

Hardware-Compiler Co-Design for Adjustable Data Power Savings

Hillery C. Hunter
IBM Corp.
TJ Watson Research Center
Yorktown Heights, NY 10598
hhunter@us.ibm.com

Erik M. Nystrom
Universal Network Machines
Santa Clara, CA 95054
enystrom@unminc.com

Daniel A. Connors
Dept. Elec. & Comp. Engr.
Univ. of Colorado
Boulder, CO 80309
dconnors@colorado.edu

Wen-mei W. Hwu
Dept. Elec. & Comp. Engr., CSL
Univ. of Illinois
Urbana-Champaign, IL 61801
hwu@crhc.uiuc.edu

Abstract

To accommodate standards changes and algorithmic improvements, functional reconfigurability is increasingly desired for media processing. Such adaptability, however, generally comes at significant power cost. This work suggests that another dimension of adaptation can be beneficial – power adaptation. Through a unique compiler–hardware approach, we (1) demonstrate an extension to the state-of-the-art in data analyzability, yielding better control over scratch-pad data management, and (2) combine this knowledge with an SRAM having variable latency and access properties, yielding adjustable power savings. Building upon the compiler techniques presented in [1], we evaluate the severity of the current on-chip storage power problem and detail how SRAM structures can be built to enable data power savings for media applications. We show how the implemented compiler techniques can be applied to other problems in the embedded/media processing domain, and present net data power savings results for a suite of media and telecommunication applications, including MPEG-2, MPEG-4, H.263, and JPEG-2000.

1. Introduction

In the embedded space, processors were traditionally small and specialized. However, increases in the performance requirements of consumer devices have necessitated multi-issue architectures with large amounts of on-chip storage. The increasing proportion of power consumed by this on-chip data storage is cause for concern. Currently, up to 50% of high-end embedded device power may be dissipated in on-chip data storage [2]. Many previous works have focused on embedded cache power consumption [2, 3], but as indicated in Table 1, numerous embedded processors have significant amounts of on-chip, non-cache SRAM, which may also consume a large amount of data storage power. Despite being partitioned or banked, the large logical capacity of many SRAMs results in higher static and dynamic power consumption relative to smaller data caches.¹ Clearly, the power consumption of non-cache data storage needs to be addressed for high-performance embedded processors.

¹Dynamic power estimates in Table 1 were calculated using XCACTI [4], assuming a 0.13 μ m technology. Dynamic power considers banking, cache associativity, and port counts; static power is based only on data array size, neglecting additional power from tag storage.

Our study of embedded applications has revealed significant opportunity for reducing the power consumed by data accesses by providing differentiated service according to data access needs (giving fast access to critical loads and using low-power storage for less access-time or bandwidth critical data). Software control is necessary to maintain predictability and preserve applicability for deadline-conscious systems, and performance degradation should be explicitly controlled to allow optimization of system-level power. For the presented scheme, amortization of any configuration cost can be ensured, being enforced by use of the compiler to guarantee extended periods of stability.

This paper suggests a physical structure and presents software algorithms for an automated, compiler-managed, approach to program-wide data power savings. Power reduction is achieved using *configurable SRAM* that provides differentiation of data access time and port counts within multi-line regions of a single on-chip structure. SRAM configuration leverages recent circuit-level techniques for power savings in unified SRAM arrays, including *Self Reverse Biasing* [5], *Floating Bitlines* [5, 6], and *V_{dd} Throttling* [7]. Each was previously applied for hardware-controlled power savings, but to increase predictability of both access time and average power consumption, we instead use them in a collaborative hardware-software approach. The compiler leverages scalable interprocedural analysis to provide both safe and optimistic data usage information, and perform joint operation scheduling and data placement to enable use of low-power SRAM regions. The combination of an SRAM that supports flexible low-power regions with compiler technology that can automatically map data to these regions is termed *configurable SRAM*, or *C-SRAM*. We first outline the C-SRAM structure, then describe the implemented compiler algorithms and their applicability to broader scheduling challenges in the embedded domain. The paper concludes with power savings results for the proposed mechanisms.

2. Configurable SRAM

Figure 1(a) shows a conceptual view of eight multi-line C-SRAM configuration regions. When not configured, the baseline data storage is dual-ported and short latency, thus not very power efficient. Figure 1(b) shows a configuration with five short latency, dual-port regions; one short latency single port region; one long latency, dual-port region; and one

Table 1: Embedded/DSP processors: data cache vs. on-chip SRAM power.

