any use of an integrator will require careful evaluation to determine the error of measurement.

The use of op-amp components in these circuits can lead to interesting problems. The op-amps introduce a slight delay, causing the observed current to be slightly out of phase or *skewed* with the observed voltage. Advanced digital oscilloscopes typically offer a mechanism for time-delay in channels to de-skew the signals, eliminating this source of error. For high-frequency oscillations, this can become a significant source of error. For our target circuit up to 100kHz, it's less than $\pm 1\%$ error as compared to the oscilloscope with a current probe. Additional error sources arise from too much gain in the op-amp circuit, or bandwidth/slew rate limitations of the op-amp in use. For this reason, it is important to carefully select each component to avoid unnecessary error sources.

While both the R_{sense} and SCD10PUR circuits are workable models for handling current measurement under the right conditions, our studies use a Tektronix TDS5104B and Tektronix TDS3034B oscilloscope, each with two P6139A voltage probes and two TCP202 current probes attached, to avoid any introduction of these types of error. This provides us with a deep waveform buffer as

well as high precision measurements with excess bandwidth available. The trigger output of the TDS5104B is connected to the external trigger input on the TDS3034B to provide synchronized data results. The timing of the current readings are de-skewed to match the voltage readings based on our square wave test circuit in section A.1.

We altered a Sitsang platform by removing the ferrite bead and 0Ω resistor components at J20, R26, R30, and R43. These were replaced with minimal wire segments to connect TCP202 current probes to, as well as the P6139A voltage probes. This provided us with the voltage and current measurements for the battery and three regulated supplies: 4.2V, 3.3V, and the PXA255 processor core. To understand how each of these supplies is used in the Sitsang design, we next present a high level view of the physical circuit layout on the Sitsang.

Sinusoid Waveform		Frequency (Hz)						
		100	1k	5k	10k	25k	50k	100k
I_{load} Peak (A)	0.5	$\pm 75mW$	$\pm 80mW$	$\pm 85mW$	$\pm 90mW$	$\pm 90mW$	$\pm 90mW$	$\pm 90mW$
	1.0	$\pm 315mW$	$\pm 345mW$	$\pm 360mW$	$\pm 425mW$	$\pm 440mW$	$\pm 565mW$	$\pm 570mW$
	1.5	$\pm 650mW$	$\pm 810mW$	$\pm 840mW$	$\pm 950mW$	$\pm 960mW$	$\pm 1.35W$	$\pm 1.35W$
	2.0	$\pm 1.15W$	$\pm 1.35W$	$\pm 1.45W$	$\pm 1.65W$	$\pm 1.95W$	$\pm 2.80W$	$\pm 3.60W$

Triangle Waveform		Frequency (Hz)						
		100	1k	5k	10k	25k	50k	100k
I_{load} Peak (A)	0.5	$\pm 50mW$	$\pm 55mW$	$\pm 60mW$	$\pm 70mW$	$\pm 70mW$	$\pm 85mW$	$\pm 85mW$
	1.0	$\pm 200mW$	$\pm 255mW$	$\pm 260mW$	$\pm 325mW$	$\pm 455mW$	$\pm 540mW$	$\pm 560mW$
	1.5	$\pm 445mW$	$\pm 530mW$	$\pm 585mW$	$\pm 900mW$	$\pm 920mW$	$\pm 1.25W$	$\pm 1.20W$
	2.0	$\pm 755mW$	$\pm 900mW$	$\pm 1.15W$	$\pm 1.30W$	$\pm 1.85W$	$\pm 2.95W$	$\pm 3.45W$

Square Waveform		Frequency (Hz)						
		100	1k	5k	10k	25k	50k	100k
I_{load} Peak (A)	0.5	$\pm 165mW$	$\pm 180mW$	$\pm 185mW$	$\pm 195mW$	$\pm 200mW$	$\pm 250mW$	$\pm 330mW$
	1.0	$\pm 550mW$	$\pm 565mW$	$\pm 635mW$	$\pm 680mW$	$\pm 695mW$	$\pm 900mW$	$\pm 1.10W$
	1.5	$\pm 1.24W$	$\pm 1.40W$	$\pm 1.45W$	$\pm 1.75W$	$\pm 1.75W$	$\pm 2.15W$	$\pm 2.50W$
	2.0	$\pm 2.05W$	$\pm 2.10W$	$\pm 2.15W$	$\pm 2.30W$	$\pm 2.35W$	$\pm 3.40W$	$\pm 3.65W$

Table 9: The error terms introduced in power measurements for various stimulus waveform types, frequencies, and amplitudes through the test circuit if V_{DC} is considered constant. V_{DC} is 0-15V, 2-3A precision HP E3610A power supply.

Vendor	Model	Type	Bandwidth	Max. Sample Rate	Min. Sample Rate	Max. Waveform Buffer
Agilent	34401A	DMM	300 kHz	1 KSa/s	n/a	0.5 kSa
Fluke	8508A	DMM	100 kHz/1MHz	100 Sa/s	n/a	n/a
Bitscope	300N	O's	100 MHz	40 MSa/s	4 kSa/s	32 kSa/chan
Picotech	ADS-212/100	O's	50 MHz	100 MSa/s	n/a	5.4 kSa/chan
Tektronix	TDS3014B	O's	100 MHz	1.25 GSa/s	n/a	10 KSa/chan
Tektronix	TDS5104B	O's	1 GHz	5 GSa/s	n/a	2 MSa/chan

Table 10: Different test and measurement manufacturers advertise products with different criteria. By using the single-shot baseline with oscilloscope models, each is compared by the same metric. The Tektronix 5104B allows user- specified waveform buffer lengths, providing variable durations.

APPENDIX B

MIBENCH RESULTS

The following sections contain the results for each of the 24 MiBench benchmark applications. Each benchmark application is briefly described, and the key characteristics are presented in table format. The miss rate over dynamic execution time, as measured by dynamically executed instructions, is presented in graph format. Each benchmark is presented twice, once for instruction caching behavior and once for data caching behavior.

B.1 Instruction Caching Results

B.1.1 bf

Table 11: MiBench *bf* performance results.

Unique PCs	2,588
Executed Blocks	104,141,294
Bytes Transferred	18,036
Evictions	0
Subset Collisions	35

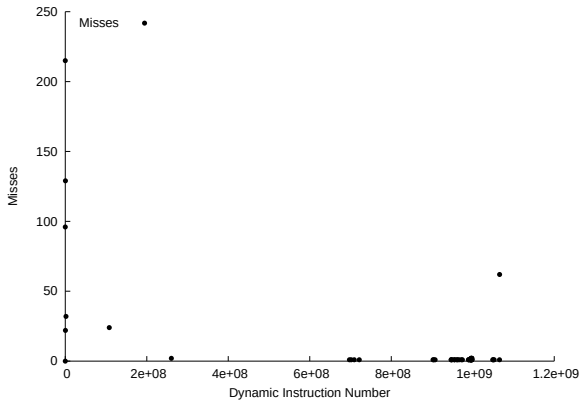


Figure 40: The miss rate over the full dynamic execution of the MiBench benchmark *bf*.

The *bf*. or blowfish, benchmark is the Blowfish symmetric block cipher with a variable length key. With a key size that can be set from 32 to 448 bits, it is commonly used as a technology that can be exported from the United States. The input data is a large ASCII file.

The required footprint for actual code executed is at most 18KB, as this number is skewed slightly higher due to the presence of subset collisions discussed previously. In reality, the hot-path code will be less than this value. However, given the 32KB on-die SRAM storage reserved for the instructions in our simulation, this demonstrates that the blowfish algorithm is a perfect application for the SoftCache framework with longs periods of stable execution briefly interrupted by a few cache misses.

B.1.2 bitcnts

Table 12: MiBench *bitcnts* performance results.

Unique PCs	3,818
Executed Blocks	171,578,140
Bytes Transferred	17,528
Evictions	0
Subset Collisions	66

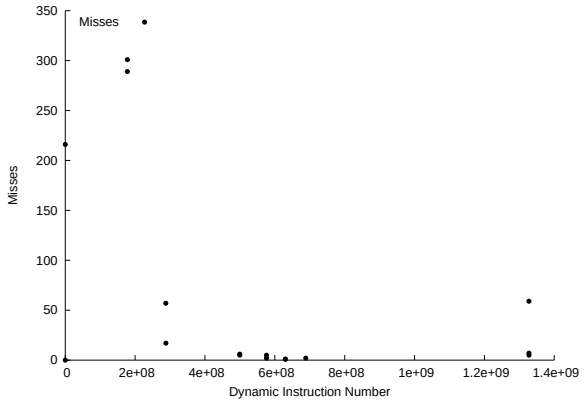


Figure 41: The miss rate over the full dynamic execution of the MiBench benchmark *bitcnts*.

The *bitcnts*, or bitcount, benchmark tests bit manipulation of integers exhaustively. By employing various methods, different patterns and evaluations are obtained. The input is a set of arrays with equal counts of 1's and 0's in the binary strings.

The amount of storage transfer between the server and client is only 17KB, easily fitting in the 32KB maximum for instruction space. This number is slightly higher than what is essential due to the subset collisions causing additional transfers. This is not the hot-code size, merely the actual transfer size.

B.1.3 cjpeg

Table 13: MiBench *cjpeg* performance results.

Unique PCs	8,399
Executed Blocks	14,095,128
Bytes Transferred	40,944
Evictions	366
Subset Collisions	148

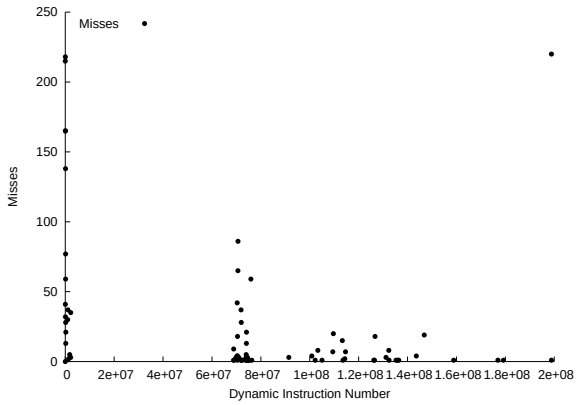


Figure 42: The miss rate over the full dynamic execution of the MiBench benchmark *cjpeg*.

The *cjpeg* benchmark implements the standard, lossy JPEG compression algorithm. This is a typical algorithm based on discrete cosine transformations (DCTs) that computationally is similar to most image format routines. The input is a large color image.

This application fails to fit well in the limited 32KB on-die instruction space. A slightly larger execution space would reduce the conflicts that start to appear in the middle of the execution. There is a minor oscillation or behavior paging from the instruction range 100M - 160M. With a few additional KB of on-die storage, or perhaps a more efficient code representation, this application would fit well in the SoftCache framework.

B.1.4 crc

Table 14: MiBench *crc* performance results.

Unique PCs	2,811
Executed Blocks	292,927,673
Bytes Transferred	12,536
Evictions	0
Subset Collisions	41

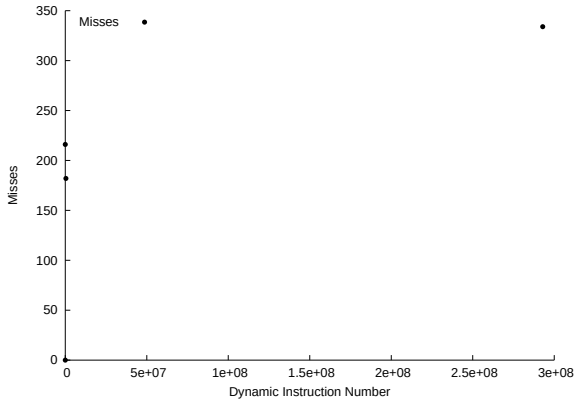


Figure 43: The miss rate over the full dynamic execution of the MiBench benchmark *crc*.

The *crc* benchmark is a 32-bit cyclic redundancy check (CRC) implementation for a test file. In general, CRC checks are used widely in communications system to verify files transferred such as in digital television signals. The input data is the sound test file from the ADPCM benchmark.

This benchmark is an excellent SoftCache application, generating the bulk of all misses during the initial execution of a few thousand instructions. After the initial misses, no misses occur until the end of the application at which point results are reported. The CRC algorithm fits in less than 12.5KB of space out of the 32KB reserved on-die.

