Submission to the 2004 Workshop on Compilers and Tools for Constrained Embedded Systems

Applying Scalable Interprocedural Pointer Analysis to Embedded Applications

Hillery C. Hunter Erik M. Nystrom Wen-mei W. Hwu
Center for Reliable and High-Performance Computing
Department of Electrical and Computer Engineering
University of Illinois at Urbana-Champaign
{hhunter, nystrom, hwu}@crhc.uiuc.edu

Abstract

This paper evaluates six different types of interprocedural pointer analyses on 22 telecommunication and media applications and describes their application to an SRAM power reduction technique. This configurable SRAM *provides differentiation of data access time and port counts within a single on-chip structure. Scheduling for configurable SRAM relies on inter-procedural dependence analysis for safe assignment of program data objects to on-chip storage regions with little or no performance degradation. It thus provides a meaningful vehicle for exploring the applicability of and need for various interprocedural analysis techniques, as well as a demonstration of how compilation and hardware techniques can be combined to achieve optimization beyond that for performance.*

1. Introduction

The non-orthogonal instruction mixes, unique features, and tight code size constraints of DSP/embedded processors traditionally necessitated programming embedded applications in assembly code. However, the increasing complexity of applications, combined with the growing popularity of relatively uniform VLIW processors, have created a need for high-level language coding and compilation. The embedded domain remains fundamentally constrained by power and energy consumption, but wider instruction issue processors are often more power- and energy-hungry than their predecessors. A new generation of compilation techniques is needed to address not only performance, but also power, in a joint hardware-software approach.

One of the more difficult static compiler analysis techniques is interprocedural pointer analysis, which provides compile-time knowledge of program memory activity by resolving the data locations reachable through a program's pointers. A number of previous works have addressed the power consumption of the memory subsystem, but many avoided the difficulties associated with pointer use in high-level language codes. Compiler-managed scratchpad buffers have been used, for example, for spill code [1]; proposals have been evaluated primarily on the basis of kernels, and so do not consider benchmarks with high-level language pointer usage; or proposals achieve power savings through modifications to hardware caches [2], thus incurring potential performance degradation, which may not be appropriate for an embedded environment. This paper evaluates the necessity for, and efficacy of, various interprocedural pointer analysis techniques on a suite of telecommunication and media applications, and illustrates their ties with a compiler-hardware memory power savings approach.

We provide two primary contributions to the developing field of embedded compilation: (1) an evaluation of the applicability and utility of six forms of inter-procedural pointer analysis to a suite of telecommunication and media applications; and (2) a demonstration of the synergy possible between such analytical compiler analysis and microarchitectural techniques designed for power reduction. Section 2 describes a selection of five pointer analysis options available for C language programs, and evaluates their applicability to classes of the studied applications. The correspondence of pointer analysis to net performance changes in classically optimized benchmark codes is also presented. Section 3 describes the *configurable SRAM* technique for power savings in on-chip SRAM, and Section 4 provides background on the compiler techniques required to schedule code for configurable SRAM. The impact of pointer analysis quality and results on net configurable SRAM power savings is then presented in Section 5.

2. Scalable Pointer Analysis

Conceptually, the goal of a pointer analysis system is to describe symbolically 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 intraprocedural register promotion, scheduling, memory dataflow, debugging, and verification.

While the goal of pointer analysis is straightforward, the realization of this goal is complicated by the variety of formulations, each with its own strengths and weaknesses with respect to accuracy and scalability. Accuracy is a measure of how closely the derived pointer relationships match those actually realizable by the program. Scalability is a measure of 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 problem, along with the usefulness of the results, has led to a large body of research [3].

