MATCHING ON-CHIP DATA STORAGE TO MEDIA AND TELECOMMUNICATION APPLICATION PROPERTIES

BY

HILLERY CATRIONA HUNTER

B.S., University of Illinois at Urbana-Champaign, 1999
M.S., University of Illinois at Urbana-Champaign, 2002

DISSERTATION

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Electrical Engineering
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2004

Urbana, Illinois

MATCHING ON-CHIP DATA STORAGE TO MEDIA AND TELECOMMUNICATION APPLICATION PROPERTIES

Hillery Catriona Hunter, Ph.D.
Department of Electrical and Computer Engineering
University of Illinois at Urbana-Champaign, 2004
Wen-mei W. Hwu, Adviser

This describes hardware and software necessary for compiler-controlled power savings in embedded data storage systems. Recent techniques for saving dynamic and static power in specialized SRAM arrays are leveraged to provide port, latency, and sleep configurability. Using advanced interprocedural pointer analysis to provide complete resolution of potential memory accesses, the compiler choses a power-saving configuration for each program data object. Data objects are then grouped according to the number of needed ports and desired access latency.

Both the hardware and software design of configurable SRAM was driven by a *data intent* characterization of telecommunication and media applications. This characterization is described, along with description of its application to other methods for power savings in embedded systems.

For configurable SRAM, data storage savings in the range of 29% static power and 6% dynamic power are achieved without sacrificing code performance. When slight performance degradation can be tolerated, the compiler uses profile feedback to realize an average of 51.7% static and 9.4% dynamic power reduction.

To my parents.

ACKNOWLEDGMENTS

I would first like to acknowledge the guidance of my adviser, Professor Wen-mei Hwu, and thank him for allowing me to pursue computer architecture studies in his IMPACT research group. My committee members, Gary Eden, Steve Lumetta, and Sanjay Patel, have also provided invaluable feedback and encouragement along the way.

Thanks are due to the many past and current members of the IMPACT research group, whose work has contributed to the infrastructure which enabled this research. I would specifically like to thank Erik Nystrom for a very productive collaboration to analyze the effects of interprocedural pointer analysis, Shane Ryoo for his overhaul of the memory profiling framework, and John Sias for innumerable bug fixes. Jame Player, Ron Barnes, Sain-Zee Ueng, and Ian Steiner have also helped with last-minute deadlines and generally provided much-needed humor and levity.

Lastly, I would like to recognize the National Science Foundation, IBM Corporation, Semiconductor Research Corporation, DARPA/MARCO GSRC Center, University of Illinois, and Department of Electrical and Computer Engineering for their financial support of my graduate studies.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Figure Page

1 INTRODUCTION

Traditional scaling trends indicate that as CMOS process technology improves and feature sizes shrink, leakage currents will soon vie with capacitive switching for dominance of overall microprocessor power consumption. In most design domains, these increased subthreshold drain-source currents demand that static power be brought under control, while dynamic power budgets are pushed to their limits by performance requirements. Traditional power mitigation techniques take a brute-force approach by changing transistor design or reducing system-wide voltage. Because processors have become more complex and application complexity will continue to grow for the foreseeable future, new, more sophisticated, power management mechanisms are needed to reign in both static and dynamic power.

In the embedded space, processors were traditionally small and specialized, but for consumer devices, increases in application breadth and complexity have necessitated multi-issue architectures, supplied by correspondingly large amounts of on-chip, multiported, data storage. Relative to logic, the proportion of power consumed for data storage is of increasing concern. Currently, up to 50% of high-end embedded device power is dissipated in on-chip data storage [1]. Many previous works have focused on

embedded cache power consumption [1], [2], but as indicated in Table 1.1, a large number of embedded processors have significant amounts of on-chip, noncache SRAM, which may consume a large amount of data storage power.[1] Some of these product SRAMs are partitioned or banked, but the sheer size of provided on-chip noncache storage still makes access cost quite high relative to data cache access.

The fourth column of Table 1.1 shows the dynamic power cost of access to each product's noncache SRAM relative to the cost of accessing its data cache. If an equal number of accesses occur to SRAM and data cache, this represents the fraction of dynamic on-chip data power consumed in noncache SRAM. In an SRAM, static power is consumed proportional to the number of cells in the storage array. If designed with similar transistor sizing, i.e., having the same leakage per data storage cell, the cache and SRAM sizes for each processor result in the relative data array static power consumption shown in the last column of Table 1.1. The consistently high cost of noncache SRAM clearly shows that there is need for consideration of noncache power in this design space.

As will be discussed further in the following section, unique characteristics of the embedded domain must be taken into account for designs in this space. Ideally, a low-power embedded memory system would maximize power reduction by providing differentiated service according to data access needs (giving fast access to critical loads and using low-power storage for less important data); use software control for predictability and to

[1]Data cache and noncache SRAM sizes are derived from processor datasheets, and dynamic power estimates calculated using XCACTI [3]. Array banking, cache associativity, and number of access ports are taken into account in the relative dynamic power calculations, and a 0.13-μm technology is assumed.

Table 1.1 Embedded/DSP processors: dynamic data cache versus on-chip SRAM power.

Processor	D-Cache Size	SRAM Size	Relative Power Cost of SRAM Access	SRAM Static Power Contribution
Atmel AT91RM3400 (ARM)	0	96 kB	100%	100%
Atmel AT91RM9200 (ARM)	16 kB	16 kB	11.9%	50%
Atmel AT91FR40162 (ARM)	0	256 kB	100%	100%
Blackfin ADSP-BF533	32 kB	36 kB	55.7%	53%
Blackfin ADSP-BF533	0	68 kB	100%	100%
Hitachi SH7750S (SH4)	16 kB	0	0%	0%
Hitachi SH7750S (SH4)	8 kB	8 kB	44.0%	50%
Infineon TC1130	4 kB	92 kB	83.4%	96%
Motorola MCF5216	2 kB	64 kB	92.5%	97%
Motorola MPC8540	32 kB	256 kB	57.6%	89%
Power405GPr	16 kB	4 kB	21.8%	20%
Power405GP	8 kB	4 kB	30.0%	33%
STMicro NOMADIK	16 kB	48 kB	55.2%	75%
TI 'C6701	0	64 kB	100%	100%
TI 'C6204	0	64 kB	100%	100%
Xilinx ML310 XC2VP30	0	340 kB	100%	100%

maintain applicability for real-time systems; and provide explicit control over cycle degradation, to allow optimization of system-level power. As explored in previous work, power can be saved by providing customized data storage arrays with varied port and latency properties [4], but this compromises the generality and reusability of both hardware and software. This work instead uses recent SRAM technologies to configure access latency and port counts in the field. A dual-ported, software-managed SRAM is assumed as a baseline, and the capability to increase access latency or shut-down either port in multi-line *configuration regions* is provided. Access differentiation is available to data stored in a noncache, software/compiler-managed SRAM. This combination of SRAM port, latency, and sleep control with compiler technology for automatically realizing power savings has been termed *configurable SRAM*, or *C-SRAM*.

1.1 Embedded Domain Background

Introduction of third generation (3G) telecommunication and media standards has accelerated development of hand-held wireless devices. Consumer demand has incited competition for rapid turn-around times, and application size and throughput requirements of 3G codes have challenged traditional labor-intensive methods of embedded development. These factors have made hand assembly coding prohibitively expensive and led to increased adoption of high-level language (HLL) programming. As a result, the importance of HLL analysis and compilation have dramatically increased.

While application development in HLLs (primarily C) greatly speeds time-to-market, it constitutes a fundamental shift in programmer mind-set. An assembly programmer aims to optimally map algorithms to processor resources, but a C programmer must express kernel operations in terms of functions, pointers, and data structures, often with little knowledge of how the compiler will schedule operations and allocate data on the target platform. Though use of more regular architecture styles for digital signal processors has made them better compiler targets [5], there are still many difficulties in scheduling code for DSPs. Inefficient HLL data usage and generic procedure invocations may make it difficult, or even impossible, for a compiler to automatically detect optimization opportunities and schedule for special features like circular buffers, packed vector parallelism, or nonstandard memory hierarchies and partitions. Some compilers implement these optimizations with programmer directives; however, if HLL source is obtained from

Table 1.2 Power and energy terminology and domain-specific concerns.

Term	*Source*	*Problem*	*Trend / Domain Constraint*
Static Power	Subthreshold source-drain leakage currents	Growing contribution to total power for sub-micron technologies	Management mechanisms must be simple for this domain (e.g., software management instead of hardware detection)
Dynamic Power	Switching (charge and discharge of capacitances)	Formerly largest contributor to total power	Correlated to design complexity ð added logic in future designs should be minimal
Total Power	Static Power + Dynamic Power	Tot. power should be minimized in this domain	Increases across processor and SRAM generations
Peak Power	Theoretical Max(Static Power + Dynamic Power) at some time point	Cooling and packing are designed for peak power to guarantee thermal failure prevention	Inclusion of thermal probes or other general-purpose domain mechanisms to detect crisis can be too expensive
Energy	(Total Power) * Time	Battery life	Always a fundamental design constraint

a vendor, addition of directives is difficult because the vendor's code structure and algorithms are unfamiliar. Future embedded processing enhancements need to avoid these manual development steps – mechanisms should be good compiler targets; the application programming model should not need modification; and provided resources should closely match the properties and needs of telecommunication and media applications so that usage and benefits are as intuitive as possible.

Table 1.2 provides basic definitions of power-related terms used throughout this dissertation. Static and dynamic power are concerns for microprocessor design in any domain. In the consumer device space, however, total power consumption and application execution time have a direct effect on battery life. Due to both real-time considerations and overall cost, power management designs should also be built upon relatively simple control mechanisms, adding minimal cost and uncertainty to the system. These factors drive the compiler-managed approach taken in this dissertation.

1.2 Contributions

The configurable SRAM approach provides software-managed data power savings with (1) no change in the programming model; (2) a mode for maintaining predictable performance; (3) inclusion of heap and stack data; and (4) low microarchitectural cost. We are aware of only one other technique for compiler-managed data power savings [6], and it has few of these characteristics. This dissertation describes an SRAM architecture for configuration, necessary compiler analysis components, and a prototype compiler implementation for data power savings. The compiler's ability to guide power savings is evaluated for twenty-two telecommunication and media applications, and synergy with traditional and aggressive code optimizations is explored.

A number of previous works have addressed power consumption in the memory subsystem, but many had significant limitations due to their avoidance of the difficulties associated with pointer use in high-level language codes. Compiler-managed scratchpad buffers have been used, for example, for spill code [7]; proposals have been evaluated primarily on the basis of kernels, and so do not consider programs with high-level language pointer usage; or studies achieve power savings through modifications to hardware caches [1], thus incurring unpredictable performance degradation, which is not appropriate for real-time and performance-constrained embedded environments.

1.3 Overview

Configuration of SRAM port count and access latency is described in Chapter 2. Discussion includes recent circuit-level techniques leveraged for power savings, microarchitectural methods for configuration storage, and comparison of SRAM configuration to other customized methods for data storage power savings. Automated techniques for joint operation scheduling and data placement have been integrated into an existing compiler framework, and are described in Chapter 3. Generality of the implemented algorithms; the role of interprocedural pointer analysis; and techniques for management of heap and stack data are discussed. The resulting utilization of various storage configurations, net power savings results, and trends are presented in Chapter 4. Savings across different code optimization levels are presented, and the benefits of various methods for compile-time feedback direction are evaluated.

The notion explored in this dissertation, i.e., differentiated data access servicing, originated from a detailed characterization of telecommunication and media application properties, and represents an attempt to match storage properties and power consumption to fundamental algorithmic needs. A characterization of *data intent* is presented in Chapter 5, along with possible applications of this characterization to other power and performance optimizations in the memory subsystem. An overview of related work is given in Chapter 6, and conclusions and future work are outlined in Chapter 7.

2 CONFIGURABLE SRAM DESIGN

Technology improvements have made transistors increasingly inexpensive to incorporate into a chip design, but a new obstacle has been encountered: the cost not of fabricating a feature, but of maintaining it, i.e., the power cost of additional transistors. CMOS SRAM *arrays* are composed of *rows* and *columns* of single-bit *cells*, with each cell being constructed of three, four, or most commonly, six, transistors. Relative to CPU logic, SRAM arrays are simple, regular structures. They consume, however, an increasing proportion of chip real estate (upwards of 50% [8]), making them a prime target for power reduction as transistor maintenance cost increases.

The SRAM used in this work results in a marginal increase in data storage structure size, but reduces the maintenance cost of a large software-managed array through explicit control of fine-grain power modes. These modes are provided by leveraging recent circuit-level techniques for power savings in unified SRAM arrays, including *self reverse biasing* [9], *floating bitlines* [9], [10], and V_{dd} *throttling* [11]. Each has been previously applied for hardware-controlled cache sleep, but to increase predictability of both access time and energy consumption, they are instead used here in a collaborative software-hardware approach. As specified by software, multiline regions are configured for the

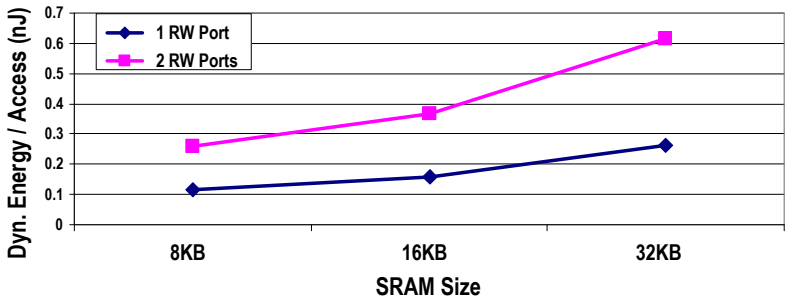


Figure 2.1 Standard methods for cutting SRAM access energy: reducing total array size and port count; values computed using CACTI 3.0 [12].

duration of program execution. Access latency is predictable, and multiple applications and application upgrades can easily be accommodated.

The chapter provides an overview of low-power, short-latency SRAM design methodology, then describes specific techniques to enable port, latency, and sleep control. A low-overhead control mechanism for multiline SRAM configuration is presented, and a conceptual view of the power trade-offs of various low-power data storage design styles is provided. This chapter concludes with an example of mapping application data to configurable SRAM.

2.1 Building Blocks: SRAM Subarrays and Macroblocks

With each processor generation, on-chip memory capacity has increased in size, necessitating novel circuit-level approaches to keep area, speed, and power within design budgets. For standard design, dynamic access cost can be decreased by reducing total SRAM size, increasing access latency, or reducing port counts. Figure 2.1 shows these

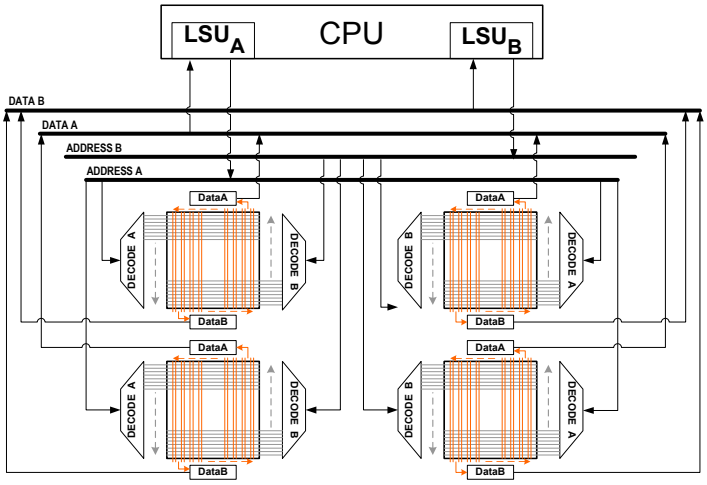


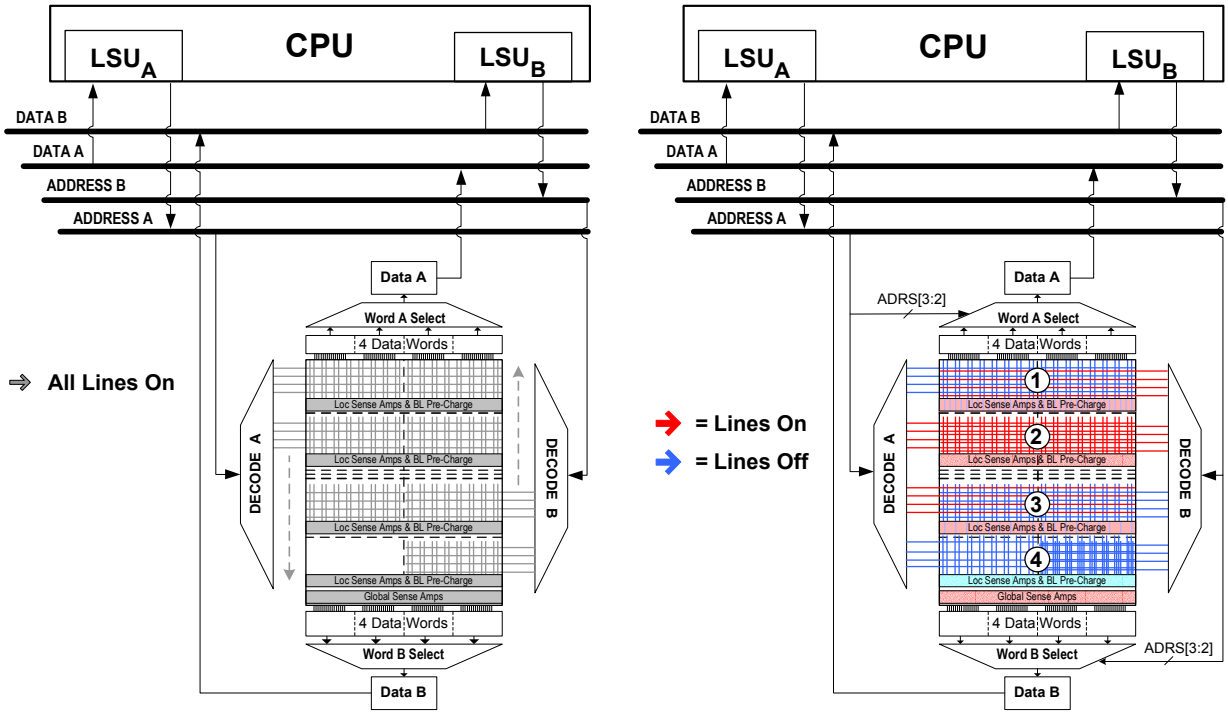
Figure 2.2 Baseline subarray SRAM architecture.

trends for size and port counts, as generated using CACTI 3.0 [12] with 0.13μm technology. Due to the strong correlation of array size with dynamic access power, ASIC (*application-specific integrated circuit*) and ASIP (*application-specific instruction processor*) designs commonly use *partitioned* SRAMs which each store one or several application variables [13]. For programmable processors this presents neither a feasible programming model, nor a scalable circuit design methodology. Instead, design complexity should be minimized by providing power savings within a uniform SRAM array design, rather than providing a collection of physically partitioned low-power memories.

Within a uniformly addressable storage space, *subarray* partitioning provides a means for favorably trading off power, area, and timing. A microarchitectural diagram of a dual-ported, noncache SRAM data array design is shown in Figure 2.2. To reduce access latency and power consumption, the wordlines and bitlines in the figure are each divided once, so there are four subarrays with separate decoders. Bitlines are shorter, reducing

the length of capacitive wires switched on each access, and wordlines are divided, reducing the number of cells activated by a given address. Subarrays have an A port and a B port, each fed with an address from a processor load-store unit (LSU). As a baseline, this work assumes an SRAM access latency of one cycle, so a new request may be issued to each of PortA and PortB once per CPU cycle. The implemented compiler technology, however, handles SRAM latencies of any number of processor cycles (see Section 3.5.2).

Subarrays alone, however, are not sufficient to tackle modern power and timing difficulties. The *macroblock* approach takes partitioning one step further without requiring replication of decode logic [14]. According to this methodology, both the bitlines and wordlines of the memory array are broken into segments, forming blocks of memory storage cells placed adjacent to one another to create an array of the desired size. This chessboardlike means of array construction allows one to view the array as easily partitionable both vertically (like traditional banking) and horizontally (into contiguous multiline regions). Figure 2.3(a) shows the conceptual subdivision of an SRAM subarray from Figure 2.2 into *macroblocks* of cells. In addition to further partitioning of bitlines and wordlines, sense amplification and bitline precharging are localized. Macroblock partitioning is designed for power and access latency reduction, but provides a logical multiline granularity for controlling the configuration of SRAM regions. *Banked*, or vertically partitioned, cache and memory structures are increasingly common in the literature and in practice [15]. However, to our knowledge, the use of horizontal partitions has not been explored for differentiated power savings (aside from hardware-managed



(a) Macroblock detail for single subarray **(b) Port-configured subarray**

Figure 2.3 SRAM ports and configurability.

sleep control of individual cache lines [16]). Layout-level implications and snapshots of the macroblock design style are beyond the scope of this work, but are available in [9].

Zhang et al. [1] indicate that for current low-power designs, caches account for up to 50% of a microprocessor's energy. While this value is realistically at the upper end of current embedded on-chip memory energy consumption, it emphasizes the importance of data power and energy in the design process. In addition to dynamic power, static (or leakage) power is of increasing concern. Previous work in the area of low-power cache design has found static leakage power to be 30% of total power in a 0.13-μm process [17]. In accordance with ITRS technology scaling estimates, Zhang et al. [1] indicate that static power may soon increase to be 50% of total power consumption. Configurable

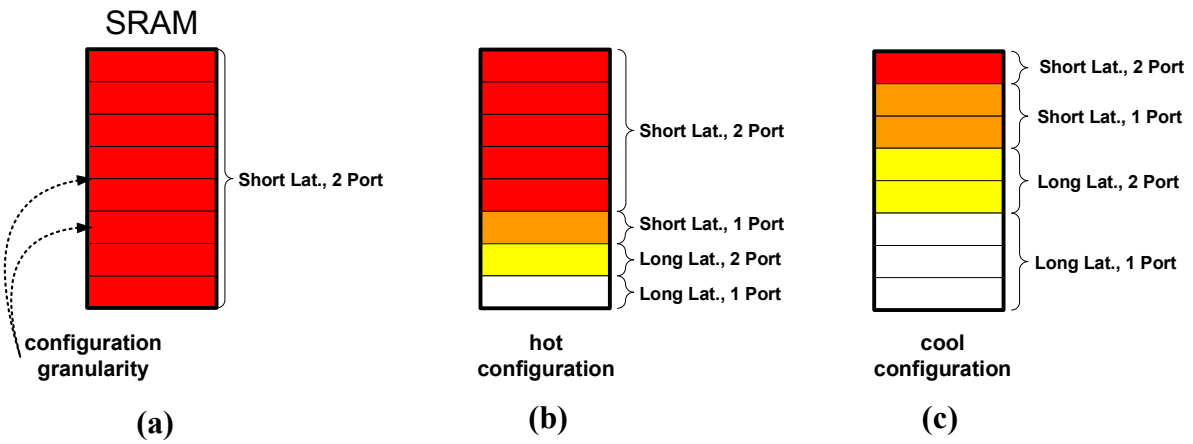


Figure 2.4 Conceptual view of *Configurable SRAM*. (a) SRAM configurable at a multiline *macroblock* [14] granularity, (b) a "hot" (high power) configuration that saves an estimated 22.8% leakage power, and (c) a cooler configuration saving 55.6% leakage power.

SRAM aims to address both these contributors to on-chip energy consumption through port and latency reduction in the field.

Figure 2.4 shows a conceptual view of configurable SRAM. Figure 2.4(a) depicts eight units of the multiline *macroblock* [14] configuration granularity. When not configured, the SRAM is dual-ported and short latency, thus not very power efficient. Figure 2.4(b) shows a configuration with five short latency, dual-ported macroblocks; one short latency single port macroblock; one long latency, dual-ported macroblock; and one long latency, single port macroblock. This results in an estimated static power savings of 22.8% over the baseline fast, dual-ported design, and the dynamic power cost of access to data in long latency regions is also reduced. However, if the compiler can exploit more code flexibility, the cooler configuration shown in Figure 2.4(c) can be used. This configuration specifies that more macroblocks be placed in each low-power configuration type, and results in

an estimated 55.6% static power savings over the baseline design. Derivation details of these power savings are given in subsequent sections.

2.2 Port Configuration

If two different versions of SRAM are built, one dual-ported and the other with a single port, there is no control needed for port "customization" and no added hardware cost. The SRAM arrays themselves are not configurable, but the compiler or programmer may choose to allocate objects to an array with multiple ports or an array with a single port. This is the approach taken in previous work [13].

A standard six-transistor SRAM cell is pictured in Figure 2.5. At the cell level, a port consists of two bitlines to read the cell's single-bit content ($BitLine$ and $\overline{BitLine}$) and a wordline for addressing the cell. Before a cell's contents may be read, both $BitLine$ and $\overline{BitLine}$ are *precharged* to the supply voltage, V_{dd}. Ninety-three percent of leakage passes through the two paths drawn: V_{dd}–to–ground and precharged–bitline–to–ground [18]. For speed, the cell access transistor, *M1*, and the nMOS pulldown, *M2*, are designed with low threshold voltage (V_t), causing high leakage currents across them. This work targets the bitline–ground path in a configurable manner.

Characterization studies we performed indicate that while overall application performance may moderately react to a drop in available memory resources, there is significant variation in the needed number of resources across an application. For `loads` not *sensitive* to the number of ports, i.e., computation is not slowed when the compiler's scheduler

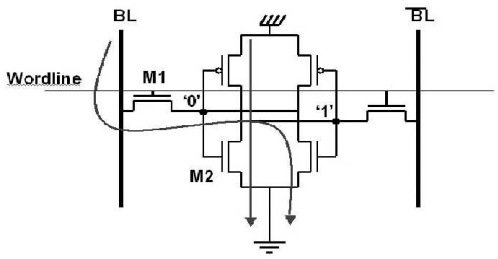


Figure 2.5 6T SRAM cell: primary leakage paths [18].

has fewer available memory ports, unneeded ports may be turned off. Essentially, if all accesses to a data object can be guaranteed to use only one of two available ports, the unneeded port can be turned off for the cells holding that object.

A simple method for turning a port off is to allow its corresponding bitlines to *float*. Floating bitlines are not precharged at each access cycle and reduce bitline–ground leakage current because the values stored in SRAM cells set the bitlines to a midrail voltage (a value somewhere between the supply V_{dd} and ground). This midrail voltage is optimal for reducing leakage current across the cell access transistors. This means, however, that these cells cannot be read, so their corresponding port is effectively "off." Floating bitlines were applied to the instruction cache and register files in [10], and in their proposal for *self reverse biasing*, Bhavnagarwala et al. [9] include a technique that allows SRAM bitlines to float.

Compiler techniques can provide guarantees as to when ports will be needed and supply information for software control of port turn-off. Because ports are explicitly disabled, configurable SRAM could go a step beyond bitline floating and also turn off the subdecoder logic for a configured port's macroblock. Available cache power estimation

tools do not accurately model decoder power, so this added power savings is not figured into C-SRAM power estimations. Because row decode delay is commonly on the critical access path [19], row decoder transistors are sized for access speed (have low V_t). Decode logic is thus prone to leakage, and is likely to benefit from explicit software control.

At the microarchitecture level, Figure 2.3(b) shows an example port configuration of a subarray as follows: Region 1–PortA off, PortB on; Region 2–both ports on; Region 3–PortA on, PortB off; Region 4–both ports off. As will be described in Chapter 3, the compiler's object placement decisions are constrained by data access characteristics and relationships among objects. Regions 1 and 3 have opposite ports open so as to allow simultaneous access to two different objects, each of which resides in a port- and power-reduced region.

Floating bitlines were applied to the instruction cache and register files in [10], but the literature lacks a control mechanism that allows their application to data storage without performance penalty. Some recent SRAMs have been designed with selective subarray precharge, which allows bitlines to float in subarrays which do not contain the currently requested data. Early decode of upper address bits is used to signal bitline precharging in a single subarray per data request. Such early decode and selective port wake-up adds to the total access time for all SRAM contents. In the general-purpose domain, this delay has been found to be inviable for level-one caches [20]. For configurable SRAM, we sought to provide a port turn-off mechanism with minimal timing penalty and granularity finer than the subarray (the macroblock). Port usage is explicitly managed by the compiler

and remains stable for the entire application run, so wake-up need not be designed into C-SRAM cycle time.

2.2.1 Overhead of port configuration

As previously indicated, Bhavnagarwala et al.'s [9] self reverse biasing (SRB) included bitline floating. Implemented in a commercial-quality design, SRB was found to have only a 3% area overhead. Because bitline floating is only a small part of the total SRB mechanism, its area cost is minimal. Heo et al. [10] appear to consider the area penalty of bitline control to be negligible and do not evaluate it. For configurable SRAM, the primary cost is not in hardware, but rather in implementation of flexible machine description and decision mechanisms so that the compiler can (a) understand the available memory resource configurations, (b) decide whether to turn ports off for a particular program variable, (c) schedule individual operations to access variables using only available ports, and (d) respect interprocedural pointer behavior which causes aliasing of program data. Solutions to these problems will be described in Chapter 3.

2.2.2 Power savings of port configuration

Studies have shown that on average, 70% of stored data bits are zero [21]. When bitlines are allowed to float, this percentage will set the bitlines to a midrail voltage indicated in [10] to reduce subthreshold leakage sufficiently to cut total static power in a dual-ported SRAM by 45%. One-half this value (22.5%) will be assumed to be the static power savings for turning off a single port, but it should be noted that additional savings

would be realized from gating a port-configured region's subdecoders. The presented results are conservative in that they do not account for savings from gating subdecoders and wordline drivers.

2.3 Latency Configuration

Typically, processor performance is highly sensitive to the latency of first-level memory. Without the ability to analyze `load` sensitivity to access latency and accordingly control data placement, there would be little reason to want to degrade SRAM/cache performance. In fact, many dynamic voltage scaling (DVS) approaches attempt to scale the V_{dd} supplied to CPU logic, not memory, during times when distant cache misses are being serviced [22], [23]. The analysis described in Chapter 3 instead allows latency to be increased in SRAM sections designated to hold data objects that are not (or are rarely) accessed on a critical schedule path. This differentiates accesses required to be fast and those that can be slowed with little or no performance impact, providing significant dynamic and static power savings.

This work assumes such differentiation can be provided using DVS and SRB. In addition to the dynamic power savings of DVS, SRB cuts leakage by 80%, and can be implemented in a programmable circuit, i.e., turned on and off [9]. Both DVS and SRB increase data access time. To simplify circuit implementation cost and configuration logic, this work assumes one latency step, so there are two possible memory operation latencies.

2.3.1 Overhead of latency configuration

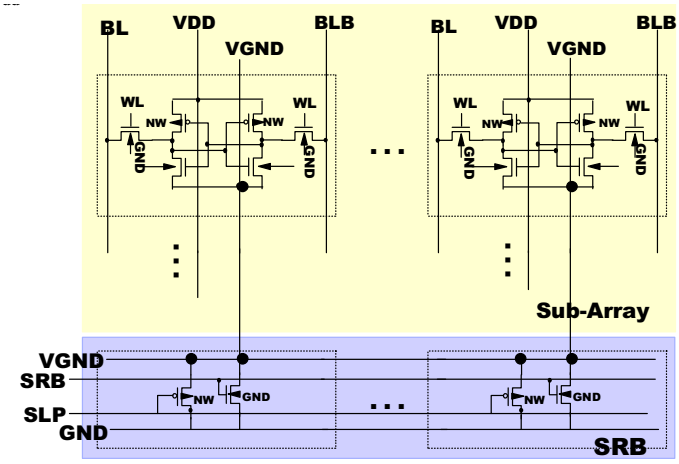
Figure 2.6 shows the circuit added to the local bitline and sense amp region of each macroblock in order to enable SRB. As mentioned, the area penalty of this circuitry, including that for bitline floating, is measured by Bhavnagarwala et al. [9] to be 3%. DVS for sleep mode is estimated in [11] to have somewhat less than 5% SRAM area overhead and have little impact on the SRAM array's power consumption [11]. With respect to fabrication cost, use of DVS will require a second metal layer. This, however, is no different than previous approaches to cache power reduction, such as the "drowsy cache," which uses a multiplexed low voltage line to create a hardware-managed low-leakage state [11].

2.3.2 Power savings of latency configuration

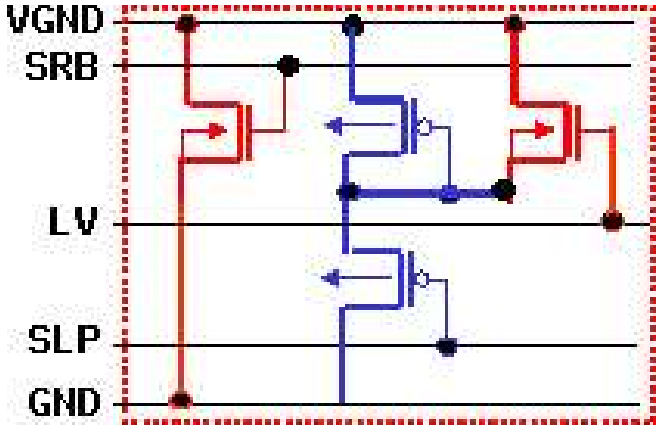
To calculate the power savings of long latency accesses, [19] provides an approximation of SRAM access time τ:

$$\tau_{access} \approx \frac{C_{BitLine}}{K'(\frac{W}{L})(V_{dd} - V_t)^2} \Delta V_{BitLine\ differential} \tag{2.1}$$

Some current commercial on-chip SRAMs provide single cycle access, including the Atmel AT91FR40162 (an ARM-based system with 256 kB SRAM) [24] and the Freescale/ Motorola MCF5216 (64 kB SRAM) [25]. A baseline single cycle access latency and a three cycle low-power latency are assumed for the majority of the results described in Chapter 4. A study of scheduling opportunities at greater latencies is also presented in



(a) Baseline cell / subarray circuitry.



(b) Added logic for programmable self reverse biasing.

Figure 2.6 Self reverse biased SRAM (SRB) [9].

Section 4.5. The TSMC[1] low-power 90-nm process technology with standard V_t transistors has a baseline $V_{dd} = 1.2$ V with nMOS and pMOS $V_t = 0.45$ V [26]. Power savings due to latency scaling from one to three cycles is computed from Equation (2.1) as follows:

$$\text{When} \quad \frac{\tau_{slow}}{\tau_{fast}} = 3, \tag{2.2}$$

$$\frac{(V_{ddShort} - V_t)^2}{(V_{ddFast} - V_t)^2} = 3, \text{ so} \tag{2.3}$$

$$V_{ddShort} = 0.88V \tag{2.4}$$

$$Power_{saved} = 1 - \frac{Power_{Short}}{Power_{Long}} \tag{2.5}$$

$$Power_{dynamic} \propto {V_{dd}}^2, \text{ so} \tag{2.6}$$

$$Power_{saved} = 1 - \frac{V_{ddShort}^2}{V_{ddLong}^2} \tag{2.7}$$

$$Power_{saved} = 1 - \frac{0.88^2}{1.2^2} = 46\% \tag{2.8}$$

If either general-purpose or high-speed TSMC process parameters are used, dynamic power reduction is greater (51% for general design and 54% for high speed technology), but this work aims to demonstrate compiler-controlled power savings on top of a low-power baseline design.

The static power reduction due to enabling self reverse biasing in long latency configurations is assumed to be 80% [9], although additional leakage savings are anticipated

[1] Taiwan Semiconductor Manufacturing Company, Ltd., *http://www.tsmc.com*.

from voltage scaling. In addition to leakage power, SRAM cell stability, i.e., the ability to maintain correct data as a bit, and surrounding bits, are read and written and disruptions are encountered from the environment, is projected to be a problem in future technologies. Stability is worsened by low V_{dd}/V_t ratios. If the parameters described above are not feasible for a given implementation and fabrication environment, there is still significant room for power savings by (a) reducing the V_{dd} drop (e.g., for $V_t = 0.45$ V, maintaining a V_{dd}/V_t ratio of 2 at V_{ddLong} still realizes a 44% drop in dynamic power); or (b) using a lower threshold voltage (e.g., as specified for the TSMC general-purpose technology $V_t = 0.45$ V). Sense amplifier design is complicated by voltage scaling, but [19] demonstrated SRAM functioning at 0.9 V–1.6 V with a novel glitch-free sense amp design. A single sense amplifier can thus be designed to handle operation at V_{ddLong} and $V_{ddShort}$.

2.4 Sleep Mode

Because both DVS and SRB are incorporated to enable port and latency configuration, sleep mode comes "for free" by voltage scaling a macroblock and turning off both ports. In current embedded processor design, noncache data and instruction memory sizing are determined based upon a summation of the needs of the anticipated application suite. When a traditional SRAM memory is sized to accommodate multiple applications, the whole memory is on, and all data is held ready, regardless of which application is

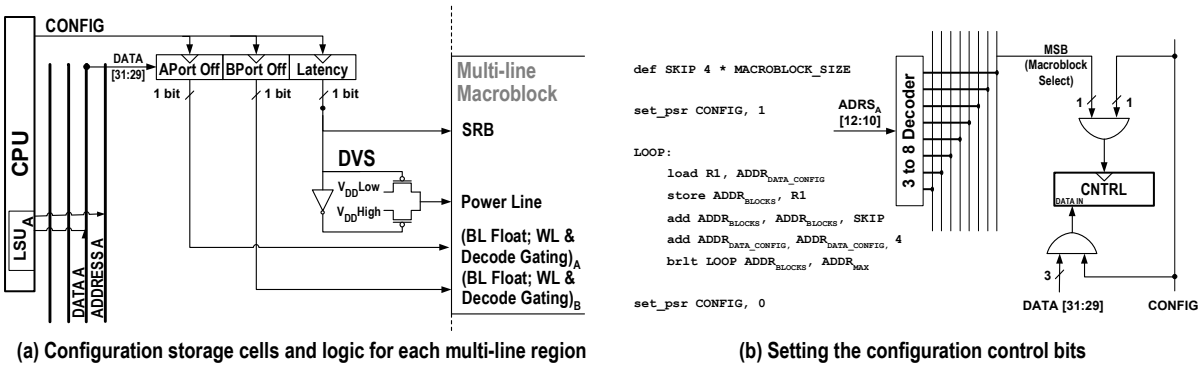


Figure 2.7 SRAM control mechanism.

executing. If a sleep configuration is provided, regions not belonging to the currently executing application can be put to sleep.

