Heap Data Allocation to Scratch-Pad Memory in Embedded Systems

Angel Dominguez, Sumesh Udayakumaran and Rajeev Barua
Department of Electrical & Computer Engineering
University of Maryland
College Park, MD 20742, U.S.A
{angelod, skumaran, barua}@eng.umd.edu

***Abstract*— This paper presents the first-ever compile-time method for allocating a portion of the heap data to scratch-pad memory. A scratch-pad is a fast directly addressed compiler-managed SRAM memory that replaces the hardware-managed cache. It is motivated by its better real-time guarantees vs cache and by its significantly lower overheads in access time, energy consumption, area and overall runtime. Existing compiler methods for allocating data to scratch-pad are able to place only global and stack data in scratch-pad memory; heap data is allocated entirely in DRAM, resulting in poor performance. Runtime methods based on software caching can place heap data in scratch-pad, but because of their high overheads from software address translation, they have not been successful, especially for heap data.**

In this paper we present a dynamic yet compiler-directed allocation method for heap data that for the first time, (i) is able to place a portion of the heap data in scratch-pad; (ii) has no software-caching tags; (iii) requires no run-time per-access extra address translation; and (iv) is able to move heap data back and forth between scratch-pad and DRAM to better track the program's locality characteristics. With our method, global, stack and heap variables can share the same scratch-pad. When compared to placing all heap variables in DRAM and only global and stack data in scratch-pad, our results show that our method reduces the average runtime of our benchmarks by 34.6%, and the average power consumption by 39.9%, for the same size of scratch-pad fixed at 5% of total data size.

***Index Terms*— heap allocation, scratch pad, SRAM, tightly coupled memory, TCM, dynamic allocation.**

I. Introduction

The proposed research presents an entirely new approach to heap allocation for embedded systems with scratch-pad memory. In embedded systems, program data is usually stored in one of two kinds of write-able memories – SRAM or DRAM (Static or Dynamic Random-Access Memories). SRAM is fast but expensive while DRAM is slower (by a factor of 10 to 100) but less expensive (by a factor of 20 or more). To combine their advantages, often a large DRAM is used to build low-cost capacity, and then a small SRAM is added to reduce runtime by storing frequently used data. The gain from adding SRAM is likely to increase in the future since the speed of SRAM is increasing by 60% a year versus only 7% a year for DRAM [28].

In desktops, the usual approach to adding SRAM is to configure it as a hardware cache. The cache dynamically stores a subset of the frequently used data. Caches have been a success for desktops – a trend that is likely to continue in the future. One reason for their success is that code compiled for caches is portable to different sizes of cache; on the other hand, code compiled for scratch-pad is usually customized for one size of scratch-pad. Binary portability is valuable for desktops, where independently distributed binaries must work on any cache size. In embedded systems, however, the software is usually considered part of the co-design of the system: it resides in ROM, and cannot be changed. Thus, there is really no harm to the binaries being customized to one memory size, as required by scratch pad. Source code is still portable, however: re-compilation with a different memory size is automatically possible in our framework. This is not a problem, as it is already standard practice to re-compile for better customization.

For embedded systems, the serious overheads of caches are less defensible. Caches incur a significant penalty in area cost, energy, hit latency and real-time guarantees. All of these other than hit latency are more important for embedded systems than desktops. A detailed recent study [11] compares caches with scratch pad. Their results are startling: a scratch pad has 34% smaller area and 40% lower power consumption than a cache of the same capacity. These savings are significant since the on-chip cache typically consumes 25-50% of the processor's area and energy consumption, a fraction that is increasing with time [11]. Even more surprising, the runtime cycle count they measured was 18% better with a scratch pad using a simple static knapsack-based [11] allocation algorithm, compared to a cache. Defying conventional wisdom, they found absolutely no advantage to using a cache, even in high-end embedded systems in which performance is important. With the superior dynamic allocation schemes proposed here, the runtime improvement will be larger. Given the power, cost, performance and real time advantages of scratch-pad, and no advantages of cache, it is not surprising that scratch-pads are the most common form of SRAM in embedded CPUs today (*e.g.,* [20], [1], [40], [54], [39]), ahead of caches. Trends in recent embedded designs indicate that the dominance of scratch-pad will likely consolidate further in the future [48], [11], for regular as well as network processors.

Although many embedded processors with scratch-pad exist, compiling program data to effectively use the scratch-pad has been a challenge. The challenge is different for global and stack variables, on one hand, and heap variables, on the other. This is explained below.

Recent advances have made much progress in compiling *global and stack variables* into scratch-pad memory. Two classes of compiler methods for allocating global and stack variables to scratch-pad exist. First, *static* allocation methods are those in which the allocation does not change at runtime; these include [10], [51], [29], [9], [50] and others not listed here. In such methods, the compiler places the most frequently used variables, as revealed by profiling, in scratch pad. Placing a portion of the stack variables in scratch-pad is not easy – [10] is the first method to solve this difficulty by partitioning the stack into two stacks, one for scratch-pad and one for DRAM. Second, recently proposed *dynamic* methods improve upon static methods by allowing variables to be moved at runtime; we know of two dynamic methods [56], [37]. Being able to move variables enables tailoring the allocation to each region in the program rather than having a fixed allocation as in a static method. Dynamic methods aim to keep variables that are frequently accessed in a region in scratch-pad during the execution of that region. The methods in [56], [37] explicitly copy variables from DRAM into scratch-pad just before a region in which they are expected to the frequently accessed. Other variables are evicted to DRAM by explicit copy out instructions to make space for incoming variables. The method in [37] is restricted to arrays accessed through affine functions; the method in [56] is fully general and can place all global and stack variables in scratch pad dynamically, even in the presence of unrestricted pointers.

Allocating heap data to scratch-pad has proven far more difficult. Indeed, as far as we know, no one has proposed a successful method to allocate a portion of the heap variables to scratch-pad memory. To see why, it is useful to understand heap variables and their available analysis techniques; an overview follows. Heap objects are allocated in programs by dynamic memory allocation routines, such as **malloc** in C and **new** in Java. They are often used to store dynamic data structures such as linked lists, trees and graphs in programs. Many compiler techniques for heap analysis group all heap objects allocated at a single site into a single heap ”variable”. Additional techniques such as shape analysis have aimed to identify logical heap structures, such as trees. Finally, in languages with pointers, pointer analysis [24], [52] is able to find all possible heap variables that a particular memory reference can access.

Having understood heap variables, let us consider why heap data is difficult to allocate to scratch-pad memory at compile-time. Two reasons for this difficulty are as follows. First, heap variables are usually of *unknown size* at compile-time. For example, linked lists, trees and graphs allocated on the heap typically have a data-dependent number of elements, and thus a compile-time-unknowable size. Thus it is difficult to guarantee at compile-time that the heap variable will fit in scratch-pad. Such a guarantee is needed for a compiler to place that heap variable in scratch-pad. Second, moving data at runtime, as is required for any dynamic allocation method to scratch-pad, usually leads to the *invalid pointer problem* if the moved data is a heap object. To see why, consider that heap data often contains pointers to other heap data, such as the child pointers in a tree node. When a heap object is moved between scratch-pad and DRAM, all the pointers into it become invalid. Updating all these pointers at runtime is prohibitely expensive since it involves scanning through entire, possibly large, heap structures at each move. Static methods avoid this problem, but lack the better per-region customization of dynamic methods.

Lacking compile-time methods for heap allocation to scratch-pad, people have investigated runtime methods, *i.e.,* methods that decide what to place in scratch-pad only at runtime; however largely they have not been successful. Primary among runtime methods is *software caching* [38], [27]. This class of methods emulate the behavior of a hardware cache in software on the scratch-pad. Since caches decide their contents at runtime, software caching decides the subset of heap data to store in scratch-pad at runtime. Software caching is implemented as follows. A tag consisting of the high-order bits of the address is stored for each cache line in software. Before each load/store, additional instructions are compiler-inserted to mask out the high-order bits of the address, access the tag, compare the tag with the high-order bits and then branch conditionally to hit or miss code. Some methods are able to reduce the number of such inserted overhead instructions [38], but much of it remains, especially for non-scientific programs and for heap data. This implementation points to the primary drawbacks of software caching: the inserted code before each load/store adds significant overhead, including (i) additional runtime; (ii) higher code size and dollar cost; (iii) higher data size and cost from tags; and (iv) higher power consumption. These overheads, especially for heap data, can easily exceed the gains from locality.

In conclusion, lacking compile-time methods and successful runtime methods for heap allocation to scratch-pad, *heap data is usually not allocated to scratch-pad at all* in modern embedded systems; instead it is placed entirely in DRAM.

Heap allocation method This paper proposes a new dynamic method for allocating a portion of the heap to scratch-pad. The method is outlined in the following three steps. First, it partitions the program into *regions* such that the start and end of every procedure and every loop is the beginning of a new region, which continues until the next region begins. This is not the only possible choice of regions; the reasons for this choice are in section IV. Second, straightforward analysis is done to determine the time-order between the regions by finding the set of possible predecessors and successors of each region. Third, copying code is inserted by the compiler at the beginnings of regions to copy in portions of heap variables into the scratch-pad; these portions are called *bins*. A cost-model driven heuristic method is used to determine which variables to copy in and what size their bins should be.

At first glance, the above method is similar in flavor to our compile-time dynamic method for global and stack data [56] in that it copies in data when the compiler expects that it will be frequently used in the next region. However its real novelty is seen in how it solves the unknown size problem and the invalid data problem mentioned earlier. How these problems are solved result in virtually every aspect of the algorithm being different from our earlier method. The solutions to the unknown size problem and the invalid data problem are described in the next two paragraphs.

First, our heap method solves the problem of unknown-size heap variables by **not** storing all the elements of a heap allocation site in its SRAM bin, but only a fixed-size *subset*. (From here on "site" is used to mean the objects allocated at a site). This fixed-size portion for each site in scratch-pad is called the *bin* for that site.. Fixed-size bins make possible compile-time guarantees that they will fit in scratch-pad. For example consider a linked list having nodes of size 16 bytes and an **unknown** number of nodes. Here, the compiler may allocate a bin of size 192 bytes for the allocation site of the list – this will hold only 192/16 = 12 nodes from the list. The total number of nodes may be larger, but only twelve are allocated to the bin and the rest to DRAM. A bin is copied into SRAM just before every region where it is accessed (unless it is already in SRAM) and is subsequently evicted before a region where it is not[1]. When a bin is evicted it is maintained as a contiguous block in DRAM; it is copied back later to SPM contiguously if needed. This ensures that the offset of a particular data object inside its bin is not changed during its lifetime, regardless of whether the bin is in SRAM or DRAM.