The basis for this work an interprocedural pointer analysis introduced in [4] and integrated with the IMPACT compiler framework [5]. This algorithm is very accurate, yet has also been shown to work very efficiently for large programs having complicated pointer usage. The analysis is an Andersen style, offset-based field sensitive, bottom-up/top-down context sensitive, heap object specializing pointer analysis that generates unique identifiers for all accessed objects. 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 benchmarks analyzed in this paper, the time to perform the most accurate form of analysis ranged from less than 0.01 second to 1 second. The applicability and utility of six different pointer analysis configurations are evaluated in this paper:

- *Andersen* style [6] *(And)* vs. *Steensgaard* style [7] *(Stgd)* : If a single location points to two different objects, *And* will continue to track the objects separately, while *Stgd* will unify them, tracking them as a single object.
- Context (in)sensitivity *(CI / CS)* : A CS algorithm is able to keep data flow along different call paths separate. For example, should functions A and B both call C, a CI algorithm may show false data flow from A into B, by way of C.
- Field (in)sensitivity *(FI / FS)* : A FS algorithm is able to distinguish between fields of an aggregate object. In a FI analysis, the C-language expressions `x.f1` and `x.f2` are equivalent to `x`.
- Heap (in)sensitivity [8] *HI / HS* : HS enables an algorithm to distinguish between different heap allocated objects even though allocated by the same call to `malloc()`.
- Zero-weight path exclusion *(ZE)*: This excludes all zero weight expressions from the pointer analysis. While not valid for general dependence analysis, Section 5 will detail its utility for placement of objects into a configurable SRAM.

2.1. Result clarity

Pointer analysis is commonly described in terms of *points-to* sets, or the locations that a given pointer may reference. The implemented analyses all generate points-to sets, but for the purpose of clarity in this work, analysis results will be described in terms of data *object* references. An object may be an array; a scalar; a structure; or a union, and may reside in local, global, or heap space. The studied benchmarks are listed in Table 1.

Table 1: Benchmarks from MediaBench [9], ETSI [10], EEMBC, JasPer [11], and independent authors.

Applications	Source (LOC)	Description
adpcm {*dec* | *enc*}	MediaBench (13K)	Intel/DVI ADPCM codec
g721 {*dec* | *enc*}	MediaBench (14K)	Voice compression according to the CCITT G.721 standard
g724 {*dec* | *enc*}	ETSI ({20K, 25K})	GSM 06.60 EFR speech transcoding, state-of-the-art digital cellular comm.
gsm {*dec* | *enc*}	MediaBench (19K)	Lossy sound compression according to the GSM 6.10 RPE-LTE standard
jpeg {*dec* | *enc*}	MediaBench (37K)	Independent JPEG Group photo decoder/encoder
h263 {*dec* | *enc*}	Independent ({19K, 21K})	H.263 video decoder/encoder, Telenor implementation
mpeg2 {*dec* | *enc*}	MediaBench ({22K, 20K})	MPEG-2 video decoder
mpg123	Independent (24K)	MPEG-2 Layer 3 audio decoder.
jpg2Kdec	Independent (43K)	JPEG-2000 Part-1 standard (ISO/IEC 15444-1) ref. dec. from JasPer Project.
autcor00	EEMBC (17K)	Autocorrelation: Code-Excited Linear Pred. (CELP) filter transfer function matching
conven00	EEMBC (17K)	Convolutional encoder: V.xx modem output stream encoding to enable error det/cor
fbital00	EEMBC (17K)	Bit allocation: data distribution into ADSL frequency bins
fft00	EEMBC (17K)	Fast Fourier Transform: 256-point complex decimation in time algorithm
viterb00	EEMBC (17K)	Viterbi decoder: embedded IS-136 channel coding

For five of the studied benchmarks, Figure 1 plots the average number of `loads` and `stores` that access an object field. Objects are divided into non-heap and heap categories, and eight combinations of the described pointer analysis styles are shown. The configuration string on the x-axis is composed of a *c* if context sensitive, a *f* if field sensitive, and an *h* if heap sensitive. In this figure, all analysis combinations are Andersen style. Analysis was performed for all benchmarks listed in Table 1, but only a subset is shown due to the marked similarity of results across groups of benchmarks – beneath the name of each selected benchmark in Figure 1 are grayed names of benchmarks that behave similarly.