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

long latency, single port region. As calculated by the proportion of total SRAM size and power savings from various configurations, this results in an estimated static power savings of 22.8% over the baseline fast, dual-port design. Dynamic power cost of access to data in long latency regions is also reduced. However, if the compiler can exploit more code flexibility (i.e. compensate for more costly memory accesses), the cooler configuration shown in Figure 1(c) can be used. This configuration specifies that more data be stored in a low-power configuration, and results in an estimated 55.6% static power savings over the baseline design.

Circuit building blocks

Subarray partitioning is common in current SRAM designs to reduce access latency and power consumption by dividing data into two or more disjoint arrays, each with separate decoders. Without requiring further decoder replication, the SRAM *macroblock* design methodology goes one step further, calling for bitlines and wordlines to be broken into segments, and bitline precharge and sense amp logic to be localized. This forms blocks of memory storage cells, which are placed adjacent to one another to form an array of the desired size [8]. Such chessboard-like means of array construction allows one to view the array as easily partitionable both vertically (like traditional banking) and horizontally (into contiguous multi-line regions). This construction provides a logical multi-line granularity for controlling access properties of stored data.

Port configuration

If two different versions of SRAM are built, one dual-ported and the other with a single port, no control is needed for port “configuration” and no overhead added to the SRAM array design. The SRAMs 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 “customized” approach taken in previous embedded work [9], but it neither scales well with application growth, nor provides a straight-forward software model.

A simple method for turning a port “off” is to allow its corresponding bitlines to *float*. Floating, or *leakage-biased*,

bitlines are not precharged on each access cycle and reduce bitline–ground leakage current because the values stored in SRAM cells set the bitlines to a mid-rail voltage. This 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 [6], 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. 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.

Latency configuration

A fundamental rule of processor tuning is performance sensitivity to first-level memory latency. While an indiscriminate increase in memory latency will likely be detrimental to performance, large parts of programs are capable of absorbing longer memory latencies. A compiler can distinguish between those accesses required to be fast from those that can be slowed with little or no performance impact, allowing latency to be *increased* in SRAM sections holding data objects that are not (or are rarely) accessed on a critical schedule path.

This work proposes use of DVS and Self Reverse Biasing (SRB) [5] to increase access latency. DVS provides dynamic power savings, while SRB cuts leakage by 80%, and can be implemented in a programmable circuit. Both DVS and SRB increase data access time. To simplify circuit implementation cost and configuration logic, this work assumes there are just two memory latencies.

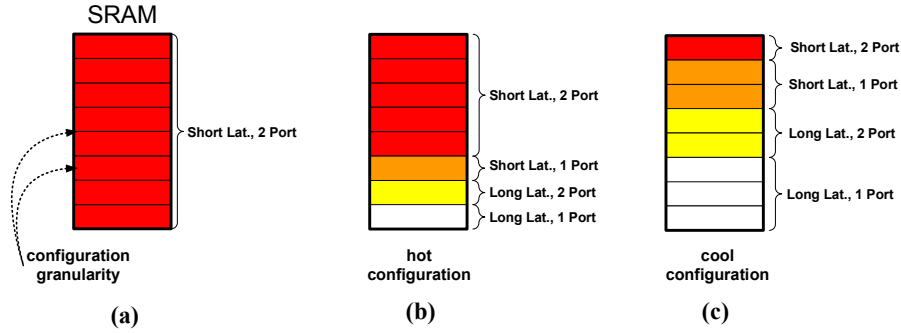


Figure 1: Conceptual view of *Configurable SRAM*. (a) SRAM configurable at a multi-line granularity, (b) A "hot" (high power) configuration that saves an estimated 22.8% leakage power over no configuration, and (c) A cooler configuration saving 55.6% leakage power, resulting from exploiting more code flexibility.

Table 2: Techniques leveraged to provide port and latency configuration.

Circuit Technique	Contributed Configurability	Added Components	Power Saved	Area Cost
Bitline Float [5][6]	Port turn-off	Precharge gating and control	22.5% leakage / port	Bitline Float + SRB = 3%
Self Reverse Biasing [5]	Leakage reduction when latency increased	SRB signal, nFET, pFET	80% leakage	Bitline Float + SRB = 3%
DVS [7]	Dynamic power reduction when latency increase	DVS signal, second V_{dd}	46% dynamic power	<2%
SRB + DVS	Both ports off + Latency increase = 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)

Overhead and power savings of the combination of techniques proposed for C-SRAM configuration are summarized in Table 2.