For the target application space, there are many objects for which a clear correlation exists between data usage and a particular high-level language routine. This occurs due to underlying properties of telecommunication and media algorithms, in which processing occurs in computational blocks, each of which has its own filter and transformation coefficients, state variables, and look-up tables. When translated from a mathematical representation to high-level language, this results in *ownership* of data objects by particular functions, indicating that performance is guaranteed to not be impacted if these objects are put in a low-power sleep state when their owner functions are not executing. Our experiments indicate that software sleep control based on this ownership property could realize data sleep time within 10% of hardware techniques such as the "drowsy cache" [11].

2.5 Setting and Storing Configurations

The macroblock level control logic (Figure 2.7(a)) for controlling configurability is only slightly more complicated than that proposed in [11] for sleep control alone. The proposed mechanism requires three one-bit cells per macroblock (`A Port Off`, `BPort Off`, and `Latency [Long]`) and, more importantly, only adds a single configuration line (`CONFIG`) to the SRAM array. A single-cell, per-line drowsy bit and associated logic were calculated by [11] to impact data array area by 2%; since the C-SRAM mechanism requires three bits per region (8–32 lines), its impact on overall area should be less than 2%. Also, since C-SRAM configuration is determined on a per-application basis, not dynamically at short intervals, configuration logic speed is less important for C-SRAM, and so may be designed with low-leakage (slow) transistors so as to have a negligible impact on overall static power consumption.

As shown in the pseudo code of Figure 2.7(b), to configure a region, the `CONFIG` line is set via a processor status register. A general register (`R1` in (b)) is then set to contain the configuration bits, and this value is sent to the control cells via a conventional `store` operation. Address decoding is commonly a three-stage process [11]. The stage of address decoding that selects a macroblock is used to select the target configuration cells. A 32-bit data line is assumed to configure eight multiline regions, leaving room for up to four configuration bits per macroblock, although only three are used in our current design. Each configuration store receives a different set of four bits from the `DATA` line. In the Figure 2.7(b) pseudo code, the `R1` data is received from the `DATA` bus and used to

set the configuration bits of the pictured macroblock and three subsequent macroblocks. After all regions have been configured via sequential `store` operations, the `CONFIG` bit is returned to zero.

2.6 Amortizing Configuration Cost

The time and power necessary to charge and discharge capacitances impose configuration costs on any mutable circuit component. While voltage scaling achieves significant dynamic and static power savings, its cost occurs when cells are brought back to the standard V_{dd} from a lower operating voltage. Where C_i is the internal capacitance of an SRAM cell, the energy loss from charging cells at a low voltage V_{ddL} up to a high voltage V_{ddH} is

$$E_{saved} = line\_size\_in\_bits * C_i * (V_{ddH} - V_{ddL})^2 * T_{CLK} \tag{2.9}$$

The energy dissipated in waking a 128-bit sleeping voltage-scaled cache line was previously found to be equivalent to its static energy consumed during 200 cycles [11]. The 0.18μm amortization time for floating bitlines was also found to be 200 cycles in [10]. While these values do decrease for smaller technologies with lower supply voltages, considering configuration overhead is an important factor for any adaptive or configurable system.

For embedded processors, battery life (and thus energy consumption) will always be a fundamental design constraint. In this context, the danger of an automatic hardware configuration mechanism to control sleep, cache way shut-down, or other data storage

properties is that the initiation cost may not be fully amortized over the time spent in a low power configuration, or throttling mechanisms to prevent performance degradation may constrain actual power savings. Because C-SRAM configurations are set at the application level, where there are many execution cycles and hundreds of thousands, or millions, of accesses to each data object, amortization constraints are easily met. For example, processing 318 global system for mobile communications (GSM) enhanced full rate (EFR) speech decoder frames (*g724dec* benchmark – 6.26 s) entailed 3.9 million memory accesses. Similarly, processing 108 s of 128 kbits/s MP3 music (*mpg123* benchmark) required 750 million memory accesses. Additionally, because reconfiguration is amortized across many operations and many cycles, this relaxes circuit design constraints, and may allow circuit designers to provide greater power savings than those possible for short-lived configurability controlled by hardware mechanisms.

Detailed listings of the average number of run-time accesses to global and heap data are provided in Table 2.1. For most applications, objects are used hundreds of thousands of times over the course of a short input (see Table 4.1 p. 82 for definitions of the *input1* used here for evaluation). The *g724* GSM-EFR cellular codec input length, for example, is 285-frames, or 5.6 s. *mpg123* is listed in Table 2.1 as a benchmark with heap utilization, but for *input1*'s conditions, only one `malloc()` site is reached. Use of dynamic memory in this benchmark requires options such as downloading a file from an `http://` site, or use of an `mmap`-based routine for accelerating parent-child process data transfer. Realistic application implementations specialized for particular processing or usage environments

Table 2.1 Average number of run-time accesses per global and heap object; *Afc(h)* IPA; input1 from Table 4.1.

Application	**Average Number Run-Time Accesses per Global Object**	**Average Number Run-Time Accesses per Heap Object**
adpcmdec	103 412.4	—
adpcmenc	103 412.4	—
g721dec	1 451 186.7	—
g721enc	1 622 598.7	—
g724dec	82 920.4	—
g724enc	338 174.1	—
autcor00	44.5	—
conven00	4508.4	—
fbital00	2346.3	—
fft00	2513.9	—
viterb00	3004.2	—
gsmdec	19 988.5	1 958 790.5
gsmenc	74 467.3	7 564 550.5
h263dec	112 051.3	1 582 541.3
h263enc	549 235.0	11 406 784.4
mpeg2dec	102 290.5	748 000.0
mpeg2enc	133 796.8	23 981 431.5
mpeg4dec	546 696.4	8 205 706.6
jpg2Kdec	122.8	2 530 268.7
mpg123	88 577.1	0.0
jpegdec	115.5	410 945.5
jpegenc	811.1	796 292.0

Table 2.2 Summary of techniques for port and latency configuration.

Circuit Technique	**Contributed Configurability**	**Added Components**	**Power Saved**	**Area Cost**
Bitline float	Port turn-off	Precharge gating and control	22.5% leakage / port	Bitline float + SRB = 3%
Self reverse biasing	Leakage reduction when latency increased	SRB signal, nFET, pFET	80% leakage	Bitline float + SRB = 3%
DVS	Dynamic power reduction when latency increase	DVS signal, second V_{dd} line	46% dynamic power	<2%
SRB + DVS	Sleep	Components included for port and latency configuration	80% static power; with another control line, enhanced SRB can provide 90% "deep sleep" power reduction	No additional area (included in port and latency configuration)

would be implemented without heap use. The low *mpg123* heap access values are thus not of concern with regard to amortization of configuration cost. Average access counts for the Embedded Microprocessor Benchmark Consortium (EEMBC) [27] benchmarks (*autcor00–viterb00*) are low, but this is to be expected because they are kernels, and in a real processing environment, would be enclosed within a larger application. For kernels, access averages scale with input size and number of kernel iterations, so data configurations are easily amortized over the course of execution within a full application.

2.7 Comparing Configurable SRAM Power to Customized Approaches

Table 2.2 summarizes the proposed C-SRAM circuit techniques and their power and area implications. In total, area overhead is anticipated to be less than 5%. To evaluate the power benefit of C-SRAM, Figure 2.8 visually summarizes the power differences between the configurable SRAM approach and previous proposals for data power savings. The front of each power "bucket" is divided into the principal contributors to dynamic

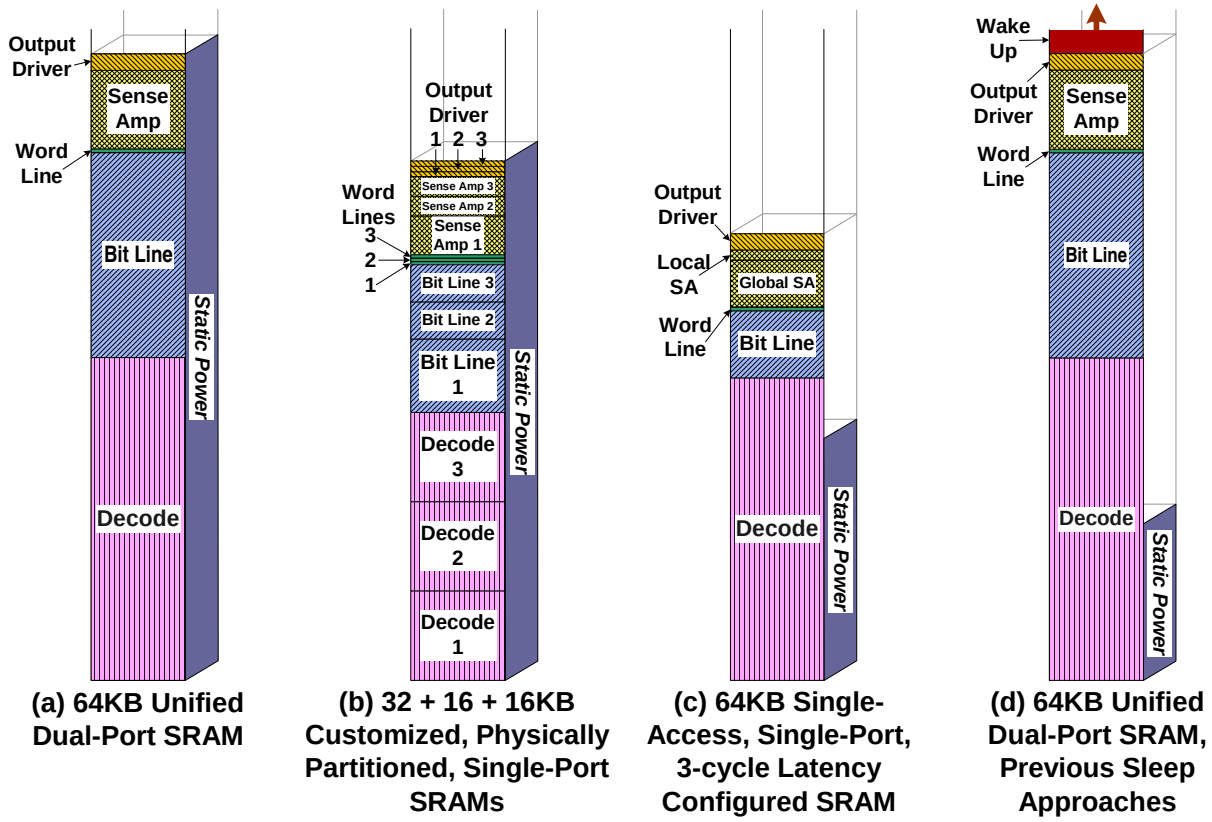


Figure 2.8 Power comparison of various SRAM approaches.

per-access SRAM power, as scaled according to results facilitated by XCACTI [3]. Static power levels are denoted on the sides of each bucket. The single-access dynamic power advantage of customized, physically partitioned memories is clear in Figure 2.8(b), where instead of a unified 64-kB SRAM (Figure 2.8(a)), three smaller memories comprise the total storage area. As shown, physical partitioning realizes significant dynamic power savings. However, such customized partitioning sacrifices hardware generality and requires complex programming, compiler, or operating system models, and is therefore not a feasible solution for programmable embedded systems. Figure 2.8(c) shows the dynamic and static power implications of a configurable SRAM array customized for single-ported, long-latency access. As far as we are aware, previous approaches to realizing data storage power savings at run time have only targeted leakage, primarily through invocation of sleep modes; as is clear here, there are also meaningful dynamic savings to be had via latency configuration. While some previous hardware-controlled sleep approaches do marginally raise dynamic power consumption, Figure 2.8(d) shows that conceptually there will be negligible change in dynamic power for sleep-based power savings, while leakage alone is likely reduced on average by 75% [11].

2.8 Application Data Configuration Example

Figure 2.9 shows a mapping of the *g721dec* application to configurable SRAM. The *g721dec* call graph is shown in Figure 2.9(a), where each block contains a single subroutine and the data objects it accesses from global data space. The mapping of these

data objects to configurable SRAM without performance degradation is pictured in Figure 2.9(b). The designation $nP - mC$ indicates the number (n) of ports used in each region, and the number (m) of Cycles needed for a `load` access. Function stack space (local variables) is left in the generic SRAM region with two ports and 1 cycle `load` latency.

The first configured segment is occupied by `power2`, a math look-up table used by the `quan` routine. Like many look-up tables in this application space, `power2` is not interactive with other objects, but is conjoined to several arrays. Only one `load` operation accesses `power2`, but this operation has little slack, so `power2` appears in a 1P–1C region—port count is customized, but access latency remains low.

The integers `in_buffer` and `in_bits` are used for application input/output to and from a file. Figure 2.10(a) shows scheduled code from the `unpack_input` function which determines the SRAM configuration needed for `in_buffer`. The machine description for this schedule reflects the default SRAM configuration–two memory ports and a one-cycle `load` latency. Dependences between operations in this schedule are drawn with black lines.

This is a control-code type region in which nonparallel bit manipulations are performed, so the dependence height (highlighted in gray) is four cycles, but overall slot utilization is low. What remains to be examined is the *latency sensitivity* of the loads to `in_buffer` and `in_bits`. In Figure 2.10(a), operation `ld0` accesses stack space (a 2P–1C region), so the `load` to `in_buffer` could be moved up into Cycle 1. However, though it

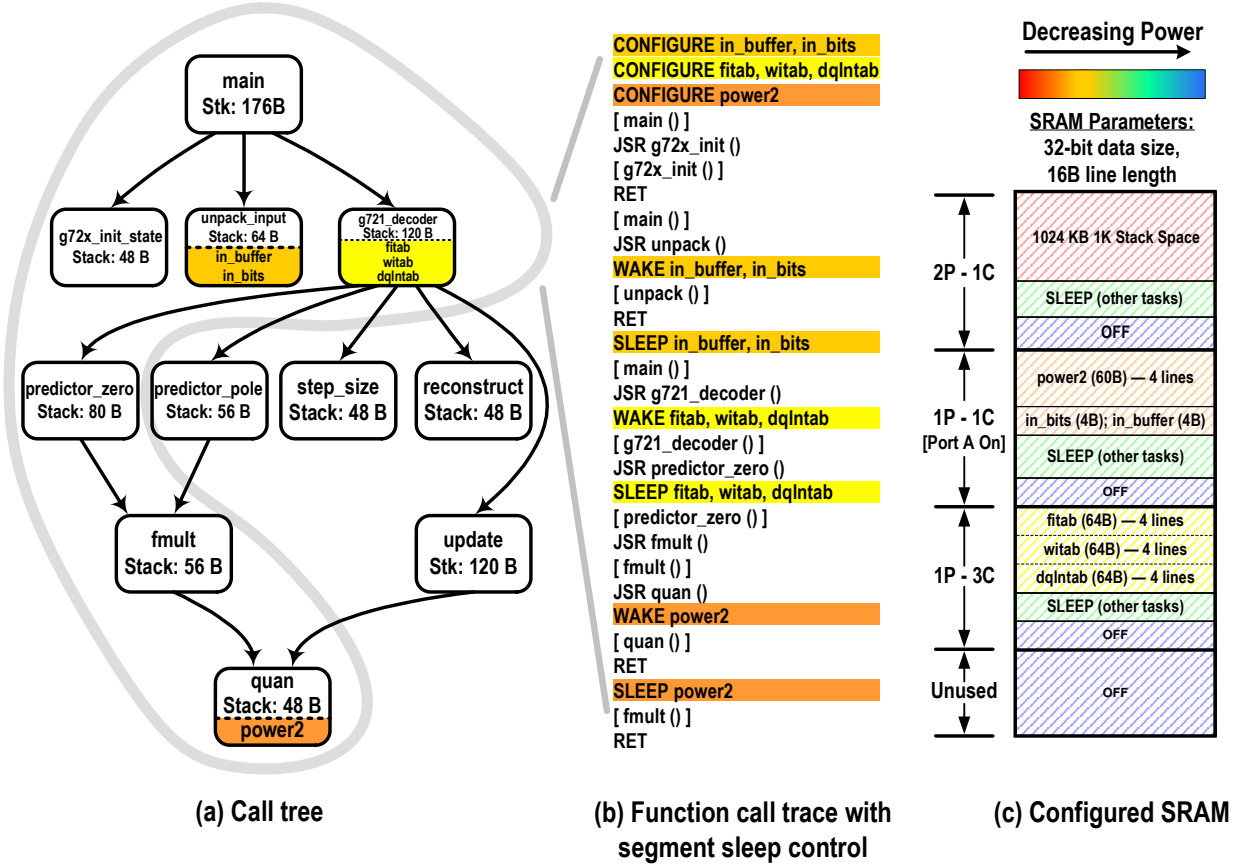


Figure 2.9 Mapping the *g721dec* application to configurable SRAM.

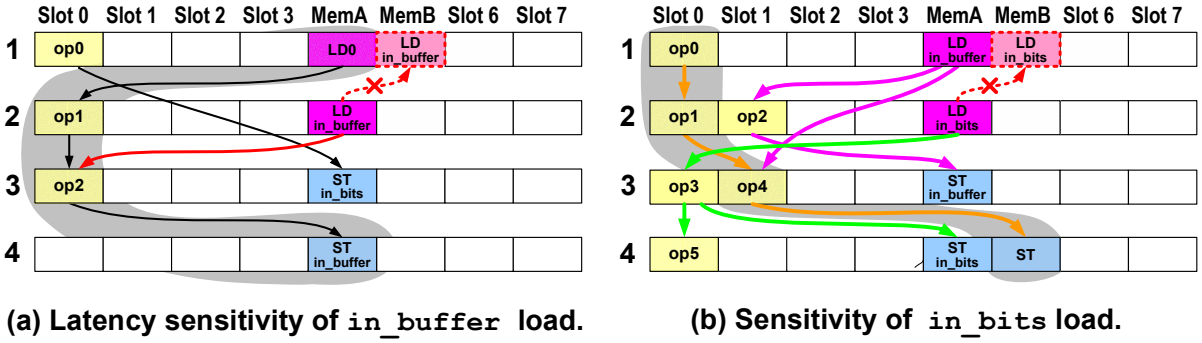


Figure 2.10 *g721dec*: determining 1P–1C storage for *IO* data `in_buffer` and `in_bits`.

is not directly on the critical path (in gray), the `load` from `in_buffer` is very close to it (via the dependence to `op2`), so increasing its latency to three would increase schedule height. This operation is thus *sensitive* to latency, i.e., forces an increase in schedule height, above a `load` latency of two.

The sensitivity of `in_bits` can be seen through the code in Figure 2.10(b). The `load` from `in_bits` is also sensitive to latencies above two, because even if moved to the first cycle, the chain of dependences following it (shown in green) would cause overall schedule height to increase. This has an interesting consequence: because the `loads` to `in_buffer` and `in_bits` are latency sensitive, and their data objects must thus remain in a 1C region, they are slightly removed from the critical scheduling path (highlighted in gray) and may be issued in separate cycles. This enables the same port (A) to be used to access both of them, and means that they may both be placed in the same 1P–1C segment without any negative performance consequence.

The arrays `fitab`, `witab`, and `dqlntab` are all look-up tables accessed by a single function, `g721_decoder`. Figure 2.11(a) shows the only code that accesses `fitab` and `witab`. The schedule shown originates from the two C code lines:

```
1:  dqsez = sr - se + sez;
2:  update (4, y, witab[i] << 5, fitab[i], dq, sr, dqsez, state_ptr);
```

As highlighted in gray in Figure 2.11(a), the critical operation dependence chain mandates a schedule height of five cycles. However, the `load` operations to `fitab` and `witab` lie off this critical path, and the `store` operations are to stack space, i.e., to data in the

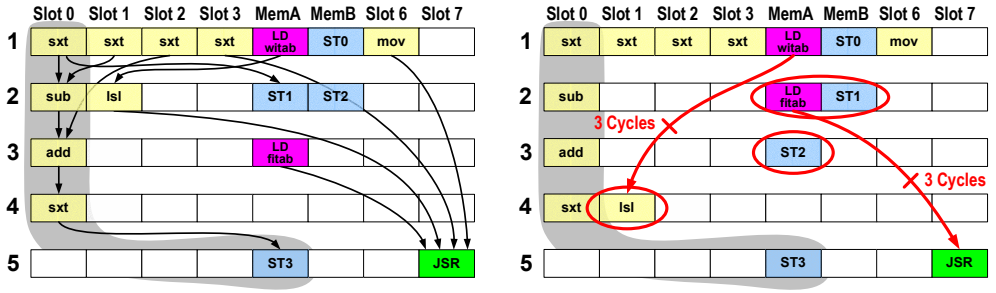


Figure 2.11 *g721dec*: accommodating 1P–3C storage for *LT* arrays `fitab` and `witab`.

2P–1C region. This allows for the schedule adjustments shown in Figure 2.11(b): the `loads` to `fitab` and `witab` use PortA in separate cycles; the `lsl` operation dependent on the `witab load` is moved two cycles later, to Cycle 4; and the `load` to `fitab` is moved to Cycle 2, thus accommodating a three-cycle latency before the `jsr` operation. These arrays are thus assigned to the 1P–3C region, in which both port availability and latency are customized for reduced power consumption.

A function call trace for the highlighted section of Figure 2.9(a) is shown in Figure 2.9(b). A sample method for sleep control has been inserted into the trace at the function granularity. This method groups and wakes objects by function ownership (exclusive use) of data objects by subroutines, relating back to the algorithmic underpinnings of this telecommunications application. Regions corresponding to each function are awoken at `jsr` operations and put to sleep when the corresponding `ret` operations are issued. It should be noted, however, that sleep control did result in one unique data layout decision in Figure 2.9(c), namely, the separation of `in_bits` and `power2` in the 1P–1C region. The array `power2` has 15 elements, and should normally only require 60 bytes of space, or 3 $\frac{3}{4}$ lines. However, `power2` is accessed by a different function than

the I/O data objects, and so for the sake of sleep control, `in_bits` and `in_buffer` would be placed in a separate region. Invoking a sleep state at the function granularity results in a reduction of total memory on-time of about 30% for a sample run of the *g721dec* application, at the cost of reduced space efficiency from alignment of same-owner data to macroblock bounds.

At the end of each configured region in Figure 2.9(b) is a space marked *Sleep*. Since the configurable SRAM space is shared among applications, multiple live datasets may be present at any given time. By turning off both ports and reducing supply voltage, the data space of the nonactive applications may easily be put into sleep mode, which preserves data while significantly reducing leakage. In addition, the SRAM array is closed with an *Off* section presumed to be in a low-power, non-data-retaining mode. This additional area represents over-design of the SRAM capacity for the purpose of accommodating additional data resulting from standards upgrades or the addition of new applications to the executable suite.

3 COMPILING FOR CONFIGURABLE SRAM

Code generation and optimization for instruction-level parallel processors have traditionally focused on improving program performance. In that context, static analyses were primarily designed to answer dependence queries which enabled code motion for increased parallelism. For future technologies, however, power concerns are surpassing performance demands, and new compiler techniques are needed to tackle the power challenge. This chapter describes use of advanced interprocedural analysis to enable differentiated data access, and presents a prototype compilation framework for automated data power savings.

3.1 Compilation Challenges

The first challenge of compilation for environments with differentiated data access is *safety*. For this discussion, transformations and scheduling decisions made by the compiler will be considered *safe* if they result in code valid for all inputs. The more subtle implications of transformations which may degrade performance beyond a real-time bound will be addressed separately in Section 3.6.

Conventional sequential programs have unit assumed latency (UAL): each operation is assumed to have been completed before the next operation is issued. When statically determining multiple operations for a very long instruction word (VLIW) architecture, nonunit assumed latency (NUAL) is more commonly used. NUAL exposes operation latencies explicitly to the compiler, which prevents interlocking by scheduling operations only when their inputs are ready. To obtain more efficient schedules, classic multi-issue digital signal processors (DSPs) have used NUAL EQ (equals) models, in which operation latencies may vary, but are guaranteed to match a specification [28]. This restriction sacrifices code compatibility if operation latencies are reduced and also makes compiler scheduling for differentiated access more difficult. Specifically, EQ code will be unsafe unless all objects accessible from a single operation appear in storage with the same latency and all accessors of a given object are scheduled with the same latency assumptions.

The NUAL LEQ scheduling model assumes that operations may complete before their indicated latency [29], [30]. Not only does this simplify implementation of precise interrupts, but for differentiated data access, it also means that data objects accessible from the same static operation may be placed in storage with different latencies. If the operation is scheduled for long latency access and instead accesses an object in a short latency storage region, LEQ code will still be correct. The LEQ model has been adopted for the StarCore DSPs [31], and is used in this work to ensure safety while increasing the compiler's ability to realize data power savings.

The LEQ scheduling model relieves some pressure on determination of data access latency, but when hardware recovery mechanisms are not provided, a complete understanding of program memory usage is still necessary to ensure safety. In particular, if port wake-up and interlocking are not provided, an undefined value is returned from SRAM if a request accesses a floating port. This work includes the use of optimistic pointer analyses which result in an incomplete view of memory behavior, and so assumes that recovery mechanisms are present in hardware (Section 3.3). However, it also includes use of an accurate formulation of interprocedural analysis, which ensures safety and increases power savings because recovery mechanisms need not be invoked.

The second challenge of compilation for embedded power savings is *performance*. Resource conflicts should be avoided (Section 3.4.1); dynamically allocated data should be handled with minimal overhead (Section 3.5.1); and performance-oriented optimizations should be synergistic with power savings (Section 4.4). Ultimately, it is particularly important for the embedded domain that it be possible to realize power savings without sacrificing performance predictability (Section 3.6).

Lastly, balancing the cost and benefit trade-offs of various configurations would be an excessive burden on an assembly programmer, and the high-level language programmer should not be expected to manage data layout. The compiler should *automate* the process of data power management. Much previous compiler analysis and dynamic optimization work has aimed to understand memory access patterns and re-organize them to improve caching characteristics. This indirectly realizes power savings because memory traffic

more frequently hits in the upper levels of a processor's memory hierarchy. Configurable SRAM, however, does not require applications to have regular access patterns; it only requires that there be regularity in the data usage, i.e., that sections of the memory space, or data objects, be used in a definable manner. With regard to compilability, one appeal of the proposed approach is thus that it does not require complex pattern analysis.

For configurable SRAM, the compiler can automate power management by leveraging and selectively creating memory operation flexibility to exploit power-saving data storage restrictions. By integrating data placement and power control with the operation scheduler, data power savings has been achieved in a free-standing, back-end module, which does not require modification of optimizing phases of the compiler. This allows arbitrary optimization levels, and frees the HLL programmer from data power management responsibilities. The compilation path developed works for general code, so no application modifications are necessary, and fast data access is provided when needed.

3.2 Framework Overview

The compiler technology described has been implemented in IMPACT [32]. IMPACT provides an infrastructure for studying instruction-level parallelism (ILP), and includes compiler modules for analysis and optimization, as well as tools for profiling, emulation, and simulation. The IMPACT modules used for code analysis and generation are shown in Figure 3.1.

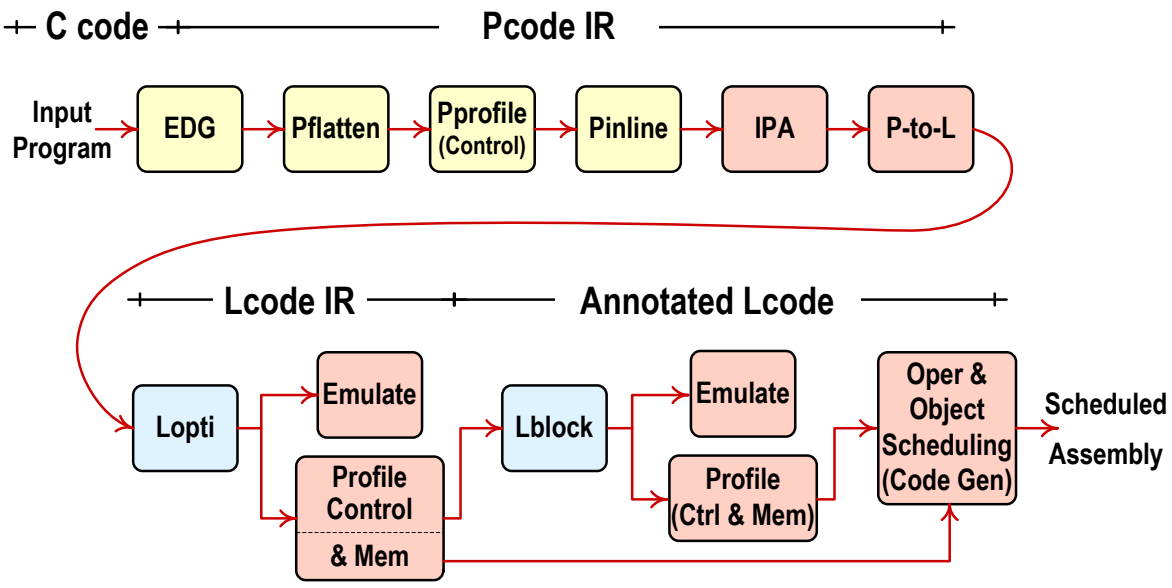


Figure 3.1 IMPACT infrastructure.

One of the goals of this work was to realize power savings without requiring programmers to augment code with pragmas or provide directives during compilation. This was achieved, and all benchmarks used for evaluation are written in standard C. The Edison Design Group (EDG) front-end is used, producing *Pcode*, IMPACT's abstract syntax tree (AST) front-end intermediate representation (IR). Pcode control flow profiling provides annotation of function and code block execution weights, used later to guide function inlining (in the *Pinline* module) and zero-weight exclusion analysis in the *IPA* module. Various formulations of interprocedural pointer analysis may be performed in the *IPA* module, as described in greater detail in Section 3.3. The AST notation is converted to IMPACT's assemblylike back-end IR, *Lcode*, in the *P-to-L* module. Classical optimizations [33] are performed by *Lopti*, and code may then follow one of two paths. Emulation

may be used to provide only a control flow profile, feeding aggressive ILP control transformations such as superblock [34] and hyperblock [35] formation in *Lblock*. If instead code is to be evaluated after only classical optimization, it is emulated and both control flow and memory profile information are collected. These profiles serve as input to the code generator, which includes operation and data placement algorithms. Modules modified and implemented for this work are shaded in red; yellow modules are used, but, for the most part, unmodified; and blue modules perform transformations which impact the compiler's ability to realize data power savings, as will be detailed in Sections 4.2 and 4.4.

Figure 3.2 provides a conceptual view of the contributions various compilation modules make toward placement of data in differentiated storage. At the top left of the figure, interprocedural pointer analysis resolves not only are pointer relationships, but assigns data objects unique identifiers, and tags objects with their size in bytes. This IPA information guides a series of performance enhancing optimizations, as well as the scheduling process. Memory profiling is optional, but when run, provides access weights for each program data object and helps to estimate the size of live heap allocated data. Finally, on the far right, the instruction scheduler uses operation slack and object use information to obtain either a schedule without anticipatable performance degradation, or a power-performance balanced schedule.

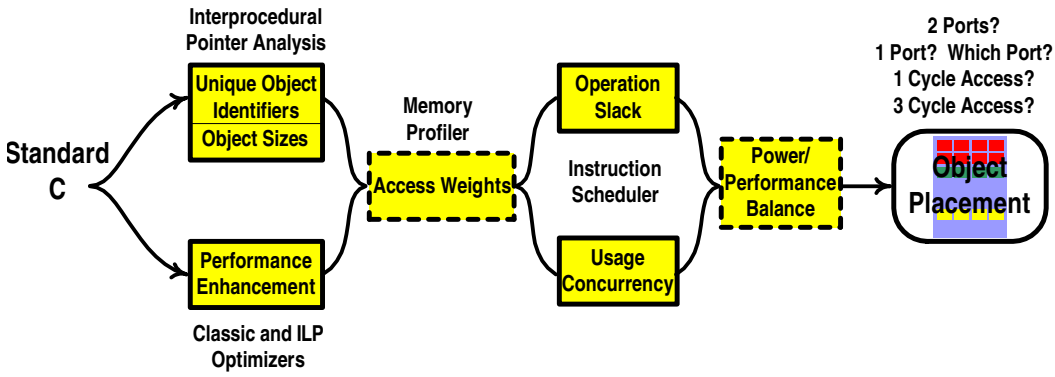


Figure 3.2 Compiler flow.

3.2.1 Studied application suite

Throughout the following chapters, reference will be made to a studied suite of telecommunication and media applications; they are listed in Table 3.1. These applications originate from an academic benchmarking suite (MediaBench [36]); the European Telecommunications Standards Institute (ETSI [37]); an industry standard suite for embedded processor evaluation, EEMBC [27]; an open-source JPEG-2000 Part1 project (JasPer [38]); and other independent authors. The size of each benchmark, measured in lines of C code, is listed in the *LOC* column in Table 3.1.

3.2.2 Terminology

The available storage configurations and accompanying compiler algorithm leverage both operation *slack* (as manifested in the ability to tolerate added `load` operation latency) and *memory slot tolerance* (as demonstrated through memory slot issue choice). By exploiting these forms of code *flexibility* present in telecommunication and media

Table 3.1 Benchmarks studied.

Benchmark	Source	LOC	Description
adpcm{*dec*|*enc*}	MediaBench	0.3K	Intel/DVI ADPCM codec
g721{*dec*|*enc*}	MediaBench	1.5K	Voice compression according to CCITT G.721 standard
g724{*dec*|*enc*}	ETSI	6K,11K	GSM 06.60 EFR speech transcoding, state-of-the-art digital cellular communication
gsm{*dec*|*enc*}	MediaBench	5K	Lossy sound compression according to the GSM 6.10 RPE-LTP standard
jpeg{*dec*|*enc*}	MediaBench	7K	Independent JPEG Group photo decoder/encoder
h263{*dec*|*enc*}	Independent	5K, 8K	H.263 video decoder/encoder, Telenor implementation
mpeg2{*dec*|*enc*}	MediaBench	9K, 7K	MPEG-2 video decoder
mpeg4dec	MoMuSys	50K	MPEG-4 simple profile video decoder
mpg123	Independent	11K	MPEG-2 Layer 3 audio decoder
jpg2Kdec	JasPer	27K	JPEG-2000 Part-1 standard (ISO/IEC 15444-1) reference decoder
autcor00	EEMBC	1K	Autocorrelation: code-excited linear predictive (CELP) filter transfer function matching
conven00	EEMBC	1K	Convolutional encoder: V.xx modem output stream encoding to enable error det/cor
fbital00	EEMBC	1K	Bit allocation: data distribution into ADSL frequency bins
fft00	EEMBC	1K	Fast Fourier Transform: 256-point complex decimation in time algorithm
viterb00	EEMBC	1K	Viterbi decoder: embedded IS-136 channel coding

applications, the compiler can derive both dynamic and static power savings with either no performance degradation or with a small *tolerance* of performance degradation, which allows higher power savings while maintaining a favorable ratio of power savings to application slowdown [39]. The compiler's ability to bound performance degradation preserves applicability to real-time applications.

Placement of program data objects into storage sections with varied properties, in conjunction with operation scheduling, is discussed in subsequent sections as the *object scheduling* problem. Object scheduling determines the number, type, and size of profitable data storage sections for a given application. It is constrained by two factors:

available schedule *flexibility* (slack + memory slot tolerance) and *tolerable performance degradation*. Slack, one component of flexibility, may be loosely defined as the number of cycles an operation can be moved earlier or later without affecting the total schedule height of the enclosing region. Load slack, previously exploited in performance-oriented scheduling techniques, is used together with memory slot issue choice to accommodate slower or restricted port accesses. If no operation flexibility were available, object scheduling would consist only of observing data usage and looking for cases where a particular constraint can be applied without penalty. For realistic applications, however, slack is present, and when an architecture provides multiple load-store units (and data ports), memory slot issue choices also become available.

In describing code properties, *object relationships* will be discussed. Objects may be (1) *conjoined* by virtue of being accessible from a common static operation (they are in the same pointer analysis points-to set), or (2) *interactive* in being accessed from within a common scheduling region, (e.g., a single basic block, loop, superblock [34], or hyperblock [35]). Objects which are neither conjoined nor interactive with any other object will be referred to as *independent* objects.

3.3 Interprocedural Pointer Analysis

Scheduling for configurable SRAM relies on interprocedural dependence analysis for safe assignment of program data objects to on-chip storage regions with little or no

performance degradation. Six different interprocedural pointer analysis (IPA) techniques are used in various combinations throughout this work:

1. *Andersen* style *(A)* [40]
2. *Steensgaard* style *(S)* [41]: If a single location points to two different objects, Andersen will continue to track the objects separately, while Steensgaard will unify them, tracking them as a single object. Steensgaard thus forces the points-to set size to one, a form of lossy alias result compression.
3. Context sensitivity *(c)* : A context-sensitive algorithm is able to keep data flow along different call paths separate. For example, should functions A and B both call C, a context-insensitive algorithm may show false data flow from A into B, by way of C.
4. Field sensitivity *(f)* : A field-sensitive algorithm is able to distinguish between fields of an aggregate object. In a field-insensitive analysis, the C-language expressions `x.f1` and `x.f2` are equivalent to `x`.
5. Heap sensitivity [42] *(h)* : Heap sensitivity enables an algorithm to distinguish between different heap allocated objects even though allocated by the same call to `malloc()`.
6. Zero-weight path exclusion *(z)*: All zero-weight expressions are excluded from the pointer analysis. While not valid for general dependence analysis, Section 3.6 will detail its utility for placement of objects into a configurable SRAM.

Conceptually, the goal of a pointer analysis system is to describe the potential targets of each pointer. Information about pointer accesses gives the compiler a view of an application's memory activity and can be critical for tasks like register promotion, scheduling, memory data flow, debugging, and verification. It is also central to automated software management of data access properties.

While the goal of pointer analysis is straightforward, the realization of this goal is complicated by the variety of possible formulations, each with its own strengths and weaknesses with respect to *accuracy* and *scalability*. Accuracy measures how closely the derived pointer relationships match those actually realizable by the program. Scalability measures the applicability of an algorithm to a range of programs with a variety of characteristics (size is not the only consideration – a 10 million line program lacking pointer use is easy to analyze). The challenges inherent to the IPA problem, along with the usefulness of its results, have led to a large body of research [43], [44].