Many telecommunication applications in this paper have trivial pointer behavior and are similar to *fbital00* in that any combination of analyses works equally well. The remaining applications behave like *gsmdec*, in which only field sensitivity affects the result; *mpeg2dec*, which requires both field and heap sensitivity; and the *jpegs*, which require a synergy between all three techniques.

The importance of heap sensitivity to achieving accurate, yet efficient interprocedural pointer analysis of SPEC programs with dynamic memory allocation is discussed in [8]. 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 liter-

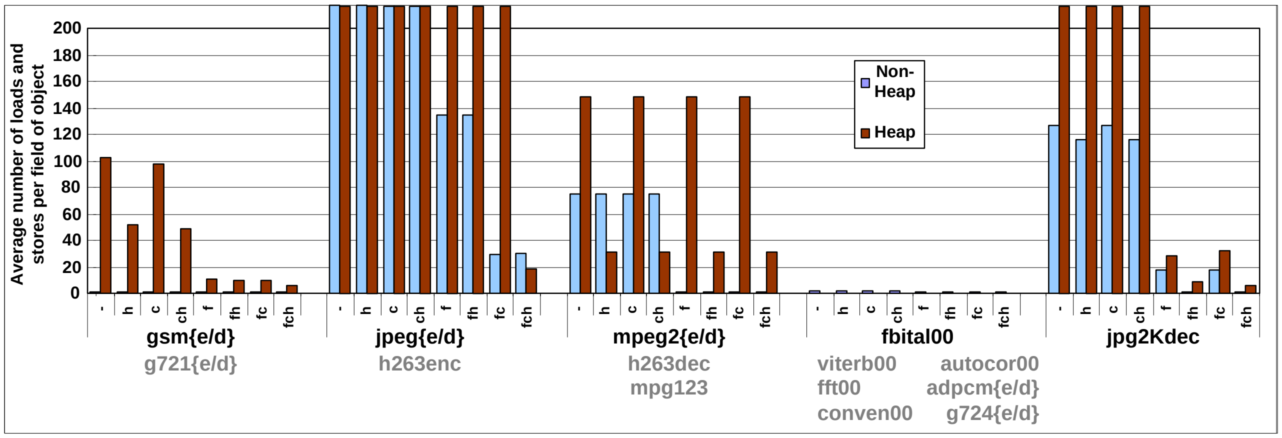

Figure 1: Average number of `loads` and `stores` that access a field in heap and non-heap objects for eight pointer analysis combinations, indicated as: *c* if context sensitive, *f* if field sensitive, and *h* if heap sensitive.

ature [12][13] and included in commercial products [14]. Correspondingly, the studied application suite is clearly divisible into two parts: approximately one-half of the applications (telecommunication applications) do not use heap-allocated variables, while others, in particular video codecs, rely relatively heavily on dynamically allocated space. For these applications, *heap sensitivity* was found to be essential to accurate pointer analysis.

2.2. Performance

The clarity of pointer analysis results impact the compiler's scheduling and optimization abilities. Figure 2 shows the relative performance of benchmarks classically optimized after several types of pointer analysis. The baseline for comparison is an Andersen style, field-sensitive, context-sensitive pointer analysis, with heap sensitivity where appropriate (*Afch*). The classical compiler optimizations performed include constant and copy propagation, dead code removal and invariant code removal; the optimized codes were scheduled for a baseline eight-wide VLIW architecture, with function unit distribution similar to that of the Texas Instruments C6x processors.

Dependences that result from pointer analysis clearly have some effect on the compiler's ability to optimize code. However, since most of these benchmarks have relatively simplistic pointer behavior, the quality of the pointer analysis does not significantly impact net performance benefit gained through classic optimization. If these are the only applications of interest and only classic optimization is desired, it would appear sufficient to use a relatively simplistic, low-cost analysis algorithm, such as context-insensitive Steensgaard analysis. However, as will be described in Sections 4 and 5, the accuracy of pointer analysis, and thus resolution achieved in understanding object references, can aid other optimizations, specifically placement of data to achieve power savings.