It is important to understand that objects may be allocated or freed from either memory – *separate free lists are maintained for each bin, and there is a unified free list for heap data that are not in bins*. The bins are moved between SRAM and DRAM, but non-bin data is always in DRAM. New objects from a site are allocated to its bin if space is available, and to DRAM otherwise. Sites having a higher data re-use factor are assigned larger bins to increase the total runtime gain from using bins. Figure 1(a) is an example showing the five allocation sites for a hypothetical program and bin size and regions-of-access for each site. Four regions 1-4 are assumed in the program, numbered in order of their timestamps (defined in section V).

[1]This is the default behavior but it is selectively changed for some regions by the optimizations in section IX.

Second, our heap method solves the problem of invalid pointers by never changing the bin offset or size for any site in the regions it is accessed. For example, figure 1(c) shows the bin layout in scratch-pad for the sites in figure 1(a), for each of the four regions in the program. (Ignore figure 1(b) for now.) It shows that the offset of each bin is always the same when it is present. For example, site A is allocated at the same offset 512 in both regions 2 & 4 it is accessed. An entire bin may be evicted to DRAM in a region it is not accessed (as revealed by pointer analysis). For example, site A is copied to DRAM in region 3. Moving a bin to DRAM temporarily results in invalid pointers that point to objects in the bin, but those invalid pointers are never dereferenced as they occur only during regions that pointer analysis has proven to not access the site.

Our heap method effectively improves runtime for three reasons. First, like a cache it allocates more frequently used data to SRAM. This is achieved by assigning larger bins to sites with high frequency-per-byte of access. Heap area is traded off with global and stack data as well – the frequency-per-byte of variables of all types (from profile data) are compared to determine which ones are actually copied to SRAM[2]. Any variable is placed in scratch-pad only if the cost model estimates that the benefits of locality exceed the cost of copying. Second, like caching our heap method is able to change the contents of the SRAM at runtime to match the requirements of different phases of the program. The allocation is dynamic, but is decided at compile-time. Third, unlike software caching, our method has no tags and no per-memory-access overhead.

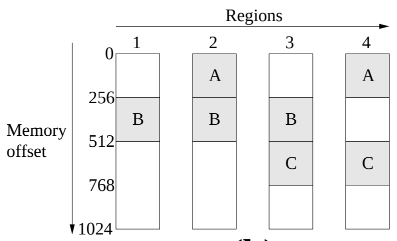
Comparison with caches The primary measure of success of our heap method is **not** its performance vs. hardware caches, but vs. all-DRAM heap allocation, the only existing method for scratch-pad. (Software caching has not been a success). There are a great many chips that have scratch-pad memory (SPM) and DRAM but no data cache; examples include low-end CPUs [43], [2], [46], mid-grade CPUs [3], [8], [7], [30], [33] and high-end CPUs [4], [31], [42]. We found at least 80 such embedded processors with SPM and DRAM but no D-cache in our search but have listed only the above eleven for lack of space. Thus our method delivers its full promised benefits for a great variety of chips.

It is nevertheless interesting to see a quantitative comparison of our method for SPM against a cache. Section XI presents such a comparison. It shows that our compile-time method is at least comparable to, or outperforms, a cache of the same area in both run-time and

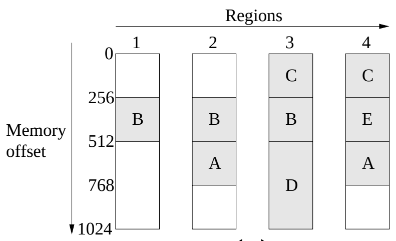
[2]The use of frequency-per-byte itself is not new. It has been used earlier for allocating global and stack variables to SPM [50], [44]. The novelty in this paper is in the solution to the unknown size and invalid pointer problems; this allows heap data to be placed in SPM.

Site	Bin size (bytes)	Regions accessed
A	256	2,4
B	256	1,2,3
C	256	3,4
D	512	3
E	256	4

(a)



(b)



(c)

Fig. 1. Example of heap allocation using our method showing (a) heap Allocation sites; (b) greedy compiler algorithm for layout – D does not fit; and (c) layout after backtracking – D fits.

energy usage.

Below, section II considers recent methods to convert heap data to stack data, to see if they can be used to allocate the heap. Section III overviews related work. Section IV overviews the steps for our method for allocating heap data, along with how it interacts with existing methods for global and stack data. Section V describes how the program is partitioned into regions. Section VI describes how the bin size for each heap variable is computed. Section VII describes layout assignment, which shows how the memory offset for each object in scratch-pad memory is computed. Section VIII how our method can be iteratively applied to find multiple allocation solutions, and choose the one with lowest estimated runtime. Section IX describes two optimizations for our method and code generation. Section X presents results. Section XI presents a comparison versus architectures containing caches. Section XII concludes.

II. Can we use heap to stack conversion to allocate to scratch pad?

At first glance, it seems that recent work on converting heap data to stack data [18], [19], [22] or to stack-like constructs called regions [17], [26] may help in allocating heap data to scratch-pad. Here is some background on these methods. These methods use escape analysis [45], [60] to try to prove that a heap data structure is never accessed outside a certain procedure. If so, the heap variable can be placed on the procedure's stack frame, instead of the heap. The advantage of stack allocation is that the high overhead of heap allocation and de-allocation is avoided. Heap de-allocation is particularly expensive for object-oriented languages since it is done using garbage collection. In contrast, heap data on the stack is de-allocated at low cost when its corresponding procedure exits. One restriction with stack allocation is that it requires fixed-size heap variables, except in some cases when the data is on the frame on the top of the stack. Since this is restrictive, region-based schemes [17], [26] have been proposed for when heap data is of unknown size. Regions, like stack frames, are associated with procedures but are physically allocated on the heap so that they can grow and shrink at runtime. A region is de-allocated when its corresponding procedure exits.

If some heap variables can be allocated to stacks or regions using these methods, then, we ask, can our global/stack method be used to allocate most heap variables to scratch-pad? Unfortunately this approach fails for most heap variables for two reasons. First, heap data structures with compile-time-unknown size cannot be allocated to scratch pad *even if they can be converted to stack or region allocation*. Dynamic data structures such as linked lists, trees and graphs almost always have compile-time-unknown size, and thus cannot be allocated to scratch-pad by stack/region conversion. Second, the fraction of heap data that is of fixed size, and can be converted to stack allocation using escape analysis, is small. Such data can be allocated to scratch pad using our stack/global method, but [22] reports that only 19% of the heap data in their benchmarks could be converted to fixed-size stack data. This low percentage is not surprising – most heap data is in dynamic data structures. Fixed-size heap variables occur mostly in object-oriented languages, where objects are often allocated on the heap so that they can be returned as results from their *method* (object-oriented function) of allocation. For most heap variables, since they are not of fixed size, stack or region allocation does not help in scratch-pad allocation.

In conclusion, allocating heap data to stack or regions reduces allocation and de-allocation overhead in all embedded systems, but cannot be used to allocate most heap data to scratch-pad.

III. Related Work

Static methods to allocate data to SPM include [50], [51], [11], [44], [29], [9], [10]. Static methods are those whose SPM allocation does not change at run-time. Some of these methods [50], [11], [44] are restricted to allocating global variables to SPM; others [51], [29], [9], [10] can allocate both global and stack variables to SPM. Some of these methods use greedy strategies to find a solution; others model the problem as a knapsack problem or an integer-linear programming problem to find a solution.

Some static allocation methods [6], [59] aim to allocate code to SPM rather than data. Other static methods [61], [53] can allocate both code and data to SPM. Their data allocation is restricted to global and stack data. The the

```
Step 1.   Partition program into regions.                                            /* Section V */
Step 2.   Compute initial heap bin sizes.                                            /* Section VI */
Step 3.   Compute consensus heap bin sizes.                                          /* Section VI */
Step 4.   Indirection optimization:                                                  /* Section IX */
            (a) Perform indirection optimization on regions where profitable.
            (b) If (indirection optimization was applied anywhere)
                  Goto Step 2.                  /* Inner iteration */
Step 5.   Do heap layout assignment.                                                 /* Section VII */
Step 6.   Do global and stack placement.                                             /* Section VII */
Step 7.   Do global and stack layout assignment.                                     /* Section VII */
Step 8.   Lazy leave-in optimization.                                                /* Section IX */
Step 9.   Iterative step:                       /* Outer iteration */                /* Section VIII */
            (a) If (estimated runtime of current solution < estimated runtime of best solution so far)
                    Update best solution so far.
            (b) If (estimated runtime of current solution == estimated runtime of solution in last iteration)
                    Goto Step 10.               /* See same solution again ⇒ end search */
            (c) If (number of iterations < THRESHOLD)
                    Goto Step 2.                /* Otherwise end search and proceed to Step 9 */
Step 10.  Do code generation to implement best allocation found.                     /* Section IX */
```

Fig. 2. Algorithm for allocating global, stack and heap data to scratch-pad memory.

goal of the work in [5] is yet another: to map the data in the scratch-pad among its different banks in multi-banked scratch-pads; and then to turn off (or send to a lower energy state) the banks that are not being actively accessed.

Dynamic methods to allocate data to SPM include [37], [56]; these are methods which can change the SPM allocation during run-time. The method in [37] can place global and stack arrays accessed through affine functions of enclosing loop induction variables in SPM. No other variables are placed in SPM; further the optimization for each loop is local in that it does not consider other code in the program. Our earlier method in [56] is a fully general dynamic method that can place all kinds of global and stack variables in SPM. It uses a whole-program analysis that aims to consider the interactions between neighboring code regions to minimize the transfer of data between SPM and DRAM while maximizing the fraction of data found in SPM.

Summarizing the above work, we can see that all of them are restricted to global and stack data. *As far as we know, our method is the first and only method to be able to place a portion of the heap data in SPM under compiler control*.

Runtime methods such as software caching [38], [27] emulate a cache in SRAM using software. The tag, data and valid bits are all managed by compiler-inserted code at each memory access. Software overhead is incurred to manage these fields, though compiler optimizes away the overhead in some cases [38]. Software caching schemes suffer from large overheads from inserted checks before every memory instruction, which are hard to optimize away especially for heap data. Lacking good methods for heap data, the current practice is to place all heap data to DRAM in systems with scratch-pad. Our proposed dynamic method promises to be the first to successfully place heap data in scratch-pad.

Some software caching schemes have been proposed for desktops that use *dynamic compilation* [32] which changes the program at runtime in RAM. Most embedded systems, however, store the program in unchangeable ROM, and dynamic compilation cannot be used. Other software caching schemes have been proposed with different goals and/or non-applicable platforms [58], [15], [21], [47], [16], [34]. For lack of space, we do not discuss these further.

Offline paging [14] derives an optimal page replacement strategy when future page references are known in advance. It cannot be used for our purposes since it makes its page transfer decisions at runtime (address translation done by virtual memory), while any compile-time dynamic method for heap data needs to associate memory transfers with static program points.