B.1.5 dijkstra

Table 15: MiBench *dijkstra* performance results.

Unique PCs	3,497
Executed Blocks	45,957,936
Bytes Transferred	15,724
Evictions	0
Subset Collisions	57

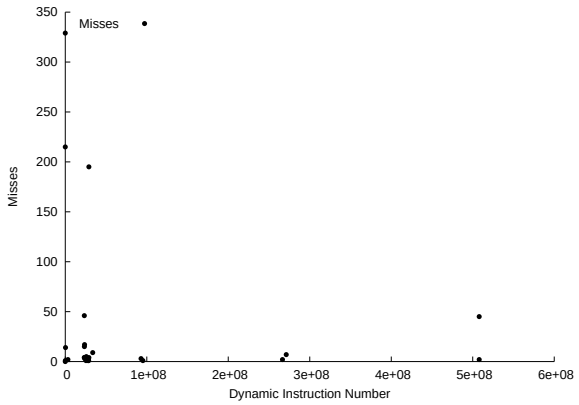


Figure 44: The miss rate over the full dynamic execution of the MiBench benchmark *dijkstra*.

The *dijkstra* benchmark constructs an adjacency matrix representation of a large graph. It then uses Djikstra's algorithm to compute the shortest path between every pair of nodes. This algorithm is $O\left(n^2\right)$ in time.

This benchmark also easily fits in the allocated 32KB space, using only 16KB of it without compression to just the steady-state code.

B.1.6 fft

Table 16: MiBench *fft* performance results.

Unique PCs	3,864
Executed Blocks	51,389,101
Bytes Transferred	18,804
Evictions	0
Subset Collisions	75

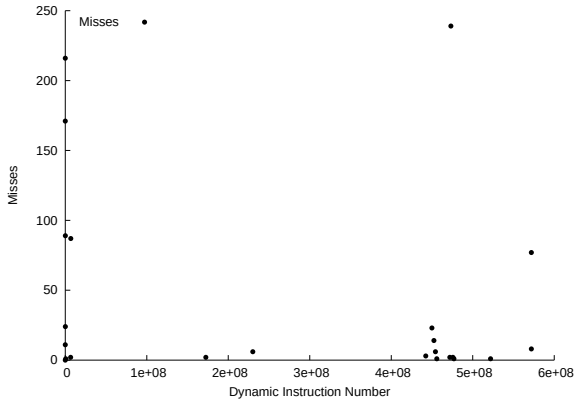


Figure 45: The miss rate over the full dynamic execution of the MiBench benchmark *fft*.

The *fft* benchmark is an implementation of the fast Fourier transform (FFT) as applied to a large array of data. This kernel is commonly used to isolate the frequencies in input signals to facilitate operations such as audio compression or noise cancellation. The array input is a series of pseudorandom amplitude and frequency values.

The benchmark easily fits in the 32KB on-die storage space, incurring a few misses as slight changes in program execution occur. Preloading of the known hot path would easily overcome all of these misses. The relatively long time between miss regions indicates that the FFT algorithm would work well in the SoftCache system.

B.1.7 gs

Table 17: MiBench *gs* performance results.

Unique PCs	8,522
Executed Blocks	81,790
Bytes Transferred	37,400
Evictions	218
Subset Collisions	64

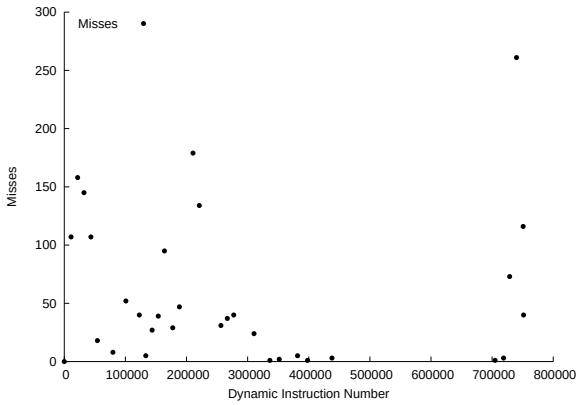


Figure 46: The miss rate over the full dynamic execution of the MiBench benchmark *gs*.

The *gs* benchmark is the open source PostScript intepreter known as GhostScript. This interpreter lacks a graphical interface, and is strictly the language interpreter. PostScript is a defacto standard for printers and some display devices such as Apple computers.

This benchmark is similar to the cjpeg benchmark in that a slightly larger on-die storage is necessary to avoid excessive swapping. The first half of the program is continually missing due to cold start *and* capacity problems. Eventually the program reaches a steady state interpreting the PostScript input file, but the delay penalties incurred from the initial swapping are likely to be too great for the SoftCache to compensate for.

B.1.8 ispell

Table 18: MiBench *ispell* performance results.

Unique PCs	3,618
Executed Blocks	110,879
Bytes Transferred	16,080
Evictions	0
Subset Collisions	60

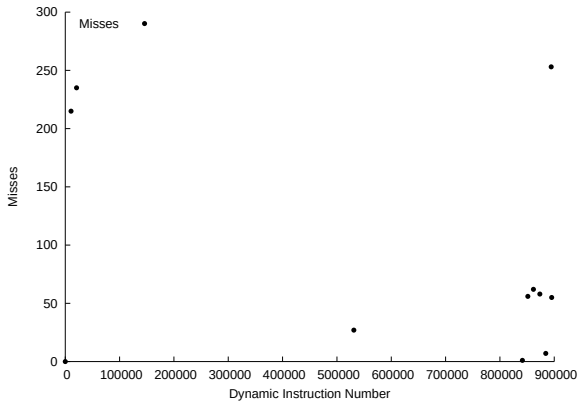


Figure 47: The miss rate over the full dynamic execution of the MiBench benchmark *ispell*.

The *ispell* benchmark is a fast spell checker similar to the UN*X "spell" but designed to run much faster. This application supports context-based checking, alternate spelling suggestions, and multiple languages. The input is a large document from the Internet as a web page.

This application easily fits in the on-die storage for instructions, and exhibits few periods of misses in the SoftCache framework. The long runtime between misses makes this a good application for the SoftCache.

B.1.9 lame

Table 19: MiBench *lame* performance results.

Unique PCs	21,332
Executed Blocks	50,883,969
Bytes Transferred	55,520,712
Evictions	1,651,653
Subset Collisions	85,301

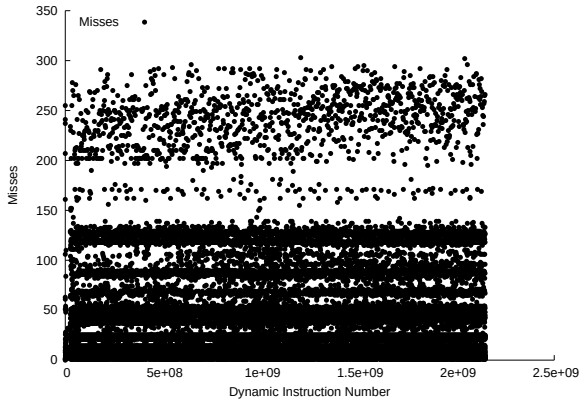


Figure 48: The miss rate over the full dynamic execution of the MiBench benchmark *lame*.

The *lame* benchmark is the open source MP3 encoder that supports both constant and variable bitrate encoding of audio files to MP3 format. This is a lossy compression algorithm similar to the JPEG algorithm. The input is a large wave file.

This benchmark is far too large to run in the SoftCache framework under any conditions. With over 1.5 million evictions and 55MB of code transferred, this is a perfect representation of the wrong class of application to run on a SoftCache framework.

B.1.10 lout

Table 20: MiBench *lout* performance results.

Unique PCs	16,279
Executed Blocks	139,206
Bytes Transferred	82,816
Evictions	2,276
Subset Collisions	188

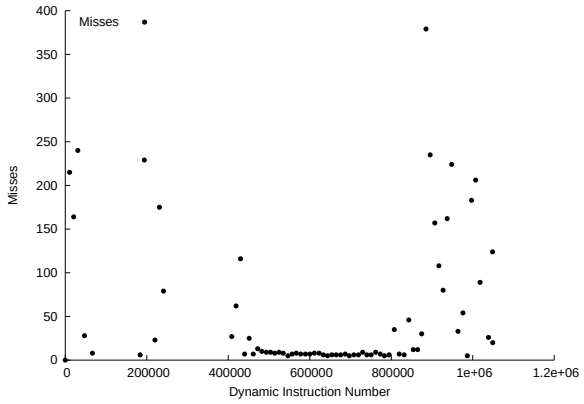


Figure 49: The miss rate over the full dynamic execution of the MiBench benchmark *lout*.

The *lout* benchmark is actually the MiBench *typeset* benchmark which is a typesetting tool for HTML. This benchmark captures the processing required to render an HTML page without actually rendering anything. This is similar to a simplistic web browser engine, such as Gecko from the Mozilla Foundation. The input is a large web page.

This benchmark is another example of a program that requires more than the 32KB on-die instruction area. With intermittent events causing large misses, and then a long-running period of misses, this application requires more storage than is available.

B.1.11 madplay

Table 21: MiBench *madplay* performance results.

Unique PCs	10,167
Executed Blocks	22,572,523
Bytes Transferred	48,644
Evictions	910
Subset Collisions	135

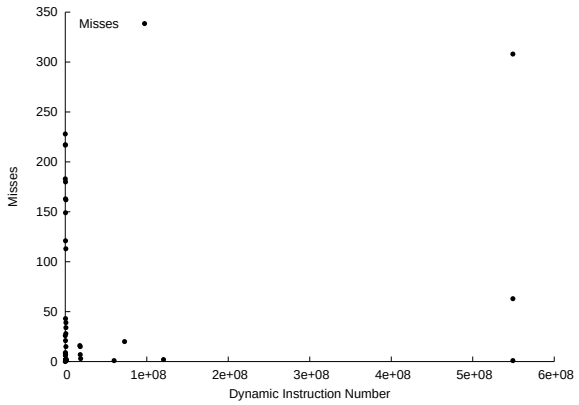


Figure 50: The miss rate over the full dynamic execution of the MiBench benchmark *madplay*.

The *madplay* benchmark is a high-quality MP3 audio decoder. This decoder supports Layer I, Layer II, and Layer III MPEG audio decoding. The input file is a large MP3 test file.

Unlike the MP3 encoder *lame*, the decoder runs quite well in the SoftCache once the initial hot-code path is found. From the range of 100M - 550M instructions, almost no SoftCache misses occur. This is excellent behavior for a SoftCache application.

B.1.12 math

Table 22: MiBench *math* performance results.

Unique PCs	5,221
Executed Blocks	488,846,352
Bytes Transferred	26,952
Evictions	0
Subset Collisions	130

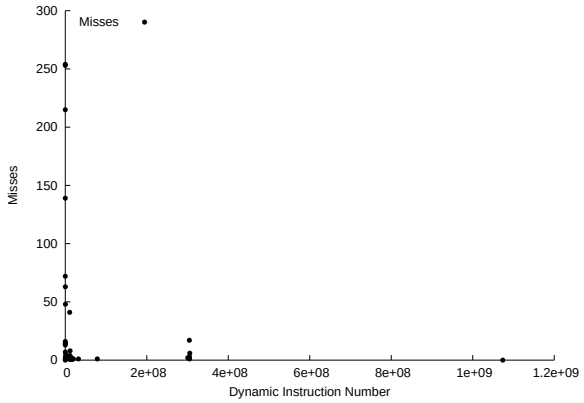


Figure 51: The miss rate over the full dynamic execution of the MiBench benchmark *math*.

The *math* benchmark, called *basicmath* in MiBench, consists of simple calculations that generally lack dedicated hardware support in embedded processors. These calculations are tasks like square root, angle unit conversions, cubic function solutions, etc. The input data is a fixed set of constants.

As can be seen from Table 22, the actual amount of storage required for the program is only 27KB out of the possible 32KB maximum. This is not the hot-code size, merely the amount of actual code transfer. The subset collisions skews this number to be higher than it normally would be. This benchmark would easily run in the SoftCache framework with no penalty once steady-state is reached. All misses are cold start misses.

B.1.13 patricia

Table 23: MiBench *patricia* performance results.