3. Configurable Data Storage

As both leakage currents and overall on-chip energy consumption become increasing concerns in the embedded domain, new methods for power management are needed to control the ever-increasing proportion of power consumed for data storage. 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 critical data); use software control for predictability and to maintain applicability for real-time systems; and provide explicit control over cycle degradation, to allow optimization of system-level power. Power can be saved by providing customized data storage arrays with varied port and latency properties, but this typically requires physical SRAM array partitioning, and thus compromises the generality and reusability of both hardware and software.

We have instead proposed use of recent SRAM technologies to configure access latency and port counts in multi-line SRAM regions. A dual-port, single cycle, software-managed SRAM is assumed as a baseline, and the capability to increase the access latency or shut-down either port in a multi-line *zone* is provided. SRAM configuration leverages recent circuit-level techniques for power savings in unified SRAM arrays, including *Self Reverse Biasing* [15], *Floating Bitlines* [15, 16], and V_{dd} *Throttling* [17]. Each was previously applied for hardware-controlled power savings, but to increase predictability of both access time and average power consumption, we instead propose to use them in a collaborative hardware-software approach. The microarchitecture-level details of these techniques are described in [18].

Our compiler analysis has found that 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) are present in

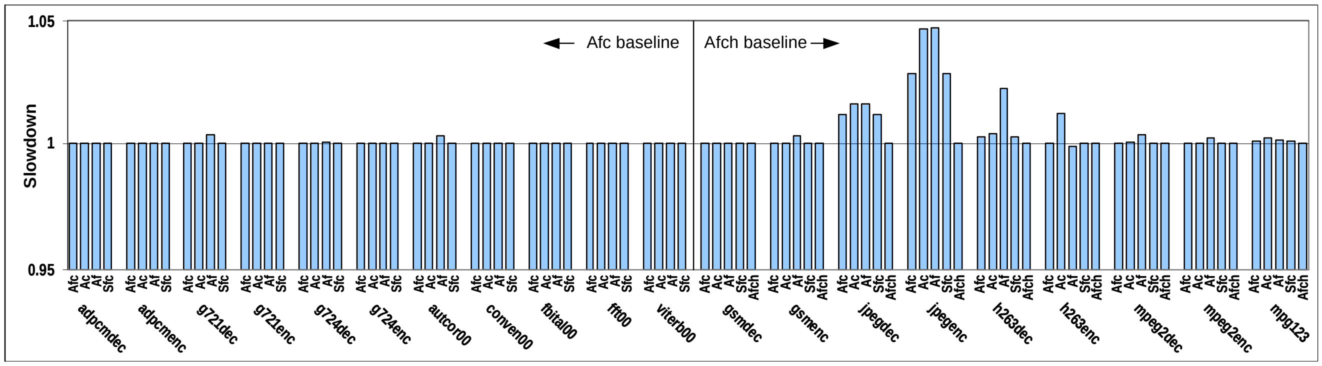

Figure 2: Performance changes realized by classical optimization after various pointer analysis combinations, indicated as: *A* if Andersen style, *S* if Steensgaard, *c* if context sensitive, *f* if field sensitive, *h* if heap sensitive.

telecommunication and media applications. With the ability to identify and exploit these forms of code *flexibility*, we realize 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 [19].

Interprocedural pointer analysis is used to provide unique identifiers for all application data objects. Objects corresponding to global variables; stack arrays, unions, or structures; and heap allocation sites are of interest to the object scheduler presented in this work. The compiler combines this memory access information with knowledge of operation slack to choose which port- and latency-customized SRAM region each object will be stored in. Dynamic and static power savings are realized with little or no performance degradation. It is important to note that where performance is degraded, it is *predictably* degraded, since the degradation is based on a static schedule, not runtime phenomena. The implemented compiler techniques, enabled by interprocedural pointer analysis, work for general code, so no application modifications are required, and fast data access is still provided when necessary.