This work uses the interprocedural pointer analysis introduced in [45]. The algorithm is very accurate, yet has also been shown to work very efficiently for large programs having complicated pointer usage. It is an Andersen style, offset-based field-sensitive, fully context-sensitive, heap object specializing pointer analysis. Indirect calls are handled by forming an optimistic call graph and then iterating between pointer analysis and call graph updates until the solutions converge. External library calls are represented by procedure stubs that mimic the appropriate pointer behavior. For the studied application

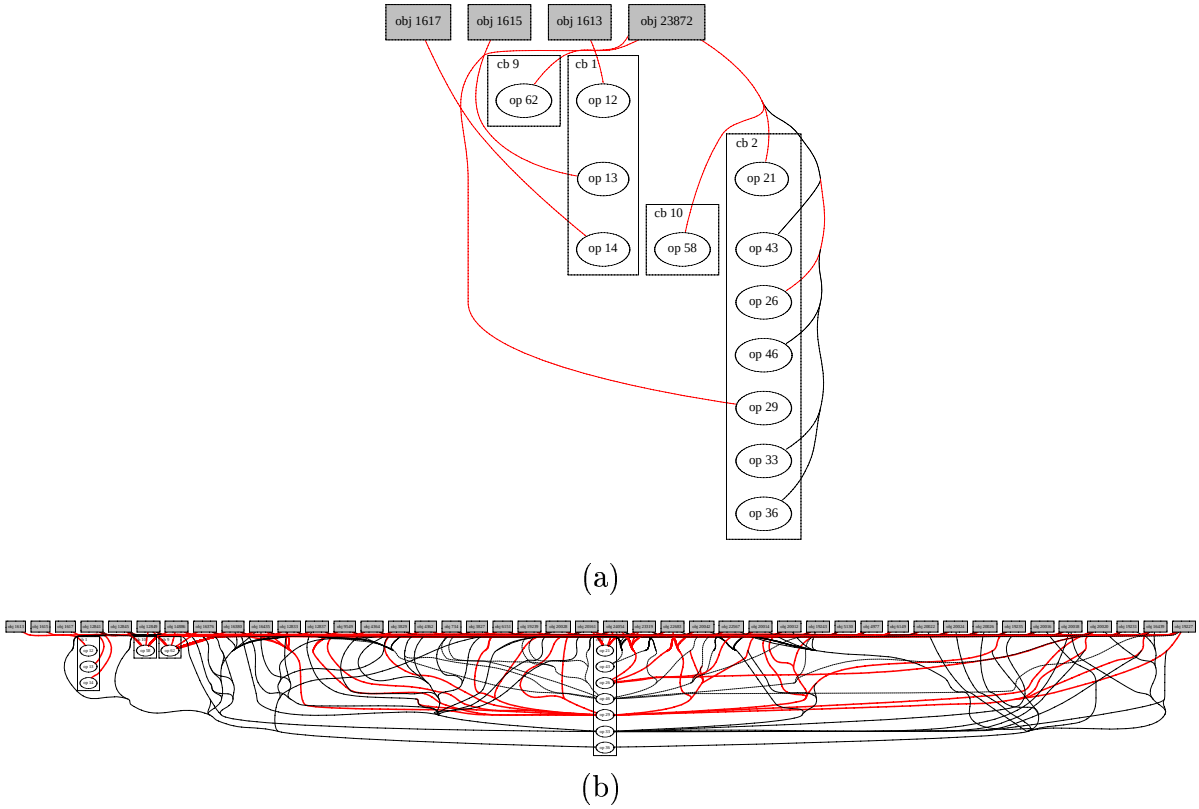


Figure 3.3 Pointer analysis accuracy for the *mpeg4dec* function `GetContextInter`. For the same memory operations, arcs represent potential object accesses, as determined by (a) an accurate *Afc(h)* formulation, and (b) a straightforward Steensgaard implementation (*S*). Note that many more objects appear to be accessed in (b).

suite, the time to perform field-sensitive, context-sensitive, heap-sensitive Andersen style analysis (*Afch*) ranged from less than 0.01 s to 1 s on a 2.8-GHz Pentium 4 system.

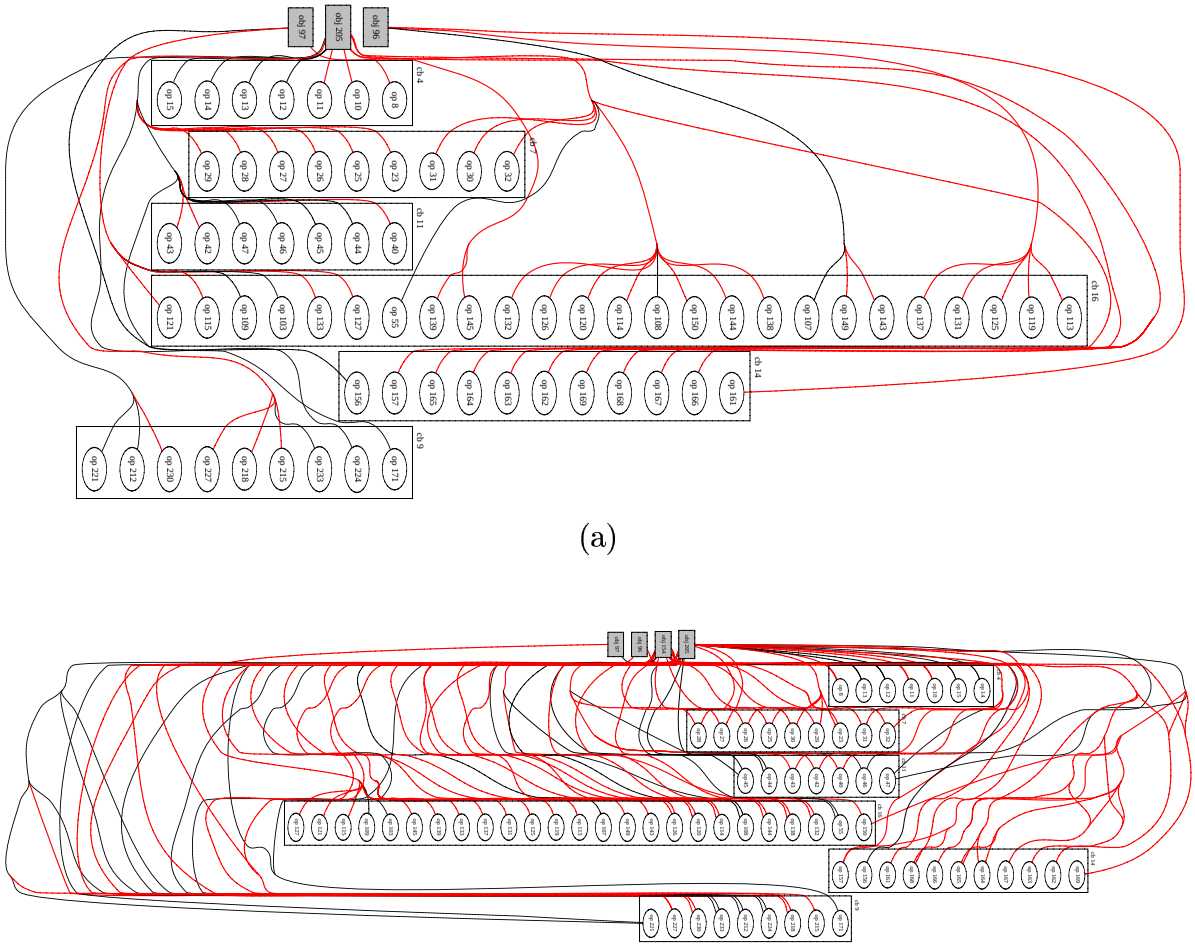
The degree of difficulty involved in obtaining accurate pointer analysis results is often highly dependent on the coding style and language features used in a program being analyzed. Across a suite of applications, the necessary pointer analysis options may vary, and in some instances the selection can make a dramatic difference in the quality of alias information. Figure 3.3 provides a graphic illustration of differences in analysis

accuracy. Memory operations in an MPEG-4 decoder function are shown as elipses, and the data objects they access appear as a horizontal row of rectangles at the top of the figure. Arcs between operations and objects indicate potential accesses, as indicated by two different pointer analysis formulations: Figure 3.3(a) shows information resulting from an accurate *Afch* analysis, while the more chaotic relationships in Figure 3.3(b) result from a Steensgaard algorithm similar to that published in [41]. It is clear that an inaccurate analysis can result in orders of magnitude increase in the number of data objects that appear to be targeted by each `load` or `store`.

Using the same representation (elliptical operations tied to rectangular objects), Figure 3.4 shows pointer analysis results for the *h263dec* `idct` function. Operation schedule slack was computed as $early\_time - late\_time$ for a scheduling run with an eight-wide, two memory slot VLIW machine model, and used to color the operation–object arcs. Red arcs represent potential access by an operation with less than two cycles of slack; solid black lines correlate with three to five cycles of slack; and dotted lines are drawn from operations with six or more slack cycles. The difference between accurate (*Afch*, Figure 3.4(a)) and inaccurate analysis (*S*, Figure 3.4(b)) is not as striking for this function as it was for `GetContextInter` in Figure 3.3. However, this `idct` function was chosen to demonstrate three important points. First, the inaccurate analysis indicates that objects 154 and 205 are *conjoined* at many operation sites (see the double object arcs coming from operations at the very bottom of Figure 3.4(b)). In source code, the two objects are unrelated, and this is clearly a false alias. Conjoined object 154 is correctly removed

by the accurate analysis (no double arcs in Figure 3.4(b)), resulting in fewer constraints during object scheduling. Second, though the path of some arcs is difficult to follow, this function also provides a demonstration of object *interactivity*. Specifically, all three objects reachable through operations in this function are interactive in extended basic block 14, i.e., the fifth row of operations in Figure 3.4(a) access objects 96, 97, and 205. Lastly, the top row of operations in Figure 3.4(a) all access program object 205, a structure holding input, output, and intermediate data for frame processing. Though all seven operations access the same object, their arcs are of three different colors, indicating increasing amounts of slack from right to left. The access properties of object 205, however, are constrained by its accessors with the least amount of slack (zero, one, and two cycles), so during object scheduling the structure will be placed in a high-power storage region (single cycle, dual-ported).

A pointer analysis framework is typically used to obtain a conservative approximation of the actual pointer behavior. For example, it will likely determine that a variable A points to a variable B when, in reality, no program input will result in such a situation. However, a conservative result will never omit a realizable relationship. This completeness is necessary to preserve the correctness of many compiler transformations. However, the new IMPACT framework optionally allows a modified pointer analysis algorithm that generates *optimistic* results; i.e., they are a subset of the realizable pointer behavior. This is done via *zero-weight path exclusion*, in which all zero-weight expressions (based on a control flow profile) are excluded from pointer analysis. It is important to note that



(a)

(b)

Figure 3.4 Pointer analysis accuracy for the *h263dec* function `idct`. For the same memory operations, arcs represent potential object accesses, as determined by (a) an accurate *Afc(h)* formulation, and (b) a straightforward Steensgaard implementation (*S*).

the pointer analysis process always excludes provably unreachable program code. Zero-weight exclusion analysis additionally eliminates code not touched during a particular execution profile.

Pointer analysis is commonly described in terms of *points-to* sets, or the locations that a given pointer may reference. This representation is shown in Figure 3.5(a), where red arcs are drawn to indicate the data objects a pointer declared as `int *p` may access. For dependence checking, such points-to sets can be converted into dependence arcs between operations. An intraprocedural operation dependence representation was previously used in IMPACT to describe pointer analysis results [46]. Data storage management, however, requires interprocedural information, and knowledge of all data objects which individual operations may access. To accomplish this, the compiler's back-end now retains a rich form of pointer analysis results through the use of *dependence objects*. A dependence object is a virtual location that maps to one (or in the case of dynamically allocated data, many) real, disjoint data objects in storage when the application is run. For this data power management work, such an object may be an array; a scalar; a structure; or a union, and may reside in local, global, or heap space. As pictured in Figure 3.5(b), static `load` and `store` operations are marked with a list of all dependence objects that they may access. In the compiler's intermediate representation (*Lcode*), these dependence object arcs also include the accessed offset into the object and size of the access. Accurate pointer analysis (Figure 3.5(c)) will result in fewer operation–object arcs, and an optimistic analysis (Figure 3.5(d)), like zero-weight path exclusion, will further reduce the number

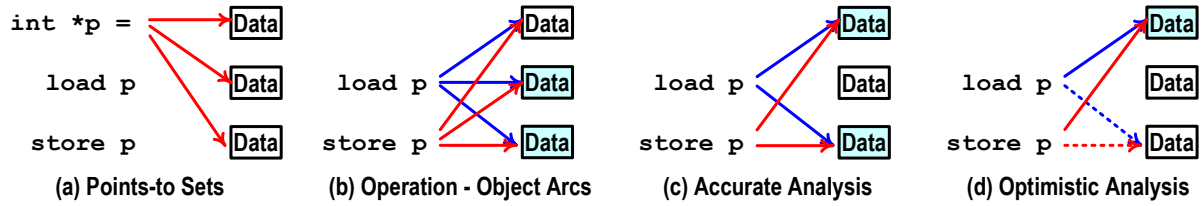


Figure 3.5 Pointer analysis representations.

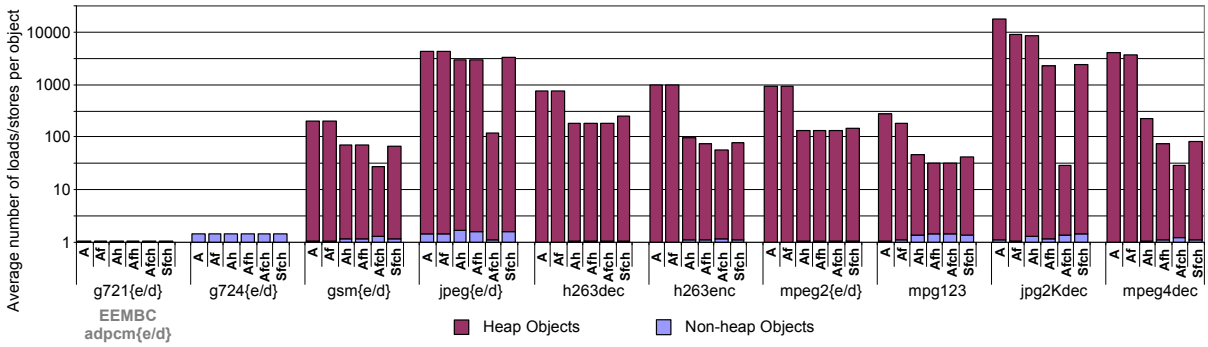


Figure 3.6 Average number of `loads` and `stores` accessing heap and nonheap objects for six IPA combinations. A = Andersen. S = Steensgaard. f = field-sensitive. c = context-sensitive. h = heap-sensitive.

of operation–object arcs. In Figure 3.5(d), optimistic analysis has eliminated the dotted pair of arcs marked by accurate analysis, so the remaining (solid) arcs do not comprise a complete representation of program memory behavior. For a given input, however, the zero-weight exclusion analysis is correct. In fact, cross-input touched code behavior is quite similar for telecommunication and media applications [47], [48]. As will be shown in Section 4.2.4, zero-weight analysis is thus complete, or near-complete, for most inputs in this domain.

3.3.1 Pointer analysis accuracy

For a selection of the studied codes, Figure 3.6 shows a log-scale plot of the average number of `loads` and `stores` that access each dependence object after applying

various combinations of pointer analysis techniques. The metric plotted indicates the number of static operations in the compiler's back-end representation which may access each dependence object. Objects are divided into nonheap and heap categories, and six combinations of the described pointer analysis options are shown. The configuration string on the x-axis is composed of a c if context-sensitive, an f if field-sensitive, an h if heap-sensitive, an A if Andersen's style, and an S if Steensgaard's style.

Analysis was performed for all benchmarks listed in Table 3.1, but only a subset is shown due to the marked similarity of results for benchmarks with relatively trivial pointer behavior. Codec pairs with similar properties are grouped and beneath the *g721* benchmarks in Figure 3.6 are grayed names of benchmarks that behave similarly. The pictured count of static operations per object is useful in (a) indicating the resolution that can be achieved by pointer analysis, i.e., the number of false aliases eliminated when more accurate pointer analysis is used; and consequently, (b) forecasting the amount of scheduling freedom available to traditional operation optimization and scheduling or to object scheduling (as will be described in Section 3.4.1).

The benchmarks show a varying sensitivity to the quality of the pointer analysis used. Many of the selected telecommunication applications have only trivial pointer behavior; thus, any combination of analyses works equally well. This can be seen in the data for the *g721s*, *g724s*, *adpcms*, and all of the EEMBC suite. One noticeable factor is that these benchmarks also lack significant heap usage. While the analysis configuration affects the results for both heap and nonheap objects, it is generally more difficult to obtain

quality information about heap object usage. The benchmarks *h263dec*, the *mpeg2s*, and *mpg123* need the ability to distinguish heap objects but are otherwise insensitive to the distinguishing of contexts and fields. The remaining benchmarks show substantial benefit from the most accurate combination of techniques. For example, the *jpegs* and *mpeg4dec* all show at least an order of magnitude reduction in apparent accesses to an object between the best and the next best configurations.

The importance of heap sensitivity to achieving accurate, yet efficient interprocedural pointer analysis of SPEC programs with dynamic memory allocation is discussed in [42]. Traditional embedded systems did not include virtual memory management and had very simplistic operating systems. Recently, however, dynamic memory allocation for embedded systems has been discussed in the literature [49], [50] and included in a few commercial products [51]. Correspondingly, the studied application suite is clearly divisible into two parts: approximately half the applications (telecommunication applications) do not use heap-allocated variables, while others, in particular video codecs (e.g., the *mpegs* and *h263s*), rely relatively heavily on dynamically allocated space. For these applications, *heap sensitivity* is essential to accurate pointer analysis.

3.3.2 Optimistic interprocedural pointer analysis: function indirection

Figure 3.7 depicts a typical form of telecommunication and media application construction in C: arcs between functions represent control flow; arcs drawn with arrows represent data flow. A top-level function (`f_top`) is responsible for buffering input from

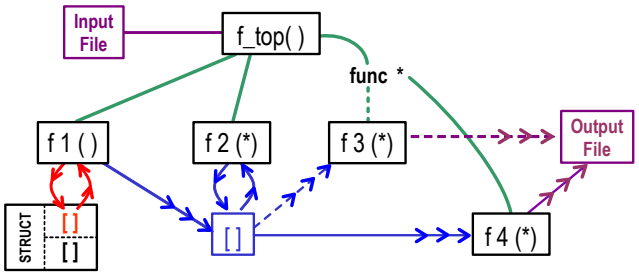


Figure 3.7 Typical telecommunication application construction.

a file and initializing and allocating variables. From this top-level routine (or routines), a series of computation kernels (`f1`, `f2`) are invoked repeatedly until all input data are processed, mimicking the block flow of an algorithm as drawn in signal processing form. Two of the functions depicted, `f2` and `f3`, are reached via an indirect function call (through the pointer `func*`). If a control flow profile indicates that `f3` is never called from `func*`, the effects of `f3` will be ignored during optimistic pointer analysis. This type of behavior occurs frequently in the studied application set (for example, in the *g721* applications, in which `*dec_routine` is used to select among the g721, g723_24, and g723_40 standards included in a single code package). When scheduling data objects for the *g721s*, the zero-weight exclusion of g723 effects is valid and will not result in an invalid object schedule, for g721 code.

3.4 General Principles for Differentiated Access

3.4.1 Object relationships

When making object placement decisions, the compiler is aware of all program memory usage, and so encounters interactions among various program data objects. Relationships among data objects can constrain object placement and make this decision process more difficult. As previously defined, the relationship between a pair of data objects is determined in two dimensions: objects may be (1) *conjoined* by virtue of being accessible from a common static operation, or (2) *interactive* in being accessed from within a single scheduling region. Placement of data not conjoined to or interactive with any other objects may be performed in isolation. Their placement in a particular configuration will not affect scheduling decisions for other objects. In contrast, if one in a set of conjoined objects is placed in a port-restricted region, all other members of the set much be accessible through that port. Similarly, if objects are interactive in a scheduling region, the placement of one into a long latency region will consume slack that might have been used to schedule another long latency `load` operation.

The accuracy of interprocedural pointer analysis can have a significant effect on the number of independent objects available. Figure 3.8 shows the number of independent objects for four formulations: inaccurate conservative *S*, accurate conservative *Afc(h)*, inaccurate optimistic *zS*, and accurate optimistic *zAfc(h)* analysis. The greatest differences in numbers of independent objects are seen between conservative and optimistic analyses. For some applications, the large differences in inaccurate and accurate accessor

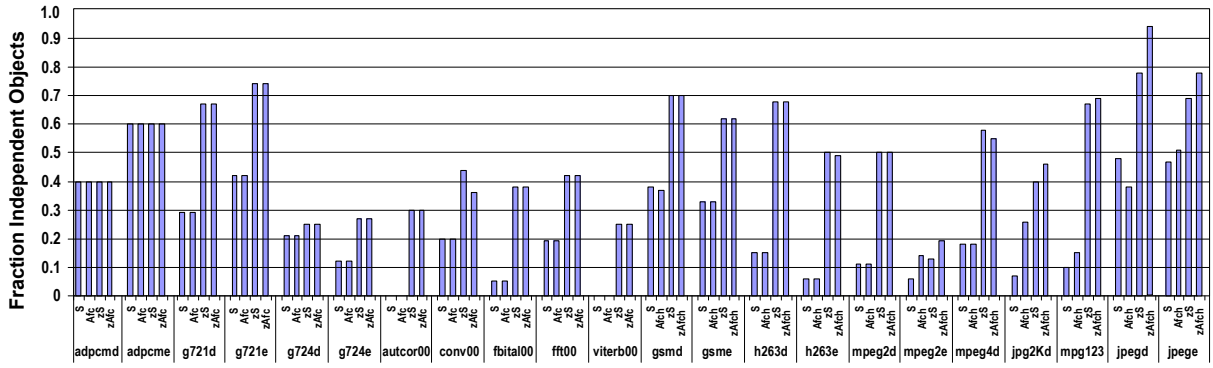


Figure 3.8 Fraction of independent objects for four IPA formulations. S = a straightforward Steensgaard implementation. $Afc(h)$ = an accurate formulation. zS = an optimistic formulation with an inaccurate foundation. $zAfc(h)$ = an optimistic formulation with an accurate foundation.

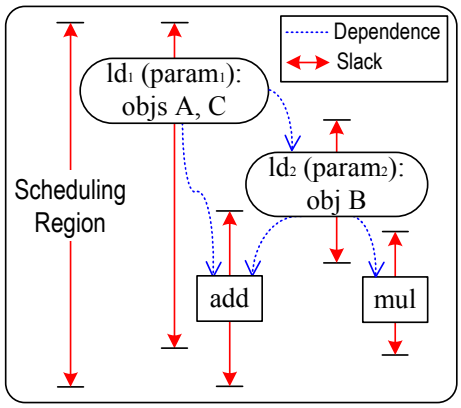


Figure 3.9 Example operation flexibility and object relationships.

counts seen in Figure 3.6 do not significantly affect object independence. Media applications, however, such as *mpeg2enc*, *jpg2Kdec*, and *mpg123*, do generally see larger numbers of independent objects for more accurate analysis, so because the more accurate *Afch* analysis *can* change the ease and correctness of object placement, its use is advisable.

In Figure 3.9, the `load` operation ld_1 accesses a location pointed to by $parameter_1$ of its function. Pointer analysis determines that the location accessed may be either data object A or C. A and C are thus *conjoined*, and object scheduling decisions must respect

the fact that they may be accessed from a common static operation. The scheduling region shown also contains an access to a third object, B, from ld_2. The scheduling decision for object B has no constraints placed on it from conjoined objects, but it is *interactive* with A and C because they are accessible from the same scheduling region, share a common dependent, the `add` operation, and B's accessor, ld_2, is dependent on ld_1. Accessor ld_1 has a significant amount of slack; assuming other accessors of A and C also have sufficient slack, objects A and C could likely be placed in a long-latency storage region to save dynamic and static power. This decision, however, delays the dependent ld_2 in the schedule, and consumes enough slack to prevent placement of B in long-latency storage.

3.4.2 Cost and benefit

Performance degradation has a direct impact on the energy consumed to execute an application because either run time is increased or cycle time must be shortened to compensate for the slowdown.[1] For some systems, neither option will be tolerable, so the implemented object scheduler's threshold for acceptable performance degradation can be set to zero, which will not change application performance. Where systemwide DVS is available, this may be the favored scheduling mode. However, if some performance degradation is tolerable, a first-order approximation of the cubic relationship between

[1] Shorter cycle times increase the power component of $Energy = Power * time$ as supply voltage is raised to speed circuit operation.

Table 3.2 Static and dynamic power savings per SRAM configuration type.

	Single-Port Long Latency (3 cycles)	Dual-Port Long Latency (3 cycles)	Single-Port Short Latency (1 cycle)
Static Power Savings (Continuous)	80%	80%	22.5%
Dynamic Power Savings (Per Access)	46%	46%	0%

power and voltage[2] indicates that a technique may be considered "power-aware" if the ratio between power savings and performance degradation is at least 3.0 [39]. This leeway to degrade performance allows the scheduler to make object placement decisions that save power at the expense of increased schedule height. In this case, *cost* and *benefit* metrics are needed to guide placement of data objects in low-power storage.

At schedule time, cost can be measured as both (a) schedule height increase and (b) anticipated increase in execution cycles according to a control flow profile. The benefit of latency and port configurations may be prioritized according to static and dynamic power savings: (1) increasing latency and restricting access to a single port saves the most power; (2) increased latency alone also saves both static and dynamic power; and (3) port restriction primarily provides static power reduction. This prioritization is derived from the power savings for each SRAM configuration, summarized in Table 3.2, and derived in Chapter 2.

[2] $Power = \alpha_{ActivityFactor} * C_{EffectiveCapacitance} * V_{dd}^2 * F_{OperatingFrequency}$. The frequency component of power is commonly approximated as being linearly proportional to voltage: $F = K_{ProportionConstant} * V_{dd}$, and if uniprocessor performance can be assumed to scale linearly with frequency, $Performance = IPC * F_{OperatingFrequency}$, making $Performance \propto Frequency \propto Voltage$. Since $Power \propto V_{dd}^3$, the first-order approximation indicates that percentage performance increase must be at least three times the power increase for a technique to be power-aware.

Configurations for individual objects should be prioritized according to the power savings realized (a) in particular configuration regions, and (b) for particular object usage patterns. Specifically, the static power differences for various configurations result in a static power cost proportional to object size. Dynamic power, on the other hand, is incurred relative to the number of `load` and `store` accesses to an object at run time. For object scheduling, dynamic power cost is anticipated using a memory profile to measure accesses for a sample input or inputs. Use of memory profile information is discussed in further detail in Section 3.6.

3.5 Prototype Implementation

To evaluate the effectiveness of C-SRAM in matching algorithmic needs and properties, a prototype C-SRAM compiler has been implemented in IMPACT. This section describes how stack and heap data were handled, the implemented greedy scheduling algorithm, the compiler's role in a multiprogramming environment, and the generality of the implemented algorithms.

3.5.1 Heap and stack data

3.5.1.1 Promotion of stack variables

In the studied application set, many local variables (stack objects) are pseudoglobal: they exist for most of the program duration and are used heavily by multiple functions

(passed by reference). To increase effectiveness of the object placement algorithm described in Section 3.5.2, the compiler first promotes arrays, structures and unions from local stack allocation to static global variable space. This enables such variables to be scheduled into a low-power region without requiring partitioning of the stack. All other stack data are excluded from the placement process and thus kept in nonconfigured (i.e., high power, dual-ported, short latency) storage.

3.5.1.2 Heap objects

Some image and video applications (e.g., JPEG and MPEG implementations) make heavy use of heap-allocated objects. For these applications, excluding heap objects from placement into low power SRAM regions would significantly limit potential data power savings. Because heap allocated objects are created dynamically, many previous approaches to software data management (e.g., for compiler-managed caching or scratch-pad placement), have not been able to handle heap data. For C-SRAM, a specialized implementation of the memory allocation procedure, such as `malloc()`, is used to force placement of heap objects into low-power SRAM regions.

The implementation chosen includes a parameter that determines the power configuration from which each `malloc()` will allocate objects. Each static `malloc()` may allocate to a different SRAM region, but all run-time data objects allocated from a single static call to `malloc()` must go to the same port and latency configuration. For applications with only a single call to `malloc()`, this constrains all heap objects to the

configuration needed by the object with the most stringent restrictions. Despite this restriction, many heap objects can be allocated into low-power regions, particularly to tolerate port restrictions.

3.5.2 Implemented algorithm

For default object scheduling, a list instruction scheduler is used on extended basic block scheduling regions. Algorithm 1 describes the steps implemented for object scheduling. Comments are enclosed in curly braces.

Prior to object scheduling, local variables are promoted, machine-independent optimizations are performed, and the IR is annotated with profile feedback. The profiler also ranks objects by their access weight and provides this information to the object scheduler. As scheduling begins, a resource representation is created from a description written in the IMPACT hmdes2 machine description language [52]. After this baseline resource availability is known, the scheduler creates a list of possible SRAM configurations, prioritized according to power savings potential.

Operation scheduling is first performed to calculate a baseline schedule for all regions (lines 8–11). This baseline assumes the fastest, but highest power, SRAM configuration: all objects may be accessed through both ports, with a single cycle `load` latency. This best-case schedule height is recorded, along with an anticipated execution time based on

Algorithm 1 Implemented scheduling steps for configurable SRAM.

Promote local variables
Perform desired level of machine-independent optimization
Profile memory access counts for all objects
Rank program data objects according to total access weight
Set baseline scheduling parameters: execution and memory resources, SRAM latency, performance tolerances ($CycleTol$ and $HeightTol$)
Set valid data configurations
Set power priority of valid configurations
for each function {Baseline run} **do**
    Create a *baseline* schedule, assuming use of all ports at the fastest available latency
    Record estimated baseline cycle time ($CurrentCycles$) and exact schedule height ($CurrentHeight$)
end for
for each object{Testing runs} **do**
    for each function **do**
        for each scheduling region **do**
            for all SRAM configurations **do**
                Mark memory opers with access restrictions of current test and previously committed decisions
                if decisions already committed for conjoined objects invalidate the current test **then**
                    Skip this test
                end if
                Schedule operations
                Record estimated cycle time ($TestCycles$) and exact schedule height ($TestHeight$)
            end for
        end for
    end for
    while no decision made {Committing global scheduling decisions} **do**
        Look at next lowest power valid SRAM configuration
        Calculate $CyclePenalty = \frac{TestCycles - CurrentCycles}{CurrentCycles}$
        Calculate $HeightPenalty = \frac{TestHeight - CurrentHeight}{CurrentHeight}$
        if $CyclePenalty \leq CycleTol$ && $HeightPenalty \leq HeightTol$ **then**
            Commit schedule
        end if
    end while
    if $CycleTol > 0$ || $HeightTol > 0$ **then**
        Update $CurrentCycles, CurrentHeight$ with post-commit values
    end if
end for
for each function{Final run} **do**
    Create a **final** schedule, respecting all object configuration decisions made during testing runs
end for

control flow profile weights. Anticipated execution time, $CurrentCycles$, is calculated as

$$CurrentCycles = \sum_{\forall Blocks} \sum_{\forall Flows} FlowWeight * FlowScheduleCycle \qquad (3.1)$$

A flow may be either a branch or fall-through path, and $FlowScheduleCycle$ is counted from the beginning of the enclosing code region, i.e., for a fall-through path, $FlowScheduleCycle$ will be the region's schedule height. Because IMPACT does not currently provide meld scheduling [53] and assumes interlocking, it must be noted that this accounting method neglects operation latencies which dangle beyond the end of a scheduling region.

Objects are greedily scheduled according to their overall profiled access counts, so as to increase potential dynamic power savings. First, the highest ranked, unplaced object is selected (line 12). For each region from which the object may be accessed, new region schedules are computed to evaluate the effects of placing the current object into a restricted SRAM configuration. Five configurations need to be considered for each object: (1) long latency and just PortA, (2) long latency and just PortB, (3) long latency and both ports, (4) short latency and just PortA, and (5) short latency and just PortB (lines 15–20). As code is scheduled for each configuration, the impact of the current test on schedule height and estimated cycle time is recorded (line 21).

As the algorithm progresses and objects are placed into configured space, some configuration options may no longer be valid, due to choices made for objects conjoined with the current one. For example, if the object currently under consideration has a conjoined

object that was previously scheduled to a *PortA-only* region, when this is discovered, tests of *Latency-and-PortB* and *PortB-only* are eliminated from the valid configurations for the current object (lines 17–18).

Once valid region schedules have been generated, statistics are summarized for all scheduling regions. Starting with the configuration that saves the greatest power, the program-wide $CyclePenalty$ (line 27) and $HeightPenalty$ (line 28) of this configuration are compared against the allowable tolerances, $CycleTol$ and $HeightTol$, respectively. If no performance degradation can be tolerated for the particular scheduling run, a strict *no schedule height increase* restriction is enforced for all object scheduling decisions, meaning that if $HeightPenalty \neq 0$, another configuration must be considered. If no schedule meets the specified tolerances, the object must remain in the default, full-power configuration. After a scheduling decision is made for an object ("committed" – line 30), $CurrentCycles$ and $CurrentHeight$ are updated and the process continues with the next highest ranked object.

The scheduling option search space decreases as the scheduler progresses, since both port and latency configuration options may be invalidated and slack consumed by previous decisions. Because objects are selected and committed sequentially, the object scheduling search space will never explode, and is simplified when greater pointer analysis accuracy reduces the total number of object access sites.

3.5.3 Algorithm bounds

General scheduling problems are NP-hard. Worst-case run time for list operation scheduling is $O(n^2)$, so an exhaustive enumeration-based approach to joint object–operation scheduling would have an unacceptable run time. To avoid this, the implemented scheduler uses the greedy algorithm described in the previous section.

The run time of object scheduling is dominated by the number of program objects (second and third columns of Table 3.3) and static memory access operations (fourth and fifth columns of Table 3.3), not just program size (third column of Table 3.1). Column 4 in Table 3.3 shows the average number of accessors per object for accurate *Afc(h)* analysis, while the much greater numbers of access sites per object in column 5 result from inaccurate S analysis. Averages for benchmarks with high numbers of access sites are generally dominated by one or two objects with many access sites. The high *jpg2Kdec* average, for example, is caused by `jpc_mqstates`, a 1560 byte array of structures, with 93 `load` sites in classically optimized *Afc(h)* code.

Figure 3.10(a) shows average data power savings for different types of objects and scheduling modes. To simplify the graph and this discussion, dynamic and static power are assumed to contribute equally to total data storage power, so the savings percentage shown is $(Static\_Power\_Saved + Dynamic\_Power\_Saved)/2$. In Chapter 4, power saved is broken into its static and dynamic components. When no performance degradation is tolerated, decisions made for independent objects are optimal because they do not consume slack or port resources other data might need. Power savings from

Table 3.3 Data object counts and accessibility; no function inlining.

Application	**Num. Global Objects,** $Afc(h)$ **IPA**	**Num. Heap Objects,** $Afc(h)$ **IPA**	**Avg. Num. Static Accessors per Object,** $Afc(h)$ **IPA**	**Avg. Num. Static Accessors per Object,** S **IPA**	**Avg. Num. Dynamic Accesses per Object** Table 4.1 *input*1
adpcmdec	5	—	2.20	2.20	103 412.40
adpcmenc	5	—	2.40	2.40	103 412.40
g721dec	21	—	5.62	5.62	1 451 186.67
g721enc	19	—	5.79	5.79	1 622 598.74
g724dec	96	—	6.55	12.31	82 920.43
g724enc	174	—	6.14	12.96	338 174.05
autcor00	20	—	3.25	3.25	44.45
conven00	20	—	3.45	4.20	4508.45
fbital00	21	—	2.52	2.52	2346.33
fft00	22	—	4.59	4.59	2513.91
viterb00	24	—	5.67	5.67	3004.21
gsmdec	137	2	7.41	13.58	47 884.90
gsmenc	137	2	7.76	13.92	182 238.29
h263dec	147	6	11.40	12.18	169 717.57
h263enc	103	28	11.55	13.70	2 869 932.58
mpeg2dec	208	4	7.86	9.84	114 473.65
mpeg2enc	158	11	11.07	203.21	1 686 009.75
mpeg4dec	267	27	12.54	277.2	1 250 074.91
jpg2Kdec	106	3	21.53	3172.77	69 759.80
mpg123	259	1	13.80	211.12	88 236.38
jpegdec	1377	2	3.04	49.70	711.38
jpegenc	1433	2	2.50	69.69	1919.76

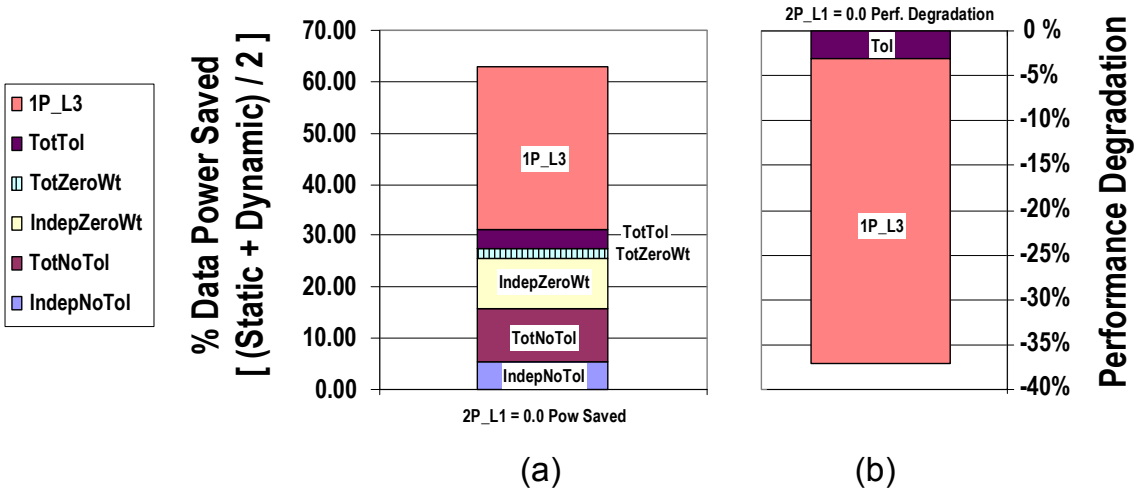


Figure 3.10 Object scheduling bounds and trade-offs.

independent object configurations thus represent an optimal lower bound on the algorithm's power savings. This savings (5.4% for classically optimized code) is shown at the bottom of Figure 3.10(a) in the *IndepNoTol* category. Scheduling decisions made for non-independent objects add an additional 10.4% to total power savings (*TotNoTol*). As previously described, optimistic zero-weight IPA significantly loosens scheduling restrictions and noticeably increases the number of independent objects. This is reflected in the much higher amount power savings attributable to independent objects (*IndepZeroWt*). When schedule degradation can be tolerated, power savings are again higher, reaching 31.1% in the *TotTol* category in Figure 3.10(a). At the upper bound of power savings is the case in which all program data is in a single-ported, long-latency configuration. The *1P_L3* category at the top of this graph reflects code scheduled for a machine with only one slot and a three cycle memory latency. In this case, maximum data power savings are assumed: 80% static and 46% dynamic, yielding 63% savings.