Unique PCs	5,635
Executed Blocks	142,654,643
Bytes Transferred	26,304
Evictions	0
Subset Collisions	114

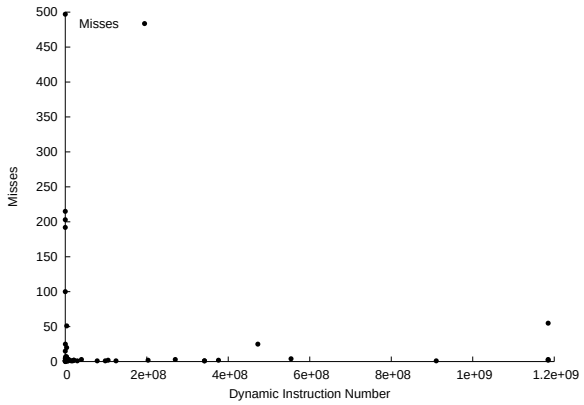


Figure 52: The miss rate over the full dynamic execution of the MiBench benchmark *patricia*.

The *patricia* benchmark is an evaluation of the Patricia trie data structure for sparse leaf nodes in trees. This type of construct is common in network routing tables. The input set is a list of IP traffic captured from a network over a period of time, with the IP numbers disguised.

This benchmark also easily fits in the 32KB of allocated space. In terms of application code, almost no misses occur once the initial program is loaded to the client, with a minor increase when a particularly complex region of the input data is parsed causing several subset collisions.

B.1.14 pgp

Table 24: MiBench *pgp* performance results.

Unique PCs	3,202
Executed Blocks	66,374
Bytes Transferred	14,732
Evictions	0
Subset Collisions	60

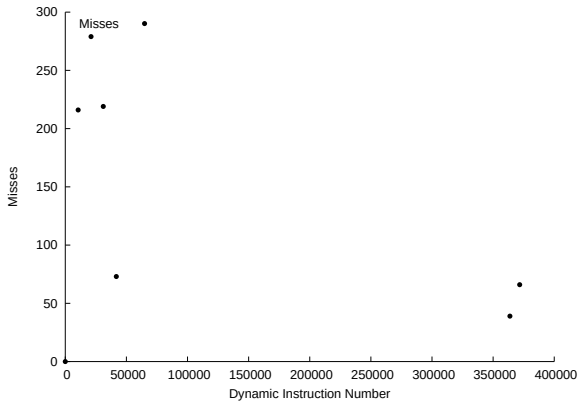


Figure 53: The miss rate over the full dynamic execution of the MiBench benchmark *pgp*.

The *pgp* benchmark is an implementation of the pretty good privacy (PGP) public key encryption system. This uses RSA encryption with digital signatures, a method developed by Phil Zimmerman. The input is a small text file, as the primary purpose of PGP is to securely exchange keys for a normal block cipher algorithm which is much faster than the PGP algorithm.

Regardless, the benchmark fits well in the SoftCache on-die SRAM for instructions and is well behaved with respect to misses. Preloading the initial misses would further improve performance.

B.1.15 qsort

Table 25: MiBench *qsort* performance results.

Unique PCs	3,490
Executed Blocks	167,108,825
Bytes Transferred	16,008
Evictions	0
Subset Collisions	57

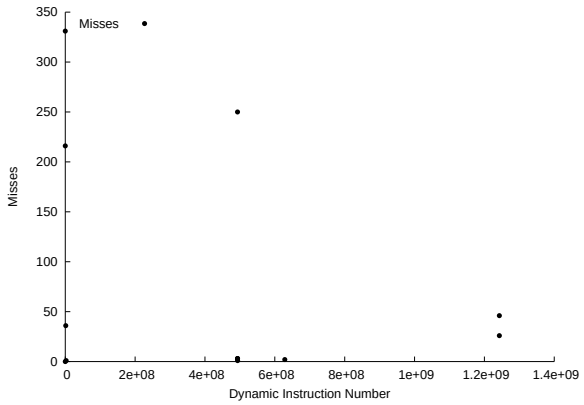


Figure 54: The miss rate over the full dynamic execution of the MiBench benchmark *qsort*.

The *qsort* test sorts a large array of strings with the quick sort algorithm. The input data set is a grouping of three-tuples, meant to represent points of data.

The entire code footprint fits into 16KB of the 32KB instruction region of on-die SRAM. Preloading of the misses that occur in the middle of the execution would further boost the application behavior with respect to the SoftCache system.

B.1.16 rawcaudio

Table 26: MiBench *rawcaudio* performance results.

Unique PCs	1,820
Executed Blocks	2,135
Bytes Transferred	8,128
Evictions	0
Subset Collisions	28

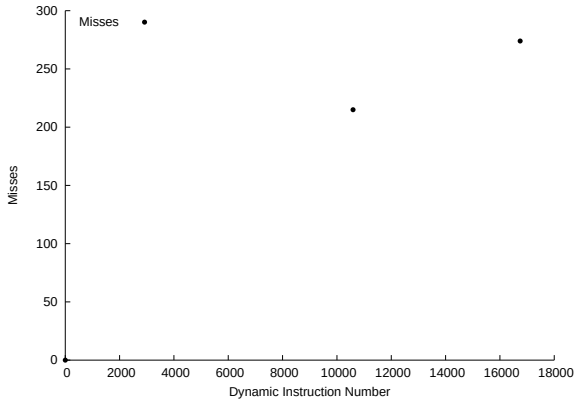


Figure 55: The miss rate over the full dynamic execution of the MiBench benchmark *rawcaudio*.

The *rawcaudio* benchmark is actually an implementation of the adaptive differential pule code modulation (ADPCM) encoder. The input is a series of 16-bit linear samples which are converted to 4-bit samples. The actual input file is a large speech sample.

This application only requires 8KB of storage in the on-die SRAM. Active preloading of the hot path would eliminate all misses entirely.

B.1.17 rijndael

Table 27: MiBench *rijndael* performance results.

Unique PCs	2,279
Executed Blocks	2,926
Bytes Transferred	10,064
Evictions	0
Subset Collisions	33

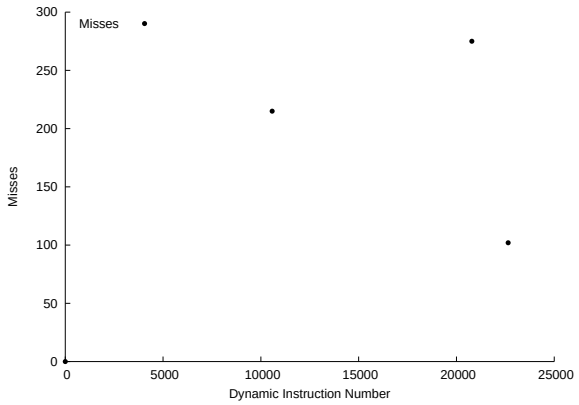


Figure 56: The miss rate over the full dynamic execution of the MiBench benchmark *rijndael*.

The *rijndael* benchmark is an implementation of Rijndael's secure encryption system. This is the algorithm selected by the National Institute of Standards and Technologies (NIST) for the Advanced Encryption Standard (AES). This block cipher algorithm is run on the same large ASCII input file as *sha*.

This benchmark fits easily into the on-die 32KB SRAM region for instructions. Moreover, it is a perfect model of the SoftCache mechanism where the application runs for a long period of time before encountering a minor phase adjustment and continuing.

B.1.18 say

Table 28: MiBench *say* performance results.

Unique PCs	6,621
Executed Blocks	5,860,106
Bytes Transferred	31,584
Evictions	0
Subset Collisions	115

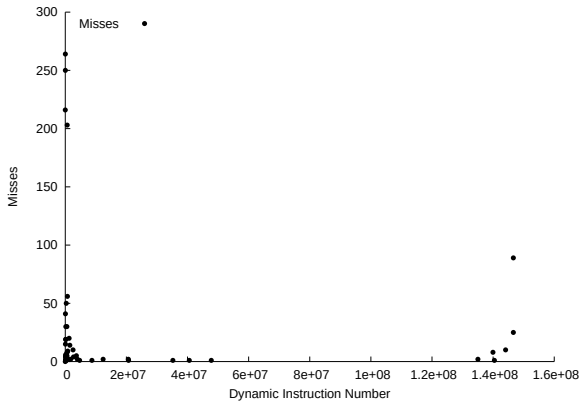


Figure 57: The miss rate over the full dynamic execution of the MiBench benchmark *say*.

The *say* benchmark is actually a speech-based benchmark built on the sphinx speech decoder. The large input is a long sequence of speech, which is handled one sequence at a time.

This application fits well within the 32KB on-die storage space. While it transfers nearly 32KB through the client-server interface, this number is inflated due to the subset collisions which overwrite some of the same code sequences.

B.1.19 search

Table 29: MiBench *search* performance results.

Unique PCs	2,293
Executed Blocks	1,067,366
Bytes Transferred	10,956
Evictions	0
Subset Collisions	42

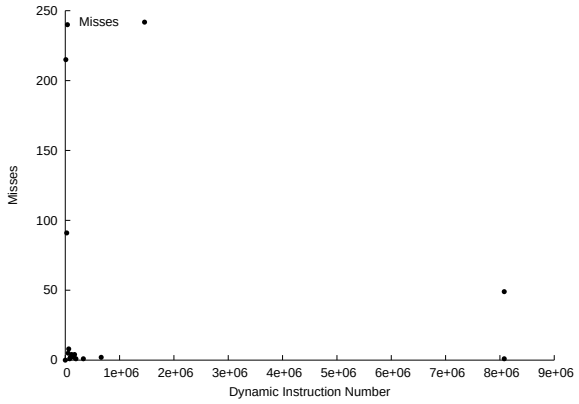


Figure 58: The miss rate over the full dynamic execution of the MiBench benchmark *search*.

The *search* benchmark is the stringsearch benchmark. This searches for specific words or phrases in a case insensitive manner. The input is a large ASCII file.

This benchmark fits well within the SoftCache on-die memory space, and exhibits excellent behavior with respect to misses. The very long period of program stability between misses is the ideal characteristic of any SoftCache application.

B.1.20 sha

Table 30: MiBench *sha* performance results.

Unique PCs	2,916
Executed Blocks	9,811,056
Bytes Transferred	13,136
Evictions	0
Subset Collisions	34

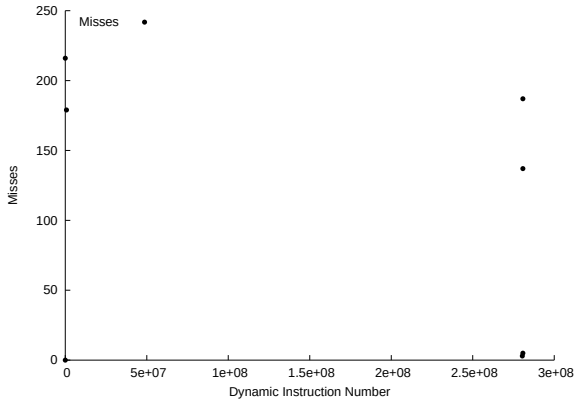


Figure 59: The miss rate over the full dynamic execution of the MiBench benchmark *sha*.

The *sha* benchmark is the secure hash algorithm (SHA) that generates 160-bit message digests for a given input. Aside from digital signatures, the SHA algorithm is also used to exchange cryptographic keys. The input data is a large ASCII file.

This benchmark is an excellent application of the SoftCache. Once the initial setup of the application is made, no misses occur until the very end when the result is being reported. This benchmark also fits easily within the 32KB limitations of the on-die SRAM.

B.1.21 susan

Table 31: MiBench *susan* performance results.

Unique PCs	3,873
Executed Blocks	3,904,530
Bytes Transferred	17,556
Evictions	0
Subset Collisions	54

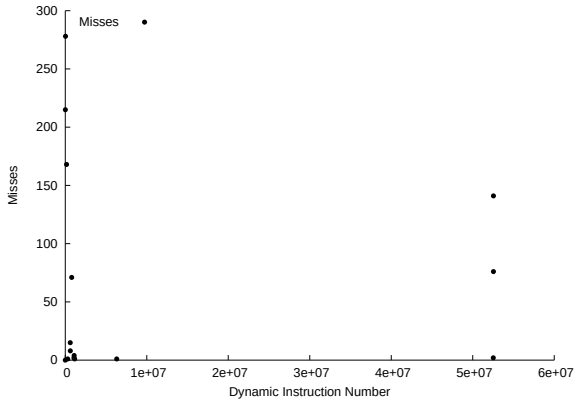


Figure 60: The miss rate over the full dynamic execution of the MiBench benchmark *susan*.