4. Object Scheduling

Placement of program data objects into storage with varied properties will be discussed in this section as the *object scheduling* optimization problem. Object scheduling determines the number, type, and size of profitable data storage configurations for a given application.

Object scheduling 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 impacting 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 while controlling performance degradation. If no performance degradation can be tolerated, object scheduling consists only of observing data usage and looking for cases where a particular constraint can be applied without penalty. For realistic applications, however, slack will be present, and when an architecture provides multiple load-store units (and data ports), memory slot issue choices also become available.

Performance degradation has a direct impact on the energy consumed to execute an application, since either runtime is increased or cycle time must be shortened. For some systems, neither option will be tolerable, thus the object scheduler's threshold for acceptable performance degradation can be set to zero, which will not change application performance. If the scheduler may save power at the expense of increased schedule height, *cost* and *benefit* metrics are used to guide the placement of data objects in low-power storage.

Cost can be measured as both schedule height increase and 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 assumed power savings for each SRAM configuration, listed in Table 2. For the circuit-level derivations of these values, the reader is referred to [18]. Configurations for individual objects should be prioritized according to the power savings realized (a) in particular configuration zones; 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 runtime. For object scheduling, dynamic power cost can be anticipated using a memory profile to measure accesses for a sample input or inputs.

Table 2: Static and dynamic power savings assumed per SRAM configuration type.

	Single Port, Long Latency	Dual Ports, Long Latency	Single Port, Short Latency
Static Power Savings (Continuous)	80%	80%	22.5%
Dynamic Power Savings (Per Access)	46%	46%	0%

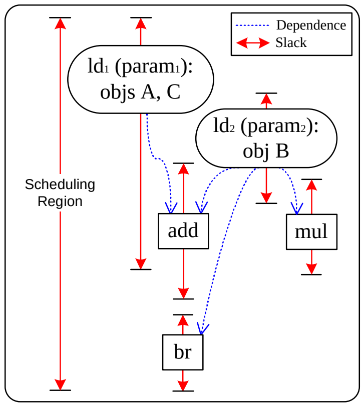

Figure 3: Example operation flexibility and object relationships.

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, loop, superblock [20], or hyperblock [21]). Data objects not conjoined to or interactive with any other objects may be scheduled in isolation. 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 zone, 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 woken up, 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. A long latency `load` operation also blocks the SRAM data output driver and a register file port at a later cycle than for the standard data access latency, causing additional potential resource conflicts.

In Figure 3, 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 and share a common dependent, the `add` operation. 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. Our compiler algorithm was able to identify a significant amount of object-based port flexibility (average of 62.7% of dynamic accesses, 67.8% of total data space) and a moderate amount of latency flexibility (average 9% of dynamic accesses, 21% of total data space) for telecommunication and media applications. When a small amount of performance degradation was tolerated (3.4%), many more objects could be scheduled to long latency regions, bringing average latency flexibility to 37.1% of dynamic accesses and 66.8% of total data space.

4.1. Compiler implementation

The IMPACT compiler framework [5] was used to generate all object and operation schedules as well as to perform all analyses. The implemented framework handles both the “perfect” case in which no performance degradation (schedule height increase) is tolerable, as well as a tunable, profile-based scheduling heuristic to enable power savings to be balanced against performance degradation.

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). For this reason, we first promote arrays, structures and unions from local to global variable space so they may be scheduled into a low-power region. All other stack data are kept in non-configured, *i.e.*, dual-port, short latency storage. Interprocedural analysis tags static `load` and `store` operations with a complete list of all objects that they may access during the course of program execution, thus providing information as to which objects are conjoined. After code optimization, object-centric memory profiling determines relative data object usage as well as access weights for individual `load` and `store` operations.