When memory resources are restricted, for a given level of code optimization, power savings will be inversely proportional to performance. As shown in Figure 3.10(b), scheduling programs with only a single, long latency port results in an average 34% performance degradation. Using the 3× $power : performance$ guideline, and taking overall system power consumption into account, the power-performance trade-off of this upper power savings bound is not favorable.

3.5.4 Physical constraints

Decisions made for conjoined or interactive data may put constraints on other objects. If a conjoined object is placed in a port-restricted region, all objects to which it is conjoined must be accessible through the selected port. Otherwise, at run time, the schedule may result in an attempt to access an object through an unavailable port. A data port can be woken up by precharging its bitlines when an attempted access is detected, but this causes a processor stall. Data power consumption is also impacted because the second set of bitlines must be charged. Despite the negative performance and power effects, such situations do not cause incorrect functionality. A second physical consideration is that a long latency `load` operation may block the SRAM data output driver at a later cycle than for the standard data access latency. This was not found to be a common occurrence for the studied applications and object schedules, so the implemented scheduler assumes that if necessary, the output drivers could be replicated to

handle this exception condition. This would, however, be a consideration for a commercial C-SRAM implementation.

By eliminating untouched code when analyzing pointer behavior, zero-weight pointer analysis leaves the impression that fewer objects can be accessed from program `loads` and `stores`. This generally increases the compiler's ability to place objects into low power regions. No change to the scheduling algorithm is necessary to leverage the optimistic zero-weight pointer analysis results. However, because the marked object usage is only representative of the input(s) used for profiling, the microarchitecture must ensure program correctness, while handling any unexpected accesses. Specifically, if dependences are missed and a datum needed in a single cycle at a particular `load` site is in a 3-cycle region, the processor must stall while the datum is accessed. The IMPACT scheduling model, however, assumes an EPIC-style processor which includes interlocking. Interlocking must thus already be provided to handle memory accesses with longer run-time latency than specified in the machine description, so use of optimistic IPA does not add a further constraint to this aspect of memory interface design.

While it does add cost to a design, current SRAMs are frequently pipelined so that accesses may be issued under an outstanding multi-cycle request. The implemented C-SRAM scheduler assumes that short-latency accesses may complete while a long-latency operation is outstanding. The accesses, however, are guaranteed to be to separate regions, and because macroblock partitioning entails localization of sense amplification and bitline precharge, this should be possible. Detailed evaluation of circuit-level feasibility of

this type of access pipelining is an item for future work. As necessary for the circuit implementation, this could involve implementing a scheduler resource model to avoid these short–under–long access cases. From inspection of current object schedules, however, we do not anticipate that results will be significantly affected.

3.5.5 Multiprogramming and caching

When considering multiprogramming and caching, three factors relate to C-SRAM: (1) whether an object is currently in SRAM storage; (2) whether an object may ever be in SRAM; and (3) whether virtual memory is used in the system.

In embedded multiprogramming environments, programs commonly share data storage, and a processor is designed or chosen based upon the total needs of the anticipated application suite. This model is necessary to ensure predictable data access and is used for real-time systems with relatively small data footprints. If all data, whether from multiple programs or a single application, will not fit in the available SRAM space, *dynamic* scratchpad allocation algorithms can be used to swap data in and out of SRAM. Such algorithms require that the compiler use cost and benefit functions to decide which data should be in SRAM and then accordingly initiate transfers. For a given input profile, the literature provides an optimal dynamic scratchpad allocation algorithm [54]. The implemented compiler provides automated power management of data once it is in C-SRAM, so is complementary to previous scratchpad work. If the necessary size of regions with

each C-SRAM configuration changes as an application executes, a dynamic allocation cost function should be easily extended to include reconfiguration cost.

Again, for the sake of predictability, and to store data not used in a straight-forward LRU fashion, embedded processors which do have a hardware cache generally also have a sizable software-managed SRAM. Product examples include Infineon's TC1130, with 4 kB hardware-managed cache and 92 kB software-managed SRAM, or the Freescale/ Motorola MPC8540, with a 32 kB L1 hardware-managed cache and 256 kB additional SRAM which may be managed by software. The standard programming methodology for these mixed systems requires that data objects be specified as *cacheable* or *uncacheable*. Data selected as cacheable can be easily removed from the object configuration process, and our methodology will still facilitate power savings for data kept in noncache SRAM.

Operating system overhead and loss of predictability also preclude virtual memory management from use in most current embedded systems. However, as previously mentioned, page-based memory systems have been discussed in the literature and included in some recent products. Configurable SRAM easily supports a page-based memory system. C-SRAM configurations can be changed at run time, so for either software-swapped scratchpad or virtual memory managed pages, regions can be configured according to the compiler's specification as it is brought into SRAM.

Pages should be sized to a multiple of the macroblock granularity and placed at an aligned macroblock bound. This ensures that the number of available configuration regions per page is known *a priori*. Configuration bits can then be stored in a page header,

specifying the power states assumed by the compiler when scheduling the enclosed data. To minimize page swap time, pages should be brought into configurable SRAM in high-power mode, i.e., with short latency access available to all architected ports. The less than or equals (LEQ) scheduling model ensures that code scheduled for long memory latency will still be correct at the higher SRAM speed. As time and resources allow, the power state of each macroblock may be configured according to the page header's specification.

3.5.6 General applicability

The implemented framework is parameterizable as follows:

- **Arbitrary machine model:** this entails not only changes in instruction issue width (number of operations per instruction word), but also the available combination of resources and operation latencies.
- **Dimensions of latency configuration:** There may be an arbitrary number of available latencies, e.g., 1, 3, and 10 cycles, thus adding additional dimensions to the configuration space.
- **SRAM access latency:** either the base (high-power) or configured (low-power) latencies may be changed.
- **Port restrictions:** access may be restricted to an arbitrary memory slot or combination of slots.

- **Performance degradation:** the compiler can create arbitrarily large amounts of slack. In addition to use for creating power–performance balanced schedules, this could be used to achieve greater power savings in systems with deadlines, but without processor DVS. If it is known, for example, that an application has a large amount of slack before its deadline for processing a frame, and system-level DVS is not available to compensate for this, the compiler could be allowed to consume available execution time (up until the deadline), thus increasing power savings within the application deadline. If a system had sufficient instruction storage, the compiler could create schedules of increasing length and power savings, with the executed code and data configuration being chosen at run time according to system and input properties and deadline lengths.

- **Stack and heap data:** local variable promotion and object scheduling of heap objects are both easily disabled, if not appropriate for the target environment.

In addition to the compiler's ability to support various dimensions and properties of SRAM configuration, the memory operation flexibility exploited for configurable SRAM could also find application for other technologies. Memory technologies currently being adopted, such as on-chip *embedded DRAM* (eDRAM), add new dimensions to the current cache/SRAM/off-chip DRAM hierarchy and require a similar decision process, which differentiates data objects based upon access properties and performance goals. These decisions are currently made by hand, and joint eDRAM/SRAM systems could benefit from automated data partitioning.

Another related partitioned memory system consists of separate X and Y SRAMS, and is used in classical DSPs. These structures are intended as the source of two separate datastreams, and the canonical X/Y partition places filter tap coefficients in the X memory and signal (filter input) data in the Y memory. Given the similarity of port restriction to division of data between two single-ported structures, the implemented C-SRAM algorithm was tested for X/Y data placement. It was found to be effective for filter-based telecommunication applications. For media programs, however, each application was found to have data conjoined to multiple disjoint object sets. Such conjointedness requires either (a) placing all sets of objects linked by overlapping points-to sets into the same memory, which caused causing severe imbalances in X/Y usage; or (b) duplicating the overlapping objects and issuing dual stores on each write (one `store` operation to each memory).

3.6 Profile Assistance

Table 3.4 lists the annotations provided by each profiling mechanism used in the process of scheduling for differentiated data access. Placement of the *Pprofile*, *Lemulate*, and *Lprofile* modules in the context of the IMPACT compiler framework was shown in Figure 3.1. To summarize, profile feedback is leveraged for increased power savings in the prototype C-SRAM compiler in the following three ways:

Table 3.4 Profiler outputs and uses.

Compiler Module	Gathered Information	Use
Pprofile	Function callsite weights	Function inlining (*Pinline*)
	Code block weights	Zero-weight exclusion interprocedural pointer analysis
Lemulate/Lprofile: Control Mode	Code block and branch weights	Aggressive ILP optimization (*Lblock*)
		Estimated performance degradation during joint code generation and data placement
Lemulate/Lprofile: Memory Mode	Global object access weights	Relative ranking of objects for greedy placement (Section 3.5.2) and determination of dynamic power benefit (Section 3.4.2)
	Heap object access weights and allocation properties	Relative ranking of objects for greedy placement (Section 3.5.2) and determination of dynamic power benefit (Section 3.4.2); Implementation of custom `malloc()` (Section 3.5.1)
	Stack usage characteristics	Size characterization for placement of non-promoted stack data in high-power storage (Section 3.5.1)
	Operation object access distributions (for each static operation, an input-specific histogram of access to multiple accessible objects)	Study of consistency of cross-input behavior

Control flow profiling:

1. *Zero-weight exclusion* interprocedural pointer analysis eliminates side effects in untouched code blocks during determination of program memory behavior. This IPA uses a control flow profile provided to the front-end IR by *Pprofile*.
2. Power savings can be favored at the cost of slight program slowdown. In this mode, the compiler uses a control flow profile to estimate performance changes resultant from data access decisions which increase code region schedule height.

Memory profiling:

3. Previous work in controlling on-chip data storage power has only directly addressed *static* power consumption (via sleep modes). The compiler algorithm presented in this work uses a greedy algorithm to chose configurations first for high access weight data objects, as determined by a memory access count profile. Not only is static

power saved by configuring data access properties, but this method also increases dynamic power savings potential, one of the unique features of the configurable SRAM approach.

A comprehensive back-end memory profiler, including mechanisms for tracing object access patterns, temporal use of data, and instruction use, was employed for the characterizations which helped define configurable SRAM design (see Chapter 5). Details of this profiler are available in [48] and [55]. To realize more favorable compilation time for object scheduling, a leaner memory profiler was integrated with IMPACT's control flow profiler in the *Lemulate* and *Lprofile* modules. This profiler, as indicated under the "Memory Mode" section of Table 3.4, tallies accesses to objects and records heap and stack allocations.

Profiling, and in particular memory profiling, is a relatively costly compilation step. In the embedded domain, the *deployment_count* : *compilation* ratio is very high, so adoption of a compiler technique is generally more dependent on net performance or power consumption, rather than changes in compilation time. For this reason, it has been assumed in this work that a light-weight memory profile is acceptable. If not permissible, several options are available:

1. Access counts may be estimated using control flow weights and static memory operation counts in combination with IPA "may access" information. The disadvantage of this technique is that access counts of globally conjoined objects (conjoined at all access sites) will not be distinguishable. Figure 3.11 shows an example of this

problem. Where `map[sfreq][2]` is referenced (lines 19, 20, B, and F), `mapbuf2` will always be accessed due to the assignment at line 9. However, even field-sensitive, context-sensitive, zero-weight IPA will still mark `mapbuf0` and `mapbuf1` as possibilities because standard IPA formulations do not distinguish array elements. As marked in lines D and H, memory profiler accounting distinguishes among these objects and allows an accurate ranking of `mapbuf2` as a frequently accessed, more power-sensitive, object than `mapbuf0` or `mapbuf1`. The inaccuracy of a control-flow-based memory estimation has been observed to be gross for this application suite, so this method has not been evaluated.

2. Objects can be ranked and scheduled according to their memory footprint. Because data size is proportional to leakage power consumption, this provides an object ranking based on static power. Power savings of scheduling decisions made on this basis were found to be within several percentage points of those made with the run time access ordering. Data sizes are known in the front-end of the compiler, so for zero-tolerance scheduling with an accurate analysis basis, this would allow rapid object scheduling, without either control or memory profiling.

Each form of profile assistance has been found beneficial for realizing increased power savings. The ability of the prototype compiler to schedule objects for various C-SRAM configurations, as well as the utility of profile feedback in improving power savings, are quantitatively evaluated in Chapter 4.

```
1:  static int *map[9][3];
2:  static int mapbuf0[9][152];
3:  static int mapbuf1[9][156];
4:  static int mapbuf2[9][44];

5:  function:  init_layer3 (...)  {
6:     ...
7:     map[j][0]=mapbuf0[j];
8:     map[j][1]=mapbuf1[j];
9:     map[j][2]=mapbuf2[j];
10:    ...
11: }

12: function:  III_dequantize_sample (...)  {
13:    ...
14:    if (block_type != 2) {
15:       int *m = map[sfreq][2];
16:       int mc = 0, cb = 0;
17:       ...
18:       if(!mc) {
19:          mc = *m++;
20:          cb = *m++;
21:          if(cb == 21)
22:             ...
23:       }
24:    }
25:    ...
26: }
```

(a) Pseudo-C code

```
A:  Block 168 [ExecutionWeight 6988]:
B:     load_int (r 33), (r 31 + 0)
C:           [IPA: Uses mapbuf0, mapbuf1, mapbuf2]
D:           [MemProf:  mapbuf0 Weight 0, mapbuf1 Weight 0, mapbuf2 Weight 6988]
E:     add_u (r 31), (r 31 + 8)
F:     load_int (r 30), (r 12 + 0)
G:           [IPA: Uses mapbuf0, mapbuf1, mapbuf2]
H:           [MemProf:  mapbuf0 Weight 0, mapbuf1 Weight, 0 mapbuf2 Weight 6988]
I:     add_u (r 12), (r 12 + 8)
J:     br_eq (r 31), 21, Block 170
```

(b) Profile annotated assembly (IMPACT *Lcode* of lines 19–21 in (a))

Figure 3.11 *mpg123* `III_dequantize_sample` function: unbalanced conjoined object access distribution.

4 EXPERIMENTAL OUTCOMES

Each of the proposed dimensions of configuration has been found profitable for power savings. This chapter describes our embedded processor model, and then discusses object schedules for classically optimized code. The power implications of obtained schedules are summarized, and the effects of interprocedural pointer analysis accuracy and optimism are presented. Optimism is shown to improve power savings, as does allowing slight performance degradation. The chapter concludes with a study of general sensitivity to global latency and port restrictions, and analysis of the effects of aggressive ILP optimization on data power savings.

4.1 Experimental Set-Up

4.1.1 Modeled architecture

Benchmarks used for the presented experiments are scheduled for a baseline eight-wide unified VLIW architecture, with function unit distribution similar to that of the Texas Instruments 'C64x processors [28]. Figure 4.1 shows the fixed assignment of functional units to slots. The modeled processor has eight integer ALUs, two of which can

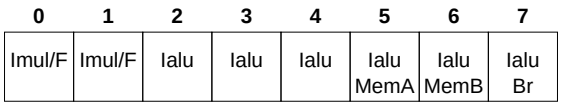


Figure 4.1 Modeled VLIW architecture issue slots.

issue integer multiplies; one branch unit; and two floating-point units. Arithmetic operations have a latency of 1 cycle; multiplies, 2 cycles; divides, 8 cycles; and floating point arithmetic, 2 cycles. By default, `loads` have a 1-cycle latency, and the processor has two load-store units. Intrinsic operations[1] are used to represent DSP instructions in the compiler, as specified in the ETSI's *g724* benchmarks. The inputs used are listed in Table 4.1.

4.1.2 Code optimization

Unless otherwise noted, control profile guided inlining is applied with an estimated touched static code size expansion ratio of up to 1.2. Classical compiler optimization is also performed on all code before operation and object scheduling, and includes constant and copy propagation, dead code removal, loop invariant code removal, and redundant load elimination. For the ILP optimized results in Section 4.4, superblock [34] and hyperblock [35] formation are performed, forming single entry, multiple exit predicated regions. Loop peeling, loop unrolling, predicate promotion, and several other ILP-enhancing optimizations are performed. Although full predication has not yet been implemented on

[1]Intrinsics are used to describe digital signal processor operations such as saturating arithmetic, normalization, and multiply-accumulation which are not found in general purpose instruction sets. They are function calls which are converted to host processor instructions for simulation.

Table 4.1 Input sets.

Applications	**Input1**	**Input2**
adpcm{*dec*|*enc*}	MediaBench default: clinton.pcm	MiBench training input: small.pcm
g721{*dec*|*enc*}	MediaBench default: clinton.pcm	MediaBench alternate input: S_16_44.pcm
g724{*dec*|*enc*}	285 frames of a synthetic harmonic signal; pitch delay varies slowly from 144 to 18.5 samples	224 frames of male speech, active speech level: -18.7dBov
gsm{*dec*|*enc*}	MediaBench default: clinton.pcm	MiBench training input: small.au
jpeg{*dec*|*enc*}	227x149 (5.6k) color image	233x174 (6.9k) color image
h263{*dec*|*enc*}	74 frames (cla7.263)	74 frames (mad13.263)
mpeg4dec	60 CIF frames (foreman.ctl)	20 CIF frames (stefan.ctl)
mpeg2{*dec*|*enc*}	4 CIF frames (I-B-B-P – mei16v2.m2v)	129 frames (1I-128P – ccm1.m2v)
mpg123	3 secs 96kbs music	108 secs 128bps music
jpg2Kdec	1024x768 color image (shed_in_field.jp2)	1024x768 color image (smokey_light.jp2)
Kernels		
autcor00	xpulsei.dat	xsinei.dat
conven00	xk5r2d.dat	xk4r2d.dat
fbital00	xtypSNRi.dat	xstepSNRi.dat
fft00	xtpulse256i.dat	xspn256i.dat
viterb00	getti.dat	togglei.dat

an embedded or DSP processor, it is assumed that a low-cost predication mechanism, such as that proposed in [56] could be used.

Unless otherwise noted, the pointer analysis used is an Andersen style, field-sensitive, context-sensitive pointer analysis, with heap sensitivity where appropriate (*Afc(h)*). During optimization, alias analysis information can be used to remove redundant memory operations. Because the impact of interprocedural pointer analysis accuracy and optimism are studied, optimization variance can result in code differences which obfuscate IPA's impact on object scheduling. To prevent this, where codes originating from multiple IPA formulations are compared against one another, optimizations requiring memory

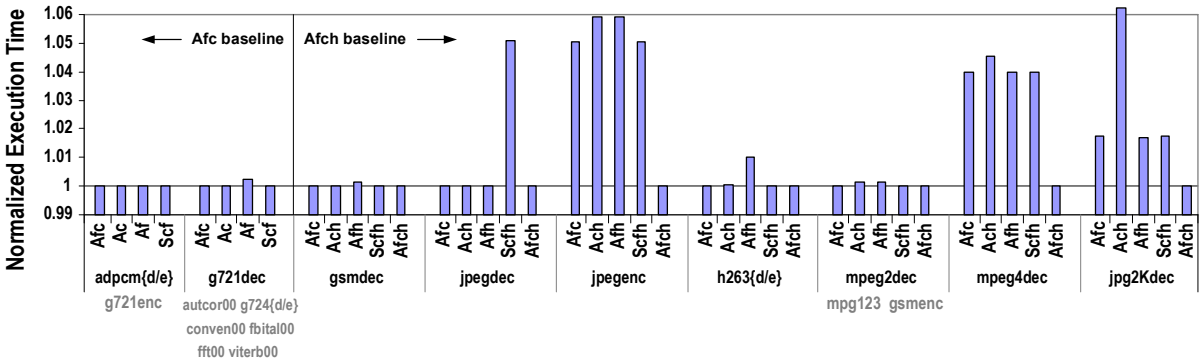


Figure 4.2 Performance changes realized by classical optimization after various IPA formations. A = Andersen. S = Steensgaard. f = field-sensitive. c = context-sensitive. h = heap-sensitive.

dependence information have been disabled. For optimistic IPA, missing memory dependences could result in incorrect code, so such optimizations are always disabled.

4.2 Classically Optimized Codes

The suite of classical transformations used for optimization closely resembles the state of current DSP compilation. This code is used here for baseline evaluation of C-SRAM efficacy.

4.2.1 Performance effects of pointer analysis

As described in conjunction with Figure 3.6, there are applications in the studied suite for which an order-of-magnitude change in the size of the points-to graph is observed when some interprocedural analysis options are enabled. Interestingly, after classical optimization, these changes have little effect on net performance. Figure 4.2 shows the relative performance of benchmarks classically optimized after several types of pointer

analysis. Results are normalized to the most accurate form of applicable analysis, *Afc* or *Afch*. Where benchmarks have similar performance characteristics, they are labeled in gray below another application.

Because around half of these benchmarks have relatively simplistic pointer behavior, pointer analysis quality does not significantly impact net performance benefit gained through classical optimization. If IPA is to be used only for classical optimization of the subset of benchmarks with simple pointer behavior, then most any algorithm will be sufficient. However, the simple behavior also means that there is no analysis cost advantage to using a less accurate algorithm. Because the C-SRAM scheduler requires pointer analysis for memory understanding and converts this to data power savings, its interaction with IPA information is different than that of performance optimizers. The following sections include results which explore the utility of accurate and optimistic IPA.

4.2.2 Schedule distributions

Exact power saved from each C-SRAM configuration will change with circuit technology and implementation. To abstract from the specific power calculations in Chapter 2, schedule distributions are first presented without consideration of their specific power implications. We first consider object schedules without performance degradation; optimistic IPA and performance-degraded schedules are evaluated in subsequent sections.

When performance degradation cannot be tolerated, the compiler is able to identify a significant number of objects to be placed in port-restricted configurations, and a

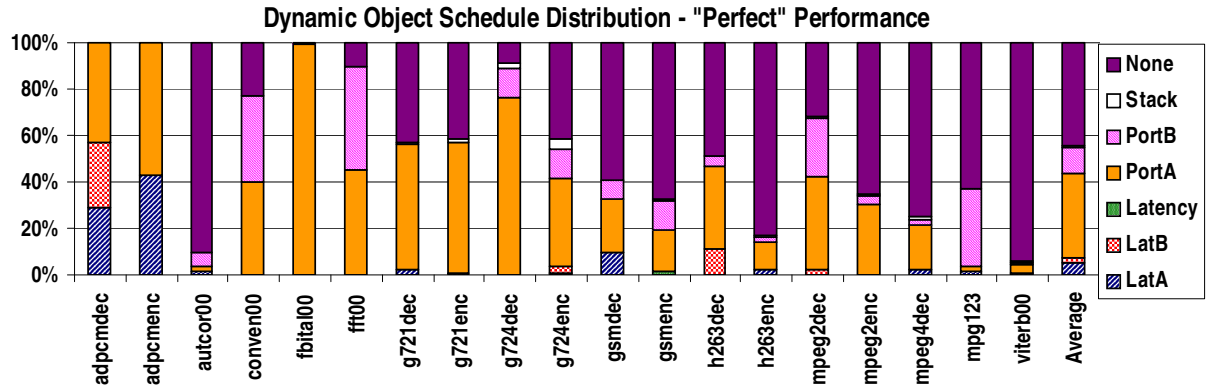


Figure 4.3 No tolerance, classically optimized schedules: percentage run-time accesses to various configurations.

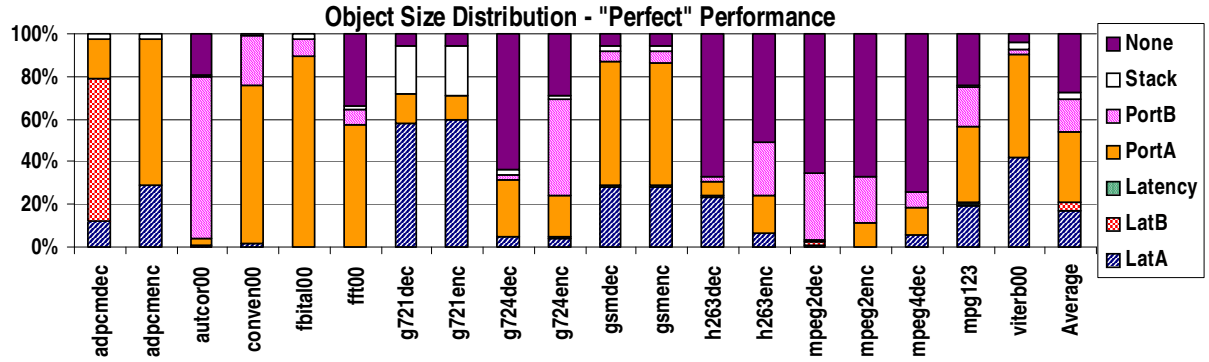


Figure 4.4 No tolerance, classically optimized schedules: percentage total data bytes in various configurations.

moderate number for placement in long-latency regions. Figure 4.3 shows the percentage of run-time accesses which hit in each configuration, and the byte-wise data footprint distribution is given in Figure 4.4. *LatA* and *LatB* indicate that data are in a long-latency region with only *PortA* or *PortB* active; *Latency* denotes a dual-ported, slow region; *PortA* and *PortB* specify full-speed, single-ported regions; and *Stack* or *None* accesses and data bytes go to dual-ported, short-latency SRAM.

Two points are of particular interest here:

1. Despite the strict zero performance degradation restriction, the compiler does find use for all five data configurations. In general, however, significant need does not arise for placement of objects in opposing port configurations (*LatA*, *Lat*, and *PortA* are the primary configurations used). While many of the *LatA* and *PortA* objects can also be placed in *LatB* and *PortB* regions, *PortA* was given preference when schedule height was unchanged. This was done to demonstrate that for an environment in which either (a) high performance demands, or (b) real-time deadlines force a zero tolerance bound, expense could be saved by only providing turn-off for one port, not both.

2. Configuration utilization varies noticeably when viewed from the perspective of run-time access counts (Figure 4.3) or object footprints (Figure 4.4). Both are important: access count distributions relate to dynamic power savings, while object size correlates with static power consumption. Vivid differences between the two distributions, such as those for the *gsms* and *g721s*, result from concentrated application access to a small number of data objects. For most benchmarks, when objects were ranked by access frequency, their relative access weights dropped exponentially by factors of approximately two. Configurations for high-weight objects thus disproportionately contribute to the access count distributions.

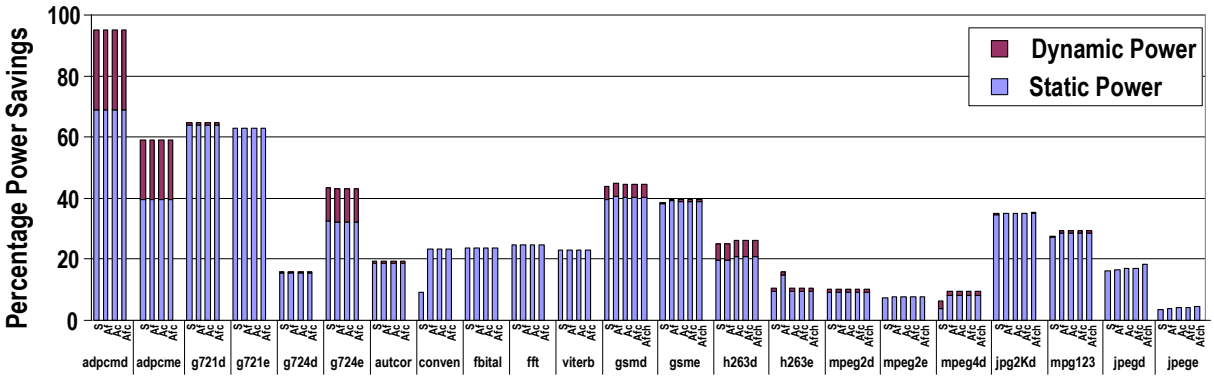


Figure 4.5 SRAM power savings for data objects allocated to configurable SRAM after various IPA formations. A = Andersen. S = Steensgaard. f = field-sensitive. c = context-sensitive. h = heap-sensitive.

4.2.3 Power savings

Figure 4.5 shows dynamic and static SRAM array power savings corresponding to application object placement decisions made after various combinations of IPA options. Reductions in data power are shown for data placement decisions made after five types of pointer analysis, indicated as c if context-sensitive, f if field-sensitive, h if heap-sensitive, A if Andersen, and S if Steensgaard. The number of IPA arcs is essentially reduced from left to right in each benchmark's cluster of results.

A value of 40% static power savings in Figure 4.5 represents a 40% decrease in static power alone. Both static and dynamic power are shown in the same column, but the two values are independent (if no SRAM data power were consumed, the column total would be 200%). On-chip data storage has been indicated to consume upwards of 50% of total processor switching power [1], and in future technologies energy dissipation from leakage (static power) may equal that of switching (dynamic power) [10]. While it is thus important to consider both components of total power consumption (static and

dynamic), their relative contributions to total SRAM and total system power will differ across technologies, so their reductions are shown independently.

Power savings should generally increase as data objects are placed in low-power configurations based on increasingly accurate, and increasingly optimistic, pointer analyses. There are several factors which level the savings seen in Figure 4.5. First, because many telecommunication applications have trivial pointer usage, and heap objects from the same malloc() site must be allocated together, IPA information useful to the object scheduler may not vary significantly across analysis types. Second, as previously stated, it was necessary to disable optimizations based on dependence information so as to have a consistent code baseline and be able to observe the effects of IPA only on object scheduling. When the optimizer is allowed to use dependence information, redundant memory operations are frequently removed, particularly when heap sensitivity is used. This alters the object scheduling problem: constraints change because the number and location of object reference sites is modified, and the relative contributions of individual objects to total dynamic power consumption shift because access sites are removed. The resulting schedules do then have power savings differences visible in IPA comparison graphs. Lastly, in some cases the scheduler is able to realize a better distribution of objects (more objects in long-latency, single-ported regions) at the cost of some of the objects originally in comparatively high power regions (single-ported). In consuming sufficient slack to place more objects in long-latency, single-ported regions, other objects may need to be

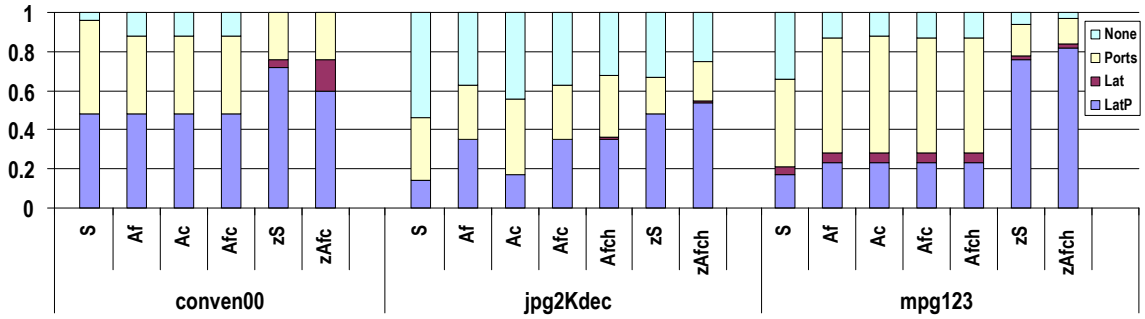


Figure 4.6 Distribution of object placement decisions.

removed from the single-ported region, thus balancing out visible power changes across IPA combinations.

With regard to this final point, the scheduling decisions made across IPA formulations can be different in important ways. For the same codes, Figure 4.6 shows several sample distributions of object placement decisions, where, by number of objects in each configuration, it is clear that IPA results have changed scheduling decisions. At the top of the *conven00* bars for Steensgaard analysis, the number of unconfigured objects is fairly small. For Andersen-style analysis, this count increases, but there was actually a net gain in power savings. With Andersen-style IPA, the number of access sites for a high access weight, large object was reduced, enabling its move from a *None* to *PortA* configuration. In the process, three other less important objects were downgraded to *None*, bringing up the *None* category count, but resulting in overall power savings.

The most noticeable cases of power reduction with increased IPA accuracy are *mpeg4dec* and *conven00*, which see static power drops of 4% and 14% due only to the accuracy of

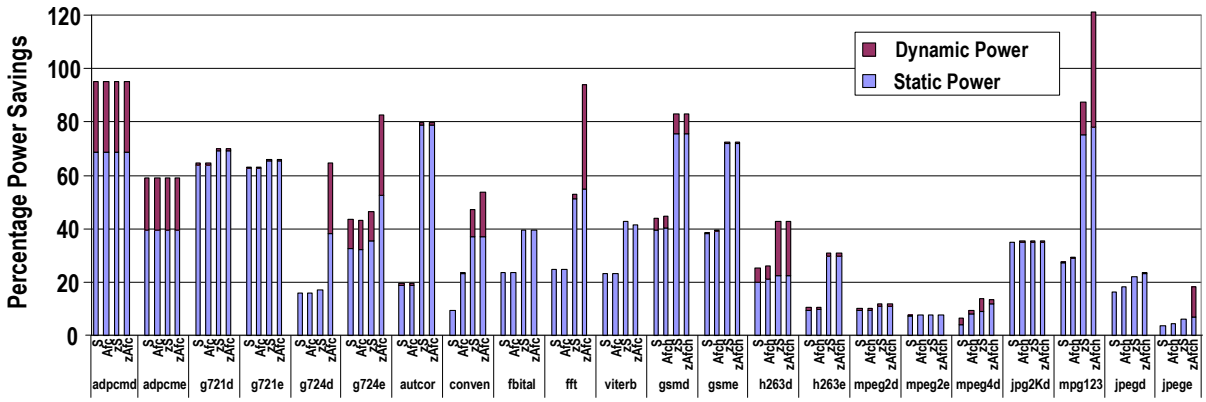


Figure 4.7 Comparing SRAM power savings for data objects scheduled following conservative and optimistic analyses. A = Andersen. S = Steensgaard. f = field-sensitive. c = context-sensitive. h = heap-sensitive. z = zero-weight exclusion.

memory usage information. Improved IPA accuracy thus *may*, not *must*, improve C-SRAM power savings. With the fast analysis framework used for this work, it is worthwhile to always use the best analysis, so as to capture these opportunities for improving results.

4.2.4 Optimistic pointer analysis

Figure 4.7 shows the positive impact IPA optimism has on object scheduling. Against the S and *Afc(h)* results from Figure 4.5, the gains of zS and *zAfc(h)* are clear. Where z-based static power savings increases, but there is little or no change in dynamic power across analysis types (e.g., for the EEMBC benchmarks), infrequently (or never) accessed objects have been moved from short to long latency storage, where static power savings is quite significant. The *autcor00* benchmark, for example, contains references to three different input types, "sine," "speech," and "pulse." For each input type, there are three

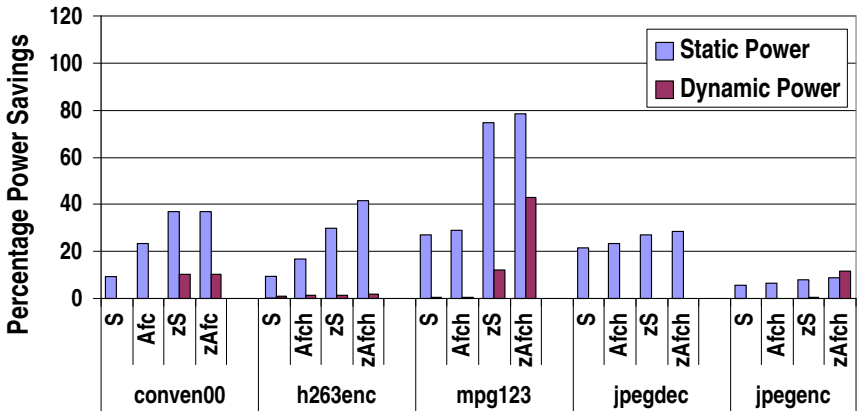


Figure 4.8 Detailed comparisons of data power savings from zero-weight pointer analysis based on accurate (*zAfc(h)*) and inaccurate (*zS*) formulations.

corresponding arrays (`input_buf`, `test_buf`, and `t_buf`) used as input and checking data. For a given program run, only one set of buffers will be used, but the other buffers are potentially accessible through pointers. Zero-weight exclusion during pointer analysis allows the unaccessed buffers to be placed in long-latency storage, because they no longer appear in the code as conjoined to the buffers actually used.

Zero-weight interprocedural analysis noticeably reduces the total number of dependence arcs present in an application. However, for data power savings, it is still important that this optimistic analysis have an accurate foundation. For a selection of the studied benchmarks, the *zAfc(h)* results in Figure 4.8 highlight the additional power savings benefit (over *zS*) which may be achieved when an accurate analysis (*Afc(h)*) underlies the zero-weight algorithm.

If zero-weight analysis missed many object access arcs that occur at run time (for inputs other than the one used for front-end control flow profiling), its use for object scheduling could have detrimental performance effects. Specifically, if an accessor was

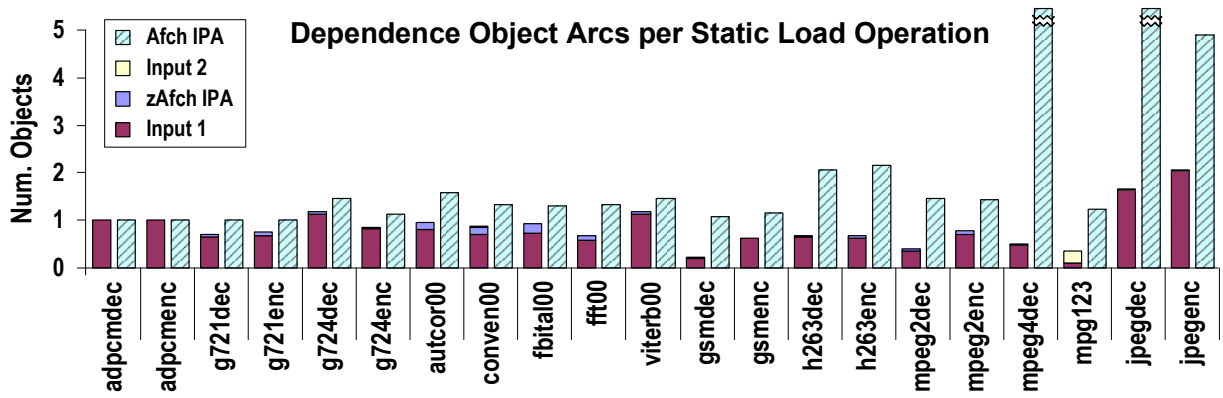


Figure 4.9 Comparison of zero-weight pointer analysis object dependence markings to real multi-input memory access behavior.

not marked by the analysis, and a datum scheduled for access in a single cycle at that `load` site is in a three-cycle region, the processor must stall while the datum is accessed. If two simultaneous accesses to a one-port region occur at run time, a smaller (one-cycle) port wake-up penalty is incurred. In this case, however, data power consumption is also impacted because the second set of bitlines must be charged.