The *susan* benchmark is designed to recognize corners and edges in images – specifically, in Magnetic Resonance Images (MRIs) of the brain. This benchmark can perform rudimentary image smoothing, brightness and threshold adjustments, etc. The large input data is a complex picture.

This is another benchmark that easily fits in the allocated 32KB of space, using a maximum of 17.5KB. This is also skewed by the subset collisions. The large initial miss rate is due to reaching steady state in the application. By using profile information, this hump could be removed by pre-loading the entire first range of misses as the code is not particular input data sensitive.

B.1.22 tiff2bw

Table 32: MiBench *tiff2bw* performance results.

Unique PCs	5,568
Executed Blocks	11,386,109
Bytes Transferred	24,404
Evictions	0
Subset Collisions	65

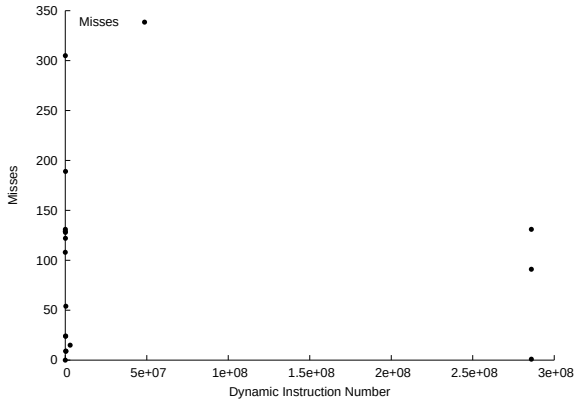


Figure 61: The miss rate over the full dynamic execution of the MiBench benchmark *tiff2bw*.

The *tiff2bw* benchmark converts a TIFF image from color to black-and-white. This type of downsampling is common in remote image processing applications to reduce bandwith and simplify feature recognition. When necessary, the raw or color version can be offloaded to upstream devices for a more complex algorithm such as selective attention.

This benchmark is another well behaved application for the SoftCache framework. This program easily fits in the 32KB on-die space allocated for programs, and has no misses beyond the initial starting of the program and the conclusion when the result is written back to a file.

B.1.23 tiffdither

Table 33: MiBench *tiffdither* performance results.

Unique PCs	7,564
Executed Blocks	200,054,165
Bytes Transferred	35,736
Evictions	0
Subset Collisions	211

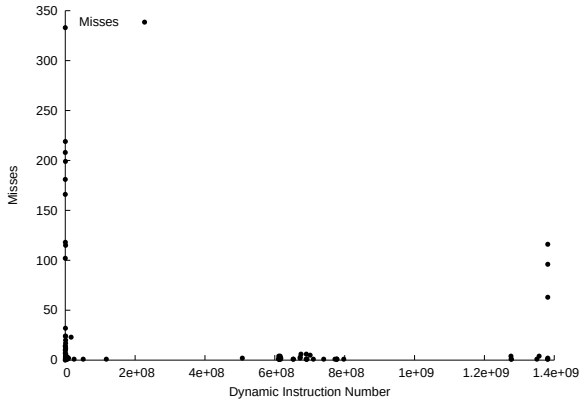


Figure 62: The miss rate over the full dynamic execution of the MiBench benchmark *tiffdither*.

The *tiffdither* benchmark is similar to the *tiff2bw* as a TIFF downsampling algorithm. The resolution and size are reduced by application of a dithering algorithm, resulting in a loss of clarity. This is another useful algorithm for more complex systems such as selective attention programs.

This benchmark needs a very small increase in the on-die storage for instructions. It encounters a minor period of thrashing in the middle of the application due to insufficient space within the 32KB limit. With a slightly larger space reserved for instructions, this benchmark would also be well behaved for the SoftCache framework.

B.1.24 toast

Table 34: MiBench *toast* performance results.

Unique PCs	3,603
Executed Blocks	4,044
Bytes Transferred	15,268
Evictions	0
Subset Collisions	20

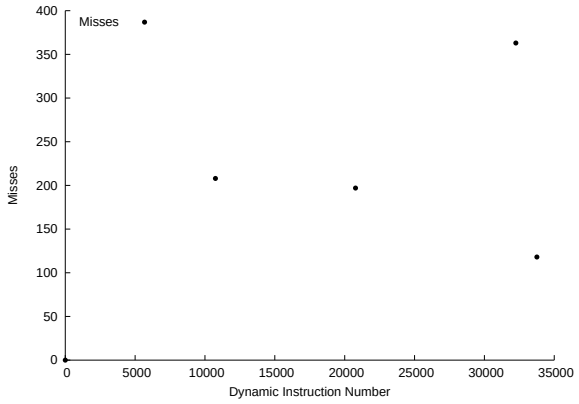


Figure 63: The miss rate over the full dynamic execution of the MiBench benchmark *toast*.

The *toast* benchmark is actually the global standard for mobile (GSM) communications encoder engine. The GSM standard is the standard for cellular communications in Europe and other countries. The GSM algorithm is based on a combination of time- and frequency-division multiple access (TDMA/FDMA) methods to encode data streams. The input is a large speech sample.

This benchmark fits well within the SoftCache on-die storage space. To avoid the misses encountered during the middle of execution, preloading the hot path would enhance the program behavior. This is another good application to run in the SoftCache framework.

B.2 Data Caching Results

B.2.1 bf

Table 35: MiBench *bf* performance results.

Unique PCs	897
MTI PCs	421
UTI PCs	476
MTI References	334,302,552
UTI References	54,224,742
Misses	378
Evictions	0

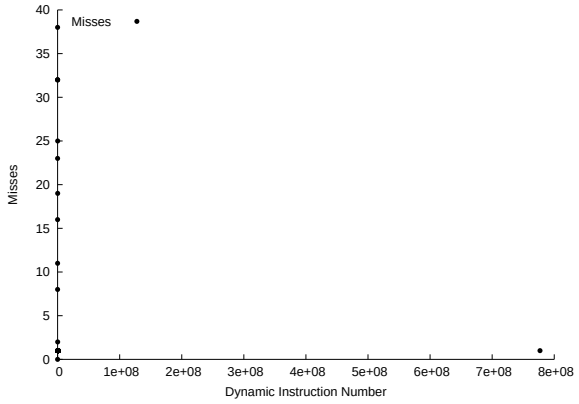


Figure 64: The miss rate over the full dynamic execution of the MiBench benchmark *bf*.

The *bf*. or blowfish, benchmark is the Blowfish symmetric block cipher with a variable length key. With a key size that can be set from 32 to 448 bits, it is commonly used as a technology that can be exported from the United States. The input data is a large ASCII file.

This application easily fits in the 32KB storage space for data on-die. Moreover, all the misses are cold-start misses and preloading the known execution path would eliminate all later misses.

B.2.2 bitcnts

Table 36: MiBench *bitcnts* performance results.

Unique PCs	1,365
Unique Addresses	1,590
MTI PCs	366
UTI PCs	999
MTI References	164,265,629
UTI References	19,133,795
Misses	187
Evictions	0

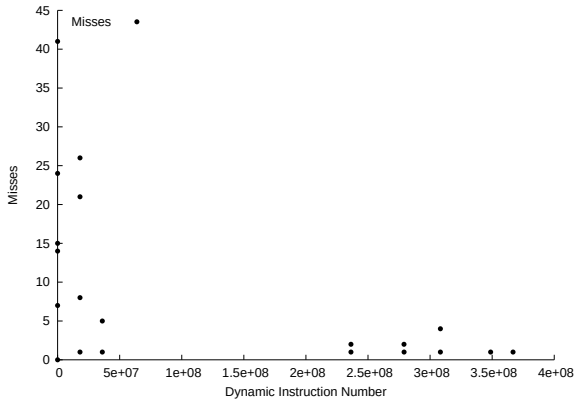


Figure 65: The miss rate over the full dynamic execution of the MiBench benchmark *bitcnts*.

The *bitcnts*, or bitcount, benchmark tests bit manipulation of integers exhaustively. By employing various methods, different patterns and evaluations are obtained. The input is a set of arrays with equal counts of 1's and 0's in the binary strings.

This application easily fits in the 32KB storage space for data on-die. Moreover, all the misses are cold-start misses and preloading the known execution path would eliminate all later misses. This program consumes a very small fraction of the available memory space.

B.2.3 cjpeg

Table 37: MiBench *cjpeg* performance results.

Unique PCs	3,078
Unique Addresses	815,720
MTI PCs	1,221
UTI PCs	1,857
MTI References	23,084,552
UTI References	16,138,543
Misses	122,477
Evictions	121,965

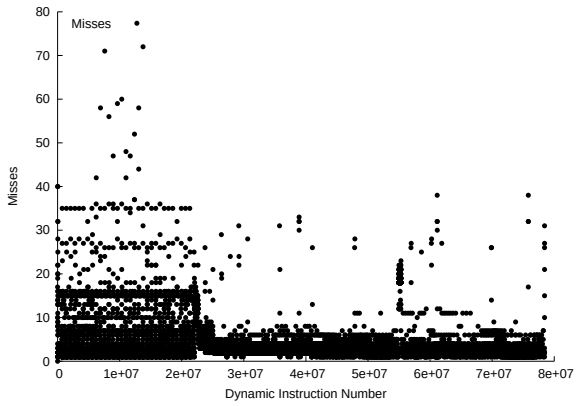


Figure 66: The miss rate over the full dynamic execution of the MiBench benchmark *cjpeg*.

The *cjpeg* benchmark implements the standard, lossy JPEG compression algorithm. This is a typical algorithm based on discrete cosine transformations (DCTs) that computationally is similar to most image format routines. The input is a large color image.

This benchmark needs a large increase in the on-die storage for instructions. It encounters a thrashing in the application due to insufficient space within the 32KB limit. With a larger space reserved for the input data, this benchmark would also be well behaved for the SoftCache framework.

B.2.4 crc

Table 38: MiBench *crc* performance results.

Unique PCs	1,037
MTI PCs	335
UTI PCs	702
MTI References	825,250,413
UTI References	159,960,596
Misses	228
Evictions	0

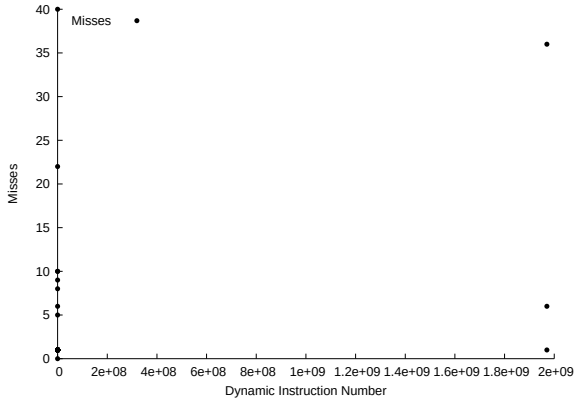


Figure 67: The miss rate over the full dynamic execution of the MiBench benchmark *crc*.

The *crc* benchmark is a 32-bit cyclic redundancy check (CRC) implementation for a test file. In general, CRC checks are used widely in communications system to verify files transferred such as in digital television signals. The input data is the sound test file from the ADPCM benchmark.

This application easily fits in the 32KB storage space for data on-die. Moreover, all the misses are cold-start misses and preloading the known execution path would eliminate all later misses.

B.2.5 dijkstra

Table 39: MiBench *dijkstra* performance results.

Unique PCs	1,283
Unique Addresses	17,480
MTI PCs	484
UTI PCs	799
MTI References	42,979,220
UTI References	74,326,910
Misses	237,372
Evictions	236,860

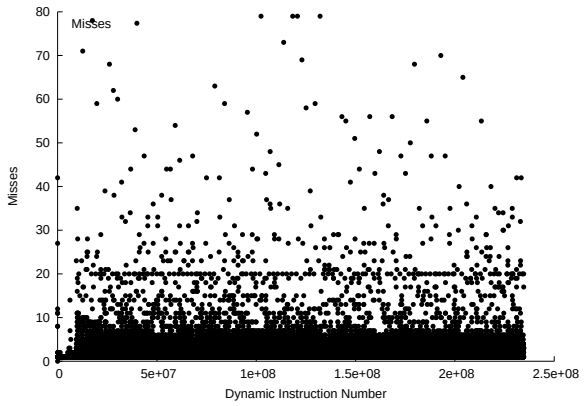


Figure 68: The miss rate over the full dynamic execution of the MiBench benchmark *dijkstra*.