3. Object Scheduling

Placement of program data objects into storage with varied properties, in conjunction with operation scheduling, will be discussed as the *object scheduling* optimization problem. Object scheduling determines the number, type, and size of profitable data storage regions for a given application. For large, high-performance embedded applications, balancing the cost and benefit tradeoffs 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, however, can effectively leverage and selectively create memory operation flexibility to exploit power-saving data storage restrictions.

The available storage configurations and accompanying compiler algorithm leverage both operation *slack* (manifested by a schedule's tolerance to added load operation latency) and *memory slot tolerance* (demonstrated through memory slot/port choice). By exploiting these forms of *code flexibility* present in telecommunication and media applications, the compiler can derive both dynamic and static power savings with either no performance degradation or a performance loss tuned to maximize the ratio of power savings to application slowdown [10]. In this way, the compiler bounds performance degradation to preserve applicability to real-time applications.

Our static compiler analysis provides an *object*-based view of application data usage, where an object may be any global variable; stack array, union, or structure; or heap allocation site (call to `malloc()`). Interprocedural pointer analysis

provides unique identifiers for all application data objects and the code locations in which each object may be accessed. The compiler combines memory object access information with knowledge of operation slack to determine the port and latency requirements of each object. After the needs of all objects are identified and all operations scheduled, objects are sorted according to their configuration options. The SRAM is then configured to match the size of each configuration group.

The implemented compiler algorithm was able to identify a significant amount of object-based port flexibility (average of 53.8% of dynamic accesses, 53.1% of total data space) and a moderate amount of latency flexibility (average 6.9% of dynamic accesses, 22.5% of total data space) for telecommunication and media applications. When a small amount of performance degradation is tolerated (2.7%), many more objects can be scheduled to long latency regions, bringing average latency flexibility to 24.6% of dynamic accesses and 56.3% of total data space (corresponding to an average of 9.4% dynamic power and 51.7% static data power savings).

3.1. Compilation environment

The compilation framework used is based on IMPACT [11]. The process used to obtain scheduled code and place objects starts with standard C code, as shown in Figure 2. Interprocedural pointer analysis results guide a series of classical optimizations, as well as the scheduling process. Memory profiling provides access weights for each program data object and helps estimate the size of active heap allocated data. Finally, the instruction scheduler uses operation slack and object use information to obtain either a schedule without performance degradation, or a power-performance balanced schedule. The following sections provide details of the com-

piler infrastructure; its applicability to other problems in the embedded/media processing domains; handling of multiprocessing and caching; treatment of local and heap variables; and joint object placement and operation scheduling.

3.2. Interprocedural pointer analysis

Conceptually, pointer analysis aims to describe the potential targets of each pointer. Pointer access information gives the compiler a view of an application’s memory activity and can be critical for tasks like register promotion, scheduling, memory dataflow, debugging, and verification. It is also central to automated software management of data access properties, since the compiler must ensure that all objects accessible through the same set of pointers must be placed in compatible configurations, particularly with regard to port assignment.

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). Due to the real-time constraints of many media applications, it is essential that an analysis be accurate,² and since application complexity is growing, analyses should be scalable. The increasing body of media codes written in C and C++ also leads us to support unmodified high-level language code, with arbitrary pointer and dynamic memory usage.

The algorithms we used for interprocedural pointer analysis are similar to, but more general than, those used in [12], which have been shown to work efficiently for large programs having complicated pointer usage. The analysis 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.

While the necessary form of pointer analysis is not always clear, the selection can sometimes be important. For the memory operations in an MPEG-4 benchmark function, Figure 3 shows which objects these operations appear to access for two pointer analysis formulations. Figure 3(a) shows the selection used in this paper, while the more chaotic relationships in Figure 3(b) result from an algorithm similar to that published in [13]. For the benchmarks analyzed in this paper, the time to perform the formulation chosen for this paper ranged from less than 0.01 second to 1 second on a 2.8GHz Pentium 4.

Pointer analysis typically provides a conservative approximation of actual pointer behavior. For example, it may determine that a variable A points to a variable B when, in

²From the perspective of latency configuration, missing dependencies could lead to unexpected long latency accesses within critical paths, thereby violating the program’s deadline constraints.

reality, no program input will result in this situation. However, a conservative result will never omit a realizable relationship. Such conservative behavior is necessary to preserve the correctness of many compiler transformations. However, our 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. While not valid for general dependence analysis, it is useful for placement of objects into a configurable SRAM. It is important to note that pointer analysis always excludes provably unreachable program code, and zero-weight exclusion analysis additionally eliminates code not touched during a particular execution profile.