IV. Method overview

Figure 2 shows the steps that our method takes to allocate all types of data – global, stack and heap – to scratch-pad memory. Although the contribution of this paper is the method for heaps, the figure shows how the heap method interacts with an existing allocator for global and stack variables, such as [56], to give a complete allocation method. The memory allocation algorithm is run in the compiler just after parsing and initial optimizations but before register allocation and code generation. This allows the compiler to integrate the transfer code insertions with the rest of the program before later compiler optimizations and final register allocation and code generation.

Here we overview the steps shown in figure 2. *Details are found later in the sections listed in the right margin for each step in the figure.* **Step 1** partitions the program into a series of regions. The region boundaries are the only program points where the allocation can change though compiler-inserted copying code. Step 1 also places timestamps on each region at compile-time showing the expected order that the regions are visited at runtime. **Step 2** computes an initial desired bin size for each heap variable for each region based upon the relative frequency-per-byte of *all* variables accessed in that region. Variables of any kind with a higher frequency-per-byte of access are thus preferred. **Step 3** computes a single bin size for each heap ”variable” (allocation site) equal to the weighted average of the initial bin sizes for that variable across all regions which access that variable; weighted by frequency-per-byte of that variable in each region. **Step 4** performs the indirection optimization which aims to reduce the number of transfers. **Step 5** computes the layout of the heap bins in scratch-pad by computing the exact memory offset for each bin. **Step 6** decides which global and stack variables to keep in scratch-pad in each region, after taking into account the space remaining after heap layout. **Step 7** computes the layout for global and stack variables. **Step 8** performs the lazy leave-in optimization, which leaves bins in scratch-pad in regions where they are not accessed, if the cost of eviction exceeds the gain from that space from other variables. **Step 9** performs an iterative step on the algorithm. Step 9(a) maintains the best solution seen so far. Step 9(b) terminates the algorithm if the iterative search is making no progress. Step 9(c) is the heart of the iteration in that it repeats the entire allocation computation, but this time with feedback from the results of this iteration. After the iterative process has exited, **step 10** generates code to implement the best allocation found among all iterations.

V. Deriving regions and timestamps

Our method defines regions and timestamps in the same way as for our method for global and stack data [56]. The presentation is restated here for completeness. A region is a contiguous portion of code in which the allocation to scratch-pad is fixed. Boundaries of regions are called ‘program points’, and thus regions can be defined by defining a set of program points. Code to transfer data between scratch-pad and DRAM is inserted only at the program points.

Program points and hence regions are found as follows. Promising program points are (i) those after which the program has a significant change in locality behavior, and (ii) those whose dynamic frequency is preferably less than the frequency of its following region, so that the cost of copying into SRAM can be recouped by data re-use from SRAM in the region. For example, sites just before the start of loops are promising program points since they are infrequently executed compared to the insides of loops. Moreover, the loop often re-uses data, justifying the cost of copying into SRAM. With the above two criteria in mind, we define program points as *(i) the start and end of each procedure; and (ii) just before and just after each loop* (even inner loops of nested loops).

Figure 3 shows an example illustrating how a program is marked with timestamps at each program point. Figure 3(a) shows the program outline. It consists of four procedures, namely *main()*, *proc-A()*,*proc-B()* and *proc-C()*, two loops that we name *Loop 1* and *Loop2*, and accesses to two variables X and Y. Only loops, procedure declarations and procedure calls are shown – other instructions and constructs are not.

Figure 3(b) shows the *Data-Program Relationship Graph* (DPRG) for the program in figure 3(a). The DPRG is a data structure introduced in [56] that helps in the marking of timestamps and thus the identification of regions. The DPRG is the program’s call graph appended with new nodes for loops and variables. In the DPRG shown in figure 3(b), there are four procedures, two loops and two variables represented by nodes. We see that oval nodes represent procedures, circular nodes represent loops and square nodes represent variables. Edges to procedure nodes represent calls; edges to loop nodes shows that the loop is in its parent; and edges to variable nodes represent memory accesses to that variable from its parent. The DPRG is usually a directed acyclic graph (DAG), except for recursive programs, where cycles occur.

Figure 3(b) also shows the timestamps (1-14) for all program points, namely the beginnings (shown on left of nodes) and ends (shown on right) of every procedure & loop node. The goal is to number timestamps in the order they are encountered during the execution. This numbering is computed at compile-time by the well-known depth-first-search (DFS) graph traversal algorithm. Our DFS marks program points in the order seen with successive timestamps. Our DFS is modified, however, to traverse and timestamp nodes *every time they are seen*, rather than only the first time. This still terminates since the DPRG is a DAG for non-recursive functions. Such repeated traversal results in nodes that have multiple paths to them from *main()* getting multiple timestamps. For example, node *proc-c()* gets timestamps 3 & 7 at its beginning, and 4 & 8 at its end[3].

The handling of conditional paths in programs is detailed in our earlier paper on global and stack variables [56]. That paper’s section 5 lists how conditional paths impact the DPRG, the regions and the timestamps.

Timestamps are useful since they reveal dynamic execution order: the runtime order in which the program points are visited is roughly the order of their timestamps. The only exception is when a loop node has multiple timestamps as descendants. Here the descendants are visited in every iteration, repeating earlier timestamps, thus violating the timestamp order. Even then, we can predict the *common case time order* as the cyclic order, since the end-of-loop backward branch is usually taken. Thus we can use timestamps, at compile-time, to reason about dynamic execution order across the whole program.

[3]This numbering of timestamps is slightly different from that in [56]. That paper used a numbering scheme where the numbering of a node depended on the numbering of its parent, *e.g.,* if the parents number was x then the n^{th} child was x.n; further an ordering between timestamps was defined. The numbering in this paper is exactly functionally equivalent and directly numbers the nodes in the order of their timestamps – this version is simpler to understand.

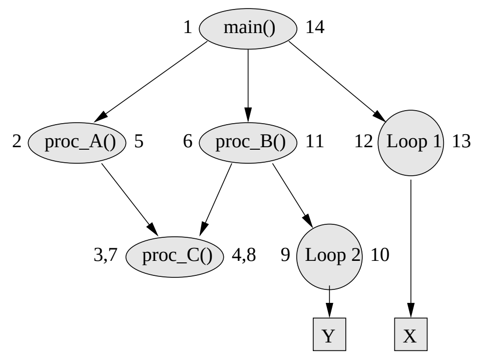
```
main () {
   proc-A()
   proc-B()
   while (...){X =...} /* Loop 1 */
}

proc-A () {
   proc-C()
}

proc-B () {
   proc-C()
   for (...){Y =...}       /* Loop 2 */
}

proc-C () { ...}
```

(a)



(b)

Fig. 3. Example showing (a) a program outline; and (b) is its DPRG showing nodes, edges & timestamps.

Timestamps have their limitations in that they cannot be derived for unstructured control flow or for inside recursive procedures. Fortunately, unstructured control flow is exceedingly rare nowadays in any domain – we only refer to arbitrary **goto** statements here; other constructs such as break and continue statements within loops, and switch statements, are okay for timestamps. Regarding recursive functions, our method is to collapse recursive cycles in the DPRG into single composite nodes, and thereafter place all stack and heap data allocated inside of recursive procedures in DRAM. *After collapsing, data in other (non-recursive) procedures in the DPRG can still be placed in scratch-pad*. Collapsing the cycles into nodes restores the acyclic nature of the DPRG, allowing our region-creation depth-first search to terminate. Recursive programs are rare in embedded programs, so we have not been motivated to investigate a more sophisticated fix. No previous method can place any data in recursive functions in scratch-pad either.

VI. Bin size computation

The bin size assignment heuristic assigns a single bin size for each heap variable in two steps. First, each region requests an initial bin size for each heap variable considering only its own accesses (Step 2 in figure 2). Second, a single consensus bin size is computed for each heap variable (Step 3 in figure 2) as a weighted average of the initial bin sizes of regions accessing that variable. This section discuss these two steps in detail.

Before looking at the algorithm for bin size computation, let us consider two intuitions on which the algorithm is based. The first intuition is that *the bin size assignment heuristic assigns larger bins to sites having greater frequency-per-byte of access*. The reason is that the expected runtime gain from placing a heap bin in scratch-pad instead of DRAM is proportional to the expected number of accesses to the bin. Thus for a fixed amount of scratch-pad space, the gain from that space is proportional to the number of accesses to it, which in turn is proportional to the frequency-per-byte of data in that space. A second intuition is also needed: the constraint of fixed-sized bins implies that *even heap variables of lower frequency-per-byte should get a share of the scratch-pad*. This intuition counter-balances the first intuition. To see why this is needed, consider that according to the first intuition alone, a heap variable with the highest frequency-per-byte in a certain region should be given all the scratch-pad space available. This may not be wise because of the fixed size constraint: doing so would mean a huge bin for the variable that would crowd out all other heap objects in all regions it is accessed, even those with higher frequency-per-byte in other regions. A better overall performance is likely if we 'diversify the risk' by allocating all bins some scratch-pad, even those with lower frequency-per-byte.

The initial bin size computation algorithm is shown in procedure **find_initial_bin_size**() in figure 4. It proceeds as follows. For every region in the program (line 1), all the variables accessed in that region are considered one by one, in decreasing order of their frequency-per-byte of access in that region (line 3). In this way, more frequently accessed variables are preferentially allocated to scratch-pad. For each variable, if it is a global or stack variable, then space is reserved for it (line 5). This reserved space is merely an estimate, however – *the global or stack variable is not actually assigned to SRAM (scratch-pad) yet*; that is done after bin size assignment by Step 6 in figure 2. This design allows for our heap method to be de-coupled from the global-stack method, allowing the use of any global-stack method.

Returning to initial bin size assignment, if the variable v is heap variable (line 6 in figure 4) then an initial bin size is computed for it (lines 7-12). Only when the frequency-per-byte of the site in the region exceeds one (*i.e.,* there is reuse of the site's data in the region), is a non-zero bin size is requested (lines 8-10). Then, the bin size is computed by proportioning the available SRAM in the ratio of frequency-per-byte among all sites accessed by that region (line 8). Only sites that having $freq\_per\_byte(S_i, R) > 1$ are included in the formula's denominator. The bin size is revised to never be larger than the variable's total size in the profile data (line 9): this heuristic prevents small variables from being allocated too-large bins. Finally, the bin size is revised to be a multiple of the heap object size (line 10), to avoid internal fragmentation inside bins.

Finally, a single final bin size is computed as a consensus among the initial bin size assignments above, as shown in procedure **find_consensus_bin_size**() in figure 4. For each heap variable v, the consensus bin size (line 16) is computed as the weighted average of the initial bin size assignments for that site across all regions that access S, weighted by the frequency-per-byte of that variable in that region.