The *dijkstra* benchmark constructs an adjacency matrix representation of a large graph. It then uses Djikstra's algorithm to compute the shortest path between every pair of nodes. This algorithm is $O\left(n^2\right)$ in time.

This benchmark needs a large increase in the on-die storage for instructions. It encounters a thrashing in the application due to insufficient space within the 32KB limit. With a larger space reserved for the input data, this benchmark would also be well behaved for the SoftCache framework.

B.2.6 fft

Table 40: MiBench *fft* performance results.

Unique PCs	1,340
Unique Addresses	136,156
MTI PCs	438
UTI PCs	902
MTI References	89,430,352
UTI References	41,886,360
Misses	141,542
Evictions	141,030

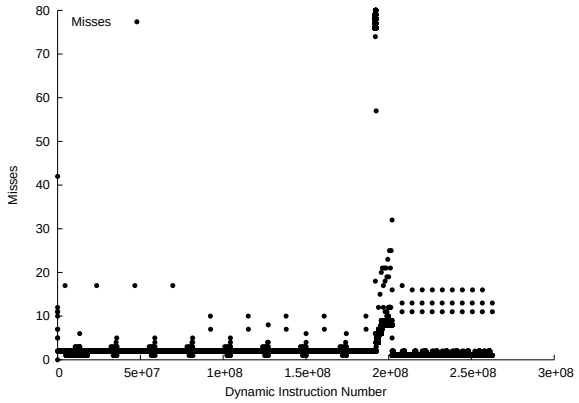


Figure 69: The miss rate over the full dynamic execution of the MiBench benchmark *fft*.

The *fft* benchmark is an implementation of the fast Fourier transform (FFT) as applied to a large array of data. This kernel is commonly used to isolate the frequencies in input signals to facilitate operations such as audio compression or noise cancellation. The array input is a series of pseudorandom amplitude and frequency values.

This benchmark needs a large increase in the on-die storage for instructions. It encounters a thrashing in the application due to insufficient space within the 32KB limit. With a larger space reserved for the input data, this benchmark would also be well behaved for the SoftCache framework.

B.2.7 gs

Table 41: MiBench *gs* performance results.

Unique PCs	3,828
Unique Addresses	32,111
MTI PCs	1,689
UTI PCs	2,139
MTI References	173,734
UTI References	29,601
Misses	3,576
Evictions	3,064

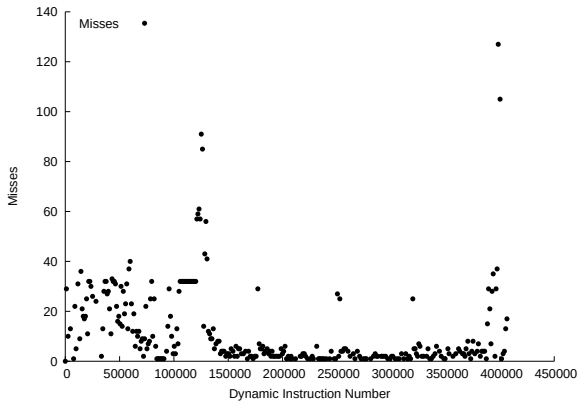


Figure 70: The miss rate over the full dynamic execution of the MiBench benchmark *gs*.

The *gs* benchmark is the open source PostScript intepreter known as GhostScript. This interpreter lacks a graphical interface, and is strictly the language interpreter. PostScript is a defacto standard for printers and some display devices such as Apple computers.

This benchmark needs a large increase in the on-die storage for instructions. It encounters a thrashing in the application due to insufficient space within the 32KB limit. With a larger space reserved for the input data, this benchmark would also be well behaved for the SoftCache framework.

B.2.8 ispell

Table 42: MiBench *ispell* performance results.

Unique PCs	1,318
Unique Addresses	71,362
MTI PCs	535
UTI PCs	783
MTI References	152,808
UTI References	49,774
Misses	6,830
Evictions	6,318

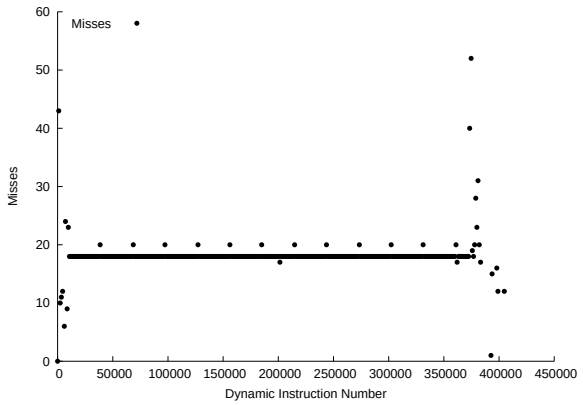


Figure 71: The miss rate over the full dynamic execution of the MiBench benchmark *ispell*.

The *ispell* benchmark is a fast spell checker similar to the UN*X "spell" but designed to run much faster. This application supports context-based checking, alternate spelling suggestions, and multiple languages. The input is a large document from the Internet as a web page.

This benchmark needs a large increase in the on-die storage for instructions. It encounters a thrashing in the application due to insufficient space within the 32KB limit. With a larger space reserved for the input data, this benchmark would also be well behaved for the SoftCache framework.

B.2.9 lame

Table 43: MiBench *lame* performance results.

Unique PCs	8,531
Unique Addresses	93,088
MTI PCs	5,045
UTI PCs	3,486
MTI References	599,509,141
UTI References	31,865,406
Misses	3,270,672
Evictions	3,270,160

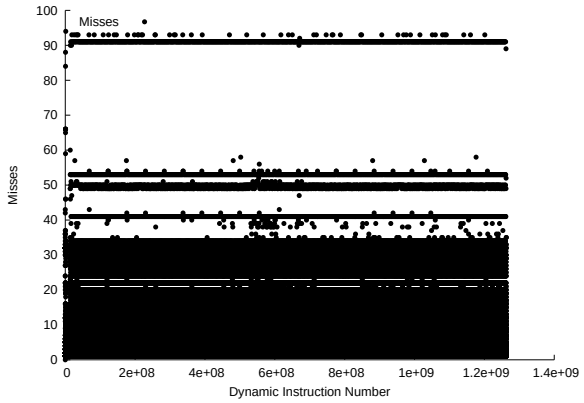


Figure 72: The miss rate over the full dynamic execution of the MiBench benchmark *lame*.

The *lame* benchmark is the open source MP3 encoder that supports both constant and variable bitrate encoding of audio files to MP3 format. This is a lossy compression algorithm similar to the JPEG algorithm. The input is a large wave file.

This benchmark is far too large to run in the SoftCache framework under any conditions. With over 3.2 million evictions and 55MB of code transferred, this is a perfect representation of the wrong class of application to run on a SoftCache framework.

B.2.10 lout

Table 44: MiBench *lout* performance results.

Unique PCs	8,355
Unique Addresses	32,493
MTI PCs	1,767
UTI PCs	6,588
MTI References	224,988
UTI References	48,990
Misses	2,212
Evictions	1,700

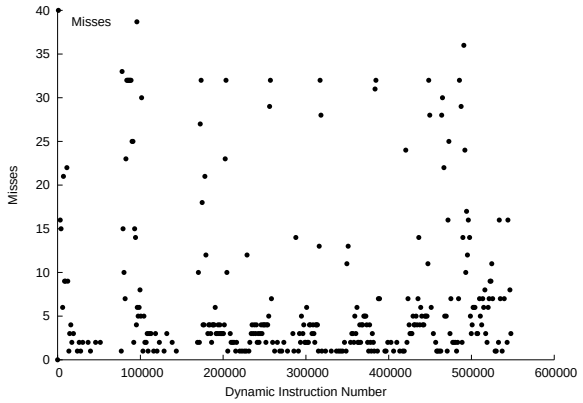


Figure 73: The miss rate over the full dynamic execution of the MiBench benchmark *lout*.

The *lout* benchmark is actually the MiBench *typeset* benchmark which is a typesetting tool for HTML. This benchmark captures the processing required to render an HTML page without actually rendering anything. This is similar to a simplistic web browser engine, such as Gecko from the Mozilla Foundation. The input is a large web page.

This benchmark is another example of a program that requires more than the 32KB on-die instruction area. With intermittent events causing large misses, and then a long-running period of misses, this application requires more storage than is available.

B.2.11 madplay

Table 45: MiBench *madplay* performance results.

Unique PCs	3,685
Unique Addresses	52,797
MTI PCs	1,472
UTI PCs	2,213
MTI References	76,344,468
UTI References	35,120,382
Misses	32,550
Evictions	32,038

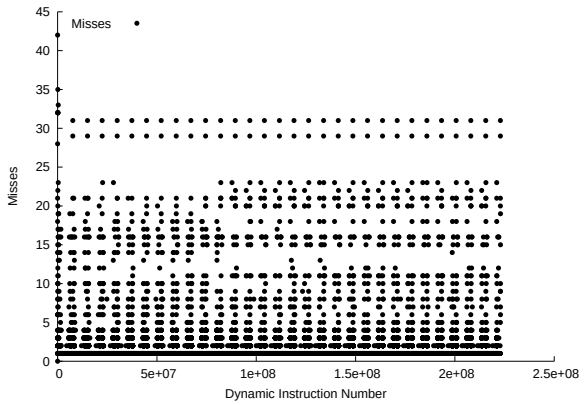


Figure 74: The miss rate over the full dynamic execution of the MiBench benchmark *madplay*.

The *madplay* benchmark is a high-quality MP3 audio decoder. This decoder supports Layer I, Layer II, and Layer III MPEG audio decoding. The input file is a large MP3 test file.

This benchmark needs a large increase in the on-die storage for instructions. It encounters a thrashing in the application due to insufficient space within the 32KB limit. With a larger space reserved for the input data, this benchmark would also be well behaved for the SoftCache framework.

B.2.12 math

Table 46: MiBench *math* performance results.

Unique PCs	1,835
Unique Addresses	2,636
MTI PCs	610
UTI PCs	1,225
MTI References	569,474,694
UTI References	101,613,946
Misses	205
Evictions	0

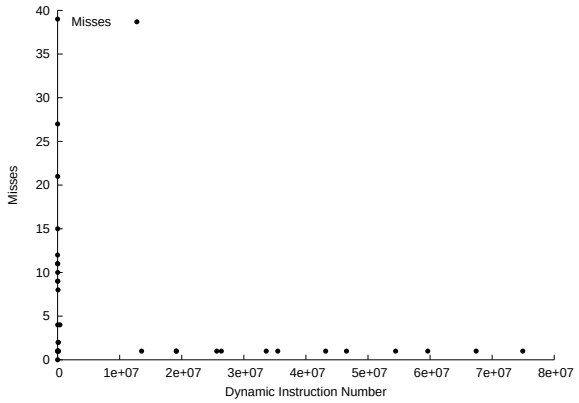


Figure 75: The miss rate over the full dynamic execution of the MiBench benchmark *math*.

The *math* benchmark, called *basicmath* in MiBench, consists of simple calculations that generally lack dedicated hardware support in embedded processors. These calculations are tasks like square root, angle unit conversions, cubic function solutions, etc. The input data is a fixed set of constants.

This application would easily fit in the 32KB data storage SRAM on-die. Each miss is a cold-start miss, and preloading the hot path would eliminate the later misses entirely.

B.2.13 patricia

Table 47: MiBench *patricia* performance results.

Unique PCs	1,948
Unique Addresses	696,967
MTI PCs	649
UTI PCs	1,299
MTI References	183,565,348
UTI References	96,640,079
Misses	131,405
Evictions	130,893

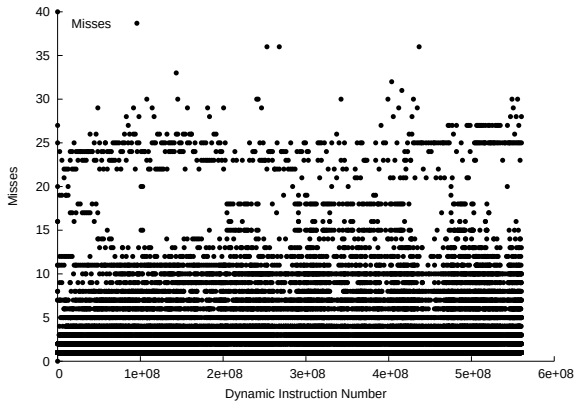


Figure 76: The miss rate over the full dynamic execution of the MiBench benchmark *patricia*.