3.3. Variable promotion

Local variables allocated in adjacent locations on the run-time stack can have very different port and latency requirements. Partitioning the run-time stack into 4 sections, so that each corresponds to a port/latency combination, and would grow and shrink within a different type of configuration region, would be unwieldy. In the studied application set, many local variables (stack objects) are pseudo-global: they exist for most of the program duration and are used heavily by multiple functions (passed by reference). To increase the effectiveness of the object placement algorithm, the compiler first promotes arrays, structures and unions from local stack allocation to static global variable space (where it is correct to do so, *e.g.*, where there is no recursion). This enables such variables to be scheduled in a low-power region without requiring partitioning of the stack. Remaining stack data is kept in non-configured, *i.e.*, high power, dual-port, short latency storage.

3.4. Heap objects

Since heap allocated objects are created dynamically, many previous approaches to software data management (*e.g.*, for compiler-managed caching or scratchpad placement), have not handled heap data. More recent high-performance applications (*e.g.* JPEG and MPEG) commonly 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. Our approach leverages a specialized implementation of the memory allocation procedure, such as `malloc()`, to force placement of heap objects into low-power SRAM regions.

The implementation chosen includes a parameter that determines the configuration from which each `malloc()` will allocate objects. Our interprocedural pointer analysis identifies all loads and stores that can access objects allocated from each static call site to `malloc()`. Thus, the object scheduling algorithm can identify the constraints on all objected allocated from the same static call to `malloc()`. 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 in the program to the object

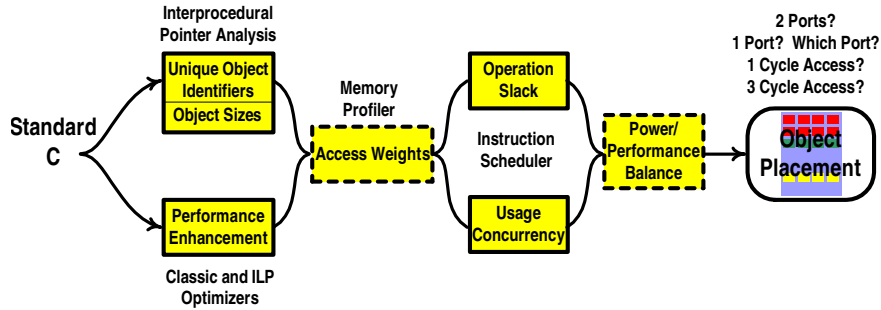


Figure 2: Compiler flow: memory usage information to enable use of low-power SRAM.

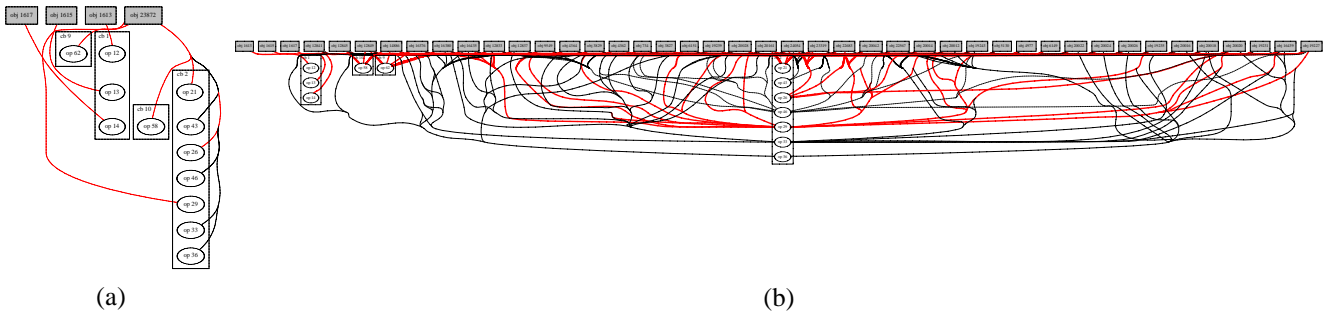


Figure 3: Pointer analysis accuracy for the *mpeg4dec* function *GetContextInter*. The upper, horizontal row of squares are data objects. The lower, vertical column of ellipses are memory operations from the function. For the same memory operations, arcs represent potential object accesses, as determined by (a) the accurate pointer analysis configuration used in this paper, and (b) a straightforward Steensgaard implementation. Note that many more objects appear to be accessed in (b).

with the most stringent schedule restrictions. Despite this restriction, many heap objects can be allocated into low-power regions.