Figure 4.9 shows two columns for each benchmark. The left is a stacked column composed of three components. Its foundation is the average number of objects accessed by each static `load` operation during a single run of the program (Input1, a training run). Added to Input1's object accesses are the additional, different, objects accessed for an evaluation run using Input2. The last component represents the objects marked by the optimistic *zAfc(h)* pointer analysis that are not accessed for the two sample runs. The right column is the average number of dependence objects marked on each `load` by the most accurate form of conservative analysis used, *Afc(h)*. Figure 4.9 shows why zero-weight analysis is profitable for object scheduling: the object dependence arcs *zAfc(h)*

draws closely match the memory behavior for benchmark inputs other than those used for the control flow profiling which drives zero-weight block exclusion.

As explained in Section 3.5.2, data configuration decisions made for objects which are neither interactive nor conjoined with other objects are made independently of decisions for other objects. Object scheduling is least constrained for (1) independent objects (because they do not consume slack or port resources that might have been allocated to another object) and (2) objects with only a single static access site. For a selection of the benchmarks studied, Figure 4.10 shows how the amount of power savings attributable to independent objects differs between accurate *Afc(h)* and optimistic *zAfc(h)* analysis. Zero-weight analysis clearly reduces constraints on the object scheduling problem, and allows many more independent object placement decisions.

A single-access object may occur, for example, when a look-up table is read-only or, for zero-weight analysis, when loop-initialized data is never used. When zero-weight pointer analysis is used, there may be objects to which no static operations have dependence arcs. Figure 4.11 shows the fraction of total global and stack objects with zero and one access sites for the most accurate pointer analysis combination used – Andersen, field-sensitive, context-sensitive, heap-sensitive (*Afc(h)*) – and the corresponding zero-weight path exclusion analysis (*zAfc(h)*). Zero-weight analysis significantly increases the number of unconstrained or lightly constrained objects.

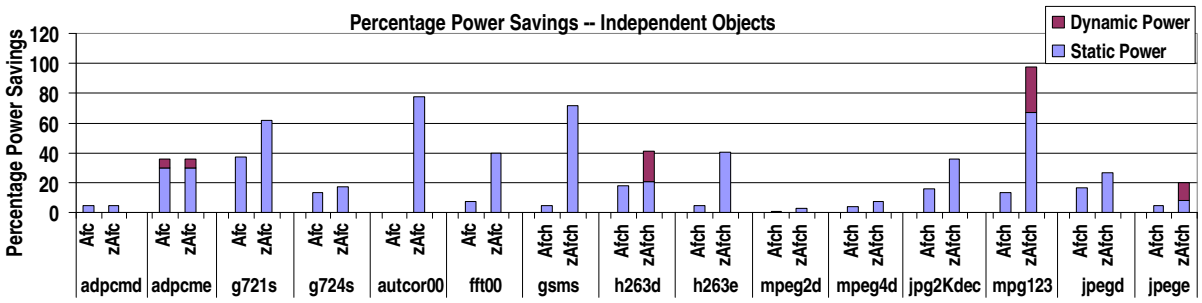


Figure 4.10 Contribution of independent data objects to total SRAM power savings.

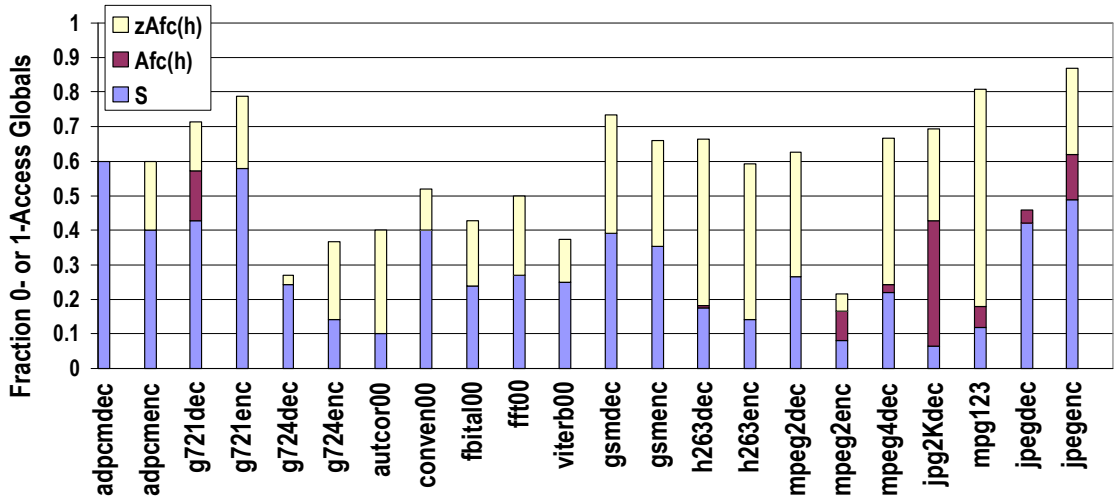


Figure 4.11 Increase in single- or no-access object counts for *Afc(h)* versus zero-weight *zAfc(h)* pointer analysis.

4.2.5 Power–performance balanced schedules

As a first-level approximation, a technique may be considered "power-aware" if the ratio between percentage power savings and performance degradation is at least 3.0 [39]. When system-level circumstances permit performance degradation, the compiler uses a control flow profile to anticipate the net performance effects of restricting ports or increasing access time of individual objects. Configuration distributions for a moderate performance degradation threshold are shown in Figures 4.12 (run-time access counts) and 4.13 (fractions of total data size). Percentage performance decay is shown as a black bar below the footprint distributions in Figure 4.13. For this moderate bound, each benchmark's performance is required to be within 5% of the best-case (2P–1C) schedule. Degradation is not uniform because for some applications, forcing additional restrictions requires "creation" of slack that pushes performance past the specified tolerance. Net power savings are shown in Figure 4.14. These results correspond to the nondegraded *Afc(h)* results shown in Figure 4.5. Allowing this slight slowdown most noticeably increases opportunity for object placement in long latency regions, accounting for the increase in dynamic power savings relative to the undegraded schedules.

The average 2.7% degradation for the "moderately" degraded schedule corresponds in Figure 4.14 to an anticipated data power savings of 31% (51.7% of static power, 9.4% dynamic power, assumed to contribute equally to total SRAM power in future technologies [10]). Assuming that on-chip data storage consumes 50% of overall processor power [1], the system-wide $power{:}performance$ ratio falls well above the threshold of

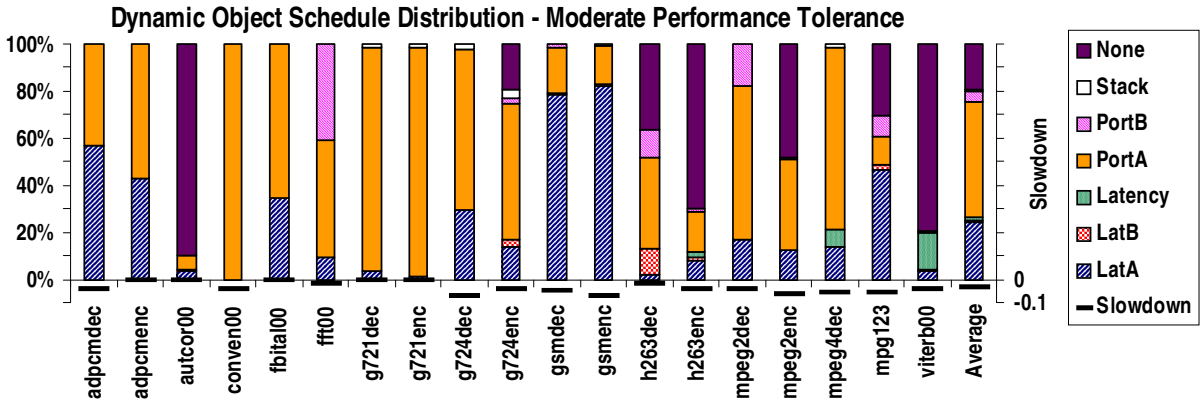


Figure 4.12 Moderate slowdown: access distributions to various configurations.

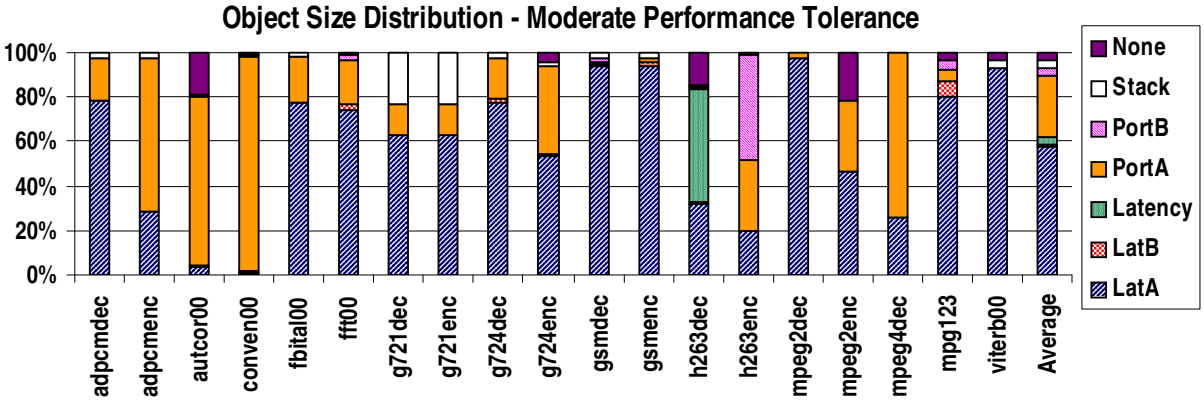


Figure 4.13 Moderate slowdown: data bytes stored in various configurations.

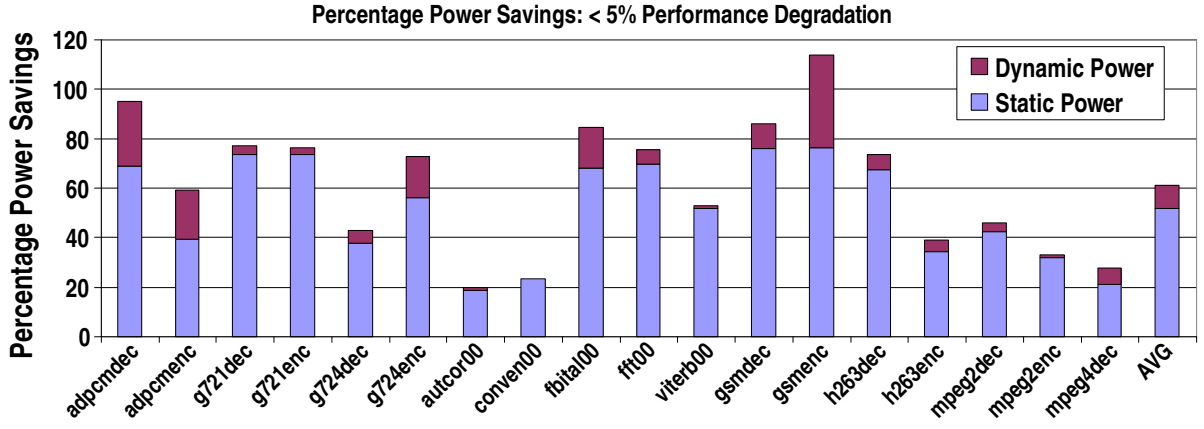


Figure 4.14 SRAM power savings with moderate performance degradation ($\leq 5\%$).

beneficial power-savings techniques. This power benefit relies on an already low-power design style, and low-power TSMC process parameters are used for calculating anticipated dynamic power savings.

4.3 Object and Configuration Statistics

Benchmark stack, global, and heap variable usage is summarized in Table 4.2. Access distributions are given for classically optimized code; Table 4.2 percentages indicated as 0.00 are fractions smaller than five one-thousandths. After local variable promotion, the stack size for each application is very small at any given time, never exceeding 1.1 kB, so the amount of mandatory high-power storage is minimal (second column of Table 4.2). Program accesses are primarily to global space for the telecommunication applications without dynamic memory allocation, and to heap for more complex codecs.

mpeg4dec and *h263enc* are good examples of codes with high heap turn-over, i.e., their maximal live heap sizes are much greater than the total amount of heap space allocated throughout program execution. While they do perform frame-wise processing, due to coding style, *jpegdec*, *jpegenc*, *h263dec*, *mpeg2dec* do not as efficiently utilize heap allocated space. The total footprint of each application, however, still easily fits within 128 kB, 256 kB, or 512 kB of data storage. The low *mpg123* heap usage is a relic of the program's implementation for desktop-domain computing, and was addressed in Section 2.6.

Table 4.2 Data sizes and access distributions, no function inlining.

Application	Max. Tot. Stack Size (kB)	Tot. Global Size (kB)	Tot. Heap Size (kB)	Max. Live Heap Size (kB)	% Access to Stack	% Access to Globs	% Access to Heap
adpcmdec	0.1	2.9	—	—	0.00	100.00	—
adpcmenc	0.1	2.9	—	—	0.00	100.00	—
g721dec	0.2	0.7	—	—	1.20	98.80	—
g721enc	0.2	0.7	—	—	1.18	98.82	—
g724dec	0.3	12.1	—	—	2.02	97.98	—
g724enc	0.3	21.3	—	—	4.27	95.73	—
autcor00	0.2	15.7	—	—	0.11	99.89	—
conven00	0.1	12.9	—	—	0.00	100.00	—
fbital00	0.2	8.6	—	—	0.00	100.00	—
fft00	0.3	20.2	—	—	0.00	100.00	—
viterb00	0.2	6.3	—	—	0.18	99.82	—
gsmdec	0.4	16.2	0.7	0.7	0.92	40.76	58.32
gsmenc	0.5	16.2	0.7	0.7	0.66	40.01	59.33
h263dec	0.4	313.2	185.5	185.5	0.11	63.36	36.52
h263enc	0.4	330.8	33 426.3	313.7	0.02	15.04	84.93
mpeg2dec	0.6	21.7	372.2	372.2	0.22	87.48	12.30
mpeg2enc	0.8	9.3	1220.7	1220.7	0.44	7.39	92.18
mpeg4dec	1.1	976.1	114 513.4	5140.9	1.62	39.07	59.30
jpg2Kdec	1.1	92.0	12 095.6	11 852.8	0.02	0.17	99.81
mpg123	0.3	240.3	33.0	33.0	0.08	99.92	0.00
jpegdec	0.4	20.5	30.7	30.7	0.62	16.12	83.26
jpegenc	0.4	24.6	143.9	143.9	1.23	41.67	57.10

After object configurations are chosen, data must be grouped into macroblocks according to configuration properties. The overhead of this alignment restriction is shown in Figure 4.15 for classically optimized code scheduled without performance tolerance into two macroblock sizes: 128 bytes and 256 bytes. For a 16 byte SRAM line size, this corresponds to eight-line and 16-line macroblocks. For most applications, the alignment overhead is trivial. Applications with a small dataset, e.g., the *g721s*, *adpcms*, and

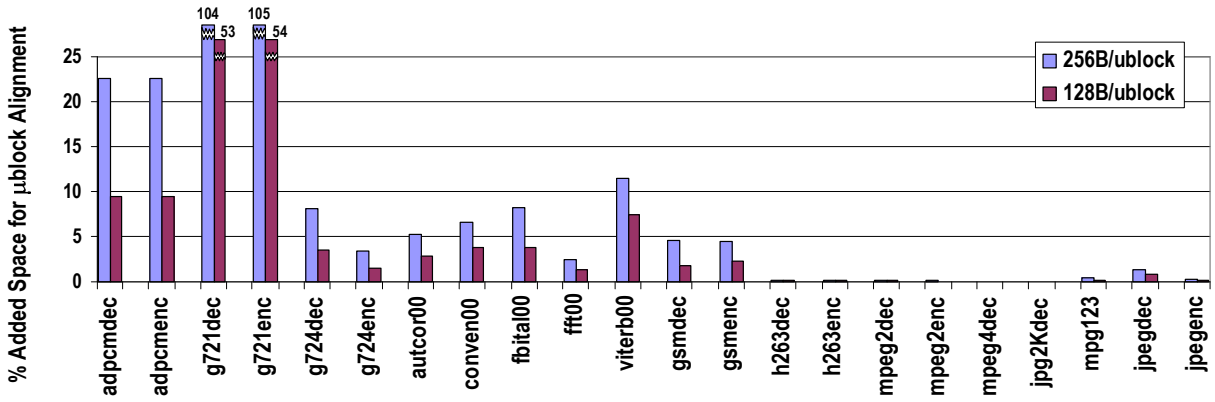


Figure 4.15 Increase in data space to due alignment of each configuration type to 128 byte and 256 byte macroblock bounds.

EEMBC kernels, are most effected by the alignment restriction, because they use only a small part of several configurations (e.g. 16 bytes of 256 bytes to hold a single object). Given low usage of opposing ports, it is worthwhile to consider providing only one port configuration for these applications, i.e., eliminating the LatB and PortB options.

4.4 Synergy of Aggressive ILP Optimization with Power Savings

Aggressive ILP optimization requires code replication of various forms: operations are duplicated and predicated on opposite predicates, loops are replicated during unrolling and peeling, and superblock and hyperblock tails are created for side exits from the main code block body. A "classic" DSP processor has little instruction storage, so other than software pipelining and limited function inlining, ILP optimizations have not traditionally been used in the embedded domain [5]. The performance benefit of code scheduled with moderate loop unrolling, peeling, and hyperblock formation bounds is

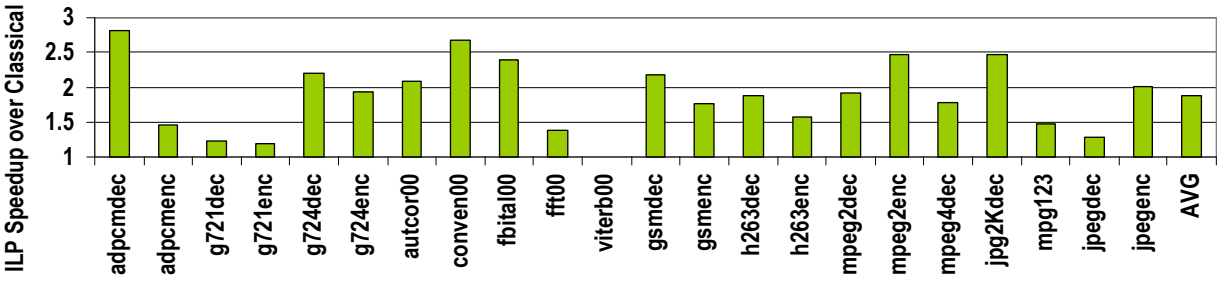


Figure 4.16 Performance improvement of ILP-optimized code.

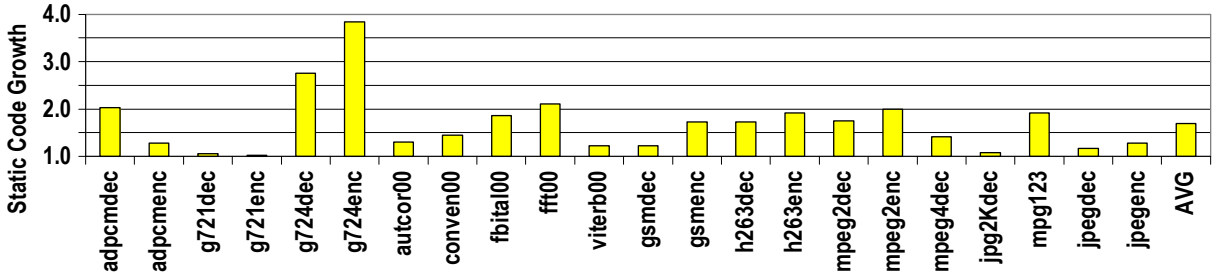


Figure 4.17 Code size expansion for aggressive ILP optimization.

shown in Figure 4.16. This speedup comes at the cost of the code size expansion shown in Figure 4.17.

Both the performance and power penalties associated with code size expansion can be mitigated through careful use of loop buffers [56]. In that work, we found that a dynamically scheduled loop buffer (one in which loops are swapped in and out at run time, under compiler direction) could realize significant dynamic power savings for fetch of ILP optimized code

Likewise, ILP optimization can have a favorable effect on data storage power, if SRAM may be compiler-managed to fit data access properties and needs. However, creation of larger scheduling regions and addition of speculative memory operations can significantly change the amount of slack and flexibility available to the object scheduler, so the effects of ILP optimization are not always positive. Figures 4.18 and 4.19 correspond with the

size and access distributions shown previously for classically optimized code. Without performance tolerance, these object schedules realize the data array power savings in Figure 4.20. For completeness, Figures 4.21, 4.22, and 4.23 show distributions and power savings for a corresponding moderate performance tolerance of less than 5%.

Given the significant code size and performance changes, it is not sufficient, however, to evaluate ILP-optimized versus classically-optimized data power savings without considering instruction and overall system power. Combining the code size and data differentiation impact of ILP optimization, the net combined instruction and data power ratios in Figure 4.24 are realized. Without performance tolerance, aggressive ILP optimization does not always yield a profitable power point relative to classically optimized code (for many benchmarks in Figure 4.24, the ratio of $\frac{ILP\_Optimized_{instruction+data\ power}}{Classical_{instruction+data\ power}}$ is less than one). With moderate performance tolerance (<5%, Figure 4.25), results are more consistently favorable.

Addition of unpredictability is inherent to the ILP optimizations performed here – speculation introduces `load` latency uncertainty, and performance will be hindered when execution leaves a profile-based trace chosen for hyperblock or superblock formation. Assuming that this drawback can be tolerated, i.e. the system has soft, not hard, real-time deadlines, when performance improves and *data+instruction power* decreases, ILP optimization is worthwhile. Cases where performance improves but power increases must be more closely evaluated. As described in Section 3.4.2, the percentage performance

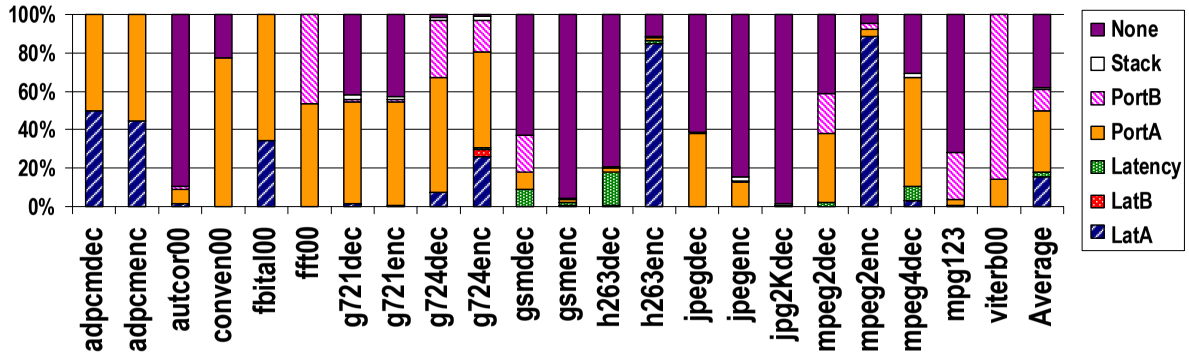


Figure 4.18 Percentage run-time accesses to various configurations: ILP-optimized code, no performance degradation tolerated.

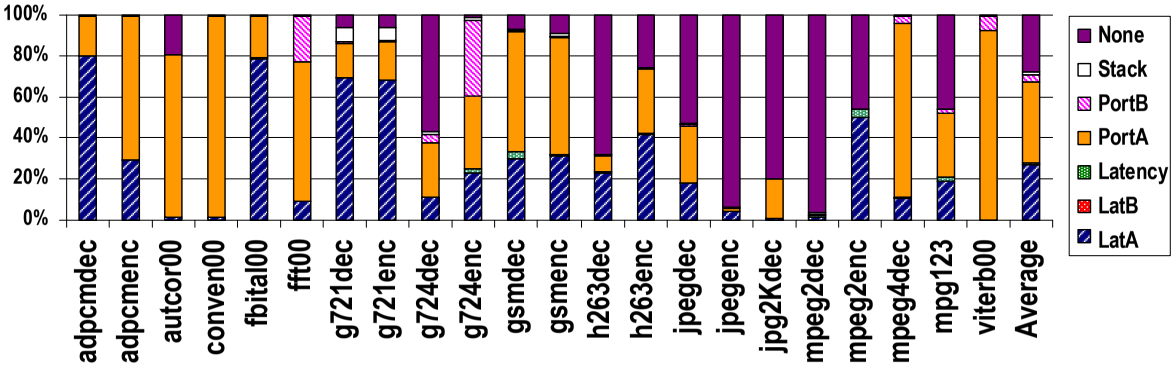


Figure 4.19 Percentage total data bytes in various configurations: ILP-optimized code, no performance degradation tolerated.

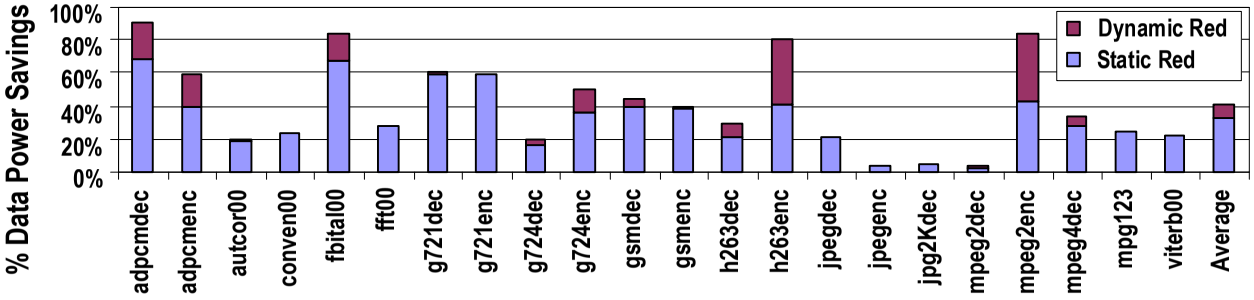


Figure 4.20 Data power savings for ILP-optimized code, no performance degradation tolerated.

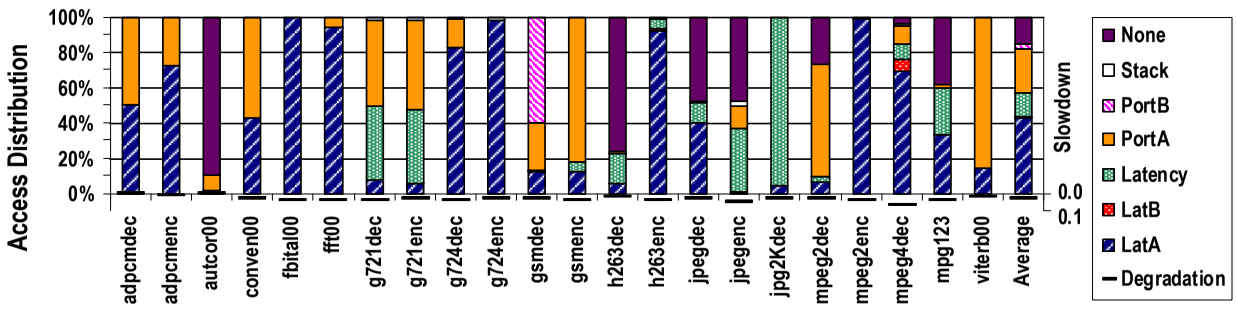


Figure 4.21 Moderate slowdown: percentage run-time accesses to various configurations: ILP-optimized code, no performance degradation tolerated.

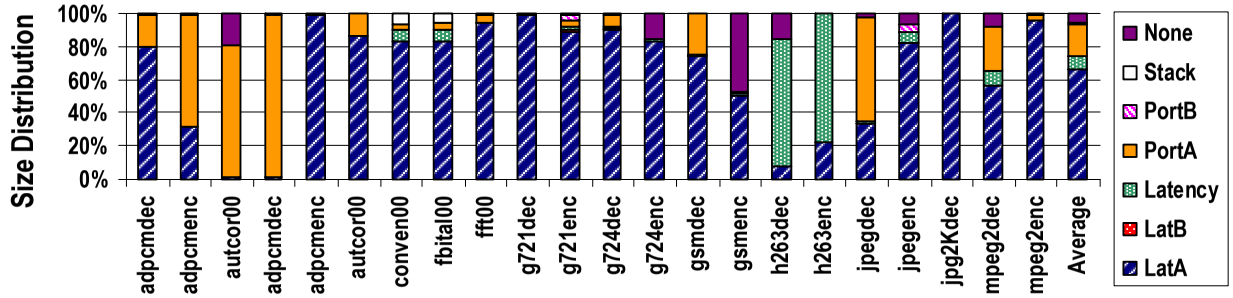


Figure 4.22 Moderate slowdown: percentage total data bytes in various configurations: ILP-optimized code, no performance degradation tolerated.

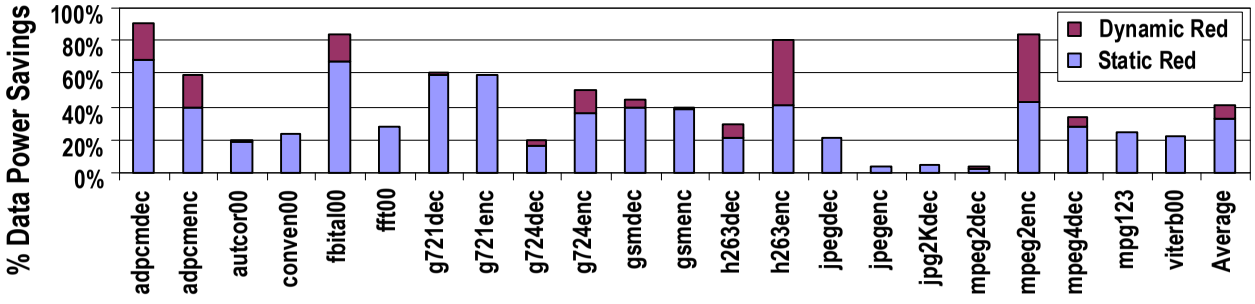


Figure 4.23 Moderate slowdown: data power savings for ILP-optimized code, no performance degradation tolerated.

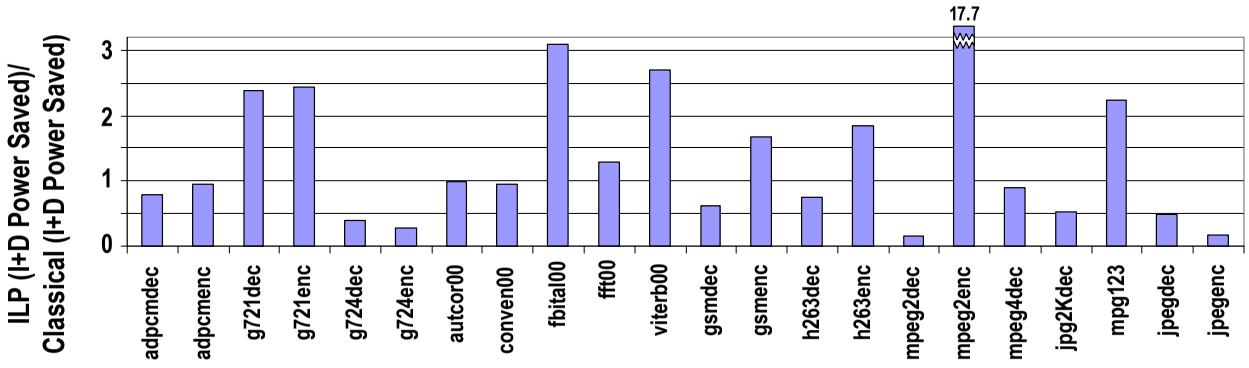


Figure 4.24 Net instruction and data power savings for ILP-optimized code relative to classical optimization.

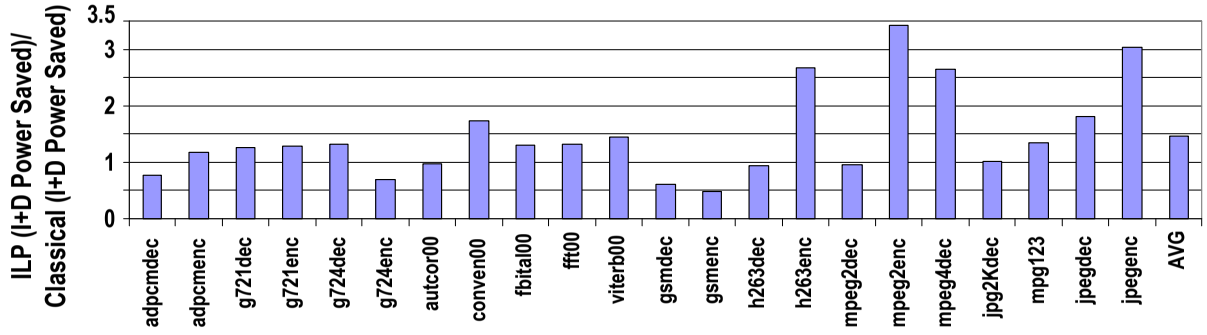


Figure 4.25 Moderate performance tolerance: net instruction and data power savings for ILP-optimized code relative to classical optimization.

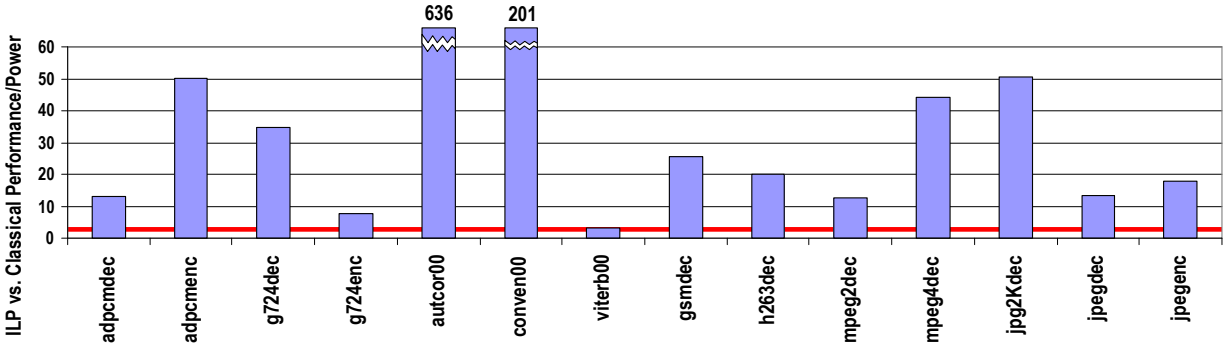


Figure 4.26 % $Performance\ improvement$: % $Power\ increase$ ratio for ILP versus classical optimization.

increase must be at least three times the resulting power increase for a technique to be considered "power-aware" [39].

Using an assumption that instruction and data storage jointly comprise one-half of processor power, Figures 4.26 and 4.27 translate the results of Figures 4.24 and 4.25 into a comparison of $Percentage\_performance\_improvement$: $Percentage\_power\_increase$ for ILP versus classically optimized code. Benchmarks are only shown if they had an $\frac{ILP\_Optimized_{instruction+data\ power}}{Classical_{instruction+data\ power}}$ ratio of less than one. A bold line at the bottom of each figure shows the 3.0 bound of this ratio. The conclusion of these results is that if a system-wide mechanism for DVS or clock gating can be provided to compensate for early completion of processing, i.e., to translate performance improvement into power savings, the combination of speedup with data power savings makes ILP optimization uniformly beneficial.

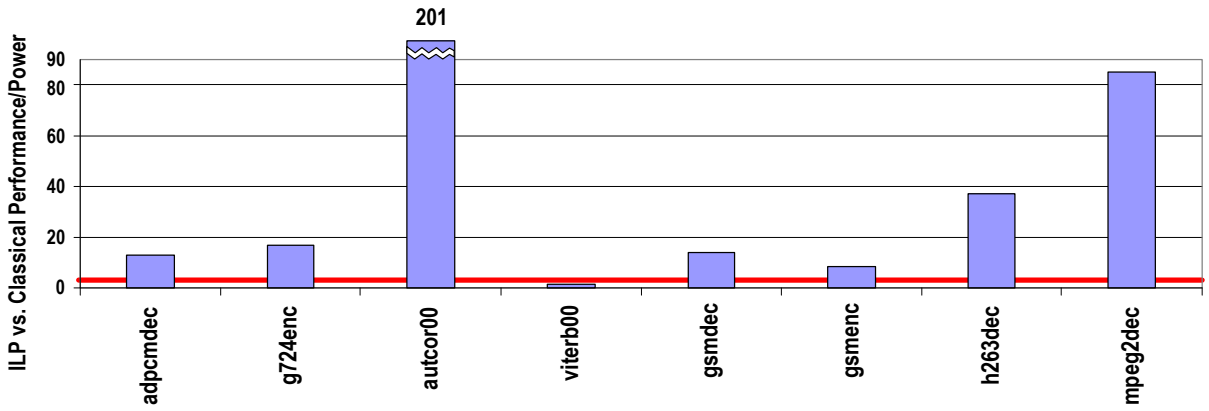


Figure 4.27 % *Performance improvement* : % *Power increase* ratio for ILP versus classical optimization, moderate performance tolerance.

4.5 Validating Results: General Sensitivity

From a general sensitivity study of program IPC (instructions per cycle) versus memory port count and latency, some interesting properties of telecommunication and media applications become apparent. As single-cycle port count is increased from left to right, Figure 4.28 shows the IPC gains realized for classically optimized codes. Without ILP-enhancing optimizations, in particular in the absence of loop transformations (unrolling and peeling), there is not generally sufficient parallelism in these applications to observe significant benefit from increasing the port count. However, the IPC step between one and two ports is virtually universal (all but the *adpcm* benchmarks, which are relatively small kernel-like applications, and primarily register allocated). Forcing all memory accesses to use a single port versus two ports can have up to a 28% impact on IPC (for *h263dec* and *h263enc*), emphasizing the importance of being able to selectively reduce

port count and provide differentiated data access. With a higher baseline memory latency of three cycles, overall IPC is lower, and is not as sensitive to port count, but the step between one and two ports is again prominent (Figure 4.29).

General sensitivity to latency is more dramatic: for a two-ported memory resource configuration, IPC drops an average of 22%, and up to 35% (*jpegdec*) as latency is increased from one to three cycles. IPC latency sensitivity is shown for each benchmark in Figure 4.30. The more dramatic latency response relative to changes observed in Figures 4.28 and 4.29 for port count is echoed in the C-SRAM scheduling distributions of Figures 4.3 and 4.4, where, without performance degradation, many more accesses can be placed in PortA and PortB configurations than in long-latency storage. If placing all data in three cycle storage realizes 63% data power savings (80% static, 46% dynamic, each contributing equally to total power), even if data access comprises one-half of total processor power, a 31.5% performance improvement does not compare favorably with the average 22% IPC drop. This again emphasizes the importance of selectively differentiated access latency.