Currently, objects are greedily scheduled according to their overall profiled access counts. The scheduling region used is an extended basic block. The highest ranked, unplaced object is selected and a baseline schedule is calculated for each region in which the object is accessed, assuming that the object may be accessed through two ports, with a single-cycle `load` latency. If objects interactive with the current object have already been committed, the reduced slack will be reflected in this baseline schedule. This bestcase schedule height is recorded, along with an anticipated execution time based on control-flow profile weights.

Next, new region schedules are computed to evaluate the effects of placing the current object into a restricted SRAM configuration. However, 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, tests of *Latency-and-PortB* and *PortB-only* are eliminated from the valid configurations for the current object.

Once the valid region schedules are generated, statistics are summarized for all scheduling regions. If no performance degradation can be tolerated for the particular scheduling run, a strict *zero schedule height increase* restriction is enforced for all object scheduling decisions. A global decision is made for the object (taking into account factors from all regions) and is *committed* based upon benefit and cost metrics, as weighted against a specified performance bound. The process then continues with the next highest ranked object.

4.2. Heap objects

The placement of heap allocated objects into a configuration zone requires a specialized `malloc()` implementation. Specialization of `malloc()` according to object size has been previously implemented to decrease allocation time [22] and minimize fragmentation [23]. Similarly, specialization according to chosen data access properties should have little impact on the overhead of dynamic memory allocation. While the separation of heap objects by pointer analysis may increase scheduling freedom (*h*-type analysis), zone assignment through a specialized `malloc()` call requires all objects created by a particular static call site to be assigned to the same configuration zone.

One possible drawback to the placement of heap objects into zones is allocation beyond the zone's capacity. Using extensive profiling, it was found that though some allocation sites spawn megabytes of space over the course of program execution, the maximum total live object size for most allocation sites is small and constant. Leveraging this characteristic, `malloc()` calls 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 into a non-configured, *i.e.*, maximum power zone.

5. Configurable SRAM/Pointer Analysis Interaction

As was described in conjunction with Figure 1, there are applications in the studied suite for which an order-of-magnitude change in the size of the points-to graph is observed across different interprocedural analysis types. Interestingly, however, after classical optimizations (Figure 2), these changes had little effect on net performance. For the scheduling of data to configurable SRAM, however, the `load`/`store`-to-object markings described by the points-to set have a much more perceptible influence on the net outcome of object scheduling, namely on the total power savings.

Figure 4 shows the dynamic and static SRAM power savings corresponding to application object schedules after each type of pointer analysis. Dynamic and static power savings are shown in the same column, so that total power on the y-axis would be 200%, *i.e.* a value of 40% static power savings represents a 40% decrease in static power alone. [16] indicates that in future technologies, energy dissipation from leakage (static power) may equal that of switching (dynamic power), so it is important to consider both components of total power consumption.

The purpose of this work is to characterize pointer usage and interprocedural analysis effects in the studied application domain, and to demonstrate the interaction of such analysis with a potential power optimization. To limit the extent of optimization-specific discussion in this section, only results for objects scheduled without performance degradation, *i.e.* without allowing schedule height to increase, will be presented. Similar trends, and greater power savings, apply to the interaction of pointer analysis with the object scheduler when given freedom to degrade net application performance.

Of most interest in Figure 4 is the additional power saved when *heap sensitivity (h)* is enabled. Heap sensitivity can significantly reduce the number of conjoined objects: for the *jpegdec* benchmark, for example, the average number of conjoined objects per memory operation was 75 for the base case (Afc), but for the heap-sensitive run (Afch), there were only 12 conjoined objects per memory operation. This noticeably reduces the run-time of the object scheduler, since many fewer constraints need to be considered during scheduling of individual objects. Viewed from another perspective, only 24.5% of *jpegdec* memory operations are indicated as accessing a single object for the baseline pointer analysis, while twice this amount (54.1%) only have an arc to a single object when heap sensitivity is used.