One possible drawback to the configuration of heap objects is allocation beyond the available configuration capacity. Based on an extensive study, we found that though some allocation sites generate megabytes of data over the course of program execution, the maximum total live object size for many allocation sites is small and constant. Leveraging this characteristic, `malloc()` calls can be specialized to allocate into a low power region. For safety, such `malloc()` calls also include a maximum spawn size. If, during the course of program execution, a given `malloc()` requests more space than determined as its maximum spawn size during compilation, the newly requested object can be placed either into a non-configured, *i.e.*, maximum power region, or into DRAM, according to the system’s specifications. Specialization of `malloc()` according to object size has been previously implemented to decrease allocation time [14] and improve speed by minimizing fragmentation [15]. Similarly, specialization according to chosen data access properties should have little impact on the overhead of dynamic memory allocation.

3.5. Algorithm implementation

For default object scheduling, a list instruction scheduler is used on extended basic block scheduling regions. Prior to object scheduling, local variables are promoted, machine-independent optimizations are performed, and code is annotated with profile feedback. The profiler also ranks objects by

their access weight and provides this information to the object scheduler. A description of the architecture is read in, and a list of possible SRAM configurations created, prioritized according to power savings potential.

Operation scheduling is first performed to calculate a baseline schedule for all regions. 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 control flow profile weights. Anticipated execution time, $CurrentCyc$, is calculated as $CurrentCyc = \sum_{\forall Blocks} \sum_{\forall Flows} FlowWeight * FlowScheduleCyc$. A flow may be either a branch or fall-through path, and $FlowScheduleCyc$ is counted from the beginning of the enclosing code region, *i.e.*, for a fall-through path, $FlowScheduleCyc$ will be the region’s schedule height.

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. 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. Increasing latency and restricting access to a single port, (1)

and (2), save the most power. Increased latency alone, (3), also saves both static and dynamic power but somewhat less than (1) and (2). Port restriction, (4) and (5), provides the least benefit, primarily through static power reduction. As code is scheduled for each configuration, the impact of the current access properties on schedule height, *HeightPenalty*, and estimated cycle count, *CyclePenalty*, is recorded.

After code generation, the possible schedules are examined, starting with the configuration that provides the greatest power savings. The corresponding schedule's *CyclePenalty* and *HeightPenalty* 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"), *CurrentCyc* and *CurrentHeight* are updated and the process continues with the next highest ranked object.

When making object access 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. 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 (they are in the same pointer analysis points-to set); and (2) they may be *interactive* in being accessed from within a single scheduling region (e.g., a single basic block, superblock [16], or hyperblock [17]). Placement of data objects not conjoined to or interactive with any other objects may be performed in isolation because their placement in a particular configuration will not affect scheduling decisions for other objects. Decisions made for conjoined or interactive data, however, 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 could result in an attempt to access an object through an unavailable port. A data port could be awoken, but this would cause a processor stall. 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. These factors sometimes cause object placement decisions to invalidate configuration options for lower priority objects. For example, if the object currently under consideration has a conjoined object that was previously scheduled to a *PortA-only* region, tests of *Latency-and-PortB* and *PortB-only* are eliminated from the valid configurations for the current object.

3.6. General applicability

The implemented compiler framework is very general. It can accommodate (a) an arbitrary functional unit distribution;

(b) more or different latency and port configurations (more latency steps, more ports); (c) different management of heap and stack data. The architect thus has freedom to change the processor and SRAM implementation and still use the compiler to identify and exploit code flexibility.

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.

Classical DSPs have separate "X" and "Y" SRAMS, intended for use as the source of two separate datastreams. The canonical X/Y partition places filter tap coefficients in X memory and signal (filter input) data in the Y memory. The implemented algorithm was tested for X/Y data placement, and 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 conjoinedness requires either (a) placing all sets of objects linked by overlapping points-to sets into the same memory, 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).

3.7. 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 [18]. The implemented compiler provides automated power management of data once it is in C-SRAM, so is complementary to previous scratchpad work.

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, as shown in Table 1. The standard programming methodology for these mixed systems requires

0	1	2	3	4	5	6	7
Imul/F	Imul/F	lalu	lalu	lalu	lalu MemA	lalu MemB	lalu Br

Figure 4: Modeled VLIW architecture issue slots.

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 non-cache 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.