VII. Layout assignment

Given the sizes of the bins computed above, the next step in our method is to compute the layout of all variables in scratch-pad memory (Steps 5-7 in figure 2). The layout refers to the offsets of variables in memory. Computing the layout is done in three steps. First, the layout of heap variables is computed (Step 5). Second, the placement of global and stack variables is computed (*i.e.*, which global and stack variables to keep in scratch-pad is decided) (Step 6). The placement takes into account the amount of space remaining in scratch-pad after heap layout. Third, the layout for global and stack variables is computed by allocating such variables in the spaces remaining in scratch-pad after heap layout. *This section only discusses how the heap layout is computed*. The placement and layout of global and stack variables is independent of this paper, and any dynamic method for global and stack variables can be used, such as [56].

Before we compute the heap layout, it is instructive to consider why heap layout is done before the placement and layout of global and stack variables. The reason the heap layout is done first is that *the layout of heap bins is more constrained than that of global and stack variables*. In particular, heap bins must always be laid out at the same offset in every region they are accessed. Thus, allowing the heap layout to have full access to the whole scratch-pad memory increases the chance of finding a layout with the largest possible number of heap variables in scratch-pad. The less constrained global and stack variables, which typically can be placed in any offset [56], can be placed in whatever spaces that remain after heap layout. *Placing heap data first does **not** mean, however, that heap variables are preferentially allocated in scratch-pad*. Global, stack and heap variables were given an equal chance to fit in scratch pad in the initial bin size computation phase. At that point, we had already reserved space for global and stack variables of high frequency-per-byte.

The layout of heap bins is computed as follows. Finding the layout involves computing the *single fixed offset* for each bin for all regions in which the bin's site is accessed. Further, different bins must not conflict in any region by being allocated to the same memory. We use a greedy heuristic that allocates bins to scratch pad in decreasing order of their overall frequency per byte of access, so the most important bins are given preference. Each bin is placed into the first block in memory that is free for all the regions accessing the bin's site. Figure 1(b) shows the result of the greedy heuristic on sites A-C listed in figure 1(a). This heuristic can, however, fail to find a free block for a bin. Figure 1(b) shows this situation – the bin for D cannot be placed since no contiguous block of size 512 is available in region 3.

To increase the number of bins allocated to scratch-pad, we selectively use a back-tracking heuristic whenever the greedy approach fails to place a bin. Figure 1(b) shows how the greedy heuristic fails to place bin D. Figure 1(c) shows how D can be placed if the offset choices for A and C are revisited and changed as shown. To find the solution in figure 1(c), our back-tracking heuristic *tries to place such a bin by moving a small set of bins placed earlier to different offsets.* This heuristic is written as a recursive algorithm as follows. To try to place a bin that does not fit, it finds the offset in scratch-pad at which the fewest number of other bins, called *conflicting bins*, are assigned. Then it recursively tries to move all the conflicting bins to non-conflicting locations. If successful, the original bin is placed in the space cleared by the moved conflicting bins. The recursive procedure is bounded to three levels to ensure a reasonable compile-time. Four levels increased the compile-time significantly with little additional benefit. An example of this method is when block D cannot be placed in figure 1(b). The offset with the minimum number of conflicts (2) is 512, and the conflicting block set is C. Thus block D is placed at offset 512 by moving C, which in turn recursively moves block A. The conflict-free assignment in figure 1(c) results. Of course, even this recursive search may fail to place a bin – in this case the corresponding heap allocation site places all its data in DRAM.

VIII. Outer iterative step

Although the scratch-pad allocation algorithm is essentially complete after layout assignment, there is an opportunity to do better by iteratively running the entire algorithm again and again, each time taking into account feedback from the previous iteration. This iterative process is depicted in step 9 of figure 2. Step 9(a) maintains the best allocation solution seen amongst all iterations seen so far. Step 9(b) terminates the algorithm if the iterative search is making no progress. Step 9(c) jumps back to step 2 to repeat the entire allocation computation, but this time with feedback from the results of the previous iteration.

The core of the iterative process is understanding what the current iteration can learn from the previous iteration. *The nature of the feedback is what was found to actually*

```
void find_initial_bin_size() {
1.  for (each region R in any order) do
2.      SRAM_remaining = MAX_SRAM_SIZE
3.      for (each variable v of any kind accessed in R sorted in decreasing frequency-per-byte(v,R) order) do
4.          if (v is a global or stack variable)
5.              SRAM_remaining = SRAM_remaining - size(v)
6.          else {                                  /* v is heap variable */
7.              if (freq_per_byte(v,R) > 1)         /* if variable v is reused in R */
8.                  initial_bin_size(v,R) = SRAM_available × $\frac{\text{freq\_per\_byte(v,R)}}{\Sigma_{\text{all accessed variables} u_i \text{ in R}} \text{freq\_per\_byte}(u_i\text{,R}) \text{ which are} > 1}$
9.                  initial_bin_size(v,R) = MIN(initial_bin_size(v,R), size of v in profile data)
10.                 initial_bin_size(v,R) = next-higher-multiple-of-heap-objects-size(initial_bin_size(v,R))
11.             else                                /* no reuse in variable v in R */
12.                 initial_bin_size(v,R) = 0
13.             SRAM_remaining = SRAM_remaining - initial_bin_size(v,R)
14. return


void find_consensus_bin_size() {
15. for (each heap variable v in any order) do
16.     consensus_bin_size(v) = $\frac{\Sigma_{all\ R} \text{ initial\_bin\_size(v,R)} \times \text{freq\_per\_byte(v,R)}}{\Sigma_{all\ R} \text{ freq\_per\_byte(v,R)}}$
17. return
```

Fig. 4. Bin size computation for heap variables.

fit in scratch-pad memory in the previous iteration, given the constraints of layout assignment. To incorporate this feedback in the next iteration, initial bin size computation (step 2) is modified in three ways. First, global and stack variables that did not fit in scratch-pad in the previous iteration do not reserve space for themselves in line 5 of figure 4. Second, heap variables that did not fit in scratch-pad in the previous iteration have their bin size reduced. In particular, the initial bin size is reduced to be the largest size that would have fit in the previous iteration. It is easy to obtain and store this information at the point the heap layout heuristic is unable to place that particular bin. Third, heap variables that fit in scratch-pad have their bin size increased to the largest size that would have fit in the previous iteration. This is determined after layout by looking at the total amount of free space left in the system and increasing the sizes of each heap bin that fit, in order of decreasing frequency-per-byte for the heap sites, until no more free space is left. This is just a heuristic – if no layout can be found with the increased bin size then a subsequent iteration will likely decrease the bin size back down.

The iterative process described above is a heuristic, and is not guaranteed to improve runtime at each iteration. Indeed, the runtime cost may even increase, and after some number of iterations it often does, so for this reason, rather than use the results of the last iteration as the final allocation, the solution with the best estimated runtime among all iterations is maintained. It is fairly straightforward to estimate the runtime cost of a solution at compile-time by counting how many references are converted from accessing DRAM to scratch-pad by a given allocation for the profile data.

A desirable feature of our iterative approach is that it is *tunable* to any given amount of desired compile-time by modifying the threshold for exiting the iterations in step 9(c). In practice, however, we found that we find close to the best solution in a small number of iterations; usually less than three to five iterations. Thereafter it may get worse, but that is no problem as the best solution seen so far is stored, and worse solutions are discarded. After exiting the iterative process (step 10 of figure 2), the best solution seen so far is implemented.

IX. Optimizations and code generation

This section discusses three optimizations to our baseline method for heap data, and code generation.

Reducing runtime and code size of data transfer code Our method copies heap bins back and forth between SRAM and DRAM. This overhead is not unique to our approach – hardware caches also need to move data between SRAM and DRAM. The simplest way to copy is a **for** loop for each bin which copies a single word per iteration. We speed up this transfer in the following three ways. First, we generate assembly-level routines that implement optimized transfers suited to the block size being copied. Copying blocks of sizes larger than a few words are optimized by unrolling the **for** loop by a small constant. Second, code size increase from the larger generated copying code are almost eliminated by placing the code in a memory block copy procedure that is called for each block transfer. Third, faster copying is possible in processors with the low-cost hardware mechanisms of Direct Memory Access (DMA) (*e.g.,* [20], [25]) or pseudo-DMA(*e.g.,* [40]). DMA accelerates data transfers between memories and/or I/O devices. Pseudo-DMA accelerates transfers from memory to CPU registers, and thus can be used to speed memory-to-memory copies via registers.

Indirection optimization A opportunity for improving our algorithm arises because of the following undesirable situation that can arise. In our strategy it is possible that in a certain region, the cost of copying a bin into scratch-pad before that region can exceed the gain in latency of accesses in the region. Leaving the bin in DRAM in such a case would improve runtime, but unfortunately *this violates correctness because of the fixed-offset requirement of our method*. To see why correctness is violated consider that a heap bin must be in the same memory bank in *all* regions that access it, as otherwise, if the bin remained in a different memory bank in a some of those regions, pointers into objects in the bin might be incorrect. Thus the optimization of leaving the bin in DRAM cannot be applied.

Fortunately, there is a way to make this optimization legal. It is legal to leave the bin in DRAM when it is not profitable to copy it into scratch-pad at the start of a region, provided, in addition *all accesses to the bin in the region are translated at runtime to convert their addresses from scratch-pad to DRAM addresses*. Doing so will ensure that all pointers to the bin – which are really invalid since they point incorrectly to scratch-pad – will become valid after translation. Scratch-pad addresses can be translated at runtime to DRAM addresses by inserting code before every memory reference that adds to each address the difference between the starting memory offset of the bin in DRAM and SRAM. In this way, a level of indirection is introduced in addressing, and hence the name of this optimization.

One may wonder why the indirection optimization is used as an optimization, and not the default scheme. Recall than the default scheme ensures that a bin is allocated at the same offset in SRAM whenever it is accessed, and uses indirection only as an exception. The reason that indirection is not used as the default is that indirection has a cost – extra code must be inserted before every access to check if its address is in SRAM and if so, to add a constant to translate it to a DRAM address. This extra code consumes run-time and energy. It is therefore profitable to apply indirection only if the cost of the transfer exceeds the overhead of indirection. For regions where a bin is frequently used, the opposite is true – the overhead, which increases with frequency of use, will increase and often far exceed the cost of transfer; so indirection is not profitable. Since regions where bins are accessed frequently make up most of the run-time the default behavior should match their requirements; indirection is used sparingly for other regions where its overhead is justified.

The indirection optimization (step 4 in figure 2) is applied as follows. For every heap variable v in the program in any order, the compiler looks at all groups of contiguous regions in which v is accessed. For each such group of regions, it estimates whether the cost of copying the bin into scratch-pad at the start of the group (a known function of the size of the block to be copied) is justified by the profile-estimated gain in access latency of accesses to v in the group. If the estimated cost exceeds the estimated gain, then the transfer of the bin to scratch-pad is deleted, and instead all references to v in the group are address-translated as described in the previous paragraph. The address translations themselves add some cost, which is included as an increase in the estimated cost above.