In contrast, the object schedule improvement observed for *gsmdec* with heap sensitivity does not occur due to a reduction in the constraints of conjoined and interactive objects, but rather through an increase in schedule slack. Specifically, the heap dependences annotated on several `jsr` (jump to subroutine) operations are removed, creating more slack on several critical `load` operations, and allowing the heap-allocated `gem_state` object to be placed into longer-latency (1-port, 3-cycle access) storage without increasing the schedule height of the accessor code blocks.

Figure 4 also includes results for object scheduling performed from zero-weight path exclusion pointer analysis,

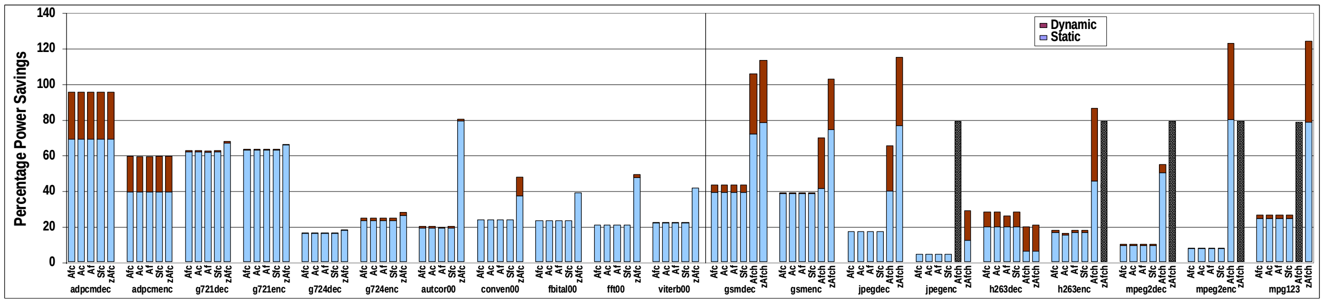

Figure 4: Power savings for data objects allocated to configurable SRAM: pointer analysis combinations indicated as: *A*=Andersen, *S*=Steensgaard, *c*=context sensitive, *f*=field sensitive, *h*=heap sensitive, *z*=zero-weight exclusion. Black bars are placeholders for values to appear in final draft.

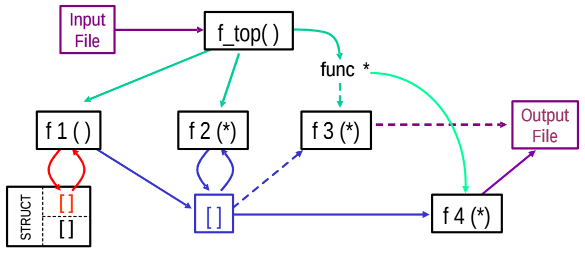

Figure 5: Typical telecom application construction.

(z). All forms of pointer analysis used exclude unreachable program code, but as explained in Section 2, zero-weight exclusion analysis also eliminates code not touched during an execution profile. This is not a generally safe form of analysis, and is not suited to general dependence analysis. However, due to several typical properties observed in the studied application suite, it is a useful baseline for object scheduling. Figure 5 depicts a typical form of telecom and media application construction in C: green lines between functions represent control flow; other arcs represent data flow. A top-level function (`f_top`) is responsible for buffering input from 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 is 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 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 routines included in a single application. 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 the *z* category in Figure 4, where static power savings increases, but there is little or no change in dynamic power across analysis types (*e.g.* 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 corresponding arrays, an `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.

Extensions for final draft

Due to space limitations in the submission guidelines, we have omitted our result graphs which clarify and justify why zero-weight interprocedural pointer analysis is meaningful for object scheduling. These results and explanation will be included in the extended final draft.

6. Related Work

[24] provides an empirical analysis of five different pointer analysis styles (including Steensgaard vs. Andersen and context- and field-sensitivity), but they do not consider media and telecommunication applications.