4. Results

In this section, we examine the significant savings achievable through our collaborative hardware–software approach to data management, and the impacts of profile feedback and ILP optimization. The studied benchmarks are listed in Table 3. While results are presented from traditional embedded kernels (from EEMBC and MediaBench), we also demonstrate the scalability of our compiler to large multimedia applications, like MPEG-2, MPEG-4, and JPEG.

Modeled architecture

Code is scheduled for an eight-wide unified VLIW architecture, with function unit distribution similar to that of the Texas Instruments ‘C6x processors [19]. Figure 4 shows the fixed assignment of functional units to slots. The processor has eight integer ALUs, two of which can 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 memory units; long latency `loads` return in 3 cycles. Intrinsic operations are used to represent DSP instructions specified in the `g724` benchmarks.

Code Optimization

Control profile guided inlining is applied with an allowable touched static code size expansion ratio of 1.2. Classical compiler optimization is also performed on all code before scheduling takes place, including constant and copy propagation, dead code removal, loop invariant code removal, and redundant load elimination. ILP-enhancing techniques used for results at the end of this section include loop unrolling,

loop peeling, predication, and hyperblock and superblock formation.

4.1. Scheduling with no performance tolerance

When performance degradation cannot be tolerated, the compiler is able to identify a significant number of objects for placement in port-restricted configurations, and a moderate number for placement in long-latency regions. Net schedule height, and thus anticipable execution time, is not increased. Figure 5 shows the dynamic and static SRAM power savings corresponding to non-degraded object schedules. A value of 40% static power savings in Figure 5 represents a 40% decrease in static power alone. For the sake of space, 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 dynamic processor power [2]. In future technologies, energy dissipation from leakage (static power) may equal that of switching (dynamic power) [6], so it is important to consider both components of total power consumption (static and dynamic).

Reductions in data power are shown for data placement decisions made after four types of pointer analysis: (1) an inaccurate pointer analysis formulation (*S*, Steensgaard analysis); (2) a very accurate set of pointer analysis options (*Afch*, Andersen style, field-sensitive, context-sensitive, heap-sensitive analysis); and (3) *zS* and (4) *zAfch*, which are optimistic, profile-enhanced versions of (1) and (2), respectively. Power savings generally increase as data objects are placed in low-power configurations based on increasingly accurate, and increasingly optimistic, pointer analyses. For some application pairs shown in Figure 5, *e.g.* `g721dec` and `g721enc`, the accuracy of analysis and the effects of profile refinement have similar effects on data power savings, so only one is shown in the figure.

For non-optimistic analyses, the strength of an interprocedural analysis formulation does not always have a significant impact on net power savings. This is because many DSP applications can be accurately analyzed with a relatively inaccurate Steensgaard formulation. However, in some cases, it does reduce constraints on critical objects, causing a noticeable change in power savings, notably for benchmarks `conv00` and `h263enc`, each of which sees a drop of around 10% in static power, due only to the improved accuracy of memory usage information between *S* and *Afch* analysis formulations. The benefits of analysis accuracy are generally observed for applications in the media and image domains, where pointer usage is more complex. Nevertheless, our best analysis runs in negligible time (0.1–1 sec), so it is worthwhile to always use the best analysis, so as to capture these opportunities for improving results.

4.2. Optimistic, profile-based refinement of IPA

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

Table 3: Benchmarks from MediaBench, ETSI, EEMBC, JasPer, and independent authors.

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

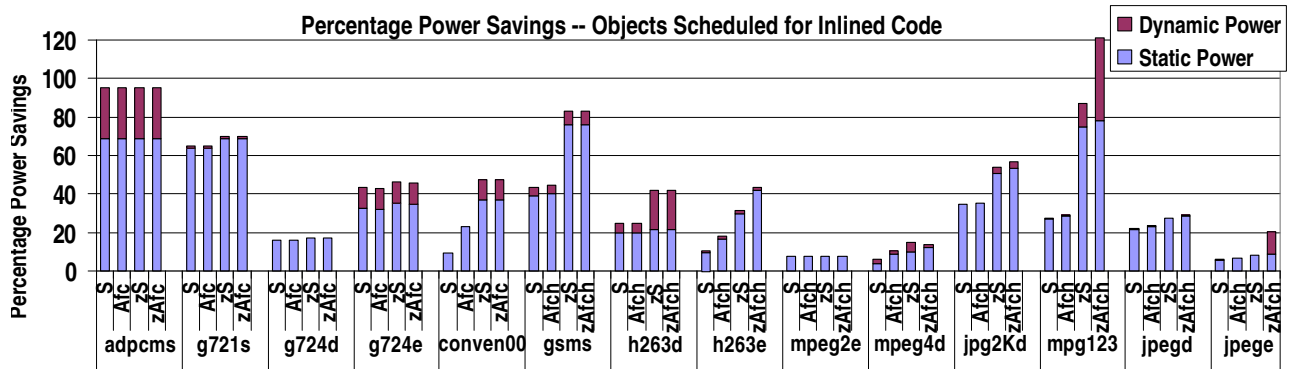


Figure 5: Data power savings for object placement after accurate (Afch), inaccurate (S), and profile-refined interprocedural pointer analysis (z).