The *patricia* benchmark is an evaluation of the Patricia trie data structure for sparse leaf nodes in trees. This type of construct is common in network routing tables. The input set is a list of IP traffic captured from a network over a period of time, with the IP numbers disguised.

This benchmark needs a large increase in the on-die storage for instructions. It encounters a thrashing in the application due to insufficient space within the 32KB limit. With a larger space reserved for the input data, this benchmark would also be well behaved for the SoftCache framework.

B.2.14 pgp

Table 48: MiBench *pgp* performance results.

Unique PCs	1,057
MTI PCs	482
UTI PCs	575
MTI References	30,717
UTI References	11,267
Misses	216
Evictions	0

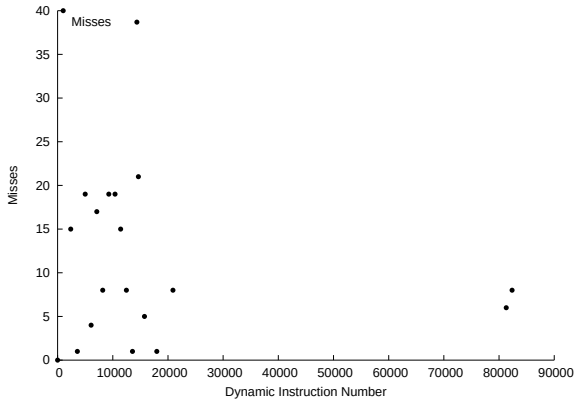


Figure 77: The miss rate over the full dynamic execution of the MiBench benchmark *pgp*.

The *pgp* benchmark is an implementation of the pretty good privacy (PGP) public key encryption system. This uses RSA encryption with digital signatures, a method developed by Phil Zimmerman. The input is a small text file, as the primary purpose of PGP is to securely exchange keys for a normal block cipher algorithm which is much faster than the PGP algorithm.

This application easily fits in the 32KB storage space for data on-die. Moreover, all the misses are cold-start misses and preloading the known execution path would eliminate all later misses.

B.2.15 qsort

Table 49: MiBench *qsort* performance results.

Unique PCs	1,233
Unique Addresses	505,276
MTI PCs	413
UTI PCs	820
MTI References	114,137,996
UTI References	63,425,373
Misses	430,952
Evictions	430,440

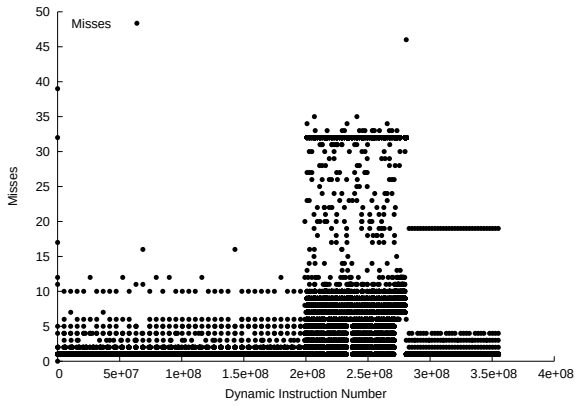


Figure 78: The miss rate over the full dynamic execution of the MiBench benchmark *qsort*.

The *qsort* test sorts a large array of strings with the quick sort algorithm. The input data set is a grouping of three-tuples, meant to represent points of data.

This benchmark needs a large increase in the on-die storage for instructions. It encounters a thrashing in the application due to insufficient space within the 32KB limit. With a larger space reserved for the input data, this benchmark would also be well behaved for the SoftCache framework.

B.2.16 rawcaudio

Table 50: MiBench *rawcaudio* performance results.

Unique PCs	647
Unique Addresses	694
MTI PCs	182
UTI PCs	465
MTI References	3,306
UTI References	676
Misses	119
Evictions	0

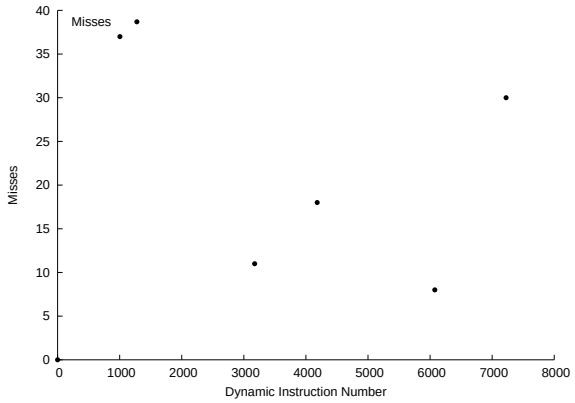


Figure 79: The miss rate over the full dynamic execution of the MiBench benchmark *rawcaudio*.

The *rawcaudio* benchmark is actually an implementation of the adaptive differential pule code modulation (ADPCM) encoder. The input is a series of 16-bit linear samples which are converted to 4-bit samples. The actual input file is a large speech sample.

This application only requires a few KB of storage in the on-die SRAM. Active preloading of the hot path would eliminate all misses entirely.

B.2.17 rijndael

Table 51: MiBench *rijndael* performance results.

Unique PCs	811
Unique Addresses	1,047
MTI PCs	237
UTI PCs	574
MTI References	4,277
UTI References	911
Misses	141
Evictions	0

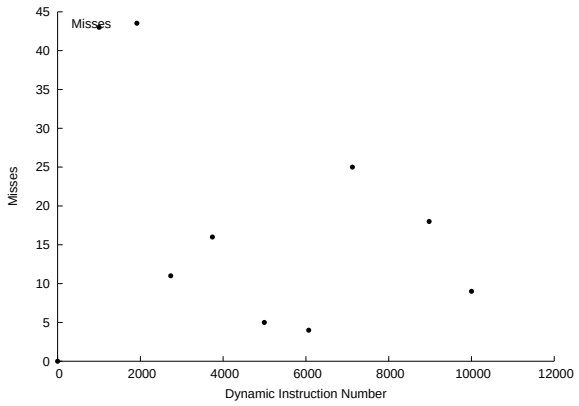


Figure 80: The miss rate over the full dynamic execution of the MiBench benchmark *rijndael*.

The *rijndael* benchmark is an implementation of Rijndael's secure encryption system. This is the algorithm selected by the National Institute of Standards and Technologies (NIST) for the Advanced Encryption Standard (AES). This block cipher algorithm is run on the same large ASCII input file as *sha*.

This benchmark fits easily into the on-die 32KB SRAM region for instructions. Moreover, it is a perfect model of the SoftCache mechanism where the application runs for a long period of time before encountering a minor phase adjustment and continuing.

B.2.18 say

Table 52: MiBench *say* performance results.

Unique PCs	2,632
Unique Addresses	572,708
MTI PCs	837
UTI PCs	1,795
MTI References	8,205,453
UTI References	36,735,235
Misses	20,127
Evictions	19,615

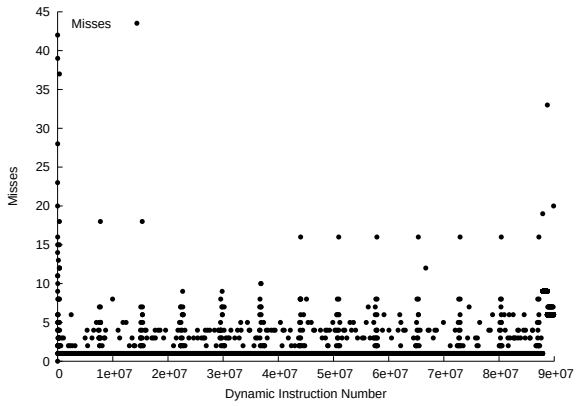


Figure 81: The miss rate over the full dynamic execution of the MiBench benchmark *say*.

The *say* benchmark is actually a speech-based benchmark built on the sphinx speech decoder. The large input is a long sequence of speech, which is handled one sequence at a time.

This benchmark needs a large increase in the on-die storage for instructions. It encounters a thrashing in the application due to insufficient space within the 32KB limit. With a larger space reserved for the input data, this benchmark would also be well behaved for the SoftCache framework.

B.2.19 search

Table 53: MiBench *search* performance results.

Unique PCs	798
Unique Addresses	10,370
MTI PCs	273
UTI PCs	525
MTI References	1,265,998
UTI References	400,836
Misses	28
Evictions	0

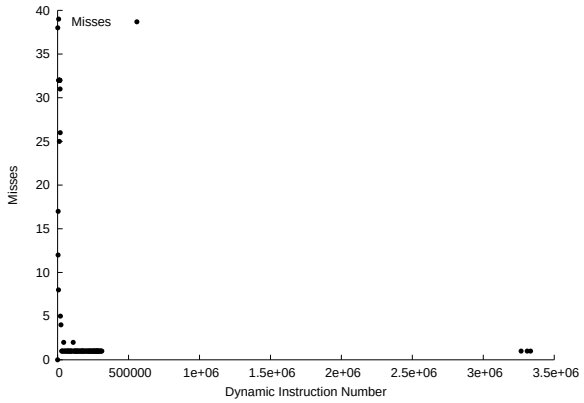


Figure 82: The miss rate over the full dynamic execution of the MiBench benchmark *search*.

The *search* benchmark is the stringsearch benchmark. This searches for specific words or phrases in a case insensitive manner. The input is a large ASCII file.

This benchmark fits well within the SoftCache on-die memory space, and exhibits excellent behavior with respect to misses. The very long period of program stability between misses is the ideal characteristic of any SoftCache application.

B.2.20 sha

Table 54: MiBench *sha* performance results.

Unique PCs	1,045
Unique Addresses	2,887
MTI PCs	322
UTI PCs	723
MTI References	32,563,258
UTI References	4,020,084
Misses	266
Evictions	0

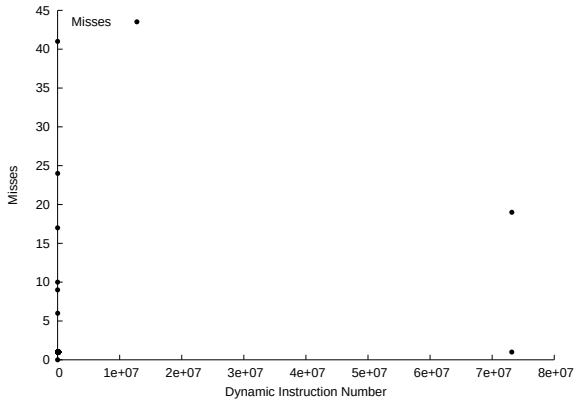


Figure 83: The miss rate over the full dynamic execution of the MiBench benchmark *sha*.

The *sha* benchmark is the secure hash algorithm (SHA) that generates 160-bit message digests for a given input. Aside from digital signatures, the SHA algorithm is also used to exchange cryptographic keys. The input data is a large ASCII file.

This benchmark is an excellent application of the SoftCache. Once the initial setup of the application is made, no misses occur until the very end when the result is being reported. This benchmark also fits easily within the 32KB limitations of the on-die SRAM.

B.2.21 susan

Table 55: MiBench *susan* performance results.

Unique PCs	1,323
Unique Addresses	226,662
MTI PCs	507
UTI PCs	816
MTI References	3,351,885
UTI References	6,431,816
Misses	7,814
Evictions	7,302

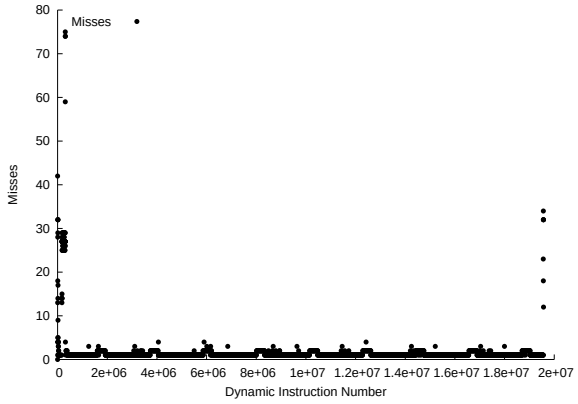


Figure 84: The miss rate over the full dynamic execution of the MiBench benchmark *susan*.