Differentiated `load` servicing through configurable SRAM is most closely related to the body of work which has studied data partitions for embedded systems with *scratchpad* resources [25][26]. In these systems, a small, low-power, software-managed SRAM is used to cache variables otherwise stored in cache or DRAM. Static allocation to scratchpad, *e.g.* [27], is a necessary compiler technique for chipsets which include scratchpad, but is in general a less flexible approach to power savings than configurable SRAM, since the size of the scratchpad and DRAM cannot be changed. Dynamic approaches to scratchpad allocation [25][28], in which the values in the scratchpad change as the program executes, have a somewhat different focus than this work, since they are concerned more with accommodation of temporal usage patterns than determining and scheduling for whole-program power savings opportunities. Many other proposals in the area of embedded memory power reduction, *e.g.*, [29], and [30], focus on conflict avoidance, data layout in the context of scratchpad management, or application-specific systems, so have a somewhat different focus and problem definition than this work. With regard to alias analysis in the memory power optimization space, some 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 [1], or proposals have been evaluated primarily on the basis of kernels and so do not use benchmarks with high-level language pointer usage. It should also be noted, however, that the configurable SRAM and object scheduling techniques are complementary to use of a small scratchpad buffer, specialized memory, or shared memory in a multi-processor SoC – if a programmer or tool has determined that particular data items are best-suited to another storage type, those objects can be removed from consideration for configurable SRAM placement, further reducing the complexity of the object scheduling problem.

7. Conclusions

Since power will continue to be a fundamental constraint of embedded systems, successful future optimizations will likely leverage both compilation and hardware techniques to yield overall power reduction. This work focused on the role of pointer analysis in the efficacy of such a task. There are three key insights: 1) Many, small telecom applications require only a very basic analysis to obtain the desired results 2) Application complexity is increasing, and, for these, it is beneficial to leverage a more accurate analysis 3) The refinement of pointer analysis results using profiling information can lead to substantial benefits.

References

[1] 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)*, pp. 100–104, 1998.

[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] M. Hind, “Pointer analysis: Haven’t we solved this problem yet?,” in *PASTE*, pp. 54–61, 2001.

[4] 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.

[5] W. Hwu *et al.*, “Compiler technology for future microprocessors,” *Proc. IEEE*, vol. 83, pp. 1625–1640, Dec. 1995.

[6] L. O. Andersen, *Program analysis and specialization for the C programming language*. PhD thesis, Univ. of Copenhagen, 1994.

[7] 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.

[8] 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.

[9] 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)*, pp. 330–335, Dec. 1997.

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

[11] ITU-T SG8, “JasPer software reference manual version 1.500.4,” Tech. Rep. ISO/IEC JTC 1/SC 29/WG 1 N2415, ISO/IEC, Dec. 2001.

[12] 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.

[13] M. Millberg, A. Postula, and A. Hemani, “An efficient dynamic memory manager for embedded systems,” in *Proceedings of the ICDA Conference*, 2000.

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

[15] 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.

[16] 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*, pp. 137–147, May 2002.

[17] 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*, pp. 148–157, May 2002.

[18] H. Hunter, S. Ryoo, J. Player, D. Connors, and W. Hwu, “Exploiting load flexibility for embedded power savings,” in *Submission to the 37th Int’l Symposium on Microarchitecture (MICRO-37)*, 2004.

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

[20] 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.

[21] 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.

[22] 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.

[23] MicroQuill, “SmartHeap technical specification.” http://www.microquill.com/smartheap/sh_tspec.html.

[24] 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*, pp. 113–123, 2000.

[25] 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.

[26] P. R. Panda, F. Catthoor, N. D. Dutt, K. Danckaert, E. Brockmeyer, C. Kulkarni, A. Vandercappelle, 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.

[27] 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.

[28] 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*, pp. 690–695, 2001.

[29] M. T. Kandemir and A. Choudhary, “Compiler-directed scratch pad memory hierarchy design and management,” in *Proceedings of the 39th Design Automation Conference*, pp. 628–663, 2002.

[30] P. Panda *et al.*, “Data memory organization and optimizations in application-specific systems,” *IEEE Design and Test of Computers*, pp. 56 – 68, 2001.