Observing the same port and latency sensitivity trends for ILP-optimized codes (Figures 4.31, 4.32, and 4.33), several differences become apparent. Clearly, overall IPC values are higher after ILP optimization, but relative changes in IPC in response to port count are greater than for classically optimized codes. Consistent IPC steps are visible not only when single-cycle port count is reduced from two to one, but also from three to two (Figure 4.31). In contrast, latency sensitivity is not as great: for a dual-ported

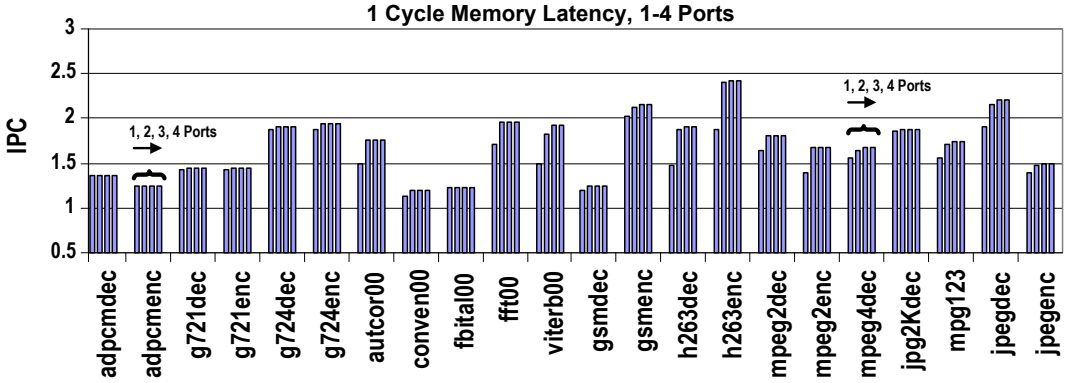


Figure 4.28 General sensitivity to port count: single cycle memory access; increasing port count from left to right; classically optimized code.

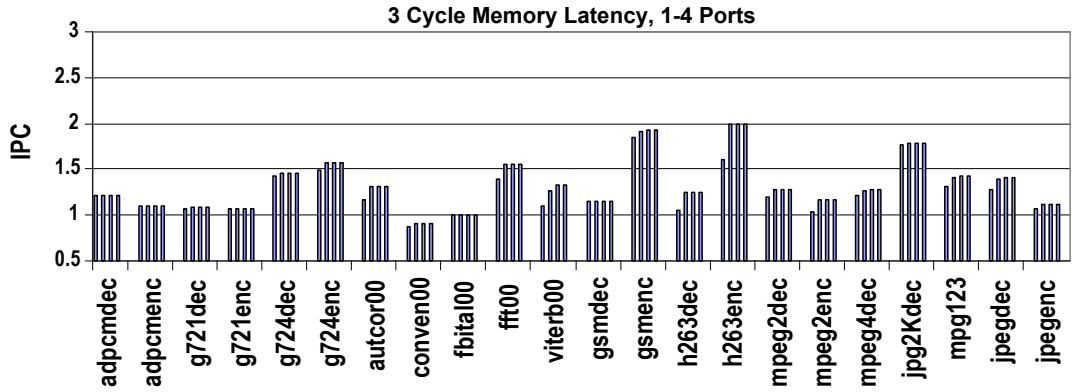


Figure 4.29 General sensitivity to port count: three cycle memory access; increasing port count from left to right; classically optimized code.

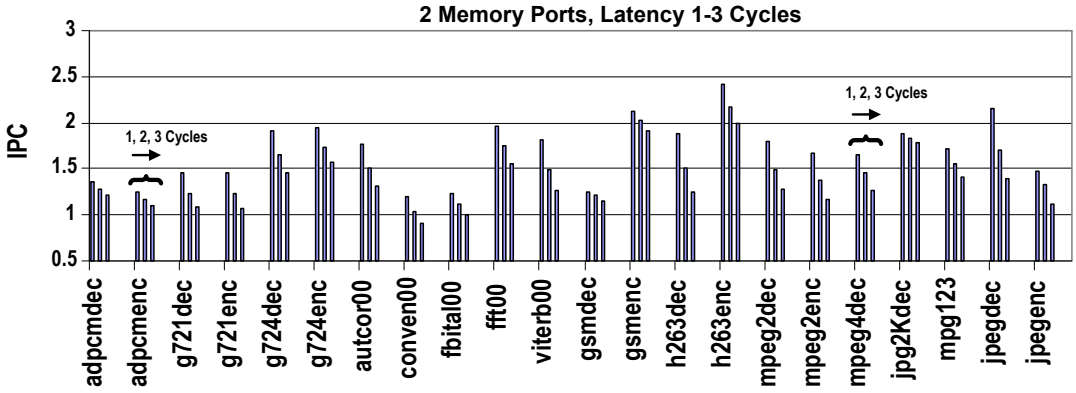


Figure 4.30 General sensitivity to memory latency: two ports for all data; increasing latency from left to right; classically optimized code.

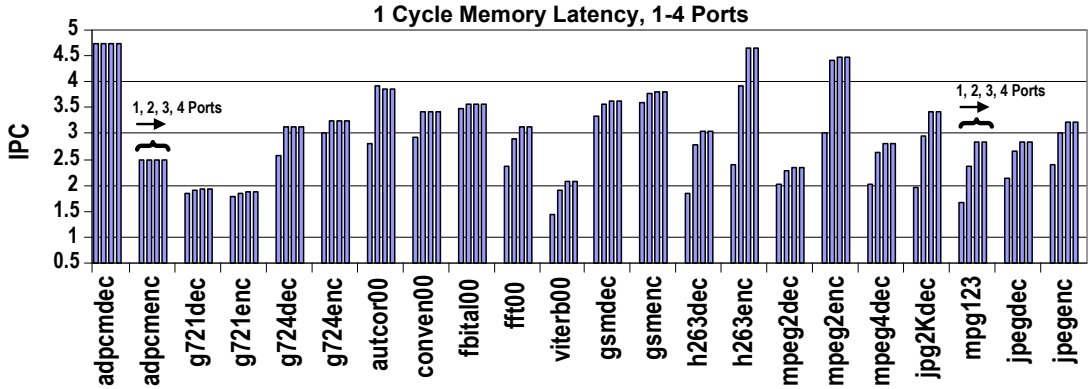


Figure 4.31 General sensitivity to port count: single cycle memory access; increasing port count from left to right; ILP-optimized code.

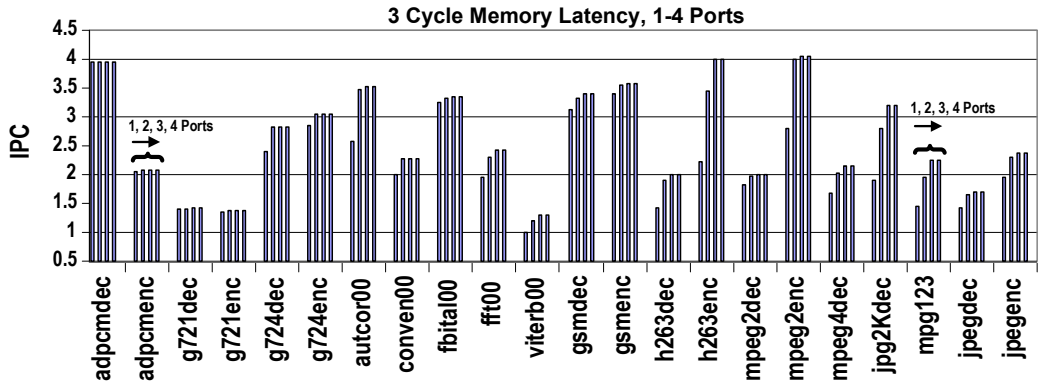


Figure 4.32 General sensitivity to port count: three cycle memory access; increasing port count from left to right; ILP-optimized code.

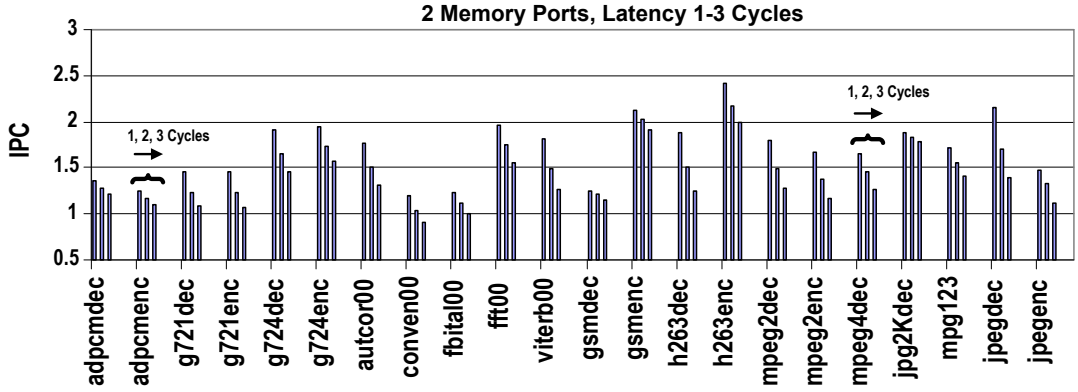


Figure 4.33 General sensitivity to memory latency: two ports for all data; increasing latency from left to right; ILP-optimized code.

baseline, classically optimized IPC drops on average 7% more than ILP-optimized code when latency is increased from one to three cycles.

Because not all SRAM, even if on-chip, will have a single-cycle access latency, the implemented framework was also validated for SRAM with higher baseline latencies. As latencies increase, overall scheduling slack can drop significantly, but C-SRAM power savings potential was still found. In particular, opportunity for port restriction remains strong because many port-restricted objects are independent, and thus, even if accessed at multiple sites, are not as frequently impacted by the overall reduction in available slack.

4.6 Summary of Findings

Key findings in this study of joint operation–object scheduling for port and latency configuration are as follows:

- As demonstrated through savings of up to 68% static and 22% dynamic power, a greedy object scheduling algorithm can find significant opportunity for data power reduction in the C-SRAM framework.

- Tolerance of port restriction and latency reduction exists in all applications, but to varying extents. As expected, the greatest opportunities for configuration generally exist in filter-based telecommunication applications, but depending upon coding style and optimization level, are also found in media codecs.

- Interprocedural pointer analysis accuracy has a very visible impact on the volume of dependence information provided to the compiler. It does not always significantly change the information needed for unconstrained object scheduling, i.e. knowledge of data independence, but it *may*. IPA accuracy thus *can* effect net data power savings.

- With a scalable analysis framework, analysis cost is low, so the most accurate available IPA should be used.

- Optimistic IPA provides information which closely matches cross-input application behavior, and allows many more objects to be placed in power-saving configurations.

- Neither control nor memory profiling are required for object scheduling, but both forms of feedback are useful for increasing power savings.

- Aggressive ILP optimization can improve data power savings, but opportunities vary widely, and must be weighted by code size expansion. Assuming that either system-wide DVS or processor clock gating is available, when the system-wide performance improvement realized through aggressive ILP optimization is included in $power : performance$ comparisons, the code and resulting data power savings are uniformly favorable.

5 DATA INTENT CHARACTERIZATION

Previous memory optimizations have targeted improvements in temporal and spatial locality, so characterizations have focused on identifying access patterns and streams in media and telecommunication applications. This work instead found opportunity for compiler-managed data power savings when application properties are described in terms of *intent* and *ownership*. These properties relate to the mathematical underpinnings of these applications, and are the basis for the characterization and architectural application of this dissertation. Data categorization cannot be the direct guide for configuration or optimization of memory, but as this chapter will show, a thorough characterization of run-time access traffic explains the efficacy and limitations of data optimizations for this domain.

In this chapter, general application properties, including common coding structure and the strong correlation of data use with particular functions, are first described. Code examples introduce four objects with common data intent types and relate their access properties to general memory optimization and C-SRAM. The seven established intent categories are then defined, and the remainder of the chapter describes how the observed

algorithmic properties have also driven design of data buffering and sleep management for embedded applications.

5.1 Algorithmic Properties

Several classes of applications exhibit regular function invocation and data usage behavior, including signal-processing and scientific codes. Of these, telecommunication, image, and video applications are particularly well-suited to execution in a specialized processing environment where ASIC blocks or custom accelerators streamline computation. On a fully programmable processor, however, execution commonly becomes inefficient.

Telecommunication and media codecs may be expressed as a cascaded series of basic signal processing functions, or *kernels*. In an ASIC design, input data enter processing blocks from an I/O buffer or memory storage, and intermediate values are passed between computational blocks. As pictured in Figure 5.1, additional data, such as look-up tables and processing state, are kept close to the kernels which use them. Such data flow is easily represented in a hardware description language (HDL) for custom design. High-level language (HLL) programming and compilation to a programmable target, however, require that the pictured communication and storage flows be translated into a sequence of accesses to generic processor memory space. In the resultant logical HLL coding style, filter tap coefficients are stored in arrays; short-cuts for mathematical operations appear as global look-up-tables in memory; and what might have originally been wires

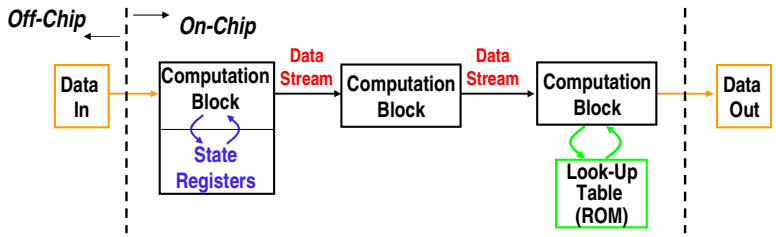


Figure 5.1 Typical ASIC telecommunication/media flow.

between function blocks in an ASIC design become arrays or structures used to transfer information across function calls. Hardwired on-chip communication paths and local variable stores are thus replaced with frequent memory accesses. These data accesses are superfluous with respect to original codec structure, but due to their original *intent* in the underpinning mathematical algorithm, are fortunately more regular than those of general-purpose applications.

Several properties recur across a broad spectrum of telecommunication and media applications: (a) reliability of usage periodicity, (b) significant temporal gaps between usage, and (c) disparity of usage among various data objects. These first two properties relate to data ownership by particular application kernels (functions) and are manifested due to call trace regularity. Data ownership also helps to account for the low number of per-object access sites, which aids the object scheduler described in Chapter 3. The latter property indicates the need for prioritization and selectivity in treatment of data objects.

Figure 5.2 shows interval-based activity for all global objects accessed during part of an execution of the *g724dec* program. Global data for telecommunication applications include mathematical short-cuts, such as powers of two, inverses, and square roots; scaling

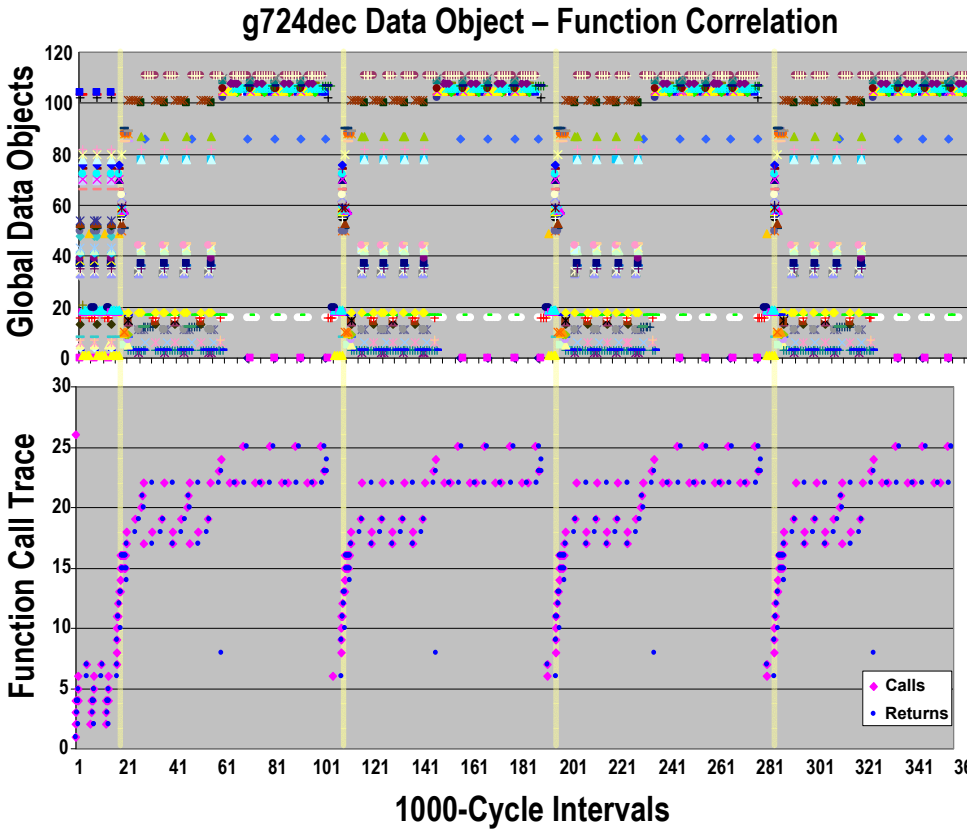


Figure 5.2 *g724dec* access patterns for all accessed global objects, correlated with the function call trace.

and quantization factors for compression and normalization; and filter coefficients. Each *g724dec* global object has been assigned a number (1–112), which appears along the y-axis at the top of Figure 5.2. Each point on the x-axis represents an interval of 1000 cycles. When any element or field of an object is accessed (read or written) in an interval, a dot appears on the graph. Along the bottom of Figure 5.2 is a similar (interval-based) graph of the *g724dec* call trace. As time proceeds from left to right, the calls and returns of individual functions (numbered 1–26) are marked by light diamonds and dark dots, respectively.

As highlighted with vertical stripes, each access to Object 60, the `dico1_lsf` array, a quantization look-up table, correlates with an execution of Function 10, `Dico_plsf_5`. This occurs because of a common trait of look-up tables in this domain: `dico1_lsf` is *owned* by, or exclusively accessible by, the function `Dico_plsf_5`.

While many objects are clearly accessed in Figure 5.2 with defined periods, it is most interesting that the majority of objects are not frequently used. The significant temporal gaps between access intervals would predict poor performance for these objects if cached using the traditional *least recently used* (LRU) hardware replacement method. This common property drives use of noncache and software-managed SRAM in the telecommunication domain. These gaps in usage also forecast inefficient $memory\_power : data\_access$ density for noncache (software-managed) memory, so reinforce the need for low-power data storage configurations.

5.1.1 Example data objects

Four code examples are used in this section to highlight typical telecommunication and media data object types. First, Figure 5.3 shows the GSM 6.10 regular pulse excitation long term predictor (RPE-LTP) decoder (*gsmdec*) `Short_term_synthesis_filtering` function. When scheduled for a processor resembling the Texas Instruments 'C64x DSP [57], this function accounts for 77.5% of the decoder execution time. Array `v` is a field of the `gsm_state` structure, a heap object allocated to serve as the program's primary cross-procedure storage. The `v` array maintains synthesis filter state information

```
struct gsm_state {
    word        dp0[ 280 ];

    word      z1;   /* preprocessing.c, offset compensation */
    longword L_z2;                    /* offset compensation */
    int       mp;           /* preprocessing.c, preemphasis */

    word      u[8];             /* Short_term_analysis_filter */
    word      LARpp[2][8];                  /* short_term.c */
    word      j;                            /* short_term.c */

    word      nrp;                /* long_term.c, synthesis */
    word      v[9];              /* short_term.c, synthesis */
    word      msr;              /* decoder.c, postprocessing */

    char      verbose;            /* only used if !NDEBUG */
    char      fast;                    /* only used if FAST */
};
```

```
static void Short_term_synthesis_filtering ( struct gsm_state * S,
    word  * rrp,  int k,  word * wt, word * sr  )
{   ...
    while (k--) {
        sri = *wt++;
        for (i = 8; i--; ) {
            /* Use coefficients LARp as multiplier values */
            sri = sri - rrp[i] * v[i];
            v[i+1] = v[i] + rrp[i] * sri;  }
        *sr++ = v[0] = sri;    }
}
```

Figure 5.3 Example state *(ST)* array: *gsmdec* `gsm_state.v`.

across `Short_term_synthesis_filtering` invocations. For classically optimized code, the compiler brings individual elements of the `v` array into the register file for each inner loop iteration, and then returns a value to memory once the loop iteration completes. These transfers account for 38% of total run time *gsmdec* `load` and `store` operations, but have a highly predictable pattern because they represent the recall and storage of kernel *state*. In the language of this characterization, the `v` array *belongs* to the `Short_term_synthesis_filtering` function because it is *state*. This array is only accessed by its *owner* function, and so optimization potential exists in putting it into a low-power state when its owner is not executing.

A second typical array type is depicted in Figure 5.4: a mathematical look-up table. The *g721dec* `quan` function is heavily used by multiple mathematical and quantization routines (when no inlining is performed, it accounts for 51.9% of benchmark execution time). The majority of *g721dec* `quan` callers use the array `power2` as the `table` parameter, *i.e.*, as a shortcut for actual computation of powers of two. This array's values are

```
static short power2 [15] = { 1, 2, 4, 8, 0x10, 0x20, 0x40, 0x80,
        0x100, 0x200, 0x400, 0x800, 0x1000, 0x2000, 0x4000} ;

/* Function quan ( ) : quantize val against table of size elements
 * Return i if table [ i -1 ] <= val < table [ i ]
 * Linear search used to simplify coding            */

static int quan  ( int val , short * table , int size )  {
        int i ;
        for  ( i = 0 ; i < size ; i++ ) {
                if ( val <  * table++  )
                        break ;
        }
        return ( i ) ;
}
/* Calls to quan( ) from quantize( ), fmult( ) , and update( ) : */
quan (SHORT , power2 , 15 ) ;
```

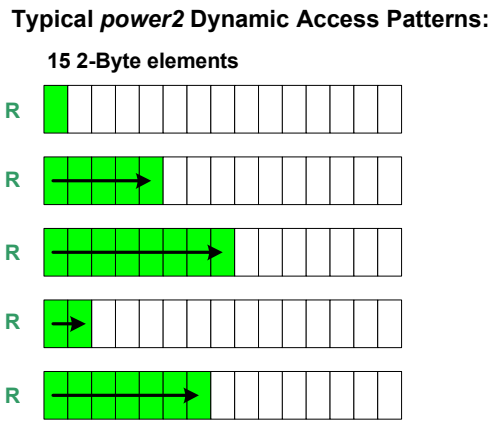


Figure 5.4 Example look-up table *(LT)* array: *g721dec* `power2`.

initialized at the time of declaration, and it is `static`, so it is never modified during the course of program execution; read-only access is one of the key features of look-up table data objects. The `size` parameter to `quan` is 15 when `power2` is used, but the `break` statement is often taken before its enclosing loop has iterated 15 times. Like the `v` array described from *gsmdec*, indexing off an induction variable makes the `power2` access pattern highly predictable, but in this case, it cannot be guaranteed that all members of the array are needed. This makes it a difficult object to analyze for pattern-based optimizations, but for access differentiation, it has a nice property: without aggressive ILP optimization, the table is accessed one element at a time. Like most look-up table accesses, the speed of this single access is critical to efficiency of its enclosing utility routine, but from a C-SRAM perspective, this object is still a candidate for port restriction. `Loads` and `stores` to `power2` account for 54% of the *g721dec* application memory traffic, so optimization of these memory accesses can significantly impact net memory power consumption.

In the context of this characterization, optimization of access power to the `power2` look-up table relies upon the fact that (a) the look-up table is a mathematical short-cut used only by one function, `quan`, and (b) since array elements are accessed individually and then used for a comparison, the schedule height of the utility routine `quan` will be directly impacted by the access latency to the `power2` array. The latter has been previously termed `load` *sensitivity* to latency, *i.e.*, if the `load` operation's latency is increased, it will negatively impact its code region's performance.

A *coefficient* array from *gsmdec* is shown in Figure 5.5. This variable `LARp` accounts for 20% of the application's total memory traffic and holds the multiplier (coefficient) values for a filter routine, *Short_term_synthesis_filtering*. Some coefficient arrays will be read-only, and are actually look-up tables. This array instead belongs to an adaptive filter, and so is accessed by three functions, *i.e.* it is not owned by a single function. As shown on the right side of the figure, the array is repeatedly read and written fully and sequentially as its values are used and then updated for the next round of computation. Coefficient array access patterns will almost always be simple to analyze. Such data is, however, always interactive because in the original algorithm it serves as input to the filter taps, or multipliers. Filter computations are critical to codec throughput and performance, so there will not usually be slack around coefficient use. As thus expected, the automated object scheduler found only opportunity for port configuration, and for a test schedule of classically optimized code, `LARp` was placed in PortB due to its interaction with the `v` and `wt` arrays.

```
void Gsm_Short_Term_Synthesis_Filter ( struct gsm_state * S,
        word * LARcr,  word * wt, word * s )
{
        word          * LARpp_j      = S->LARpp [ S->j     ];
        word          * LARpp_j_1    = S->LARpp [ S->j ^=1 ];
        word          LARp [8];

        Decoding_of_the_coded_Log_Area_Ratios ( LARcr, LARpp_j );

        /* Coefficients_N_M ( ) assigns values to LARp [ 0 ... 7 ] */
        Coefficients_0_12 ( LARpp_j_1, LARpp_j, LARp );
        /* LARp_to_rp modifies LARp values according to current values */
        LARp_to_rp ( LARp );
        Short_term_synthesis_filtering ( S, LARp, 13, wt, s );

        Coefficients_13_26 ( LARpp_j_1, LARpp_j, LARp);
        LARp_to_rp ( LARp );
        Short_term_synthesis_filtering ( S, LARp, 14, wt + 13, s + 13 );

        Coefficients_27_39 ( LARpp_j_1, LARpp_j, LARp);
        LARp_to_rp ( LARp );
        Short_term_synthesis_filtering ( S, LARp, 13, wt + 27, s + 27 );

        Coefficients_40_159 ( LARpp_j, LARp );
        LARp_to_rp ( LARp );
        Short_term_synthesis_filtering ( S, LARp, 120, wt + 40, s + 40);
}

static void Short_term_synthesis_filtering ( struct gsm_state * S,
        word  * rrp,  int k,  word * wt, word * sr  )
{       ...
        while (k--) {
                sri = *wt++;
                for (i = 8; i--; ) {
                        /* Use coefficients LARp as multiplier values */
                        sri = sri - rrp[i] * v[i];
                        v[i+1] = v[i] + rrp[i] * sri;  }
                *sr++ = v[0] = sri;     }
}
```

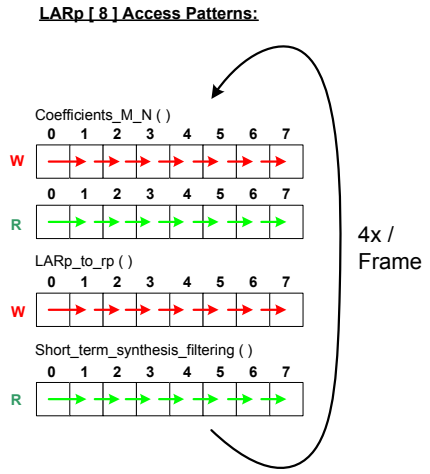


Figure 5.5 Example coefficient *(CO)* array: *gsmdec* `LARp`.

The final data object described here is of the *temporary* type. Temporaries are large, indexed, intraprocedural data which may be used to build tables, maintain local copies of input or output data, or save records of what has already been processed. Their access through array indices generally prevents allocation to a register file, and so they can account for nontrivial amounts of application memory traffic. The `workspace` array in Figure 5.6 is used to buffer data between the column and row passes of the *jpegdec* Inverse Discrete Cosine Transform (IDCT) computation. As a temporary object, `workspace` is not conjoined to other objects, but as part of a tight computation kernel, it is interactive with several other variables. The stored data is processed as an 8×8 matrix, resulting in the irregular access patterns shown on the right side of Figure 5.6. Most data relayout techniques would fail to optimize access to `workspace`, and for classically optimized

```
oid jpeg_idct_ifast (j_decompress_ptr cinfo, jpeg_component_info * compptr,
      JCOEFPTR coef_block, JSAMPARRAY output_buf, JDIMENSION output_col)

    ...
    int * wsptr;
    int workspace[64];    /* buffers data between passes */

    /* Pass 1: process columns from input, store into work array. */
    ...
    wsptr = workspace;
    for (ctr = 8; ctr > 0; ctr--) {
            if (condition) {
                    ...
                    [ Set 1 col of workspace; sequential col-order write ]
                    wsptr++;  /* advance pointers to next column */
                    continue; }
            ...
            [ Set 1 column of workspace; non-sequential col-order write ]
            wsptr++;  /* advance pointers to next column */
    }

    /* Pass 2: process rows from work array, store into output array. */

    wsptr = workspace;
    for (ctr = 0; ctr < 8; ctr++)  {
            ...
            [ Use 1 row of workspace; non-sequential row-order read ]
            ...
            wsptr += DCTSIZE;          /* advance pointer to next row */
    }
```

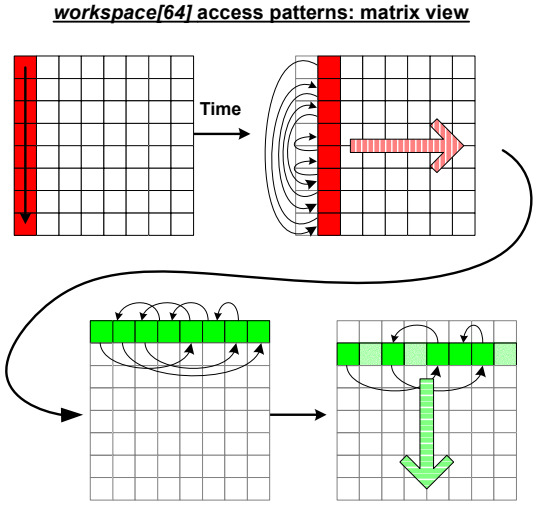


Figure 5.6 Example temporary *(TP)* array: *jpegdec* `workspace`.

code, the C-SRAM compiler was likewise unable to place `workspace` in a low-power configuration.

5.1.2 Data *intent* categories

Hand source-code and dynamic memory trace analysis has been performed on over 250 data objects in 11 applications. This analysis aims to establish a uniform understanding of both telecommunication (traditionally deemed "stream symbol processing") and media algorithms ("block pixel processing"), and explain the reasons for efficacy and limitations of existing approaches to memory optimization.

In addition to studying general usage patterns, this characterization included viewing individual data objects in terms of (a) contribution to overall memory traffic, (b) access patterns, (c) data element size, (d) number of elements/fields; (e) scaling with future

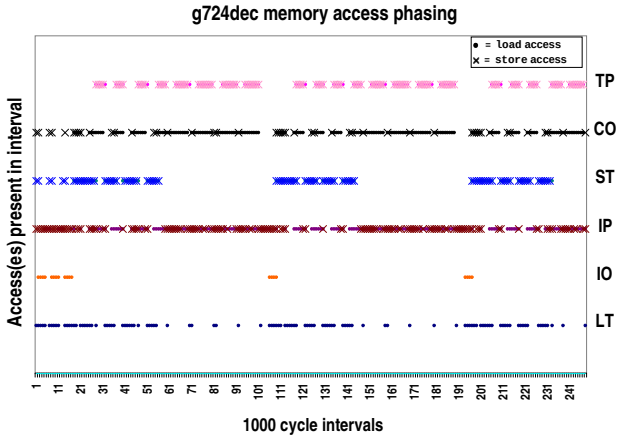


Figure 5.7 *g724dec* memory traffic phasing: arrays categorized according to Table 5.1.

standards, and (f) the role of their storage in a kernel-level algorithmic flow. Each of the seven categories in Table 5.1 originates from a property in the block-flow view of an algorithm's computation, and each category carries with it properties that should be present regardless of architecture-specific processor features. With respect to latency sensitivity, for example, it was predicted (and shown) that look-up table access is generally sparse and off the critical path. I/O items, however, generally need to be accessed rapidly, and placement in a long-latency SRAM segment generally stalls computation.

g724dec memory accesses from Figure 5.2 are revisited in Figure 5.7, where they are grouped by the data intent categories of Table 5.1. The categories in the figure are comprised of sets of arrays and structs, and so the gaps in access indicate that there are significant periods of time when particular data are not accessed. This periodic access refers back to function ownership, and as will be described in Section 5.3, can also be applied to to compiler management of sleep modes.

Table 5.1 Data object categories.

Cat	Meaning/Usage	Common Size	Ideal Memory/ Storage Construct	Data Scaling with Future Standards
LT	Look-up tables (e.g., math short-cuts, nonadaptive coefficients, scan orderings, quantization and saturation values). Commonly used just by 1 function. Generally random and partial seq. reads, full seq. accesses possible.	$S(E) = 2B, 4B$; Apps have mix of small, e.g., $N(E) = 16$, & large, e.g., $N(E) = 1024$ tables; Aprox. 4-25 tables/ application;	Two types of LUT storage:(a) low port-count, random look-up, (b) resources specialized for streamed access of full table, $P_r > P_w$.	Program access is as ROM, but writes are necessary for initialization. Most LUTs do not scale with total input size (are accessed for each frame, regardless of how many frames); however, with more complicated processing, e.g., length of a filter ($N(E)$) increases. Also, tables are inevitably different for different standards and coding styles, so reuse across standards and implementations is unlikely.
IO	Input/Output: Generally data access from/written to a file. In original algorithm, represents the input/output streams across a channel, commonly sent to the processor via a buffer. Sometimes just a temporary buffer for file I/O. May be accessed by multiple functions before/after file access.	Generally moderate to large $N(E)$ to hold a frame or block of data. Generally 1 array for reading input, 1 for writing output.	General case is full seq. access. Access will be on or near critical path since data feed rest of computation. $P_r \approx P_w$.	Size of I/O will likely grow for future devices, since frame sizes/screens will be larger. Correspondingly, both I/O buf sizes and rate of data I/O must rise to provide given frame rate. Caveat: for standards achieving higher compression, e.g., H.264/MPEG-4 AVC, I/O buf could be smaller.
IP	Interprocedural (interblock) communication (pass intermediate vals among blocks in the original algorithmic flow). Values most commonly written and read fully and seq (all elements used). Sometimes written/read in multi-element sections. For image algorithms, traversal in column-major or a scan order may occur, which makes access non-seq. Generally 1 producer function (loop) and 1 or more consumer function(s)/loop(s). *Categorization relates to original blocks, not physical procedures in code, i.e., is independent of inlining.*	$S(E)$ almost always $2B$, but $4B$ conceivable for accumulated variables not quantized/ saturated before further processing; Arrays moderate in size; 2-25 IP objects/ application	$P_r \approx or \succeq P_w$.	Interprocedural communication will likely be reasonably stable across standards; e.g., if images grow in size, for a given compression standard (e.g., MPEG-4), the means of processing will still be the same, i.e., on a macroblock (16x16) or sub-macro-block (8x8) basis.

Continued on next page.

Note: P_r = number of read ports; P_w = number of write ports; B = byte; *seq.* = sequential access, i.e., A[0],A[1],A[2], etc.; $N(E)$ = number of elements in array; $N(F)$ = number fields in struct; and $S(E)$ = size of elements in array.

Table 5.1 – *continued.*

Cat	Meaning/Usage	Common Size	Ideal Memory/ Storage Construct	Data Scaling with Future Standards
ST	State (preserved between function invocations). Commonly large `structs` with separate fields for state of various functions, so used by multiple functions. If saved in an array, generally full, seq write and read by one function across invocations (sometimes with a homing/initialization write). Update may be conditional	Small/ moderate total state size; Generally 1 `struct` or aprox. 10 arrays per application; $S(E)$ most commonly $2B$	Small RAM (must be read-writable) which is off except when its associated function is active. $P_r \approx P_w$.	Since associated with functions, for the same standard, will remain constant regardless of input size. As complexity of future standards increases, more processing blocks/filters may be added (and thus more functions), so the amount of encoder/decoder state could increase.
CO	Coefficients (adaptive). Virtually always accessed fully and seq 1x/consumer function invocation, also generally produced fully and seq. Generally either updated by one function and consumed by one other, or updated and consumed by the same function. Update may be conditional.	Small arrays, generally $N(E) = 8{-}10$; $S(E) = 2B$	Small RAM (must be read-writable) which is off except when its associated function is active. $P_r = P_w$.	Will only slightly increase with future standards, since filters may become slightly longer – advanced signal processing generally entails addition of more or slightly longer filters, not, e.g., $3\times$ longer ones.
TP	Temporary vals within 1 function (intraprocedural data, but too large to fit in register file, and need to be indirectly accessed, so cause memory traffic). May be used to store local copies of input/output parameters, help build tables, store intermediate computation values, or save records to indicate what has been processed. Would be non-persistent state native to a processing element in ASIC context. Appear as producer/consumer writes/reads from a single func Access almost always appears as many partial seq writes/reads per function invocation.	Generally several arrays per application (3-8). Appear as $S(E) = 1, 2,$ and $4B$ arrays of moderate/ large $N(E)$.	Buffer of short temporal existence. TP objects generally associated with low-leaf functions which are not callers, so buffer can be reused frequently. Wide range of sizes of objects, so flexible buffer sizing. $P_r \approx P_w$.	Likely stable with future standards.
OTHER	Access to nonarray/structure data and data not in a canonical category (such as debug access).		Small catch-all buffer which is multiported and always on, since accesses are considered unpredictable.	

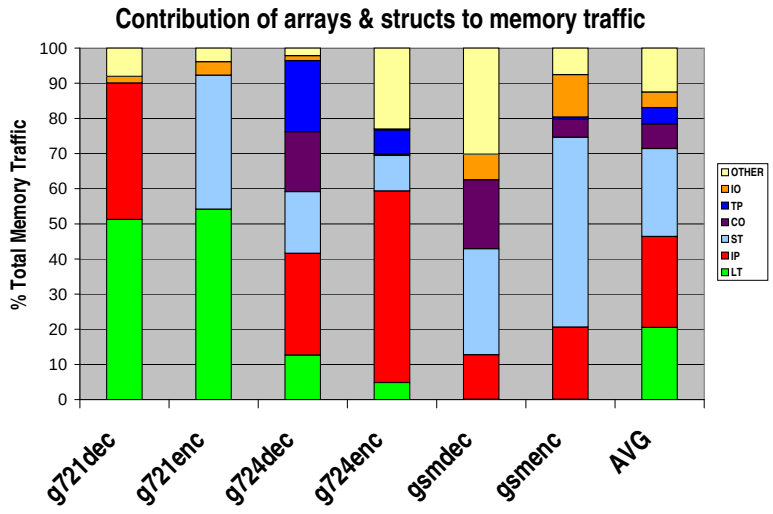


Figure 5.8 Run-time distribution of *intent* categories.