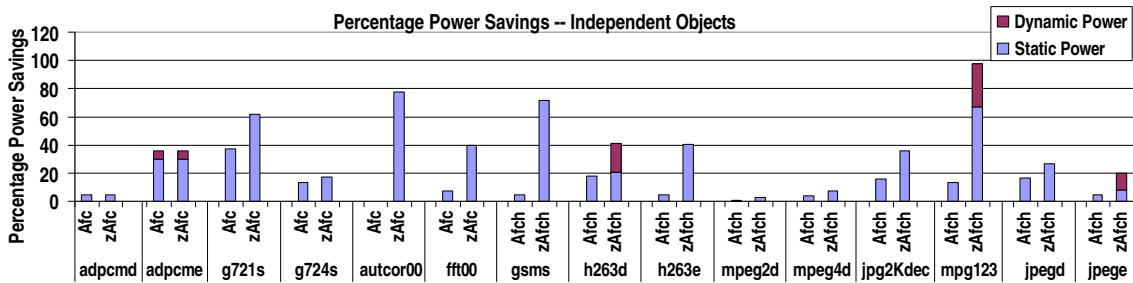


Figure 6: Contribution of independent data objects to total SRAM power savings.

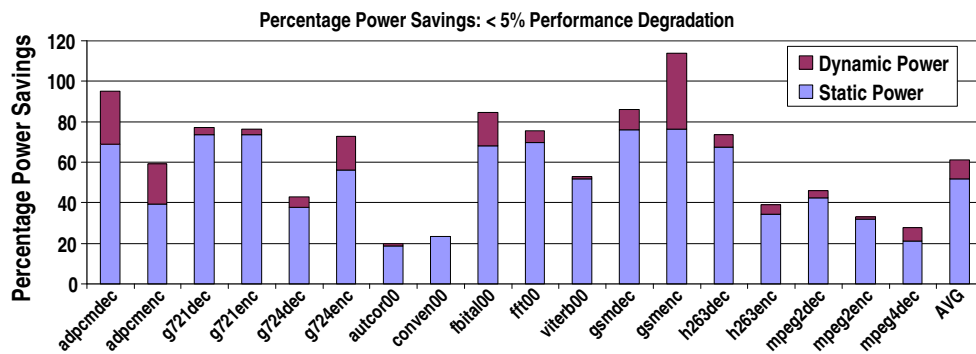


Figure 7: SRAM power savings with moderate performance degradation ($\leq 5\%$).

that this optimistic analysis have an accurate foundation. The difference between *zAfc* and *zS* results in Figure 5 highlight the additional power savings benefit (over *zS*) which may be achieved when an accurate analysis (*Afc*) underlies the zero-weight algorithm.

As explained in Section 3, data configuration decisions made for objects which are neither interactive nor conjoined with other objects are made independently of decisions for other objects. Since they do not consume slack or port resources that might have been allocated to another object, these object placement decisions are optimal. For a selection of the benchmarks shown in Figure 5, Figure 6 shows the amount of power savings which is attributable to independent objects. Zero-weight analysis clearly reduces the constraints on the object scheduling problem, and allows many more independent object placement decisions.

4.3. Profile-guided performance degradation

Object scheduling results for a moderate performance degradation bound (no benchmark degraded more than 5% for the reference input) are shown in Figure 7. These results correspond to the non-degraded *Afc* results shown in Figure 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.

When performance degradation is allowed, the compiler uses a control flow profile to anticipate the net performance effects of restricting ports or increasing access time of individual objects. The average 2.7% degradation for our “moderate” performance tolerance schedule corresponds in Figure 7 to an anticipated data power savings of 51.7% static power and 9.4% dynamic power. Assuming that on-chip data storage consumes 50% of overall processor power [2], the system-wide *power : performance* ratio falls well above the threshold of 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.