The *susan* benchmark is designed to recognize corners and edges in images – specifically, in Magnetic Resonance Images (MRIs) of the brain. This benchmark can perform rudimentary image smoothing, brightness and threshold adjustments, etc. The large input data is a complex picture.

This benchmark needs a large increase in the on-die storage for instructions. It encounters a thrashing in the application due to insufficient space within the 32KB limit. With a larger space reserved for the input data, this benchmark would also be well behaved for the SoftCache framework. The interesting behavior of this application is that the very large range of addresses touched result in a consistent low miss rate. This is due to the iterative walk through the image, continuously fetching new data at a fixed rate to the application without revisiting old data.

B.2.22 tiff2bw

Table 56: MiBench *tiff2bw* performance results.

Unique PCs	2,150
Unique Addresses	27,892
MTI PCs	851
UTI PCs	1,299
MTI References	57,981,267
UTI References	313,557
Misses	7,010
Evictions	6,498

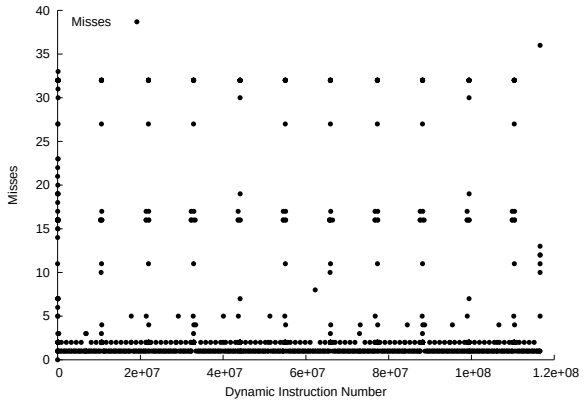


Figure 85: The miss rate over the full dynamic execution of the MiBench benchmark *tiff2bw*.

The *tiff2bw* benchmark converts a TIFF image from color to black-and-white. This type of downsampling is common in remote image processing applications to reduce bandwith and simplify feature recognition. When necessary, the raw or color version can be offloaded to upstream devices for a more complex algorithm such as selective attention.

This benchmark needs a large increase in the on-die storage for instructions. It encounters a thrashing in the application due to insufficient space within the 32KB limit. With a larger space reserved for the input data, this benchmark would also be well behaved for the SoftCache framework.

B.2.23 tiffdither

Table 57: MiBench *tiffdither* performance results.

Unique PCs	2,619
Unique Addresses	27,135
MTI PCs	1,096
UTI PCs	1,523
MTI References	166,737,501
UTI References	91,260,043
Misses	26,247
Evictions	25,735

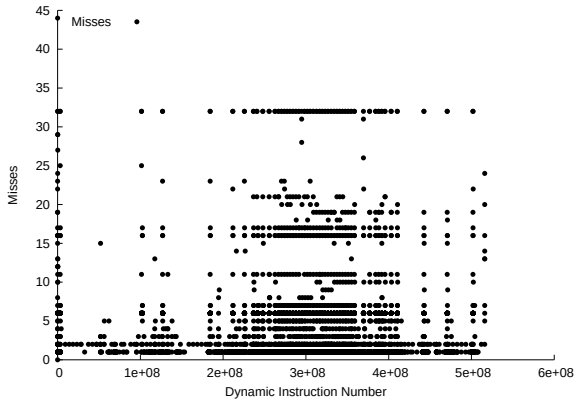


Figure 86: The miss rate over the full dynamic execution of the MiBench benchmark *tiffdither*.

The *tiffdither* benchmark is similar to the *tiff2bw* as a TIFF downsampling algorithm. The resolution and size are reduced by application of a dithering algorithm, resulting in a loss of clarity. This is another useful algorithm for more complex systems such as selective attention programs.

This benchmark needs a large increase in the on-die storage for instructions. It encounters a thrashing in the application due to insufficient space within the 32KB limit. With a larger space reserved for the input data, this benchmark would also be well behaved for the SoftCache framework.

B.2.24 toast

Table 58: MiBench *toast* performance results.

Unique PCs	1,402
Unique Addresses	1,469
MTI PCs	431
UTI PCs	971
MTI References	7,534
UTI References	1,640
Misses	206
Evictions	0

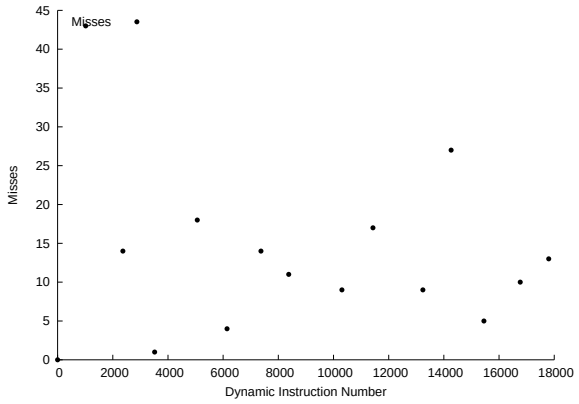


Figure 87: The miss rate over the full dynamic execution of the MiBench benchmark *toast*.

The *toast* benchmark is actually the global standard for mobile (GSM) communications encoder engine. The GSM standard is the standard for cellular communications in Europe and other countries. The GSM algorithm is based on a combination of time- and frequency-division multiple access (TDMA/FDMA) methods to encode data streams. The input is a large speech sample.

This benchmark fits well within the SoftCache on-die storage space. To avoid the misses encountered during the middle of execution, preloading the hot path would enhance the program behavior. This is another good application to run in the SoftCache framework.

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 *PLDI*, pages 302–312, June 1994.

[3] Gianluca Albera and R. Iris Bahar. Power and Performance Tradeoffs using Various Cache Configurations. In *ISLPED*, August 1998.

[4] Gene M. Amdahl. Validity of the single processor approach to achieving large scale computing abilities. In *Proc. AFIPS 1967 Spring Joint Computer Conf. 30 (April)*, pages 483–485, 1967.

[5] Simon Baatz, Christoph Bieschke, Matthias Frank, Carmen Kühl, Peter Martini, and Christoph Scholz. Building Efficient Bluetooth Scatternet Topologies from 1-Factors. In *Proceedings of the IASTED Intl Conference on Wireless and Optical Communications*, July 2002.

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

[7] Rajeshwari Banakar, Stefan Steinke, Bo-Sik Lee, M. Balakrishnan, and Peter Marwedel. Comparison of Cache- and Scratch-Pad based Memory Systems with respect to Performace, Area and Energy Consumption. Technical report, Universität Dortmund, September 2001.

[8] 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 *10th International Workshop on Hardware/Software Codesign*, May 2002.

[9] 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.

[10] Robert Bedichek. Talisman-2 — A Fugu System Simulator. `http://bediche k.or g/ - robert/talisman2 /`, August 1999.

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

[12] Bryan Black, Bohuslav Rychlik, and John P. Shen. The Block-Based Trace Cache. In *ISCA-26*, May 1999.

[13] Rastislav Bodik, Rajiv Gupta, and Mary Lou Soffa. Complete Removal of Redundant Expressions. In *PLDI*, pages 1–14, 1998.

[14] David Brooks, Vivek Tiwari, and Margaret Martonosi. Wattch: a framework for architectural-level power analysis and optimizations. In *Proceedings of the International Symposium on Computer Architecture*, pages 83–94, 2000.

[15] Bruno De Bus, Dominique Chanet, Bjorn De Sutter, Ludo Van Put, and Koen De Bosschere. The design and implementation of FIT: a flexible instrumentation toolkit. In *Workshop on Program Analysis for Software Tools and Engineering*, 2004.

[16] Naehyuck Chang, Kwanho Kim, and Hyung Gyu Lee. Cycle-Accurate Energy Measurement and Characterization With a Case Study of the ARM7TDMI. In *IEEE Transactions on Very Large Scale Integration Systems*, April 2002.

[17] Samprit Chatterjee and Bertram Price. *Regression Analysis by Example*. Wiley and Sons, ISBN 0-471-88479-0, 1991.

[18] Cristina Cifuentes and Antoine Fraboulet. Interprocedural Data Flow Recovery of High-Level Language Code from Assembly. Technical Report 421, University of Queensland, December 1997.

[19] Cristina Cifuentes and Antoine Fraboulet. Intraprocedural Static Slicing of Binary Executables. In *Proceedings of the International Conference on Software Maintenance*, pages 188–195, 1997.

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

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

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

[23] Gilberto Contreras, Margarent Martonosi, Jinzhan Peng, Roy Ju, and Guei-Yuan Lueh. Xtrem: A power simulator for the intel xscale core. In *LCTES*, 2004.

[24] 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.

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

[26] Datasheet. Intel PXA27x Family. Technical report, Intel, http://www.intel.com/design/-embeddedpca/applicationsprocessors/302302.htm 2004.

[27] Peter J. Denning. The working set model for program behavior. In *Communications of the ACM*, volume 11, No. 5, pages 323–333, May 1968.

[28] Peter J. Denning. Virtual Memory. In *Computing Surveys*, volume 2, No. 3, pages 153–189, September 1970.

[29] Peter J. Denning. Before Memory was Virtual. In *In the Beginning: Recollections of Software Pioneers*. Robert Glass, ed., IEEE Press, 1997.

[30] Karel Driesen and Urs Holzle. Improving Indirect Branch Prediction With Source- and Arity-based Classification and Cascaded Prediction. Technical report, UC-Santa Barbara, March 1998.

[31] Sandhya Dwarkadas, Nikolaos Hardavellas, Leonidas Kontothanassis, Rishiyur Nikhil, and Robert Stets. Cashmere-VLM: Remote Memory Paging for Software Distributed Shared Memory. In *Proceedings of the Intl Parallel Processing Symposium and the Symposium on Parallel and Distributed Processing*, pages 153–159, 1999.

[32] Intel Engineering. Intel PXA255 Processor Design Guide. Technical report, Intel Corp., 278964-001, March 2003.

[33] Intel Engineering. Intel PXA255 Processor Developer's Manual. Technical report, Intel Corp., 278693-001, March 2003.

[34] Intel Engineering. Intel XScale Microarchitecture for the PXA255 Processor (User's Manual). Technical report, Intel Corp., 278796, March 2003.

[35] LeCroy Engineering. Accurate Instantaneous Power Measurements. Technical report, LeCroy, Inc., AN29 0499 5M TECH 1999.

[36] Tektronix Engineering. XYZs of Oscilloscopes. Technical report, Tektronix, Inc., 03W-8605-2 2001.

[37] Tektronix Engineering. Bandwidth Alone $\neq$ Measurement Accuracy. Technical report, Tektronix, Inc., 55W-19248-2 2005.

[38] Tektronix Engineering. Power Measurement and Analysis Primer. Technical report, Tektronix, Inc., 55W-18412-0 2005.

[39] Keith I. Farkas and Norman P. Jouppi. Complexity/Performance Tradeoffs with Non-Blocking Loads. Technical report, DEC, March 1994.

[40] Kristián Flautner, Nam Sung Kim, Steve Martin, David Blaauw, and Trevor Mudge. Drowsy Caches: Simple Techniques for Reducing Leakage Power. In *International Symposium on Computer Architecture*, 2002.

[41] M.D. Flouris and Evangelos P. Markatos. The Network RamDisk: Using Remote Memory on Heterogeneous NOWs. In *Cluster Computing*, volume 2, pages 281–293, 1999.

[42] Dominique Foata and Doron Zeilberger. The Collector's Brotherhood Problem Using the Newman-Shepp Symbolic Method. In *Algebra Universalis*, 49(2003), 387-395.

[43] Daniel H. Friendly, Sanjay J. Patel, and Yale N. Patt. Putting the Fill Unit to Work: Dynamic Optimizations for Trace Cache Microprocessors. In *Proceedings of the International Symposium on Microarchitecture*, 1998.

[44] Joshua B. Fryman, Chad M. Huneycutt, Hsien-Hsin S. Lee, Kenneth M. Mackenzie, and David E. Schimmel. Energy Efficient Network Memory for Ubiquitous Devices. Technical report, Georgia Institute of Technology, GIT-CERCS-03-05, 2003.