A consequence of the indirection optimization is that scratch-pad space for a some bins is freed in address-translated regions. To profitably use this freed space, we iteratively return to the start of our allocation to re-compute the bin sizes. This iteration is shown in step 4(b) of figure 2. The iterative process exits to step 5 only when the indirection optimization cannot be applied to any additional groups of regions in an iteration.

Lazy leave-in optimization The lazy leave-in optimization tackles the opposite problem to the indirection optimization. Our default behavior is to copy a bin out to memory in regions where it is not accessed. Instead in some cases, it may be profitable to leave a bin in scratch-pad even in regions where it is not accessed, if the cost of copying the bin out to DRAM exceeds the benefit of using that scratch-pad space for other variables. Unlike the indirection optimization there are no legality concerns since there is no correctness constraint for regions which do *not* access a variable.

One way to do this optimization here as a post-pass to layout assignment. This is shown in step 8 of figure 2. After layout assignment we know what other variables were assigned to the space evicted by a bin. If the profile-estimated net gain in latency from these other variables is less than the estimated cost of the transfer, then the compiler lazily leaves the bin in scratch-pad and does not bring the other variables in. The implementation of this optimization is similar to that for the indirection optimization in that an estimated gain vs. estimated cost comparison is applied to contiguous groups of regions, but this time the groups are of regions which do *not* access a bin. A special iterative step for this optimization is not needed since the entire algorithm iterates in the next step any way (step 9 of figure 2).

Code generation Once the iterative process is complete, the code generation for the method happens in step 10 of figure 2. The code generation is straight-forward and has three aspects. First, the memory transfers are inserted where-ever the method has decided. Second, addressing of heap variables does not need modification since they are addressed (usually through pointers) in the same way. Third, calls to malloc() are replaced by calls to a new wrapper function around malloc(). This wrapper first searches for space in fast memory *for that bin* using a new free-list for fast memory. An argument is passed to the wrapper malloc specifying which site is calling it. In this way the wrapper becomes aware of which site is calling it, so that it can look in the bin free list for that site. If space cannot be found in that site's bin free list, then malloc is called on the original unified free-list in slow memory. The code for malloc() is the same for fast and slow memory, but works on different free lists. Similar

Application	Source	Description	Lines of code	Data size (bytes)	Run-time (Mcycles)	% of accesses to heap	% of data that is heap
Huff	Public Domain	Adaptive Huffman encoding	1012	17844	56.1	27.83	55.41
Dhrystone1.1	Dhrystone	Performance benchmark application	455	10912	14.63	14.9	0.01
Susan	MIBench	Image smoothing edge/corner detection	2125	378110	113.9	13.14	38.45
GSM	EU GSM consortium	RPE/LTP speech compression	6828	16466	506.5	8.18	3.88
KS	PtrDist	Minimum Spanning Tree for graphs	686	29316	2.6	25.38	18.12

Fig. 5. Application programs.

modifications are made for other heap memory allocation routines such as calloc() and realloc(). Free() functions are also modified to release the block to the either the site's bin free list or the unified non-bin free list, depending on the address of the object to be freed.

X. Preliminary results

This section presents preliminary results by comparing our proposed method for heap data against the usual practice of placing heap data in DRAM, for a variety of compiler and architecture configurations. We have implemented our algorithm in a GCC cross-compiler targeting the Motorola M-Core [40] embedded processor. The dynamic method for global and stack variables in [56] is also implemented in the same GCC M-Core compiler. The results presented in this work do not include two optimizations discussed earlier, those being (1) indirection optimization and (2) recursive back-tracking. The indirection optimization is optional and is not applied. In the layout phase, instead of recursive back-tracking a simplified two-level search and swap heuristic is used. We note that adding in these two optimizations to our overall method can only improve its performance. A cycle-accurate M-Core simulator is used to collect results.

The memory characteristics and applications are as follows. An external DRAM with 20-cycle latency and an internal SRAM (scratch-pad) with 1-cycle latency is simulated in the default configuration. The default configuration has an SRAM size which is 5% of the total data size in the program. In addition, the effect of varying DRAM latency and SRAM size are measured in the experiments. The DRAM size, of course, is assumed to be large enough to hold all program data. The applications evaluated are shown in figure 5. The applications selected all have at least some heap data.

Pseudo-DMA and DMA are simulated by counting the estimated costs of those mechanisms in the simulator. A *pseudo-DMA* function is an software implementation of a direct memory access control function that can be used with many low-end embedded processors that lack hardware DMA controllers for external memory traffic. It relies on hardware support to load multiple registers at once from memory. We apply this to transfer blocks of 24 bytes or larger by issuing a pair of instructions that load a range of registers at once starting from a base memory location in a dram memory bank. This allows pipelined access of these contiguous memory words to be loaded much faster than truly random accesses to a dram bank issued with individual load and store instructions. An N-word transfer involves N*DRAM_LATENCY/2 for the memory cycle cost, and a 10 instruction cycle base overhead plus an additional cycle for every 4 words transferred. *Real DMA* accelerates memory transfers from external memory banks without involving the cpu directly and are present in most modern mid to high-level embedded systems containing scratchpad, cache or both as their first-level memory hierarchy and external memory controllers. We model this as a cycle cost of N*DRAM_LATENCY/4 cycles per N words plus a 10 cycle overhead for the DMA transfer function for each variable transferred.

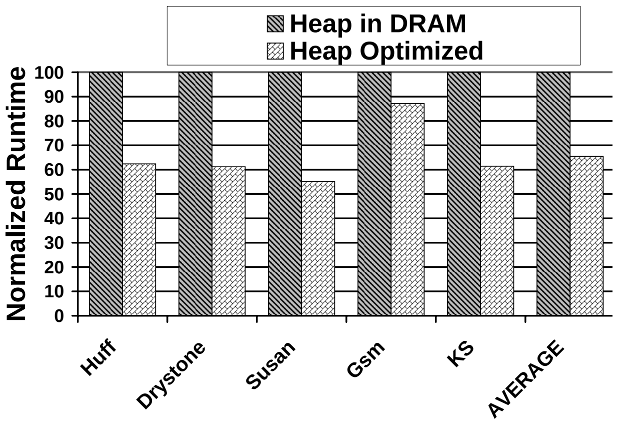


Fig. 6. Runtime gain from using our method vs. allocating heap data in DRAM.

Runtime gain Figure 6 compares the normalized runtime from our method versus from the existing practice of placing all heap data in DRAM. For each benchmark the SRAM size is the same in both configurations – 5% of the combined global, stack and heap data size in that program. Without our method, this SRAM is used only by global and stack data; with our method the SRAM is shared by global, stack and heap data. In both cases, global and stack data is allocated by the best existing method for global and stack data, which is the one in [56]. *The figure shows that the average runtime* *reduces by 34.6% by using our method for the exact same architecture*. The large improvements show the potential of our method to reduce runtime of the application beyond the state-of-the-art today. The results in figure 6 and the rest of the paper use DMA for the memory transfers in our method. However we have measured that the gain from our method only reduces slightly to 32.7% with all-software transfers (not shown); this shows that our method is profitable even without DMA. The reason the gain reduces only slightly without DMA is that both our method and the baseline for global and stack [56] rely on memory transfers; a change in the transfer mechanism affects both and changes their ratio only a little. The cost of the transfers is quantified later.

Why do we do well? Before looking at additional experiments, it is insightful to look at why an improvement of 34.6% can be obtained with an SRAM of only 5% of the total data size of the program. We identify three reasons. **First**, it is well-known that a small fraction of frequently used data usually accounts for a large fraction of the accesses in the program. This is often referred to informally as the *ninety-ten rule* [28]: on average 10% of the data accounts for 90% of the accesses. Consequently, we find that our method is able to place the most frequently used heap variables, either fully or in large part, in SRAM – this is verified later in figure 9. Without our method, they all go to DRAM, which is much slower. A **second** reason for the significant improvement is that our earlier global and stack method, and our current heap method, are both *dynamic*. Thus even though the SRAM is 5% of the total data size, by sharing this space across different frequently used data variables in different regions, *it is possible to place more than 5% of the frequently used data in SRAM*. Note that a scratch-pad improves performance for the same reasons as a cache. Just as even a small cache can significantly improve run-time [28], it is not surprising that a small scratch-pad can too. **Third**, the benchmarks selected have a significant fraction of their accesses going to heap, and thus our method to optimize for heaps does well. For other benchmarks (not selected) where the fraction of heap accesses is zero or small, heap allocation is, of course, not a problem, and hence, it does not need SRAM placement.

Here we look at two examples from our benchmarks which illustrate why some heap data is accessed frequently. First, *Susan* is a typical image-processing application which stores the image in a large stack array. It performs iterative smoothing on small chunks of the image at a time; the chunks and associated look-up tables needed for smoothing are stored on the stack and on the heap. Because smoothing accesses each pixel many times, the most-frequently used chunk data is allocated to SRAM, but the infrequently used large image array and less-frequently used chunk data are placed in DRAM. Second, *Huffman* performs Huffman encoding, meant for data compression. Here most of the program data is on the heap and there is little other data. There are four heap variables of total sizes 60 bytes, 260 bytes, 14Kb, and 4Kb. The first is used to store the encoder structure; the second holds the coded bits of the character currently being encoded; the third stores the alphabet used; and the fourth holds the current block array used in encoding. The last two are somewhat frequently used with frequency-per-byte of about 150 but only a small portion of them fit in SRAM. The first two, however, are very highly used with frequency-per-bytes of about 40000 and 1000, respectively, and our method is able to place them in SRAM.

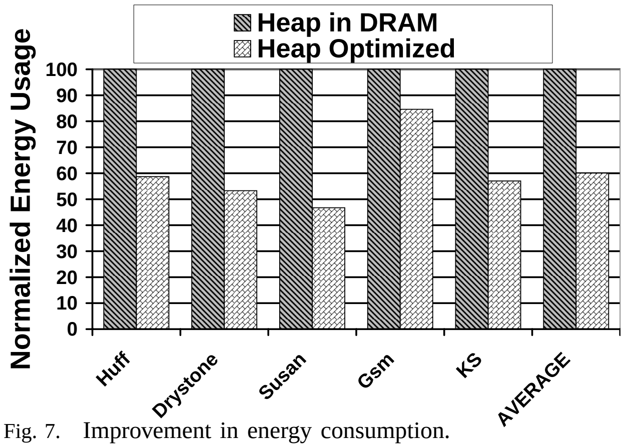


Fig. 7. Improvement in energy consumption.