5. Related Work

Cache adaptation

Configurable SRAM is a form of adaptation to application characteristics. The term *Complexity Adaptive Processor* was coined in 1998 by Albonesi to mean one whose resources adapted to the needs of an application [20]. Following this, most adaptive approaches have targeted the expense of cache associativity by allowing ways to be disabled during periods of high cache hit rates. Among these proposals are [2], [21], [22], and [23].

Each of these adaptive cache methods only explores one dimension of configuration – ways, line size, or total cache size, 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 excep-

tion of [2], 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.

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 [24], [7], and [25]. Some of these techniques have only been demonstrated for instruction caching and most rely on hardware mechanisms.

The Drowsy cache [7], [26] periodically puts all lines into a data-maintaining state, and then incurs a cycle penalty to wake accessed lines. Decay caches [25] accelerate eviction of unused cache lines by turning off lines containing data which has not been recently accessed. 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 non-cache 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, and have swings in the amount of power consumed on a per-line basis, making them less attractive for the embedded domain.

Compiler-controlled sleep for leakage savings

Zhang *et al.* [27] 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 [24]. [24] leverages data-reuse analysis to place data that will not be used by current computation into a state-preserving sleep mode. The authors 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 it 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.

Data re-layout

In the general-purpose domain, profile guidance has been employed for data re-layout to increase hardware cache spatial locality [28],[29]. 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 hardware ef-

fects, the C-SRAM approach changes hardware behavior so as to directly, and predictably, save power.

6. Conclusions

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 addressing the power challenges of high-performance embedded systems.

Acknowledgments

The authors would like to thank Shane Ryoo for his contributions to the memory profiler used in this work. This work was funded by the DARPA/MARCO Gigascale Systems Research Center, and Hillery Hunter was partially supported by an IBM PhD Fellowship.

References

- [1] H. Hunter, E. Nystrom, S. Ryoo, and W. Hwu, "Applying scalable interprocedural pointer analysis for power reduction of embedded data storage," in *Workshop on Compilers and Tools for Constrained Embedded Systems*, Sept. 2004.
- [2] 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*, pp. 136–146, June 2003.
- [3] 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*, pp. 1065–1071, 2002.
- [4] M. Huang et al., "L1 data cache decomposition for energy efficiency," in *IEEE/ACM Int'l. Symp. Low Power Electronics and Design*, pp. 10–15, 2001.
- [5] 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*, pp. 251–252, June 2003.
- [6] 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*, 2002, pp. 137–147.
- [7] 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*, 2002, pp. 148–157.
- [8] 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)*, pp. 400–405, 2001.
- [9] F. Catthoor, ed., *Unified Low-Power Design Flow for Data-Dominated Multi-Media and Telecom Applications*. Boston, MA: Kluwer Academic Publishers, 2000.
- [10] S. Gochman et al., "The Intel Pentium M processor: Microarchitecture and performance," *Intel Technology Journal*, vol. 07, pp. 21–36, May 2003.
- [11] W. Hwu, "Compiler technology for future microprocessors," *Proceedings of the IEEE*, vol. 83, pp. 1625–1640, Dec. 1995.
- [12] 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.
- [13] B. Steensgaard, "Points-to analysis in almost linear time," in *Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pp. 32–41, 1996.
- [14] B. Zorn, D. Detlefs, and A. Dosser, "Memory allocation costs in large C and C++ programs," Tech. Rep. CU-CS-665-93, University of Colorado at Boulder, Department of Computer Science, 1993.
- [15] MicroQuill, "SmartHeap technical specification." http://www.microquill.com/smartheap/sh_utspec.html.
- [16] 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.
- [17] 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)*, pp. 45–54, Dec. 1992.
- [18] 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*, pp. 276–286, Oct. 2003.
- [19] Texas Instruments Incorporated, *TMS320C6000 CPU and Instruction Set Reference Guide*, Mar. 1999.
- [20] D. Albonesi, "Dynamic IPC/clock rate optimization," in *Proceedings of the 25th Annual International Symposium on Computer Architecture*, pp. 282–292, June 1998.
- [21] D. Albonesi, "Selective cache ways: On-demand cache resource allocation," *Journal of Instruction-Level Parallelism*, vol. 2, May 2000, <http://www.jilp.org/vol2>.
- [22] 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*, pp. 214–224, 2000.
- [23] 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*, pp. 245–257, 2000.
- [24] 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*, pp. 76–85, 2003.
- [25] 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*, pp. 240–251, June 2001.
- [26] 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.
- [27] 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.
- [28] 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)*, pp. 140–153, 2002.
- [29] B. Calder, C. Krantz, 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)*, pp. 139–149, 1998.