[45] Joshua B. Fryman, Chad M. Huneycutt, Hsien-Hsin S. Lee, Kenneth M. Mackenzie, and David E. Schimmel. Energy Efficient Network Memory for Ubiquitous Devices. In *IEEE MICRO*, Sep/Oct 2003.

[46] Joshua B. Fryman, Chad M. Huneycutt, and Kenneth M. Mackenzie. Investigating a Soft-Cache using Dynamic Rewriting. In *FDDO-4*, November 2001.

[47] Matthew R. Guthaus, Jeffrey S. Ringenberg, Dan Ernst, Todd M. Austin, Trevor Mudge, and Richard B. Brown. MiBench: A free, commercially representative embedded benchmark suite. In *IEEE 4th Annual Workshop on Workload Characterization*, December 2001.

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

[49] William R. Hamburgen, Deborah A. Wallach, Marc A. Viredaz, Lawrence S. Brakmo, Carl A. Waldspurger, Joel F. Bartlett, Timothy Mann, and Keith I. Farkas. Itsy: Stretching the bounds of mobile computing. In *IEEE Computer*, volume 34, pages 28–37, 2001.

[50] Dan W. Hammerstrom and Edward S. Davidson. Information Content of CPU Referencing Behavior. In *Proceedings of the International Symposium on Computer Architecture*, 1977.

[51] Tegze P. Haraszti. *CMOS Memory Circuits*, pages 113–124,146–150. Kluwer Academic Publishers, Norwell, Massachusetts, 2000.

[52] Paul Havinga and Gerard Smit. Energy-efficient wireless networking for multimedia applications. In *Wireless Communications and Mobile Computing*. Wiley and Sons, 2000.

[53] Kim Hazelwood and James E. Smith. Exploring Code Cache Eviction Granularities in Dynamic Optimization Systems. In *Proceedings of the International Symposium on Code Generation and Optimization*, 2004.

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

[55] John L. Henneessy and David A. Patterson. *Computer Architecture: A Quantitative Approach (2nd Ed)*, page Chapters 1 and 5. Morgan Kaufmann, San Francisco, California, 2000.

[56] Paul Horowitz and Winfield Hill. *The Art of Electronics (2nd Ed.)*. Cambridge University Press, Cambridge, United Kingdom, 1989.

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

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

[59] Livia Iftode, Kai Li, and Karin Petersen. Memory Servers for Multicomputers. In *Proc. of the IEEE Intl Computer Conference*, pages 543–547, 1993.

[60] Sotiris Ioannidisgif, Evangelos P. Markatosgif, and Julia Sevaslidou. On Using Network Memory to Improve the Performance of Transaction-Based Systems. In *Intl Conference on Parallel and Distributed Processing Techniques and Applications*, 1998.

[61] Canturk Isci and Margaret Martonosi. Identifying Program Power Phase Behavior Using Power Vectors. In *Workshop on Workload Characterizations*, November 2003.

[62] Canturk Isci and Margaret Martonosi. Runtime Power Monitoring in High-End Processors: Methodology and Empirical Data. In *IEEE International Symposium on Microarchitecture*, November 2003.

[63] NTT Japan. BLUEBIRD Project. 2003. http://www.ntt s.co.jp/java/-bluegrid/en/ .

[64] Howard Johnson and Martin Graham. *High-Speed Digital Design: A Handbook of Black Magic*. Prentice Hall PTR, New Jersey, USA, 1993.

[65] Teresa L. Johnson and Wen-Mei W. Hwu. Run-Time Adaptive Cache Hierarchy Management via Reference Analysis. In *Proceedings of the 24th Annual International Symposium on Computer Architecture*, 1997.

[66] Stephen Jourdan, Lihu Rappoport, Yoav Almog, Mattan Erez, Adi Yoaz, and Ronny Ronen. eXtended Block Cache. In *HPCA-6*, January 2000.

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

[68] T. Kilburn, D.B.G. Edwards, M.J. Lanigan, and F.H. Sumner. One-level storage system. In *IRE Transactions*, EC-11(2):223–225 1962.

[69] Dongkyun Kim, J.J Garcia-Luna-Aceves, Katia Obraczka, Juan-Carlos Cano, and Pietro Manzoni. Power-Aware Routing Based on The Energy Drain Rate for Mobile Ad Hoc Networks. In *Proceedings of the IEEE Intl Conference on Computer Communication and Networks*, Oct 2002.

[70] Martin Kubisch, Seble Mengesha, Daniel Hollos, Holger Karl, and Adam Wolisz. Applying ad-hoc relaying to improve capacity, energy efficiency, and immission in infrastructure-based WLANs. Technical report, Technical University Berlin, July 2002.

[71] Hsien-Hsin S. Lee and Gary S. Tyson. Region-Based Caching: An Energy-Delay Efficient Memory Architecture for Embedded Processors. In *Proceedings of the International Conference on Compilers, Architectures, Syntesis on Embedded Systems*, 2000.

[72] Evangelos P. Markatos and George Dramitinosgif. Implementation of a Reliable Remote Memory Pager. In *USENIX*, pages 177–190, 1996.

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

[74] Carlos Molina, Antonio González, and Jordi Tubella. Dynamic Removal of Redundant Computations. Technical Report UPC-DAC-1998-022, Universitat Politècnia de Catalunya, Barcelona, Spain, April 1998.

[75] James 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.

[76] Csaba Andras Moritz, Matthew Frank, Walter Lee, and Saman Amarasinghe. Hot Pages: Software Caching for Raw Microprocessors. Technical Report MIT-LCS-TM-599, Massachusetts Institute of Technology, 1999.

[77] Andreas Moshovos and Gurindar S. Sohi. Speculative Memory Cloaking and Bypassing. In *International Journal on Parallel Programming*, 1999.

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

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

[80] 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.

[81] Vlad-Mihai Panait, Amit Sasturkar, and Weng-Fai Wong. Static Identification of Delinquent Loads. In *Proceedings of the International Symposium on Code Generation and Optimization*, 2004.

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

[83] 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.

[84] Chanik Park, Junghee Lim, Kiwon Kwon, Jaejin Lee, and Sang Lyul Min. Compiler-Assisted Demand Paging for Embedded Systems with Flash Memory. In *International Conference on Embedded Software*, 2004.

[85] Erez Perelman, Greg Hamerly, and Brad Calder. Picking Statistically Valid and Early Simulation Points. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, 2003.

[86] Dionisios N. Pnevmatikatos and Evangelos P. Markatos. On Using Network RAM as a Non-volatile Buffer. In *Cluster Computing*, pages 295–303, 1999.

[87] Christian Poellabauer and Karsten Schwan. Power-Aware Video Decoding using Real-Time Event Handlers. In *IEEE International Conference on Pervasive Computing and Communications*, September 2002.

[88] Christian Poellabauer and Karsten Schwan. Energy-Aware Media Transcoding in Wireless Systems. In *International Workshop on Wireless Mobile Multimedia*, March 2004.

[89] 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.

[90] Jude A. Rivers and Edward S. Davidson. Reducing Conflicts in Direct-mapped Caches with a Temporality-based Design. In *Proceedings of the 1996 International Conference on Parallel Processing*, 1996.

[91] John L. Ross and Mooly Sagiv. Building a Bridge between Pointer Aliases and Program Dependences. volume 1381, pages 221–?, 1998.

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

[93] D. Sayre. Is Automatic "Folding" of Programs Efficient Enough to Displace Manual? In *Communications of the ACM*, volume 12, No. 12, pages 656–660, December 1969.

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

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

[96] P. Shivakumar and Norman P. Jouppi. CACTI 3.0: An Integrated Cache T iming, Power, and Area Model. Technical report, Compaq WRL, August 2001.

[97] Tajana Simunic, Luca Benini, and Giovanni De Micheli. Energy-efficient design of battery-powered embedded systems. In *International Symposium on Low Power Electronics and Design*, August 1999.

[98] Amit Sinha and Anantha Chandrakasan. Jouletrack - a web based tool for software energy profiling. In *Design Automation Conference*, pages 220–225, 2001.

[99] Saurabh Sinha and Mary Jean Harrold. Interprocedural Control Dependence. In *International Symposium on Software Testing and Analysis*, 1998.

[100] 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.

[101] Emin G. Sirer, Robert Grimm, Arthur J. Gregory, and Brian 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.

[102] 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 *International Symposium on System Synthesis*, October 2002.

[103] Bjorn De Sutter, Bruno De Bus, Koen De Bosschere, and Saumya Debray. Combining global code and data compaction. In *Workshop on LCTES*, 2001.

[104] S. Takahashi, H. Nishino, K. Yoshihiro, and K. Fuchi. System design of the ETL Mk-6 computers. In *Information Processing (Proc. IFIP Congress 62)*, volume Amsterdam, The Netherlands: North Holland Publishing Co., page 690, 1963.

[105] Vivek Tiwari, Sharad Malik, and Andrew Wolfe. *Power analysis of embedded software: a first step towards software power minimization*. Readings in hardware/software co-design. Kluwer Academic Publishers, Norwell, MA, USA, 2001.

[106] Gary S. Tyson and Todd M. Austin. Improving the Accuracy and Performance of Memory Communication Through Renaming. In *Proceedings of the International Symposium on Microarchitecture*, 1997.

[107] Gary S. Tyson, Matthew Farrens, John Matthews, and Andrew R. Pleszkun. A Modified Approach to Data Cache Management. In *Proceedings of the International Symposium on Microarchitecture*, 1995.

[108] Osman S. Unsal, Raksit Ashok, Israel Koren, C. Mani Krishna, and Csaba Andras Moritz. Cool-Cache for Hot Multimedia. In *Proceedings of the International Symposium on Microarchitecture*, 2001.

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

[110] Perry H. Wang, Jamison D. Collins, Hong Wang, Dongkeun Kim, Bill Greene, Kai-Ming Chan, Aamir B. Yunus, Terry Sych, Stephen F. Moore, and John P. Shen. Helper Threads via Virtual Multithreading On An Experimental Itanium 2 Processor-based Platform. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, 2004.

[111] Maurice V. Wilkes. Slave Memories and Dynamic Storage Allocation. In *IEEE Transactions EC-14*, pages 270–271, April 1965.

[112] Maurice V. Wilkes. Computers Then and Now. In *Journal of the Association for Computing Machinery*, pages 1–7, January 1968.

[113] Emmet Witchel and Mendel Rosenblum. Embra: Fast and Flexible Machine Simulation. In *SIGMETRICS*, 1996.

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

[115] 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.

[116] W. Ye, Narayanan Vijaykrishnan, Mahmut T. Kandemir, and Mary Jane Irwin. The design and use of simplepower: a cycle-accurate energy estimation tool. In *Design Automation Conference*, pages 340–345, 2000.

[117] Michael Zhang and Krste Asanović. Highly-Associative Caches for Low-Power Processors. In *Kool Chips Workshop, 33rd International Symposium on Microarchitecture*, 2000.

VITA

Josh Fryman was born someplace in rural Kentucky, USA. Moving like a gypsy while his parents were climbing the corporate ladder, Josh managed to live in several states, from coast to coast. Josh obtained his B.S. degree in Computer Engineering from the University of Florida, and his Ph.D. degree from Georgia Institute of Technology. Josh is married to Hathai Sangsupan, and has one child, Brieana Fryman.

DOCUMENT

This document was typeset in LaTeX, using the Georgia Institute of Technology thesis template designed by Chuck Wilson. All development, simulation, numerical analysis, etc., was done on a set of Linux workstations running on various platforms. Graphs and plots were made using a mixture of gnuplot *(www.gnuplot.info)* and ploticus *(ploticus.sourceforge.net)*. Numerical analysis was done in a mixture of gnumeric *(www.gnome.org/projects/gnumeric)*, maple *(www.maplesoft.com)*, and hand-coded C programs. Circuit diagrams were made using xcircuit *(xcircuit.ece.jhu.edu/xcircuit.html)*. Other figures were made with a combination of dia *(www.gnome.org/projects/dia)*, xfig *(www.xfig.org)*, sketch a.k.a. skencil *(www.skencil.org)*, and tgif *(bourbon.usc.edu:8001/tgif)*.

No Microsoft products were used in any way during the creation of this document.

No Microsoft products were used in this research.