Energy gain Figure 7 compares the energy consumption of application programs with our method for heap data versus placing heap data in DRAM. An M-core power simulator [13], [12], kindly donated by that group, is used to obtain energy estimates for instructions and SRAM. This is an instruction-level power simulator similar to [49], [55]; its instruction power numbers were measured using an ammeter connected to an M-core hardware board. DRAM power is estimated by a detailed DRAM power simulator we built into the M-core simulator. It uses the DRAM power model described in [35], [36] for the MICRON external DDR Sychronous DRAM chip. The DRAM chip size is set equal to the data size in the energy model. Both the CPU and DRAM use aggressive energy saving technologies. The figure shows that *we measure an average reduction of 39.9% in energy consumption* for our applications by using our method vs. placing heap data in DRAM. This result demonstrates that our approach has the potential to not only significantly improve runtime, but also energy consumption.

The rest of section presents additional numbers which provide more insight into the method, or show the effect of varying certain architectural parameters on our method.

Additional experiments Figure 8 lists some method statistics for each application. Columns two and three list the number of regions and the number of heap allocation sites, respectively, for each benchmark. Column four shows the average code size increase from the

Appli--cation	# of regions	# of heap sites	Code growth (%)	Transfer run-time (%)	Transfer energy (%)
Huff	78	4	1.41	9.1	8.1
Dhry	61	2	4.40	12.5	11.3
Susan	81	4	0.17	0.0	0.0
GSM	294	3	0.54	7.7	9.7
KS	116	5	2.85	0.4	0.4
Average			1.87	5.9	5.9

Fig. 8. Method statistics.

inserted memory transfer code; it averages a modest increase of 1.87% of the original code-size without our method. Columns five and six list the run-time and energy consumption of our inserted transfer code, respectively, as a percentage of the run-time and energy consumption of the original code without our method. The overheads of the transfers may seem high at 5.9% each for both run-time and energy, but these overhead numbers have already been accounted for in all other results in this paper. For example, the run-time improvement from our method averages 34.6% *net* with these overheads; without transfers the run-time improvement would have been 34.6 + 5.9 = 40.5%. In conclusion, the gain from memory latency far outweighs the cost of the transfers.

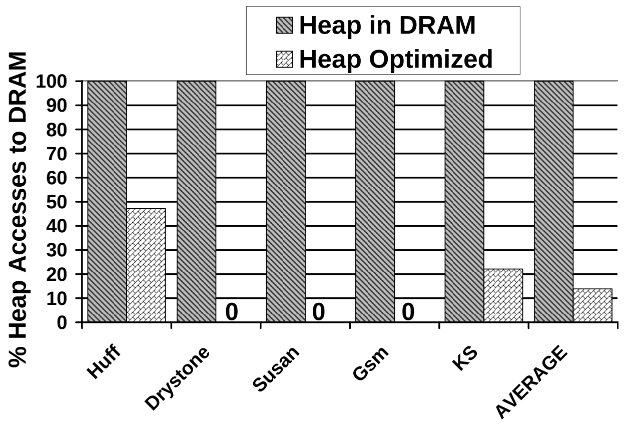


Fig. 9. Percentage of heap memory accesses going to DRAM for each benchmark.

Figure 9 shows the net reduction in percentage of memory accesses to heap data going to DRAM because of the improved locality to SRAM afforded by our method. The number of DRAM accesses is increased by the transfer code but is reduced much more by the increased locality afforded by the SRAM bins. Considering both effects, the average net reduction across benchmarks is a very significant 86.2% reduction in heap DRAM accesses. Analyzing the results shows that our method was able to place many important heap variables into SRAM without involving transfers, explaining the high reduction in DRAM accesses for heap data and showing the benefit of the lazy leave-in optimization. This was correlated with a small increase in transfers for less important stack and global variables, which were evicted to make room for the more frequently accessed heap variables allocated.

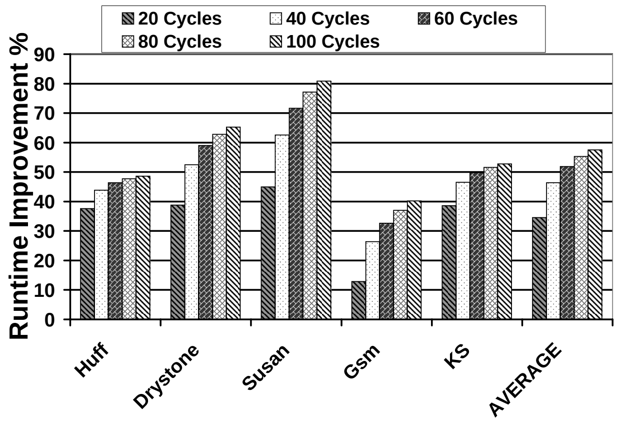


Fig. 10. Effect of varying DRAM latency on runtime gain from our method.

Figure 10 shows the effect of increasing DRAM latency on the runtime gain from our method. Since our method reduces the number of DRAM accesses, the gain from our method is greater with higher DRAM latencies. The figure shows that the runtime gain from our method vs heap allocation in DRAM increases from 34.6% with a 20-cycle DRAM latency to 55.3% with a 100-cycle DRAM latency.

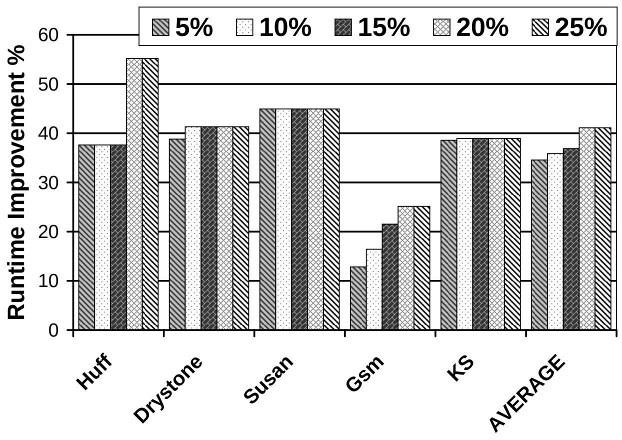


Fig. 11. Effect of varying SRAM size on runtime gain from our method, where SRAM size is expressed as percentage of total data size for application.

Figure 11 shows the effect of increasing SRAM size on the percentage gain in runtime from our method. The SRAM size is expressed as the percentage of the total data size for the application. The runtime gain from our method varies from 34.6% to 41.1%, when the scratch-pad size percentage is varied from 5% to 25%. From this we see that increasing the SRAM space beyond 5% gives only a small additional benefit. This is because of only a small fraction of the program data is frequently used. A similar effect is seen for caches: a very large cache does not yield much better performance than a moderately sized cache [28].

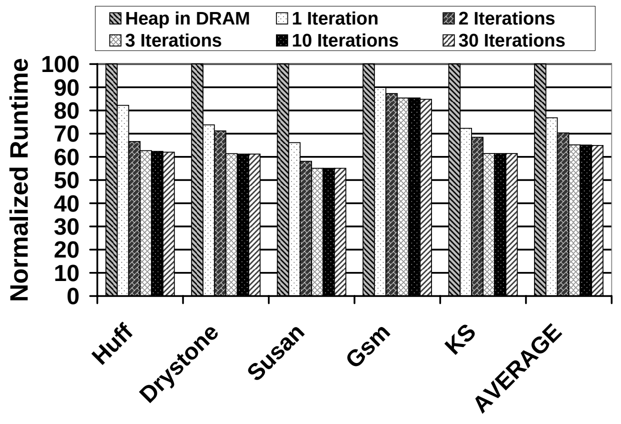


Fig. 12. Effect of varying number of iterations of our search process on runtime.

Figure 12 shows the effect of varying the number of iterations of our iterative search on the runtime of the application. The iterative step referred to is shown in step 9 of figure 2. From figure 12 we see that most of the gain (34.6%) is obtained from the first 3 iterations. Only an additional 0.3% gain in runtime is seen from using thirty iterations. Considering the small gain from many iterations, to limit compile-time we have limited our search process to 3 iterations for the rest of the paper.

XI. Comparison with caches

This section compares the performance of our method for scratch-pad memories (SPM) versus alternative architectures using either caches alone; or cache and SPM together.

It is important to note that our method is useful regardless of the results of a comparison with caches because there are a great number of embedded architectures which have a SPM and DRAM directly accessed by the CPU, but have no data cache. Examples of such architectures include low-end chips such as the Motorola MPC500 [43], Analog Devices ADSP-21XX [2], Motorola Coldfire 5206E [41]; mid-grade chips such as the Analog Devices ADSP-21160m [3], Atmel AT91-C140 [8], ARM 968E-S [7], Hitachi M32R-32192 [30], Infineon XC166 [33] and high-end chips such as Analog Devices ADSP-TS201S [4], Hitachi SuperH-SH7050 [31], and Motorola Dragonball [42]. We found at least 80 such embedded processors with no D-cache but with SRAM and external memory (usually DRAM) in our search but have listed only the above eleven for lack of space. These architectures are popular because SPMs are simple to design and verify, and provide better real-time guarantees for global and stack data [62], power consumption, and cost [6], [53], [59], [11] compared to caches. For these architectures our method delivers run-time and energy reductions averaging 34.6% and 39.9%, respectively, compared to the best previous method.

Nevertheless, it is interesting to see how our method compares against processors containing caches. We compare three architectures (i) an SPM-only architecture; (ii) a cache-only architecture; and (iii) an architecture with both SPM and cache of equal area. To ensure a fair comparison the total silicon area of fast memory (SPM or cache) is equal in all three architectures and roughly equal to the silicon area of the SPM in section X (which holds 5% of the data for each benchmark)[4]. For an SPM and cache of equal area the cache has lower data capacity because of the area overhead of tags and other control circuitry. Area and energy estimates for cache and SPM are obtained from Cacti [23], [63]. The cache simulated is direct-mapped (this is varied later), has a line size of 8 bytes, and is in 0.5 micron technology. The SPM is of the same technology but we remove the decoder, tag memory array, tag column multiplexers, tag sense amplifiers and tag output drivers in Cacti that are not needed for SPM. The Dinero cache simulator [57] is used to obtain run-time results; it is combined with Cacti's energy estimates per access to yield the energy results.

Figure 13 shows the normalized run-time for different architecture/compiler pairs. The first and second bars for each benchmark are with and without our heap allocation method for the SPM-only design. Global and stack variables are mapped to SPM in either case. The third bar is for the cache only architecture. The fourth and fifth bars are with and without our heap allocation method for the SPM and cache design. In the fourth bar when our method is used with a cache, then all the less-frequently used data that our method presumes is in DRAM is placed in cached DRAM address space instead; thus the slow memory transfers are accelerated. By comparing the second and third bars we see that our method on an SPM virtually equals the run-time of a cache-only architecture; both provide significant acceleration over the heap in DRAM case(first bar). By comparing the fourth and fifth bar we see that our method gives an average run-time which is 10.6% faster for the SPM + cache case.