Correlation of categories in Table 5.1 to telecommunication application data variables was performed by hand; trace analysis of code scheduled for an eight-wide TI 'C64x-like machine model [57] reveals that the four primary data types, look-up tables, interprocedural communication, state, and coefficients, dominate application memory traffic. For several applications with diverse distributions, traffic categorization is quantified in Figure 5.8.

To quantify ownership relationships, Figure 5.9 shows the average and maximum number of objects referenced by each function in a selection of benchmarks. IPA conservatism causes identification of ownership relationships which do not appear at run-time; these larger ownership counts are labeled as "static objects." As correlates with the significant reduction in operation–object arcs realized through optimistic zero-weight analyis, at run time, fewer objects are touched, resulting in many fewer ownership relationships (marked as "dynamic objects").

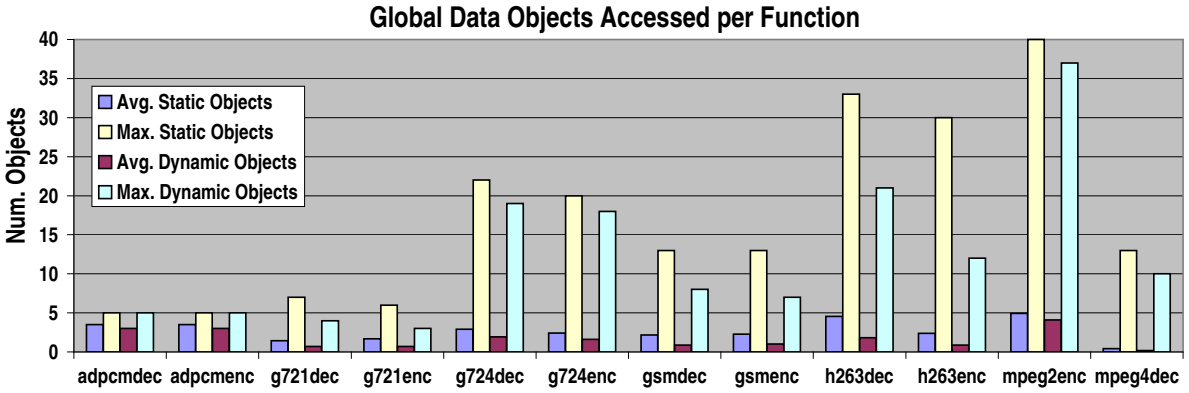


Figure 5.9 Dynamic and static function *ownership* of objects.

5.2 Application of Intent to Buffering

Rather than applying an understanding of data intent to tolerance of data access *restrictions*, this section describes focusing *improved* access latency on key application data objects. Traditional approaches to improving data access entail increasing memory bus width, providing additional memory ports, enlarging the processor register file, or duplicating register file ports. These are costly solutions, and disregard computational patterns and the resulting data access trends. In telecommunication applications, large volumes of memory traffic arise from three of the defined intent categories: arrays which store filter routine state variables, mathematical tables for look-up operations, and filter coefficients. Various telecommunication applications use these array types to differing extents, but through memory trace analysis, we observed that memory traffic is often dominated by requests to one or two relatively small arrays. Understanding usage of these constructs provides opportunities for inexpensive reduction of memory demand.

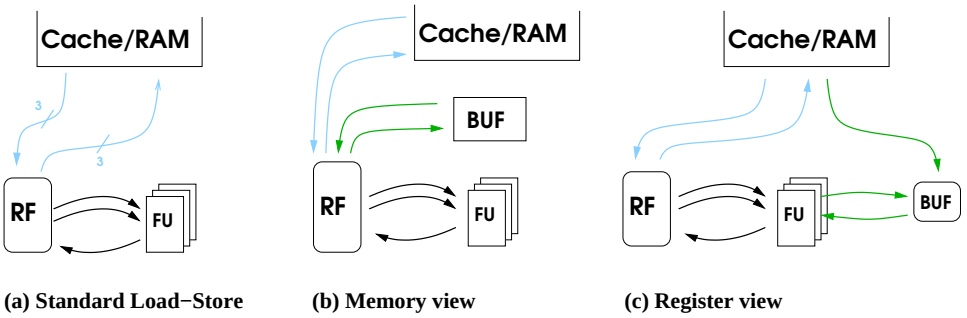


Figure 5.10 Data access models.

In contrast to "ideal" data movement through a specialized codec implemented as an ASIC, the load-store architectural paradigm (Figure 5.10(a)) specifies that an instruction set only have `load` and `store` operations to memory, and that all other operations utilize data residing in registers. For standard intraprocedural register allocation, the register file stores intrakernel values, while cross-kernel communication is routed out, through the register file, to memory. Values normally directly available to kernels in hardware, such as look-up tables and nonadaptive coefficients (constants), are also routed in and out of the register file for each routine.

This construction results in large amounts of memory traffic which are contrary to fundamental codec structure. Memory accesses (`loads` or `stores`) were found to be 17.2% of dynamic operations for the studied suite of telecommunication applications. This is far beyond the memory traffic necessary for data input to and output from a codec. If a compiler can safely identify all instructions accessing a memory location, it may assign that item to the general purpose register file. When data are allocated to the register file of a statically scheduled processor, they are accessed with shorter latency, a higher degree of parallelism, less energy consumption, and increased predictability.

In [58], we explored opportunities present in telecommunication reference codes for automatically reducing memory traffic by means of small on-chip buffers similar to compiler-managed scratchpads. Two specialized buffering structures were used to capture regular memory traffic observed during execution of telecommunication applications. Candidate variables and data structures were identified by the compiler with memory profile information and then allocated to either a first-in-first-out (FIFO) or static buffer structure residing on-chip. Consumer-producer communication streams (the interprocedural intent category) benefit from FIFO buffers, whereas filter coefficients and mathematical look-up tables may use static (read-only) buffers. Having used memory profiling tools to demonstrate the potential for reduction in memory traffic, compiler heuristics and algorithms were then developed to automatically select these critical arrays.

In this work, the term *buffer* refers to a very small on-chip structure, much smaller than the on-chip SRAM used for C-SRAM, which is the primary processor data store. The manner in which such a buffer is accessed has implications upon both processor instruction set architecture (ISA) and performance. Two methods will be discussed here: a *memory view* and a *register view* of buffer integration into a load-store architecture.

The memory view is shown in Figure 5.10(b). In this configuration, the buffer serves as an alternative data source for the register file. Data may be transfered to/from the register file out of either standard memory or the buffer. Figure 5.11(a) shows how these transfers occur: in addition to standard memory `load` and `store` opcodes, the ISA contains `ld_buf` and `st_buf` operations which initiate transfers between the buffer

and register file. Since the buffer is much smaller than a processor's memory space, `buf` operations require significantly fewer addressing and offset bits than standard memory operations. Since variable instruction set architectures have been shown to be viable and beneficial in the embedded domain [59], [60], [61], `buf` operations enable reduction in static code size.

In this memory view, computation instructions continue to access their operands from the processor register file. This approach minimizes changes to both the instruction set architecture and compiler code generation. Before scheduling, the compiler changes memory operations targeting buffer-allocated arrays into `load_buf` and `store_buf` operations. Performance improvement is realized since access to an on-chip buffer may reasonably occur within a single cycle, as opposed to the multicycle latency which must be assumed for standard memory operations. The disadvantage of this memory view is that elements from the buffer, like standard memory locations, are brought into the register file and contribute to intraprocedural register pressure.

An alternate approach to the memory view is the *register view,* in which a buffer is accessed as a register file. The register view is illustrated conceptually in Figure 5.10(c) and in more detail in Figure 5.12. This allows computation instructions to refer to buffer data via their register operand specifiers. For this work, we assume an on-chip buffer has 16 entries, making it significantly smaller than modern physical register files. Moreover, we found that a buffer requires no more than one-half the read ports of a general register file, since at most one source for an instruction will come from a buffer. Even if write ports

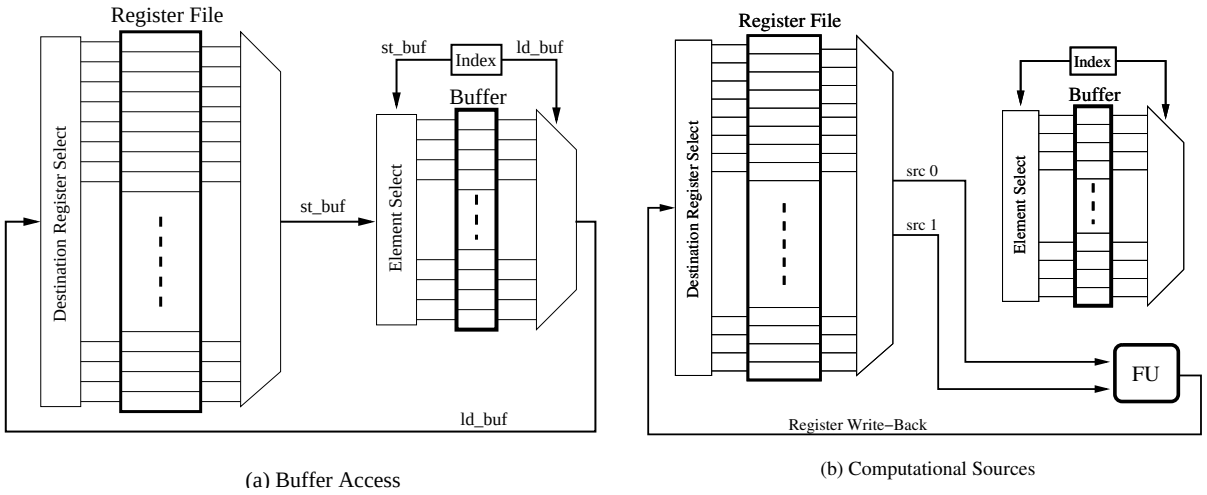


Figure 5.11 Memory buffer view.

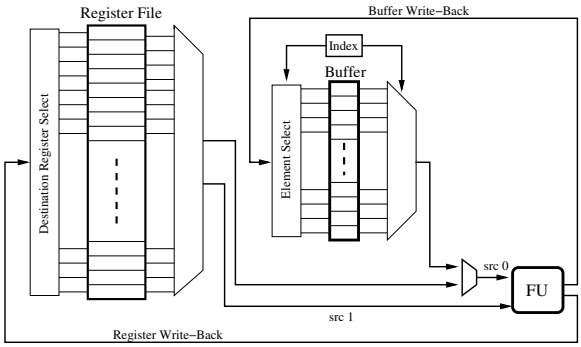


Figure 5.12 Register view: computational sources.

are still provided (*i.e.*, the buffer is not static), the size of the buffers will be significantly less than the general register file. Thus, buffer access should comfortably fit into the timing constraints of a register file access. One interesting issue is register forwarding for buffer contents. Buffer writes were found to typically be sufficiently temporally ahead of buffer reads that no register forwarding is needed, so register bypass paths from buffer writes to buffer read need not be provided.

For the register view, the static buffer is accessed with an index from a processor register file entry. This index is typically calculated by a sequence of instructions that

Algorithm 2 Identifying static buffer candidates.

1: Identify array names and lengths, forming a candidate array list, **A**
2: Tag load and store operations with array information
3: Optimize and schedule the application code
4: Profile `load` and `store` accesses to each array in **A**
5: Sort arrays by dynamic access count
6: Remove arrays with *length* greater than threshold l from candidate array list

implement a table look-up action. There could be multiple static buffers in a processor, in which case, a buffer number is needed to specify the buffer accessed. However, for the eight telecommunication applications evaluated for buffer allocation, a single buffer was sufficient to provide substantial reduction in memory traffic.

For either the memory or register view, a FIFO buffer was assumed to have standard first-in-first-out behavior which relies on two hardware pointers, *head* and *tail*. The head and tail pointers were assumed to be automatically incremented after each access. A *FIFO write* places a new element at the current tail position and automatically increments the tail pointer. A *FIFO read* returns the value at the head position and causes the head pointer to be incremented. A *peek FIFO* access returns the value at the head position without incrementing the tail pointer. Details of compiler-managed allocation of data to a FIFO buffer are available in [58].

The process used to identify static buffer candidate arrays is described as Algorithm 2. This method is interprocedural, and also includes tracking of dynamically allocated arrays, so it accounts for whole-program memory traffic. When an array is chosen for buffer allocation, all accesses to that array are rerouted to the buffer to ensure program correctness despite use of a profile-based method. An array chosen for buffer allocation

remains in its buffer for the course of program execution. The method used for array allocation thus has similarities to a pseudo-interprocedural register allocation. By specifying particular arrays to be persistent across function invocations, much of the benefit of interprocedural register allocation is achieved without the implementation and analysis complexity of true interprocedural register allocation.

Experimental results for the "memory view" buffer access model show that 27.97% of the memory accesses in these applications can be served by one or two 16-bit by 16-element buffers (Figure 5.13). This results in an average execution cycle count reduction of 7.5% (Figure 5.14), and a decrease in power consumption by 28% (Figure 5.15). In *g721enc* and *g721dec*, more than 50% of the memory traffic is captured, resulting in a 19% cycle count reduction and 50% power savings. More detailed description of buffer efficacy and derivation of the above power results are available in [58].

5.3 Application of Intent to Sleep Control

The common *ownership* relationship between data objects and high-level language routines means that performance will not be impacted if objects are put in a low-power sleep state when their owner functions are not executing. Our study of a compiler sleep management technique to exploit this property has indicated average on-time of data space will be 30%; overall memory on-time (including storage of data assumed to always be on, such as function stack variables) averages to 60%. Assuming sleep mode realizes 80% reduction in leakage power, and leakage represents one-half of total SRAM

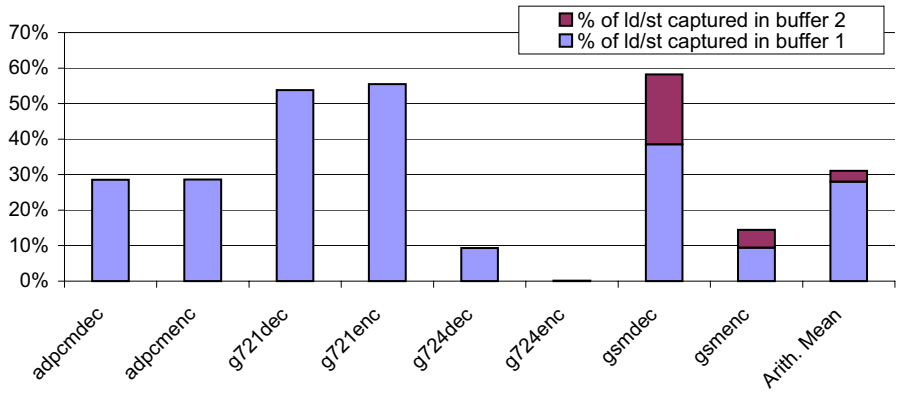


Figure 5.13 Run-time memory traffic captured by allocating key arrays to an on-chip "memory view" buffer.

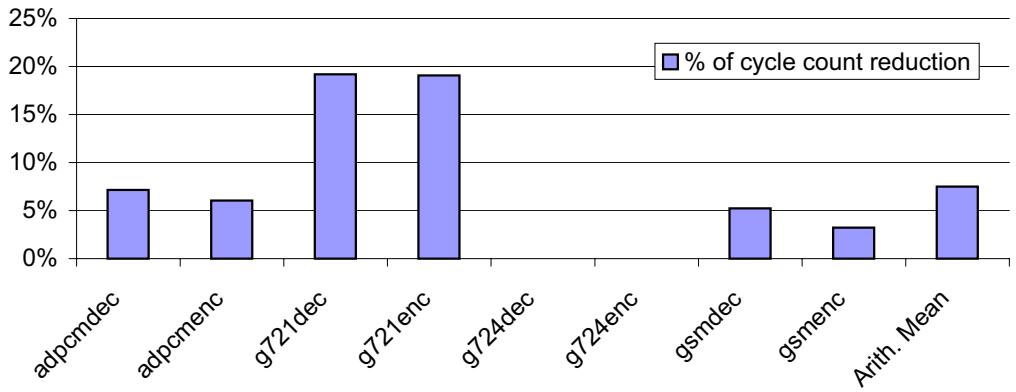


Figure 5.14 Performance improvement from allocation of key arrays to an on-chip "memory view" buffer.

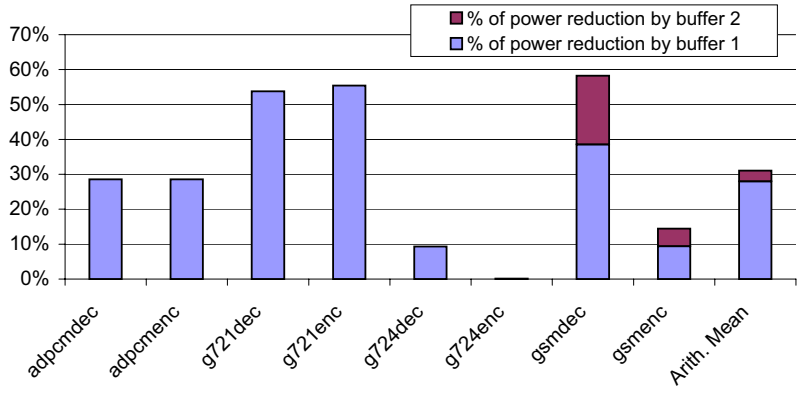


Figure 5.15 Power savings from access to an on-chip "memory view" buffer.

power, this corresponds to a 17% reduction in data storage power. This compiler sleep management technique will be termed *sleepy memory*.

The goal of allocating objects to power-controlled memory regions is to reduce the power consumption of the memory array by leveraging regularity and infrequency of data object usage in telecommunication and media codecs. Not all memory objects will be candidates for allocation in a sleepy region. For example, a data structure may be used very frequently throughout an application's execution, e.g., to pass information about current input stream properties to multiple functions. To accommodate data objects which are not suited to explicit power management, a sleepy memory needs a *standard* region which functions as a normal memory array without sleep control. Note that if C-SRAM is used, the standard region is not necessarily *None* space; it could also be single-ported or long latency. The standard area of the memory array could also be implemented with the tag and decode logic necessary to make it cacheable, or could be a software-managed cache, depending upon which is most appropriate for a particular application or processing environment. This type of scratchpad/cache partitioning was explored by Ranganathan et al. in [62].

The simple sleep mechanism studied leverages a memory profile to establish ownership relationships, much like zero-weight IPA. Program data objects with common owners are grouped into macroblocks. Groups are then put into a low-power sleep mode, and only awoken when the software turns them on at a call to their owner, or when an actual

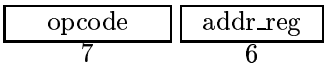
(unpredicted) access arrives. A state-maintaining sleep mode (like that provided by C-SRAM) is assumed, so during sleep data is preserved, but will take several cycles to become accessible. As is the case for C-SRAM configurations, if a data access request occurs to a region currently asleep, that request is not invalid, but its servicing is delayed until the appropriate wake time has lapsed.

Baseline support for sleepy memory requires extension of the ISA by only three operations: `Wake`, `Sleep`, and `Configure`. Necessary fields for these instructions are shown in Figure 5.16. The operations shown follow the traditional *register direct* memory addressing convention; depending upon available opcode and encoding space and address calculation resources, `Wake` and `Sleep` operations with additional addressing modes could be added to the ISA. The sleepy memory results which will be presented assume availability of register-indirect power management modes, due to the code generation mechanism used. Restriction to register direct addressing would be a minimal extension with no performance impact.

5.3.1 Compilation for sleepy memory

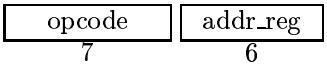
The first step in use of sleepy memory is to determine *sleep candidates*, or program data objects that are eligible for allocation in a memory region with sleep control. For this study, sleep candidates are global and heap-allocated objects. Use of a memory profile to schedule power modes makes the proposed method analogous to optimizations such as hyperblock formation, loop peeling, and loop unrolling, which are commonly

WAKE SEGMENT WS
Format: WS *addr_reg*



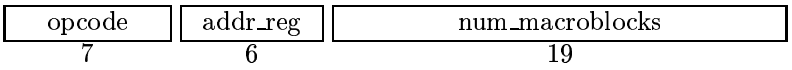
Functionality: Transition the region which begins at the address specified in Register [*addr_reg*] from its current mode to *Normal*

SLEEP SEGMENT SS
Format: SS *addr_reg*



Functionality: Transition the region which begins at the address specified in Register [*addr_reg*] from its current mode to the power-reduced *Sleep* mode

CONFIGURE SEGMENT CS
Format: CS *addr_reg num_macroblocks*



Functionality: Create a region beginning at the address specified in Register [*addr_reg*] and extending for $num\_macroblocks * macroblock\_granularity$ bytes.

Figure 5.16 Instructions for compiler-managed owner-based sleep.

implemented using execution profiles. Not all code transformations can safely be applied based upon a profile, and in fact, optimization and transformation of data access based upon profile information may be inadvisable. However, like use of optimistic IPA for C-SRAM scheduling, use of a profile to schedule region sleep control is safe because if the profile is incorrect, it will only result in a delay while the accessed region is awaken.

Algorithm 3 describes the *caller-induced* method used for sleep management. This method is analogous to caller-saved register allocation, in which a routine saves the registers it needs before calling a subroutine. In this case, however, regions currently

being used are put into a low-power mode as a subroutine is called. This potentially causes code size expansion due to insertion of additional instructions around each call site, but in practice, this effect is negligible.

Of greater concern is the performance penalty potentially incurred in invoking sleep and awakening regions. In the embedded domain, many processors have branch delay slots, and compilers are often unable to completely fill these slots. This creates bubbles in the scheduled code which negatively impact performance. The number of delay slots is generally between three [31] and five [28]. With caller-induced sleep, the branch delay slots following *Jump to Subroutine* and *Return from Procedure* (`JSR and RET`) operations can be filled with `Sleep Segment` (`SS`) operations. This thus hides the additional instructions necessary for putting memory regions into low-power mode when subroutines are invoked.

The average and maximum number of `Sleep Segment` operations required per subroutine call are shown in Figure 5.17. Values shown in this graph are run-time counts, and thus give a fairly accurate picture of the penalty of caller-induced sleep. The average number of sleep operations per subroutine call is very low, generally no greater than two or three, which corresponds nicely with the number of delay slots generally present following a `JSR` branch. It was also found that the average distance between a function call and data use is 4 cycles. Thus if delay slots are already full, `Wake` and `Sleep` operations can instead be merged into function header code.

Algorithm 3 Caller-induced sleep state

```
1:  for all Objects in the program do
2:    Determine candidate utilization
3:    if utilization < utilization_threshold then
4:      if candidate_size <= remaining_space then
5:        Candidate will be allocated to a sleepy region
6:        Insert Configure Segment operations into beginning of program
7:        for all Routines which access this object candidate do
8:          Insert Wake Segment operations at beginning of routine
9:          At each JSR, insert Sleep Segment operations
10:       end for
11:     end if
12:   end if
13: end for
```

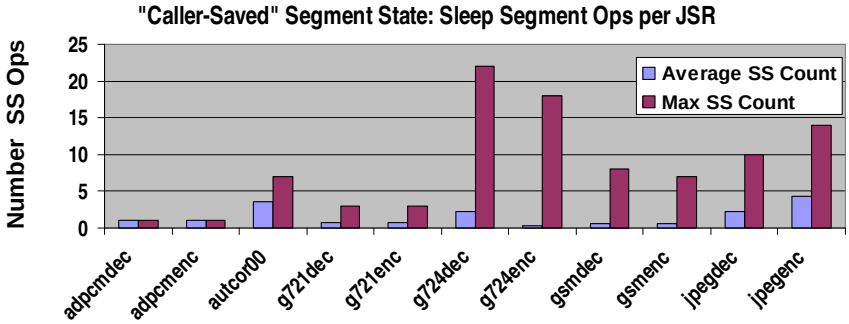


Figure 5.17 Sleep operation statistics.

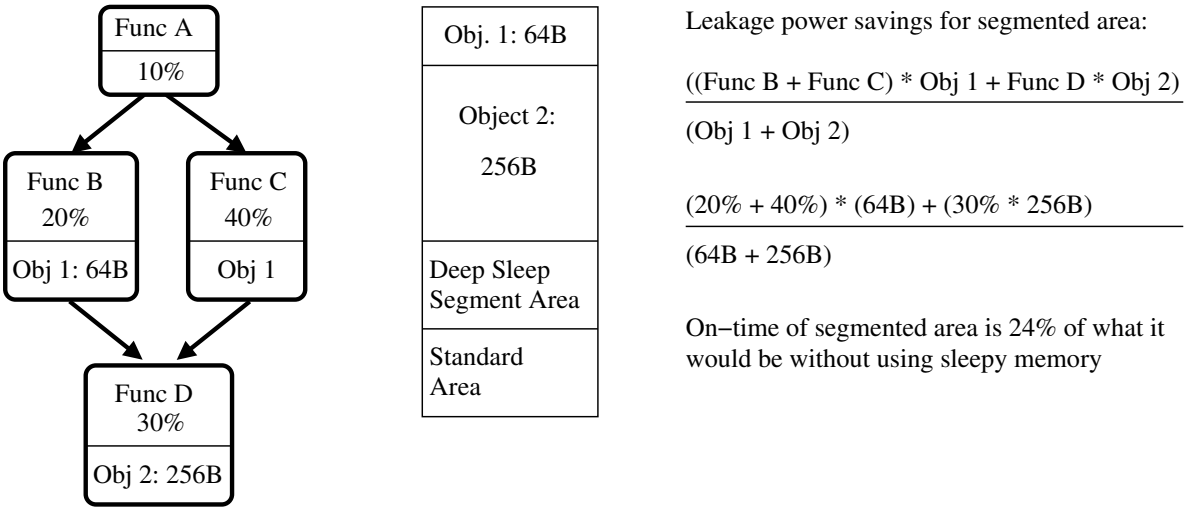


Figure 5.18 Sample sleepy memory allocation and utilization.

Figure 5.18 gives a simple example calculation of utilization, or awake time. Each function (A, B, C, and D) is shown with the percentage of program execution time it accounts for during an evaluation run of its application. Functions B and C, for example, each access Object 1, so this 64 byte object must be awake for 60% of the total application execution time.

For program-wide memory `Wake` and `Sleep` scheduled according to Algorithm 3, net utilization of memory space is shown in Figure 5.19. For a single object, utilization is calculated as

$$object\_utilization = \frac{num\_cycles\_awake}{tot\_program\_cycles} \tag{5.1}$$

Utilization is a weighted metric, whereby

$$memory\_utilization = \sum_{all\_objects} object\_utilization * object\_size \tag{5.2}$$

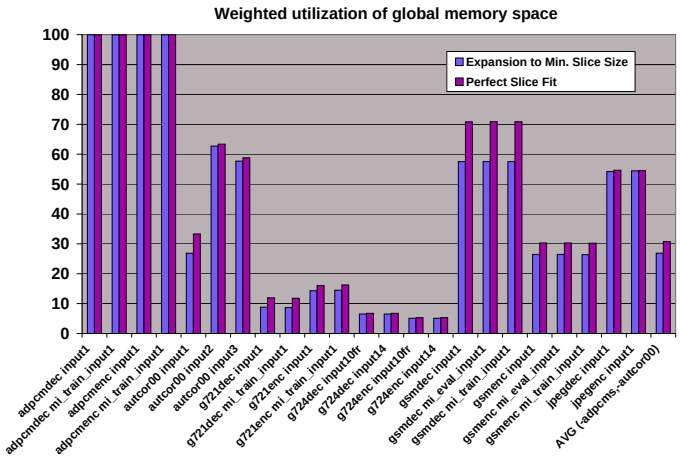


Figure 5.19 Macroblock utilization with "caller-induced" sleep.

Utilization for the lighter bars in Figure 5.19 (*Perfect Region Fit*) is weighted by the exact object size, which implies a perfect fit of objects to their sleepy regions. For the sake of simplifying region control hardware, it seems prudent to assume a minimum sleep *granularity*. The effect of a minimum sleep size on utilization can be seen in the dark er bars in Figure 5.19 (*Expansion to Minimum Segment Size*) for a sample sleepy region size of 32 bytes, or two lines for common embedded SRAM sizing. A small granularity is used here for the sake of comparison against hardware-managed sleep modes like the Drowsy Cache [11]. Sleep could also be compiler-controlled at larger granularity, e.g., within the C-SRAM framework at macroblock boundaries. Effective use of this technique at large granularities would require more in-depth layout optimization than included in the scope of this work.

Total memory utilization (both sleepy and standard space) is calculated assuming that memory space used to accommodate function stacks is always on. The sizes of all

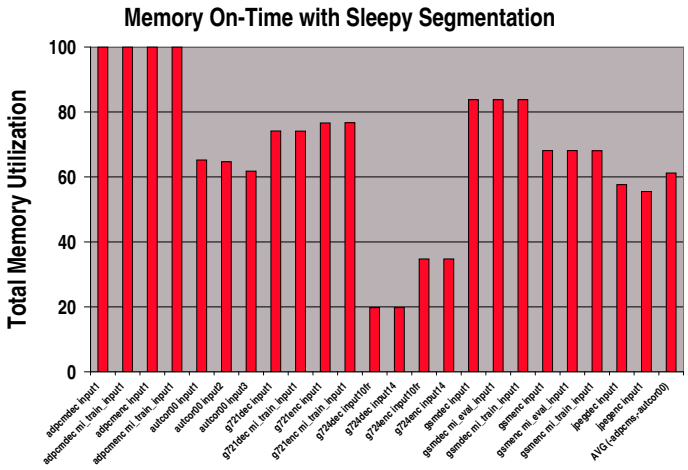


Figure 5.20 Total utilization of all data space with caller-induced sleep.

function stacks were recorded, and based upon the execution trace, the maximal run-time sum of function stack space is determined for each input run. The input with the largest total live stack space is used as the baseline for determination of total memory utilization, where

$$tot\_mem\_util = \frac{(tot\_sleepy\_util * tot\_sleepy\_size) + (100\% * max\_stack\_size)}{tot\_sleepy\_size + max\_stack\_size} \quad (5.3)$$

These results are shown in Figure 5.20.

5.3.2 General applicability

Sleepy memory is not exclusively an embedded technique – it could reside at any level of a processor memory hierarchy. For the telecommunication and media applications for which results are presented above, region allocation is done assuming an embedded context in which the sleepy memory is the nearest level of the memory hierarchy. For

scientific or other nonembedded classes of applications which exhibit regular activity patterns, but have much larger data objects, sleepy memory could be implemented at a further (e.g., L3) cache level, where software management of multiline regions could replace or supplement prefetching. As the power consumption of caches becomes of increasing concern, use of a power-managed sleepy structure could be used to alleviate power concerns, while maintaining favorable performance.

6 RELATED WORK

6.1 Cache Adaptation

Configurable SRAM is a form of adaptation to application characteristics. A *Complexity Adaptive Processor* whose resources adapted to the needs of an application was coined in 1998 by Albonesi [63], [64]. Adaptive processing was originally primarily envisioned for performance improvement, but more recently, adaptive cache designs to improve memory system power consumption have been proposed.

Most have targeted the expense of cache associativity by allowing ways to be disabled during periods of high cache hit rates. Among these proposals are [1], [65], and [66]. Albonesi proposed splitting caches into vertical subarrays (i.e., using wordline partitioning) to be able to change the degree of associativity at little cost to the cache array area [65].

While Albonesi envisioned software control of the cache's associativity at the application granularity, Balasubramonian et al. [66] expanded on Albonesi's design and proposed automated dynamic detection of application needs and program phases. Balasubramonian et al.'s cache is a subbanked, joint L1/L2 structure in which the sizes, associativities,

and latencies of the two levels can be varied. Using hardware performance monitor counters for branch frequency, IPC, and miss rate, the proportion of the cache array dedicated to each cache level is varied.

The *accounting cache* is described in [67]. A requested item is first sought via a *primary access* to a limited number of ways, and if not present, then a *secondary access* probes the remaining ways for the datum before being sent to the next level of the cache hierarchy. An LRU approximation is used to disable unneeded ways of the cache, rather than software instructions [65] or performance counter data [66].

Zhang et al. [1] adjust cache associativity not by shutting down (turning off) ways, but rather by *concatenation*. This reduces associativity and allows the full array to be utilized while still saving dynamic power, because fewer ways are accessed per datum request. Specifically, Zhang et al. use way shutdown to save static leakage via gated-V_{dd} circuitry.

Veidenbaum et al. [68] note that optimal data cache line size (as measured by the amount of memory traffic generated and the miss rate) changes over the course of an application's execution. Veidenbaum et al.'s mechanism uses hardware monitoring to adjust cache line size dynamically based on application accesses to the line. This is achieved by building variable-sized *virtual lines* from short *physical* cache lines.

Each of these adaptive cache methods only explores one dimension of configuration, and thus does not have the potential and flexibility of the evaluated set of SRAM configurations. Previous approaches are generally hardware methods, and so introduce performance and power variation into the system. With the exception of [1], the above studies consider general-purpose workloads, and none provides a clear explanation of the relation between application data usage properties and the efficacy of their adaptation.

The FlexCache uses IPA and other compiler support to improve software-managed caching [69]. SRAM tags are managed in software, allowing application-specific line sizes, associativity, and replacement policies with a five cycle access latency. The compiler may also allocate data to noncache space (not virtual memory mapped) with one cycle latency, but this is only done for scalar register spills and fills. The FlexCache work is thus focused on static analysis techniques to understand locality, rather than memory operation flexibility.

6.2 Cache Sleep

6.2.1 Hardware sleep management

While we are not aware of previous work which leverages concurrent configuration of latency and ports, there are several existing mechanisms for reducing SRAM leakage by controlling cache sleep states, including [6], [11], and [16]. Some of these techniques have only been demonstrated for instruction caching and most rely on hardware mechanisms.

The drowsy cache [11], [70] periodically puts all lines into a data-maintaining state, and then incurs a cycle penalty to wake accessed lines. Decay caches [16], [71] accelerate eviction of unused cache lines by turning off lines containing data which has not been recently accessed. The decay cache is based on use of run-time counters, not a software management mechanism. A hardware-based per-line mechanism masks the inefficiency of cache replacement algorithms, rather than substituting a new mechanism for hardware caching. Decay is not applicable to noncache SRAM because the sleep mode used does not maintain data values. Both decay and drowsy caches use per-line control mechanisms so incur significant configuration area expense in the memory array design. Both techniques also entail performance loss which is not predictable, making them less attractive for the embedded domain.

6.2.2 Compiler-controlled cache sleep

Zhang et al. [72] propose a compiler mechanism to place lines of an instruction cache into either a state-preserving or state-destroying mode by dynamically scaling supply voltage. The technique only applies to instruction caching, and does not address the power consumption of data storage.

We are aware of only one work that takes a compiler-based approach to data cache sleep management [6]. Data-reuse analysis is leveraged to place data that will not be used by current computation into a state-preserving sleep mode. Zhang et al. [6] discuss the effects of several code transformations on the performance and power savings of their

technique, but their overall performance degradation is not predictable and can be worse than that of hardware-managed cache sleep techniques. Their approach also significantly differs from ours in that theirs does not statically resolve object relationships, does not reschedule code to achieve further power savings, and does not necessarily achieve power savings without performance loss.

6.3 Embedded Compiler Technology for Memory Optimization

In contrast to the automation-driven approach of this work, compilation technology for most embedded platforms require much more programmer intervention. Many compilers require *pragmas*, or programmer directives, to trigger optimizations or force scheduling for special DSP features, particularly for data storage components. As an example, high-level language use of the Analog Devices BlackfinDSP `n`-element circular buffer requires the non-standard C/C++ notation `my_array[i%n]` [73]. If HLL source is obtained from a vendor or written by a developer other than the engineer responsible for code compilation, addition of pragmas to optimize memory usage may be difficult. By leveraging accurate interprocedural information and automatically controlling operation scheduling and data placement, this dissertation thus provides an important step forward in embedded compilation for low-power devices.

Profiling has been leveraged to improve C-SRAM power, but is not yet ubiquitous in the embedded domain. Profilers have been provided in several commercial embedded development packages, most notably through GreenHills, Texas Instruments, and Analog

Devices. Texas Instrument's tools allow monitoring of code coverage and cache performance, so represent a form of memory profiling [74], [75]; the compiler package for the Blackfin DSP includes a tool for parsing and displaying program execution profile information at function granularity [73]; and the Green Hills ST100 DSP family CodeBalance tool [76] allows developers to enter constraints for tuning the performance versus code size trade-off of the ST100 variable-width instruction set architecture. The control flow profiling used for this dissertation has finer granularity (extended basic blocks, including exit branch flow weights) than that for the Blackfin processor, and the implemented memory profiling tools supply more detailed, and more general, information about data usage than those provided by Texas Instruments.

A survey of software-based embedded memory optimization techniques is available in [77]. The authors note that embedded memory optimization differs fundamentally than that for the general-purpose domain because embedded processors are domain-specific and can fairly be tailored to observed application profile properties. These observations support the domain-specific and feedback-directed approaches taken in this work.

6.4 Scratchpad Allocation

Differentiated `load` servicing through configurable SRAM is related to studies of data partitions for embedded systems with *scratchpad* resources. In these systems, a small, low-power, software-managed SRAM is used to cache variables under programmer or compiler control. Scratchpad based compiler-managed caching has been well-explored

from the perspective of software heuristics to minimize the average latency of memory accesses. While the objective of data allocation is thus different (performance, not power), the two approaches of closest relation to this work are (a) [78], which assigns objects to a scratchpad (non-hardware-cache SRAM) on the basis of data usage profiles, and (b) [79], which assumes a mixed embedded DRAM/scratchpad SRAM/cache system. Both are explained in more detail below. Thorough summaries of other work in compiler-controlled caching are provided in [80] and [81].

Static allocation to scratchpad (e.g., [78], [79], and [82], is a necessary compiler technique for chipsets that include on-chip software-managed SRAM to reduce data access latency, but is a less flexible approach to power savings than configurable SRAM because the size of the scratchpad and DRAM cannot be changed. Sjodin et al. [78] assign objects to on-chip RAM according to their frequency of use. Nonscalar global objects are allocated based upon a static profile estimation. Scalars and non-globals are not addressed, and Steensgaard pointer analysis is used. Panda et al. [79] assume embedded DRAM or SRAM is available as a one-cycle scratchpad memory. They find that in general, joint scratchpad and cache memory provide the best performance solution. Their partitioning strategy is based on variable types: (1) scalars and constants are always allocated to the scratchpad; (2) arrays larger than the scratchpad size are allocated to global memory; (3) arrays with overlapping lifetimes are mapped to separate spaces so as to minimize cache conflicts; and (4) variables that are frequently accessed, and within whose lifetime many other variables are accessed also, are allocated to the scratchpad. If programs

are executed as threads, the scratchpad space is assumed to be partitioned into sections dedicated to each program, thus minimizing context-switch overhead because program scratchpad regions are independent of one another. This is the same assumption made for standard C-SRAM usage.