Figure 14 shows the normalized energy consumption for the same configurations as in figure 13. By comparing the second and third bars we see that our method on an SPM has 8.1% lower energy consumption than a cache-only architecture. By comparing the fourth and fifth bar we see that our method gives an average energy usage which is 10.6% lower for the SPM + cache case.

In conclusion the results in figures 13 and 14 show that our method equals or slightly outperforms a cache-only architecture; and provides slightly better run-time and energy in an SPM + cache architecture. The differences are not great though; so we can only say that the

[4]Actually since cache must be a power of two in size and Cacti has a minimum line size of 8 bytes, the sizes of caches are not infinitely adjustible. To overcome this difficulty we first fix the size of cache whose SPM-equivalent in area holds the nearest to 5% of the data size. Then an SPM of the same area is chosen; this is easier since SPM sizes are less constrained.

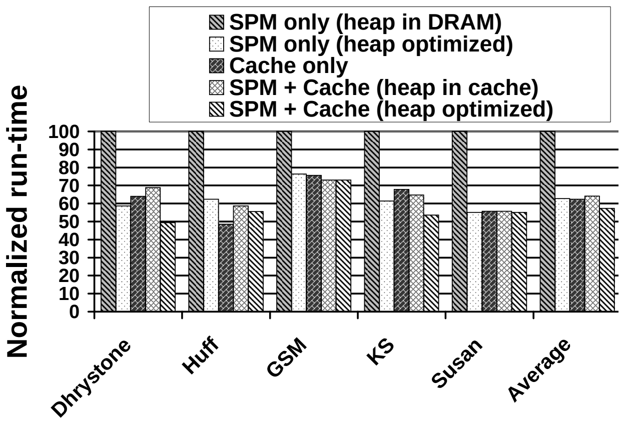


Fig. 13. Normalized run-time for different architecture/compiler pairs for architectures containing different combinations of SPM and cache.

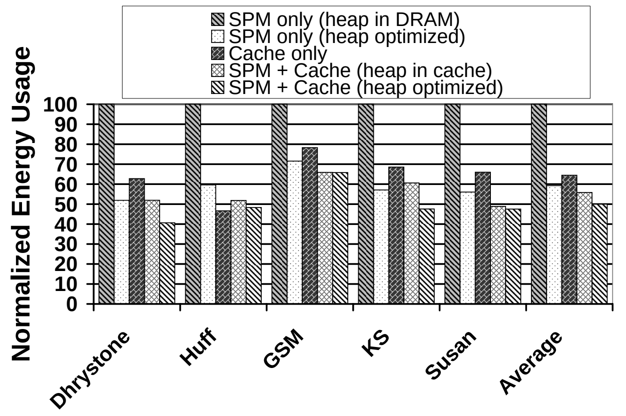


Fig. 14. Normalized energy usage for different architecture/compiler pairs for architectures containing different combinations of SPM and cache.

numbers are comparable or slightly better. We believe it is remarkable for a compile-time method for heap data to equal or out-perform a cache – something many thought was not possible.

It is interesting to speculate on strengths and weaknesses of our method vs. caches. First, like caches our method gives preference to more frequently accessed sites by allocating them larger bins in SPM. Second, one downside of our method is that a cache retains the used subset of a heap variable in SRAM, while our method retains a fixed subset. Third, an advantage of our method is that it avoids copying infrequently used data to fast memory; a cache copies in infrequent data when accessed, possibly evicting frequent data. On the whole, the results indicate that we come out slightly ahead vs. caches.

Despite the comparable performance vs. caches our method still has merit because of two other advantages of SPMs over caches not apparent from the results above. First, it is widely known that for global and stack data, SPMs have significantly better real-time guarantees than caches [62], [53], [10]. Second, other researchers have repeatedly demonstrated a significant energy and run-time savings for benchmarks containing only global and stack data for an SPM vs. a cache of the same area [6], [53], [59], [11]. For these reasons, if an embedded task set contains some tasks of only global and stack data and other tasks having heap data as well, our method will enable the designer to use a SPM alone and avail of the advantages above to the fullest extent; instead of using an SPM + cache architecture necessary without our method for the heap.

Figures 15 and 16 measure the impact of varying cache associativity on the run-time and energy usage, respectively, on our cache-only architecture. The figures show that the run-time is comparable with increasing associativity and the energy gets worse; for this reason a direct-mapped cache is used in the earlier experiments in this section. The numbers for 4-way associative for two benchmarks are unavailable because for those benchmarks, the size of the 5% data size cache becomes smaller than for a cache with a single cache line (8 bytes) because of the significant area overhead of the parallel comparators and other control logic.

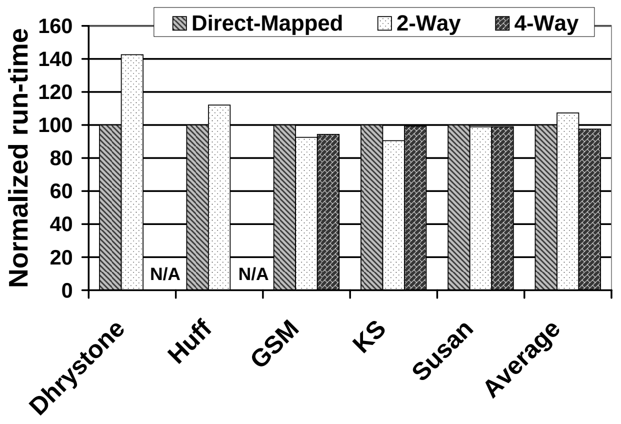


Fig. 15. Normalized run-time for different set associativities for a cache-only configuration.

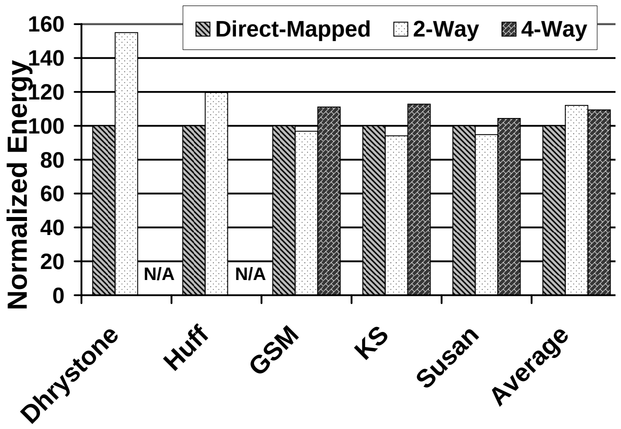


Fig. 16. Normalized energy usage for different set associativities for a cache-only configuration.

XII. Conclusion

This paper presents the first compile-time method for allocating a portion of the heap data in scratch-pad memory. Compile-time placement of heap data in scratch-pad is complicated by two factors. First, the size of heap structures is usually data-dependent and thus is not knowable at compile-time. Consequently it is difficult guarantee at compile-time that a given heap structure will fit in scratch-pad. Second, moving heap data between scratch-pad and DRAM, required by all dynamic allocation methods, results in pointers pointing into a moved block to become invalid after movement, violating correctness.

The presented method for allocating heap data to scratch-pad solves the above problems as follows. First, the problem of unknown size heap structures is solved by placing only a fixed-size portion of the heap structure, called a bin, in scratch-pad memory. Second, the problem of invalid pointers upon movement of bins is solved by ensuring that the bin location is the same at all program points where the heap structure is accessed. However, for better scratch-pad utilization, bins can be moved to other locations at program points where the heap structure is not accessed. More frequently accessed heap structures are allocated larger bins in scratch-pad to improve runtime. With our method, global, stack and heap variables can share the same scratch-pad. When compared to placing all heap variables in DRAM and only global and stack data in scratch-pad, our results show that our method reduces the average runtime of our benchmarks by 34.6%, and the average power consumption by 39.9%, for the same size of scratch-pad fixed at 5% of total data size.

References

[1] M. Adiletta, M. Rosenbluth, D. Bernstein, G. Wolrich, and H. Wilkinson. The Next Generation of Intel IXP Network Processors. *Intel Technology Journal*, 6(3), August 2002. http://developer.intel.com/technology/itj/2002/volume06issue03/.

[2] *ADSP-21xx 16-bit DSP Family*. Analog Devices, 1996. http://-www.analog.com/processors/processors/ADSP/index.html.

[3] *SHARC ADSP-21160M 32-bit Embedded CPU*. Analog Devices, 2001. http://www.analog.com/processors/processors/sharc-/index.html.

[4] *TigerSharc ADSP-TS201S 32-bit DSP*. Analog Devices, Revised Jan. 2004. http://www.analog.com/processors/processors/-tigersharc/index.html.

[5] Federico Angiolini, Luca Benini, and Alberto Caprara. Polynomial-time algorithm for on-chip scratchpad memory partitioning. In *Proceedings of the 2003 international conference on Compilers, architectures and synthesis for embedded systems*, pages 318–326. ACM Press, 2003.

[6] Federico Angiolini, Francesco Menichelli, Alberto Ferrero, Luca Benini, and Mauro Olivieri. A post-compiler approach to scratch-pad mapping of code. In *Proceedings of the 2004 international conference on Compilers, architecture, and synthesis for embedded systems*, pages 259–267. ACM Press, 2004.

[7] *ARM968E-S 32-bit Embedded Core*. Arm, Revised March 2004. http://www.arm.com/products/CPUs/ARM968E-S.html.

[8] *Atmel AT91C140 16/32-bit Embedded CPU*. Atmel, Revised May 2004. http://www.atmel.com/dyn/resources/prod_documents/-doc6069.pdf.

[9] Oren Avissar, Rajeev Barua, and Dave Stewart. Heterogeneous Memory Management for Embedded Systems. In *Proceedings of the ACM 2nd International Conference on Compilers, Architectures, and Synthesis for Embedded Systems (CASES)*, November 2001. Also at http://www.ece.umd.edu/∼barua.

[10] Oren Avissar, Rajeev Barua, and Dave Stewart. An Optimal Memory Allocation Scheme for Scratch-Pad Based Embedded Systems. *ACM Transactions on Embedded Systems (TECS)*, 1(1), September 2002.

[11] R. Banakar, S. Steinke, B-S. Lee, M. Balakrishnan, and P. Marwedel. Scratchpad Memory: A Design Alternative for Cache On-chip memory in Embedded Systems. In *Tenth International Symposium on Hardware/Software Codesign (CODES)*, Estes Park, Colorado, May 6-8 2002. ACM.

[12] Kathleen Baynes, Chris Collins, Eric Fiterman, Brinda Ganesh, Paul Kohout, Christine Smit, Tiebing Zhang, and Bruce Jacob. The performance and energy consumption of three embedded real-time operating systems. In *Proc. Fourth Workshop on Compiler and Architecture Support for Embedded Systems (CASES'01)*, pages 203–210, Atlanta GA, November 2001.

[13] Kathleen Baynes, Chris Collins, Eric Fiterman, Brinda Ganesh, Paul Kohout, Christine Smit, Tiebing Zhang, and Bruce Jacob. The performance and energy consumption of embedded real-time operating systems. *IEEE Transactions on Computers*, 52(11):1454–1469, November 2003.

[14] L.A. Belady. A study of replacement algorithms for virtual storage. In *IBM Systems Journal*, pages 5:78–101, 1966.

[15] J. K. Bennett, J. B. Carter, and W. Zwaenepoel. Adaptive software cache management for distributed shared memory architectures. In *Proc. of the 17th Annual Int'l Symp. on Computer Architecture (ISCA'90)*, pages 125–135, 1990.

[16] Azer Bestavros, Robert L. Carter, Mark E. Crovella, Carlos R. Cunha, Abddsalam Beddaya, and Sulaiman A.Mirdad. Application-level document caching in the internet. In *Proceedings of the Second Intl. Workshop on Services in Distributed and Networked Environments (SDNE)'95*, pages 125–135, 1990.

[17] Lars Birkedal, Mads Tofte, and Magnus Vejlstrup. From region inference to von neumann machines via region representation inference. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 171–183. ACM Press, 1996.

[18] Bruno Blanchet. Escape analysis: correctness proof, implementation and experimental results. In *Proceedings of the 25th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 25–37. ACM Press, 1998.

[19] Bruno Blanchet. Escape analysis for object-oriented languages: application to java. In *Proceedings of the 14th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 20–34. ACM Press, 1999.

[20] David Brash. *The ARM architecture Version 6 (ARMv6)*. ARM Ltd., January 2002. White Paper.

[21] Martin C. Carlisle and Anne Rogers. Software caching and computation migration in Olden. *Journal of Parallel and Distributed Computing*, 38(2):248–255, 1996.

[22] Jong-Deok Choi, Manish Gupta, Mauricio Serrano, Vugranam C. Sreedhar, and Sam Midkiff. Escape analysis for java. In *Proceedings of the 14th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 1–19. ACM Press, 1999.

[23] *Cacti 3.2*. P. Shivaumar and N.P. Jouppi, Revised 2004. http://-research.compaq.com/wrl/people/jouppi/CACTI.html.

[24] Manuvir Das. Unification-based pointer analysis with directional assignments. In *Proceedings of the SIGPLAN '00 Conference on Program Language Design and Implementation*, pages 35–46, Vancouver, BC, June 2000.

[25] Document No. ARM DDI 0084D, ARM Ltd. *ARM7TDMI-S Data sheet*, October 1998.

[26] Niels Hallenberg, Martin Elsman, and Mads Tofte. Combining region inference and garbage collection. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation*, pages 141–152. ACM Press, 2002.

[27] G. Hallnor and S. K. Reinhardt. A fully associative software-managed cache design. In *Proc. of the 27th Int'l Symp. on Com-*

puter Architecture (ISCA), Vancouver, British Columbia, Canada, June 2000.

[28] John Hennessy and David Patterson. *Computer Architecture A Quantitative Approach*. Morgan Kaufmann, Palo Alto, CA, third edition, 2002.

[29] Jason D. Hiser and Jack W. Davidson. Embarc: an efficient memory bank assignment algorithm for retargetable compilers. In *Proceedings of the 2004 ACM SIGPLAN/SIGBED conference on Languages, compilers, and tools for embedded systems*, pages 182–191. ACM Press, 2004.

[30] *M32R-32192 32-bit Embedded CPU*. Hitachi/Renesas, Revised July 2004. http://documentation.renesas.com/eng/products/-mpumcu/rej03b0019_32192ds.pdf.

[31] *SH7050 32-bit CPU*. Hitachi/Renesas, Revised Sep. 1999. http://documentation.renesas.com/eng/products/mpumcu-/e602121_sh7050.pdf.

[32] C. Huneycutt and K. Mackenzie. Software caching using dynamic binary rewriting for embedded devices. In *Proceedings of the International Conference on Parallel Processing*, pages 621–630, 2002.

[33] *XC-166 16-bit Embedded Family*. Infineon, Revised Jan. 2001. http://www.infineon.com/cmc_upload/documents/036/812/-c166sv2um.pdf.

[34] Arun Iyengar. Design and performance of a general-purpose software cache. *Journal of Parallel and Distributed Computing*, 38(2):248–255, 1996.

[35] Jeff Janzen. Calculating Memory System Power for DDR SDRAM. In *DesignLine Journal*, volume 10(2). Micron Technology Inc., 2001. http://www.micron.com/publications/-designline.html.

[36] *128Mb DDR SDRAM data sheet*. (Dual data-rate synchronous DRAM) Micron Technology Inc., 2003. http://www.micron.com/-products/dram/ddrsdram/.

[37] M.Kandemir, J.Ramanujam, M.J.Irwin, N.Vijaykrishnan, I.Kadayif, and A.Parikh. Dynamic Management of Scratch-Pad Memory Space. In *Design Automation Conference*, pages 690–695, 2001.

[38] Csaba Andras Moritz, Matthew Frank, and Saman Amarasinghe. FlexCache: A Framework for Flexible Compiler Generated Data Caching. In *The 2nd Workshop on Intelligent Memory Systems*, Boston, MA, November 12 2000.

[39] *CPU12 Reference Manual*. Motorola Corporation, 2000. (A 16-bit processor). http://e-www.motorola.com/brdata/PDFDB/-MICROCONTROLLERS/16_BIT/68HC12_FAMILY/REF_MAT/-CPU12RM.pdf.

[40] *M-CORE - MMC2001 Reference Manual*. Motorola Corporation, 1998. (A 32-bit processor). http://www.motorola.com/SPS/-MCORE/info_documentation.htm.

[41] *Coldfire MCF5206E 32-bit CPU*. Motorola/Freescale, Revised 2002. http://www.freescale.com/files/dsp/doc/fact_sheet/-CFPRODFACT.pdf.

[42] *Dragonball MC68SZ328 32-bit Embedded CPU*. Motorola/Freescale, Revised April 2003. http://www.freescale.com/-files/32bit/doc/fact_sheet/MC68SZ328FS.pdf.

[43] *MPC500 32-bit MCU Family*. Motorola/Freescale, Revised July 2002. http://www.freescale.com/files/microcontrollers/doc/-fact_sheet/MPC500FACT.pdf.

[44] P. R. Panda, N. D. Dutt, and A. Nicolau. On-Chip vs. Off-Chip Memory: The Data Partitioning Problem in Embedded Processor-Based Systems. *ACM Transactions on Design Automation of Electronic Systems*, 5(3), July 2000.

[45] Young Gil Park and Benjamin Goldberg. Escape analysis on lists. In *Proceedings of the ACM SIGPLAN 1992 conference on Programming language design and implementation*, pages 116–127. ACM Press, 1992.

[46] *LPC2290 16/32-bit Embedded CPU*. Philips, Revised Feb. 2004. http://www.semiconductors.philips.com/acrobat_download/-datasheets/LPC2290-01.pdf.

[47] Ioannis Schoinas, Babak Falsafi, Alvin R. Lebeck, Steven K. Reinhardt, James R.Larus, and David A. Wood. Fine-grain Access Control for Distributed Shared Memory. In *Proceedings of the Sixth International Conference on Architecture Support for Programming Languages and Operating Systems*, pages 297–306, 1994.

[48] Compilation Challenges for Network Processors. *Industrial Panel, ACM Conference on Languages, Compilers and Tools for Embedded Systems (LCTES)*, June 2003. Slides at http://www.cs.purdue.edu/s3/LCTES03/.

[49] Amit Sinha and Anantha Chandrakasan. JouleTrack - A Web Based Tool for Software Energy Profiling. In *Design Automation Conference*, pages 220–225, 2001.

[50] Jan Sjodin, Bo Froderberg, and Thomas Lindgren. Allocation of Global Data Objects in On-Chip RAM. *Compiler and Architecture Support for Embedded Computing Systems*, December 1998.

[51] Jan Sjodin and Carl Von Platen. Storage Allocation for Embedded Processors. *Compiler and Architecture Support for Embedded Computing Systems*, November 2001.

[52] Bjarne Steensgaard. Points-to analysis in almost linear time. In *Symposium on Principles of Programming Languages (POPL)*, St. Petersburg Beach, FL, January 1996.

[53] S. Steinke, L. Wehmeyer, B. Lee, and P. Marwedel. Assigning program and data objects to scratchpad for energy reduction. In *Proceedings of the conference on Design, automation and test in Europe*, page 409. IEEE Computer Society, 2002.

[54] *TMS370Cx7x 8-bit microcontroller*. Texas Instruments, Revised Feb. 1997. http://www-s.ti.com/sc/psheets/spns034c/spns034c.pdf.

[55] V. Tiwari and M. T.-C. Lee. Power Analysis of a 32-bit embedded microcontroller. *VLSI Design Journal*, 7(3), 1998.

[56] Sumesh Udayakumaran and Rajeev Barua. Compiler-decided dynamic memory allocation for scratch-pad based embedded systems. In *Proceedings of the international conference on Compilers, architectures and synthesis for embedded systems (CASES)*, pages 276–286. ACM Press, 2003.

[57] *DineroIV Cache simulator*. J. Edler and M.D. Hill, Revised 2004. http://www.cs.wisc.edu/ markhill/DineroIV/.

[58] Osman S. Unsal, Rakshit Ashok, Israel Koren, C. Manik Krishna, and Csaba Andras Moritz. Cool-cache for hot multimedia. In *Proceedings of the International Symposium on Microarchitecture*, pages 274–283, 1990.

[59] Manish Verma, Lars Wehmeyer, and Peter Marwedel. Cache-aware scratchpad allocation algorithm. In *Proceedings of the conference on Design, automation and test in Europe*, page 21264. IEEE Computer Society, 2004.

[60] Frdric Vivien and Martin Rinard. Incrementalized pointer and escape analysis. In *Proceedings of the ACM SIGPLAN 2001 conference on Programming language design and implementation*, pages 35–46. ACM Press, 2001.

[61] Lars Wehmeyer, Urs Helmig, and Peter Marwedel. Compiler-optimized usage of partitioned memories. In *Proceedings of the 3rd Workshop on Memory Performance Issues (WMPI2004)*, 2004.

[62] Lars Wehmeyer and Peter Marwedel. Influence of onchip scratchpad memories on wcet prediction. In *Proceedings of the 4th International Workshop on Worst-Case Execution Time (WCET) Analysis*, 2004.

[63] S.J.E. Wilton and N.P. Jouppi. Cacti: An enhanced cache access and cycle time model. In *IEEE Journal of Solid-State Circuits*, 1996.