In their study of scratchpad storage, Sjodin et al. [78] found that for small programs, static allocations (i.e., data placement choices made at compile-time) are sufficient. However, large programs require precise profile information, and do not realize a significant drop in off-chip memory traffic because they tend to have a greater number of memory accesses to non-global data. This emphasizes the importance of the technique described in Section 3.5.1 for configuration of dynamically allocated data, as well as the ability to perform page-based power configuration. Dynamic approaches to scratchpad allocation [54], [83], in which the values in the scratchpad change as the program executes, have a somewhat different focus than this work because they are concerned more with accommodating temporal usage patterns than determining and scheduling for whole-program power savings opportunities. However, as described in Section 3.5.5, a dynamic allocation algorithm could be used to ensure that data is in C-SRAM.

6.5 Custom Partitioned Data Layouts

Several approaches to custom memory partitioning are described in [4]. These include specialized memory hierarchies, partitioning such that frequently accessed variables appear in a small memory, and grouping variables that are accessed temporally close to one

another so as to enable sleep modes when a partition's variables are not needed. Some previous work assumes that data access patterns are known and only explores the partitioning problem; other work evaluates approaches using just computation kernels. Our ability to provide detailed memory profile information for applications allows exploration of memory performance and power techniques using high-level language code and whole applications.

The problem of allocating data to memory such that partitions are accessed infrequently is formulated in [84]; Farrahi et al. suggest that simulation data be used to determine temporal and spatial use of program memory, however their proposals are evaluated on randomly generated data, not actual application code. Profile information is used in [85] to group memory objects into categories according to frequency of access. Frequently accessed data is then placed in small SRAMS, while less commonly used data remains in large memories. While these techniques realize differentiation in `load` access properties, they break the memory into separate entities.

DTSE (Data Transfer and Storage Exploration) is a CAD tool to simplify data allocation to disjoint memories during the process of custom chip design [13], [86], [87]. It includes the *Atomium* toolset, which can be used to perform source-to-source C code transformations for simplifying custom multimedia processor design. Source-to-source transformations to, for example, ungroup array accesses could improve utilization of

configurable SRAM. The DTSE tools, however, require user input to specify transformations, memory hierarchy properties, and timing constraints; and many assumptions are also made about how subsequent CAD tools/compilers process C code.

The DTSE designers have also proposed including floorplan-based memory latency variation in custom memory designs [88]. Papanikolaou et al. [88] suggest that the most heavily accessed memories in a custom layout should be placed closest to the processing datapath so as to have the shortest wiring. Their latency variation, however, is not adjustable in the field because it results from differences in physical routing length.

6.6 Data Relayout for Cache Performance

Cache performance may be improved if data is stored in the most common program access order. In the embedded domain, data layout has been primarily addressed with compiler techniques; for general media applications, hardware remapping [89] has also been proposed. Compiler-based data remapping for embedded systems is explored in [90], but pointer-intensive applications are targeted. We have not found pointer chasing to be a problem in telecommunications or media algorithms. Palem et al. [90] state that dynamic memory remapping is too costly for embedded systems, so compiler techniques are necessary. Rather than changing algorithmic memory access patterns to fit a constrained memory system, Palem et al. aim to improve data layout relative to the application access sequence [90]. Though the technique is automated, handles dynamically allocated-data,

and is profile-based, it addresses different problems than we have encountered, so will not be discussed further here.

The goal of the Impulse project is to make the layout of data in memory more closely resemble the order in which it is needed for computation [89], [91]. The Impulse memory controller serves as a detached interface between the processor chip and off-chip memory (DRAM). The memory controller remaps addresses from disjoint physical and virtual memory locations to contiguous *shadow space* addresses, and the on-chip caches may then be filled with these tightly packed accesses. Though it deals with a different level of the memory hierarchy than that addressed in this dissertation, a remapper like Impulse could open opportunity for use of configurable SRAM in a more general purpose processing domain because the properties of access to remapped data would be better known than those of the general application.

In the general-purpose domain, profile guidance has been employed for data re-layout to increase hardware cache spatial locality [92], [93]. These works minimize cache misses (i.e., average access latency), given a fixed instruction stream and caching mechanism. Power is only impacted peripherally as L2/L3 request servicing is diminished – performance is the primary target. Our integration with the static instruction scheduler instead allows modification of operation-level data access behavior (in terms of expected latency and port resource utilization). Rather than mitigating the effects of hardware, the configurable SRAM approach changes hardware behavior so as to directly, and predictably, save power.

Profile-guided data layout optimization has also been proposed for dynamically allocated data [94]. A memory allocation library is proposed to enable profile-guided data layout. The two optimizations evaluated are field reordering and instance interleaving (placing identical fields from different instances of a data structure near one another in memory). Benchmarks are from the Olden suite, the Worchester Polytechnic Institute benchmark suite (WBS), and other sources, and include a ray tracing and virtual image rendering application.

6.7 Performance/Power Trade-Offs: Other Approaches

The implemented compiler techniques for configurable SRAM allow a trade-off to be made between application performance and memory system power consumption. As explained in previous sections, most existing memory subsystem power management approaches make these trade-off decisions at run time in hardware, so they are not explicitly controlled. System performance versus power consumption trade-offs have, however, been examined in light of increased CPU-cache bus power consumption in deep submicron technologies [95]. Increased bus power consumption is anticipated as a result of the increasing $wire : gate$ capacitance ratio as feature sizes become smaller. Givargis et al. [95] recommend careful choice of design parameters so as to correctly balance the cache and bus performance/power trade-off.

6.8 Technology: Leakage Control

An overview of static (design-time techniques related to transistor properties and sizing) and dynamic (in-the-field adjustments to circuit voltages) approaches to leakage power reduction is presented in [10]. Heo et al. [10] term the bit-line floating technique leveraged for port turn-off (see Section 2.2) *leakage biased bitlines*. Using leakage biased bitlines to shut-off all access to particular SRAM banks, they achieve 24% L1-I-cache and 50% register file leakage energy reduction. Their control methodology requires predictability of access, so only the instruction cache was considered. Configurable SRAM provides power savings when access can be limited to just one port (it is not required that all ports be turned off), and the implemented compiler technology extends the applicability of bit-line floating to include data storage.

Leakage has been shown to be exponentially dependent on temperature [96]. Due to this high correlation, *activity migration*, in which computation is migrated to a cooler section of the die when heat thresholds are exceeded, is shown to potentially reduce power by around 10%. For configurable SRAM, any macroblocks configured for lower power will have lower operating temperatures, but in particular, voltage scaling to increase latency will result in cooler portions of SRAM. To maintain a favorable overall SRAM temperature, and thus further reduce leakage, cooler (long-latency) regions could be interleaved with regions configured for faster access.

6.9 Pointer Analysis

Most approaches to scratchpad data allocation have avoided the difficulties associated with pointer use in high-level language codes. Scratchpads have been used, for example, for spill code [7], or proposals have been evaluated primarily on the basis of kernels and so do not handle benchmarks with high-level language pointer usage.

The scheduling process described in Chapter 3 can make use of an optimistic subset of the possible pointer relationships. While the optimistic result is obtained through a different process, the concept of using optimistic pointer information has been used for guiding data speculation for EPIC processors [97], [98]. However, zero-weight exclusion analysis uses control flow profiling in conjunction with an accurate pointer analysis process instead of memory dependence or alias profiling to obtain the approximation. The final representation using dependence objects is similar to the location sets (LOCs) used by Ghiya et al. [99], differing only in the naming of heap objects and the representation of structure field accesses.

6.10 Memory Characterization

Characterization of memory access types by both intent (i.e., look-up table, coefficient, intermediate data, etc.) and characteristic patterns (stride/random) appears to be unique. One work worth mentioning, however, is [100], which characterizes sources of cache load misses. Using Olden, SPEC2000, and pointer-intensive benchmarks, Sair et al. [100] classify load misses into one of four miss access patterns: next-line, stride,

same-(heap)-object, and pointer-based. Each of these patterns corresponds to previously proposed prefetching methods, and it is found that these categories cover greater than 90% of benchmark cache misses. Sair et al. suggest that a categorization of load miss behavior may be useful in guiding profile-based optimization, and propose a hardware mechanism for collecting a load miss profile at run time. Their work does not evaluate telecommunication or media algorithms, so its categorization does not provide the level of detail we contend is necessary for effective power and performance optimization in the target domain. Pointer-chasing, for example, is not a problem in the studied application set, and observation of same-heap-object cache misses does not necessarily imply anything about the predictability of those misses.

Memory characterization and analysis studies have been published for several classes of benchmarks. These studies, however, focus on simulator results in and of themselves, rather than use of profile information to derive new data storage designs or improve the compilation process. Work by Fritts [101], Fritts et al. [102], and Fritts and Wolf [103] has provided general characterizations of such application properties as instruction cache size for given cache hit rates, and data working set sizes. Berkeley's Multimedia Workload is characterized in [104].

In the multimedia application space, Berg [105] provides a characterization of data flow patterns as: horizontal (access to a single row), vertical (column access), or blocked (output of an image block). However, the focus of his classification is on the input/output

reads and writes of images before and after each kernel, and because kernels are considered in isolation, a more complete understanding of application data flow and access patterns is not presented. The interprocedural characterization presented in this dissertation goes beyond Berg's by putting kernel memory traffic behavior in the context of a whole application, where data is not just read and written once at the boundaries of kernels; adding a temporal dimension to memory traffic analysis; and applying application understanding to power reduction.

Latency tolerance of `loads` in eight SPEC95 benchmarks was evaluated by Srinivasan and Lebeck [106]. Tolerance was measured for an out of order dynamically scheduled superscalar processor by allowing run-ahead execution until a load's results were needed. Their approach does not leverage or provide compiler analysis, and the notion of tolerance is not applied to change program execution or data access properties.

In a novel approach to custom memory design, Grun et al. [107] have used data access properties to guide custom memory design. Their method uses a profile to divide variables into two categories: those exhibiting temporal locality, and those likely to benefit from spatial locality. These variables are then divided into two custom-sized caches, one with short lines; the other with long lines. They thus use the idea of data properties guiding memory optimization, but look at only two basic dimensions, only consider single-application SoCs, and do not directly target power consumption (reduction in power is only a side-effect of higher cache hit rates due to partitioning).

7 SUMMARY AND FUTURE WORK

As system and software complexity increase, future power mitigation techniques will be successful only if their design effectively coordinates multiple layers of the system hierarchy. This work has proposed and evaluated configurable SRAM, a means for leveraging low-power circuit techniques to expose on-chip SRAM port, latency, and sleep configuration to software. While maintaining applicability to real-time and performance-constrained embedded systems, C-SRAM uses compiler analysis to jointly control data power management and operation scheduling, requiring no change in the system programming model, and providing means for stack and heap data power savings. Circuit, microarchitecture, compiler, and programming constraints were all considered in the design, realizing an effective hardware–software collaboration for power savings.

The call-outs in Figure 7.1 summarize contributions made by this work in the context of data storage power and optimization. Each quadrant lists several previous approaches to power savings, classified by their primary control mechanism and power target. Previous approaches were explained in greater detail in Chapter 6; each generally falls into just one control/power quadrant.

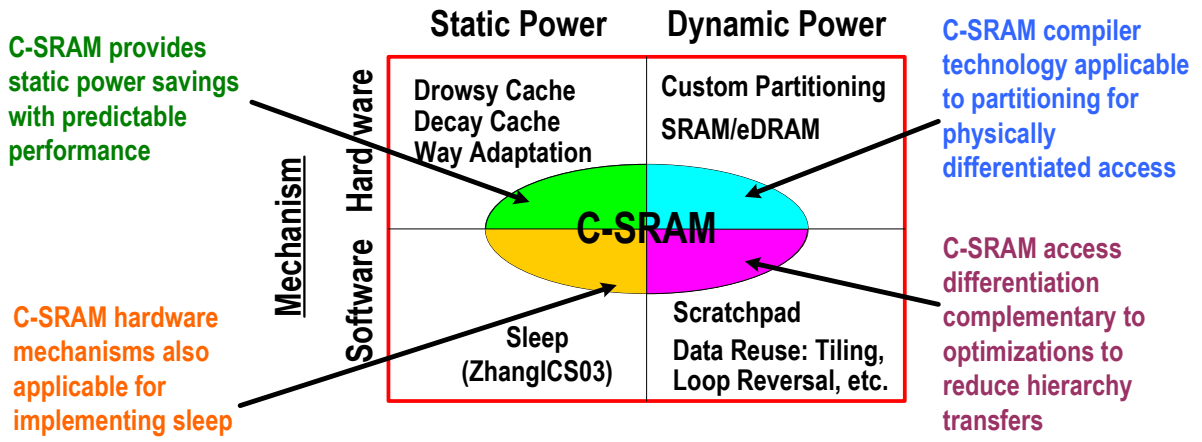


Figure 7.1 Domain space.

As technology advances, static and dynamic power must each be considered, and as application complexity increases, effective power management will rely on hardware–software collaboration. General-purpose approaches to data storage power reduction have focused on caches, using hardware-managed heuristics to turn off cache lines and ways, realizing primarily static power savings. Previous approaches to embedded power optimization reduced dynamic power by customizing memory sizes or dividing accesses between SRAM and embedded DRAM at design time. The hardware component of the C-SRAM approach (control hooks for changing access latency and number of available ports) presents opporunity for both static and dynamic power reduction, and provides a sleep mechanism for further power savings. Unlike previous approaches at the top of Figure 7.1, C-SRAM is explicitly managed by the compiler, so maintains applicability to real-time and performance-constrained systems. The analysis components of the C-SRAM compiler can also be leveraged to guide custom partitioning, but the compiler's

control is greater because operation scheduling is integrated with data management decisions.

Object scheduling is based on a solid foundation of scalable, accurate interprocedural pointer analysis, allowing more precise data control than the previously published compiler approach to data sleep management (Zhang et al. [6], bottom left quadrant of Figure 7.1). C-SRAM is complementary to scratchpad partitioning algorithms (they ensure that data will be in on-chip SRAM), and may be used in conjunction with performance optimizations, including those aimed at improving data reuse.

Future work

The algorithms and prototype compiler described in Chapter 3 are effective in realizing data power savings, but there are several extensions of this work which would likely realize additional benefit. First, it can be predicted that modulo scheduling will provide an efficient means for tolerating additional load latency by adding stages to the modulo schedule. In most cases observed in the studied applications, the initiation interval should remain the same height, and only a small performance loss will be incurred in the loop prologue and epilogue.

Second, conjoined objects limit the scheduler's ability to place important objects into lower-power zones. Such situations can occur when the same procedure is called with many different parameters, as is sometimes the case with math and other utility routines in the studied application set. There is an opportunity to either create specialized copies

of such multiuse procedures, thereby separating the uses of the objects, or, for read-only objects, replicating the objects for each consumer procedure. By cloning the procedures or objects, scheduling decisions can be made independently. Where the number of accesses and cycle counts are high enough to amortize the cost of re-configuration, it may also be possible to specialize per-function or per-region object access latencies or port restrictions, and then specify reconfiguration to occur at run time.

REFERENCES

[1] C. Zhang, F. Vahid, and W. Najjar, "A highly configurable cache architecture for embedded systems," in *Proc. 30th Annual Int'l Symposium on Computer Architecture*, June 2003, pp. 136–146.

[2] P. Petrov and A. Oralioglu, "Power efficient embedded processor IP's through application-specific tag compression in data caches," in *Design Automation and Test in Europe Conference*, 2002, pp. 1065–1071.

[3] M. Huang et al., "L1 data cache decomposition for energy efficiency," in *IEEE/ACM Int'l. Symp. Low Power Electronics and Design*, 2001, pp. 10–15.

[4] L. Benini, A. Macii, and M. Poncino, "Energy-aware design of embedded memories: A survey of technologies, architectures, and optimization techniques," *ACM Transactions on Embedded Computing Systems*, vol. 2, pp. 5–32, Feb. 2003.

[5] J. Glossner et al., "Trends in compilable DSP architecture," in *Proceedings of the IEEE Workshop on Signal Processing Systems (SiPS)*, 2000.

[6] W. Zhang, M. Karakoy, M. Kandemir, and G. Chen, "A compiler approach for reducing data cache energy," in *Proceedings of the 2003 International Conference on Supercomputing*, 2003, pp. 76–85.

[7] K. D. Cooper and T. Harvey, "Compiler–controlled memory," in *Proc. 8th Int'l Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '98)*, 1998, pp. 100–104.

[8] W. Tang, R. Gupta, and A. Nicolau, "Power savings in embedded processors through decode filter cache," in *Proceedings of Design, Automation and Test in Europe*, 2002.

[9] A. Bhavnagarwala, S. Kosonocky, M. Immediato, D. Knebel, and A. Haen, "A pico-Joule class, 1 GHz, 32 KByte x 64b DSP SRAM with self reverse bias," in *Proceedings of the 2003 Symposium on VLSI Circuits*, June 2003, pp. 251–252.

[10] S. Heo, K. Barr, M. Hampton, and K. Asanović, "Dynamic fine-grain leakage reduction using leakage-biased bitlines," in *Proceedings of the 29th Annual International Symposium on Computer Architecture*, May 2002, pp. 137–147.

[11] K. Flautner, N. S. Kim, S. Martin, D. Blaauw, and T. Mudge, "Drowsy caches: Simple techniques for reducing leakage power," in *Proceedings of the 29th Annual International Symposium on Computer Architecture*, May 2002, pp. 148–157.

[12] P. Shivakumar and N. Jouppi, "CACTI 3.0: an integrated cache timing, power, and area model," Western Research Laboratory, Tech. Rep. WRL 2001/2, Aug. 2001.

[13] F. Catthoor, Ed., *Unified Low-Power Design Flow for Data-Dominated Multi-Media and Telecom Applications.* Boston, MA: Kluwer Academic Publishers, 2000.

[14] A. Bhavnagarwala, S. Kosonocky, and J. Meindl, "Interconnect-centric array architectures for minimum SRAM access time," in *Proceedings of the 2001 International Conference on Computer Design (ICCD '01)*, 2001, pp. 400–405.

[15] T. Juan, J. Navarro, and O. Temam, "Data caches for superscalar processors," in *Proceedings of the International Conference on Supercomputing*, 1997, pp. 60–67.

[16] S. Kaxiras, Z. Hu, and M. Martonosi, "Cache decay: Exploiting generational behavior to reduce cache leakage power," in *Proceedings of the 28th Annual International Symposium on Computer Architecture*, June 2001, pp. 240–251.

[17] A. Agarwal, H. Li, and K. Roy, "DRG-Cache: A data retention gated-ground cache for low power," in *Proceedings of the 39th Design Automation Conference*, 2002.

[18] C. Kim and K. Roy, "Dynamic Vt SRAM: A leakage tolerant cache memory for low voltage microprocessors," in *Proceedings of the International Symposium on Low Power Electronics and Design (ISLPED'02)*, 2002, pp. 251–254.

[19] J. S. Caravella, "A low voltage SRAM for embedded applications," *IEEE Journal of Solid-State Circuits*, vol. 32, pp. 428–432, March 1997.

[20] S. Yang and B. Falsafi, "Near-optimal precharging in high-performance nanoscale CMOS caches," in *Proceedings of the 36th Annual International Symposium on Microarchitecture (MICRO-36)*, Dec. 2003, pp. 67–78.

[21] L. Villa, M. Zhang, and K. Asanović, "Dynamic zero compression for cache energy reduction," in *Proc. 33rd Int'l Symp. on Microarchitecture*, 2000, pp. 214–220.

[22] H. Li, C.-Y. Cher, T. Vijaykumar, and K. Roy, "VSV: L2-miss-driven variable supply-voltage scaling for low power," in *Proceedings of the 36th Annual International Symposium on Microarchitecture (MICRO-36)*, Dec. 2003, pp. 19–28.

[23] D. Marculescu, "On the use of microarchitecture-driven dynamic voltage scaling," in *Proceedings of the Workshop on Complexity-Effective Design (WCED)*, 2000.

[24] Atmel, *AT91 ARM Thumb Microcontrollers*, 2632C–ATARM–03/04 ed., Mar. 2004.

[25] Freescale Semiconductor, Inc., *MCF5216 Integrated Microcontroller Product Brief*, MCF5216PB/D ed., Mar. 2004.

[26] Taiwan Semiconductor Manufacturing Company, Ltd., "Nexsys 90-nm technology for SoC," http://www.tsmc.com/download/enliterature/90_bro_2003.pdf, Tech. Rep. 6000-1/04.03, Apr. 2003.

[27] Embedded Microprocessor Benchmark Consortium (EEMBC). http://www.eembc.hotdesk.com/about%20eembc.html, Dec. 2004.

[28] Texas Instruments Incorporated, *TMS320C6000 CPU and Instruction Set Reference Guide*, Mar. 1999.

[29] B. Rau, V. Kathail, and S. Aditya, "Machine-description driven compilers for EPIC processors," Hewlett Packard, Tech. Rep. HPL-98-40, Sept. 1998.

[30] M. S. Schlansker, B. R. Rau, S. Mahlke, V. Kathail, R. Johnson, S. Anik, and G. Abraham, "Achieving high levels of instruction-level parallelism with reduced hardware complexity," Hewlett-Packard Laboratory, Tech. Rep. HPL-96-120, November 1994.

[31] StarCore DSP Technology, *SC140 DSP Core Reference Manual*, June 2000.

[32] W. Hwu, "Compiler technology for future microprocessors," *Proceedings of the IEEE*, vol. 83, pp. 1625–1640, Dec. 1995.

[33] A. Aho, R. Sethi, and J. Ullman, *Compilers: Principles, Techniques, and Tools*. Reading, MA: Addison-Wesley, 1986.

[34] W. W. Hwu et al., "The superblock: An effective technique for VLIW and superscalar compilation," *The Journal of Supercomputing*, vol. 7, pp. 229–248, Jan. 1993.

[35] S. A. Mahlke et al., "Effective compiler support for predicated execution using the hyperblock," in *Proc. 25th Annual Int'l Symp. on Microarchitecture (MICRO-25)*, Dec. 1992, pp. 45–54.

[36] C. Lee, M. Potkonjak, and W. Mangione-Smith, "MediaBench: A tool for evaluating and synthesizing multimedia and communications systems," in *Proc. 30th Int'l Symp. on Microarchitecture (MICRO-30)*, Dec. 1997, pp. 330–335.

[37] ETSI TC-SMG, "Digital cellular communications system; Enhanced Full Rate (EFR) speech transcoding (GSM 06.60)," European Telecomm. Standards Institute, Tech. Rep. ETS 300 726, Mar. 1997.

[38] International Telecommunication Union Telecommunication Standardization Sector, Study Group 8 (ITU-T SG8), *JasPer Software Reference Manual*, 1.500.4 ed., Dec. 2001.

[39] S. Gochman et al., "The Intel Pentium M processor: Microarchitecture and performance," *Intel Technology Journal*, vol. 07, pp. 21–36, May 2003.

[40] L. O. Andersen, "Program analysis and specialization for the C programming language,", Ph.D. dissertation, University of Copenhagen, 1994.

[41] B. Steensgaard, "Points-to analysis in almost linear time," in *Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, 1996, pp. 32–41.

[42] E. Nystrom, H.-S. Kim, and W. Hwu, "Importance of heap specialization in pointer analysis," in *Proceedings of the Workshop on Program Analysis for Software Tools and Engineering*, 2004.

[43] M. Hind and A. Pioli, "Which pointer analysis should I use?," in *Proceedings of the 2000 ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2000, pp. 113–123.

[44] M. Hind, "Pointer analysis: Haven't we solved this problem yet?," in *Proceedings of the Workshop on Program Analysis for Software Tools and Engineering*, 2001, pp. 54–61.

[45] E. Nystrom, H.-S. Kim, and W. Hwu, "Bottom-up and top-down context-sensitive summary-based pointer analysis," in *Proceedings of the 11th Annual Static Analysis Symposium*, Aug. 2004.

[46] B.-C. Cheng, "Compile-time memory disambiguation for C programs,", Ph.D. dissertation, University of Illinois at Urbana-Champaign, May 2000.

[47] H. Hunter and W. Hwu, "Code coverage and input variability: Effects on architecture and compiler research," in *Proceedings of the Int'l Conference on Compilers, Architecture, and Synthesis for Embedded Systems*, Oct. 2002, pp. 79–87.

[48] H. C. Hunter, "Characterization of memory activity in media and telecommunications applications," M.S. thesis, University of Illinois at Urbana-Champaign, May 2002.

[49] S. Shalan and V. Mooney, "A dynamic memory management unit for embedded real-time system-on-a-chip," in *Workshop on Compilers, Architecture, and Synthesis for Embedded Systems*, 2000.

[50] M. Millberg, A. Postula, and A. Hemani, "An efficient dynamic memory manager for embedded systems," in *Proceedings of the International Conference on Chip Design Automation (ICDA)*, 2000.

[51] L. Martinot, "Dynamic memory allocation optimizes integration of Blackfin processor software," *Analog Dialogue*, vol. 37, July 2003.

[52] J. Gyllenhaal, W. Hwu, and B. Rau, "HMDES version 2 specification," University of Illinois, Tech. Rep. IMPACT-96-03, 1996.

[53] S. Abraham, V. Kathail, and B. Deitrich, "Meld scheduling: Relaxing scheduling constraints across region boundaries," in *Proceedings of the 29th Annual International Symposium on Microarchitecture (MICRO-29)*, Dec. 1996, pp. 308–321.

[54] S. Udayakumaran and R. Barua, "Compiler-decided dynamic memory allocation for scratch-pad based embedded systems," in *Proceedings of the ACM Int'l Conference on Compilers, Architecture, and Synthesis for Embedded Systems*, Oct. 2003, pp. 276–286.

[55] H. Hunter and W. Hwu, "Memory profiling: Expanding the 3G developer's bag of tricks," in *Workshop on Compilers and Tools for Constrained Embedded Systems*, Oct. 2003.

[56] J. Sias, H. Hunter, and W. Hwu, "Enhancing loop buffering of media and telecommunications applications using low-overhead predication," in *Proceedings of the 34th Annual International Symposium on Microarchitecture (MICRO-34)*, Dec. 2001.

[57] J. Eyre and J. Bier, "VLIW architectures for DSP," in *Proceedings of the International Conference on Signal Processing Applications and Technology*, 1999.

[58] H. Hunter, C.-W. Li, N. Carter, and W. Hwu, "Capturing telecommunication application memory traffic," University of Illinois, Tech. Rep. IMPACT-03-01, 2003.

[59] STMicroelectronics, *ST120 DSP-MCU Programming Manual*, Dec. 2000.

[60] J. Moreno, V. Zyuban, U. Shvadron, F. Nesser, J. Derby, M. Ware, K. Kailas, A. Zaks, A. Geva, S. Ben-David, S. Asaad, T. Fox, M. Biberstein, D. Naishlos, and H. Hunter, "An innovative low-power high-performance programmable signal processor for digital communications," IBM Research, Tech. Rep. RC 22568 (W0209-079), Sept. 2002.

[61] ARM Limited, *The ARM9E-S Thumb Family*, ARM DOI 0079A ed., 2002.

[62] P. Ranganathan, S. Adve, and N. Jouppi, "Reconfigurable caches and their application to media processing," in *Proceedings of the 27th Annual International Symposium on Computer Architecture*, 2000, pp. 214–224.

[63] D. Albonesi, "The inherent energy efficiency of complexity-adaptive processors," in *Proceedings of the Power-Driven Microarchitecture Workshop at ISCA*, 1998, pp. 107–112.

[64] D. Albonesi, "Dynamic IPC/clock rate optimization," in *Proceedings of the 25th Annual International Symposium on Computer Architecture*, June 1998, pp. 282–292.

[65] D. Albonesi, "Selective cache ways: On-demand cache resource allocation," *Journal of Instruction-Level Parallelism*, vol. 2, May 2000, http://www.jilp.org/vol2.

[66] R. Balasubramonian, D. Albonesi, A. Buyuktosunoglu, and S. Dwarkadas, "Memory hierarchy reconfiguration for energy and performance in general-purpose processor architectures," in *Proc. 33rd Int'l Symp. on Microarchitecture*, 2000, pp. 245–257.

[67] S. Dropsho, A. Buyuktosunoglu, R. Balasubramonian, D. Albonesi, S. Dwarkadas, G. Semeraro, G. Magklis, and M. Scott, "Integrating adaptive on-chip storage structures for reduced dynamic power," in *Proceedings of the 2002 International Conference on Parallel Architectures and Compilation Techniques (PACT '02)*, 2002, pp. 141–152.

[68] A. Veidenbaum, W. Tang, R. Gupta, A. Nicolau, and X. Ji, "Adapting cache line size to application behavior," in *Proceedings of the 1999 International Conference on Supercomputing*, 1999, pp. 145–154.

[69] C. A. Moritz, M. I. Frank, and S. Amarasinghe, "FlexCache: A framework for flexible compiler generated data caching," in *Proceedings of the 2nd Intelligent Memory Workshop*, Nov. 2000.

[70] N. S. Kim, K. Flautner, D. Blaauw, and T. Mudge, "Drowsy instruction caches–leakage power reduction using dynamic voltage scaling and cache sub-bank prediction," in *Proc. 35th Annual Int'l Symp. on Microarchitecture (MICRO-35)*, 2002, pp. 219–230.

[71] P. Juang, P. Diodato, S. Kaxiras, K. Skadron, Z. Hu, M. Martonosi, and D. Clark, "Implementing decay techniques using 4T quasi-static memory cells," *IEEE Computer Society Computer Architecture Letters*, vol. 1, pp. 180–219, Sept. 2002.

[72] W. Zhang, J. Hu, V. Degalahal, M. Kandemir, N. Vijaykrishnan, and M. J. Irwin, "Compiler-directed instruction cache leakage optimization," in *Proc. 35th Annual Int'l Symp. on Microarchitecture (MICRO-35)*, 2002, pp. 208–218.

[73] Analog Devices, Inc., Digital Signal Processor Division, *VisualDSP++ 3.0 C/C++ Compiler and Library Manual for Blackfin^{TM} DSPs*, 82-000410-03 ed., Apr. 2002.

[74] Texas Instruments Incorporated, "Code coverage and multi-event profiler user's guide." Literature Number SPRU624A, Jan. 2003.

[75] Texas Instruments Incorporated, "Using cache analysis tool to improve cache utilization." Application Report SPRA863, Jan. 2003.

[76] Green Hills Software Inc., "Embedded ST100 development guide: MULTI 2000." Pub ID D16B-I0201-89NG, 2000.

[77] P. R. Panda, F. Catthoor, N. D. Dutt, K. Danckaert, E. Brockmeyer, C. Kulkarni, A. Vandecappelle, and P. G. Kjeldsberg, "Data and memory optimization techniques for embedded systems," *ACM Transactions on Design Automation of Electronic Systems*, vol. 6, no. 2, pp. 149–206, 2001.

[78] J. Sjödin, B. Fröderberg, and T. Lindgren, "Allocation of global data objects in on-chip RAM," in *Workshop on Compilers, Architecture, and Synthesis for Embedded Systems*, Dec. 1998.

[79] P. R. Panda, N. D. Dutt, and A. Nicolau, "On-chip vs. off-chip memory: the data partitioning problem in embedded processor-based systems," *ACM Transactions on Design Automation of Electronic Systems*, vol. 5, no. 3, pp. 682–704, 2000.

[80] M. Verma, S. Steinke, and P. Marwedel, "Data partitioning for maximal scratchpad usage," in *Proceedings of the Asia and South Pacific Design Automation Conference*, 2003.

[81] O. Unsal, R. Ashok, I. Koren, C. Krishna, and C. Moritz, "Cool-cache for hot multimedia," in *Proceedings of the 34th Annual International Symposium on Microarchitecture (MICRO-34)*, Dec. 2001, pp. 274–283.

[82] O. Avissar, R. Barua, and D. Stewart, "An optimal memory allocation scheme for scratch-pad based embedded systems," *ACM Transactions on Embedded Computing Systems*, vol. 1, pp. 6–26, Nov. 2002.

[83] M. T. Kandemir, J. Ramanujam, M. J. Irwin, N. Vijaykrishnan, I. Kadayif, and A. Parikh, "Dynamic management of scratch-pad memory space," in *Proceedings of the 38th Design Automation Conference*, 2001, pp. 690–695.

[84] A. Farrahi, G. Téllez, and M. Sarrafzadeh, "Memory segmentation to exploit sleep mode operation," in *Proceedings of the 32nd Design Automation Conference*, 1995, pp. 36–41.

[85] L. Benini, A. Macii, and M. Poncino, "A recursive algorithm for low-power memory partitioning," in *Proc. Int'l Symposium on Low Power Electronics and Design*, 2000, pp. 78–83.

[86] F. Catthoor, S. Wuytack, E. D. Greef, F. Balasa, L. Nachtergaele, and A. Vandecappelle, *Custom Memory Management Methodology*. Boston, MA: Kluwer Academic Publishers, 1998.

[87] Interuniversity Microelectronics Center (IMEC), "Atomium project." http://www.imec.be/design/atomium/, Dec. 2004.

[88] A. Papanikolaou, M. Miranda, F. Catthoor, H. Corporaal, H. D. Man, D. D. Roest, M. Stucchi, and K. Maex, "Methodology for propagating technology trade-offs over memory modules to the application level," in *Proceedings of the 3rd PA3CT Symposium*, Sept. 2003.

[89] B. K. Mathew, S. A. McKee, J. B. Carter, and A. Davis, "Design of a parallel vector access unit for SDRAM memory systems," in *Proc. of the 6th IEEE Symp. on High-Performance Computer Architecture (HPCA-6)*, Jan. 2000, pp. 39–107.

[90] K. V. Palem, R. M. Rabbah, V. J. Mooney III, P. Korkmaz, and K. Putaswamy, "Design space optimization of embedded memory systems via data remapping," in *Proceedings of the Joint Languages, Compilers, and Tools for Embedded Systems (LCTES) and Software and Compilers for Embedded Systems (SCOPES) Conference*, June 2002.

[91] J. Carter, W. Hsieh, L. Stolloer, M. Swanson, L. Zhang, E. Brunvand, A. Davis, C.-C. Kuo, R. Kuramkote, M. Parker, L. Schaelicke, and T. Tateyama, "Impulse: Building a smarter memory controller," in *Proc. of the 5th IEEE Symp. on High-Performance Computer Architecture (HPCA-5)*, Jan. 1999, pp. 70–79.

[92] S. Rubin, R. Bodik, and T. Chilimbi, "An efficient profile-analysis framework for data-layout optimizations," in *Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, 2002, pp. 140–153.

[93] B. Calder, C. Krintz, S. John, and T. Austin, "Cache-conscious data placement," in *Proc. 8th Int'l Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '98)*, 1998, pp. 139–149.

[94] D. N. Truong, F. Bodin, and A. Seznec, "Improving cache behavior of dynamically allocated data structures," in *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT '98)*, 1998, pp. 322–329.

[95] T. D. Givargis, J. Henkel, and F. Vahid, "Interface and cache power exploration for core-based embedded system design," in *Proceedings of the 1999 IEEE/ACM International Conference on Computer-Aided Design*, 1999, pp. 270–273.

[96] S. Heo, K. Barr, and K. Asanović, "Reducing power density through activity migration," in *Proceedings of the International Symposium on Low Power Electronics and Design (ISLPED'03)*, 2003, pp. 217–222.

[97] Y. Wu and Y. Lee, "Accurate invalidation profiling for effective data speculation on epic processors," in *International Conference on Parallel and Distributed Computing Systems*, 2000.

[98] J. Lin, T. Chen, W.-C. Hsu, P.-C. Yew, R. Ju, T.-F. Ngai, and S. Chan, "A compiler framework for speculative analysis and optimizations," in *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implemantation*, 2003, pp. 289–299.

[99] R. Ghiya, D. Lavery, and D. Sehr, "On the importance of points-to analysis and other memory disambiguation methods for C programs," in *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implemantation*, 2001.

[100] S. Sair, T. Sherwood, and B. Calder, "Quantifying load stream behavior," in *Proc. of the 8th IEEE Symp. on High-Performance Computer Architecture (HPCA-8)*, Feb. 2002, pp. 197–208.

[101] J. Fritts, "Architecture and compiler design issues in programmable media processors,", Ph.D. dissertation, Princeton University, 2000.

[102] J. Fritts, W. Wolf, and B. Liu, "Understanding multimedia application characteristics for designing programmable media processors," in *Proceedings of the SPIE Photonics West Media Processors Conference*, 1999, pp. 2–13.

[103] J. Fritts and W. Wolf, "Instruction fetch characteristics of media processing," in *Proceedings of the SPIE Photonics West Media Processors Conference*, 2002, pp. 72–83.

[104] N. Slingerland, "Architectures for multimedia," M.S. thesis, University of California at Berkeley, 2000.

[105] S. G. Berg, "A cache-based prefetching memory system for mediaprocessors,", Ph.D. dissertation, University of Washington, 2002.

[106] S. Srinivasan and A. Lebeck, "Load latency tolerance in dynamically scheduled processors," in *Proceedings of the 31st Annual International Symposium on Microarchitecture (MICRO-31)*, Nov. 30–Dec. 2 1998, pp. 148–159.

[107] P. Grun, N. Dutt, and A. Nicolau, "Access pattern based local memory customization for low power embedded systems," in *Proceedings of Design, Automation and Test in Europe*, 2001, pp. 778–784.

VITA

Hillery Catriona Hunter was born in Bellflower, California. She began electrical engineering studies at the University of Illinois at Urbana-Champaign in 1996. She spent the 1997-1998 academic year at the Technical University of Munich, Germany, from which she received the 1998 Outstanding International Student Prize. In 1999, she received her B.S.E.E. from the University of Illinois, with Highest Honors, and an International Minor in Germanic Studies. Continuing at the University of Illinois for graduate studies under Professor Wen-mei Hwu, she received her M.S.E.E. in 2002 and Ph.D. in Electrical Engineering in 2004. As a graduate student, she was the holder of the national Tau Beta Pi, National Science Foundation, and IBM PhD Fellowships, in addition to the University of Illinois Distinguished Fellowship and Electrical and Computer Engineering Department Kohler Fellowhip. She will be joining the IBM T.J. Watson Research Center in Yorktown Heights, New York, in 2005.