Dynamic Allocation for Scratch-Pad Memory using Compile-Time Decisions

Sumesh Udayakumaran and Angel Dominguez and Rajeev Barua
University of Maryland

In this research we propose a highly predictable, low overhead and yet dynamic, memory allocation strategy for embedded systems with scratch-pad memory. A *scratch-pad* is a fast 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 energy consumption, area and overall runtime, even with a simple allocation scheme.

Scratch-pad allocation primarily methods are of two types. First, software-caching schemes emulate the workings of a hardware cache in software. Instructions are inserted before each load/store to check the software-maintained cache tags. Such methods incur large overheads in runtime, code size, energy consumption and SRAM space for tags and deliver poor real-time guarantees just like hardware caches. A second category of algorithms partitions variables at compile-time into the two banks. However, a drawback of such static allocation schemes is that they do not account for dynamic program behavior. It is easy to see why a data allocation that never changes at runtime cannot achieve the full locality benefits of a cache.

We propose a dynamic allocation methodology for global and stack data and program code that, (i) accounts for changing program requirements at runtime (ii) has no software-caching tags (iii) requires no run-time checks (iv) has extremely low overheads, and (v) yields 100% predictable memory access times. In this method data that is about to be accessed frequently is copied into the scratch-pad using compiler-inserted code at fixed and infrequent points in the program. Earlier data is evicted if necessary. When compared to a provably optimal static allocation, results show that our scheme reduces runtime by up to 39.8% and energy by up to 31.3% on average for our benchmarks, depending on the SRAM size used. The actual gain depends on the SRAM size, but our results show that close to the maximum benefit in run-time and energy is achieved for a substantial range of small SRAM sizes commonly found in embedded systems. Our comparison with a direct mapped cache shows that our method performs roughly as well as a cached architecture.

Categories and Subject Descriptors: B.3.2 [**Memory Structures**]: Design Styles—*Primary Memory*; C.3 [**Special-Purpose And Application-Based Systems**]: Real-time and Embedded Systems; D.3.4 [**Programming Languages**]: Processors—*Compilers*

General Terms: Performance
Additional Key Words and Phrases: Memory Allocation, Scratch-Pad, Compiler, Embedded Systems, Software managed cache, Software caching

1. INTRODUCTION

Memory systems generally are organized using a variety of devices which serve different purposes. Devices like SRAM and ROM are fast but expensive. On the other hand devices

Author's address: Dept. of Electrical and Computer Engineering, University of Maryland, College Park, MD, 20742.
Permission to make digital/hard copy of all or part of this material without fee for personal or classroom use provided that the copies are not made or distributed for profit or commercial advantage, the ACM copyright/server notice, the title of the publication, and its date appear, and notice is given that copying is by permission of the ACM, Inc. To copy otherwise, to republish, to post on servers, or to redistribute to lists requires prior specific permission and/or a fee.
© 20YY ACM 0164-0925/20YY/0500-0001 $5.00

like DRAM and tape drives are slower but being cheaper can be used to provide capacity. Designing a memory system therefore involves using small amounts of faster devices like SRAM along with slower devices like DRAM to obtain satisfactory performance while keeping a check on the overall dollar cost.

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 big success for desktops, a trend that is likely to continue in the future. Using non-cached SRAM or scratch-pads is usually not feasible for desktops; one reason is the lack of binary-code portability. Scratch-pad code is not portable across different sizes of scratch-pad because all existing compile-time methods for data[1] allocation to scratch-pad (including the method in this paper) require that the scratch-pad size be known; otherwise they cannot reason about what variables will fit in the scratch-pad. This contrasts with cache allocation which is decided only at run-time; and hence does not require compile-time knowledge of the size of cache. Binary portability is valuable for desktops, where independently distributed binaries must work on any cache size. However, in embedded systems, software is configured along with the hardware in the factory and rarely changes thereafter. So embedded system designers can afford to customize the SRAM to a particular size to reap the additional cost savings from customization.

For embedded systems, the serious overheads of caches are less defensible. An alternative that is instead prevalent is to use compiler managed SRAM or scratch-pad. Studies [Banakar et al. 2002] have shown that scratch-pad's use 34% lesser area, consume 40% lower power 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 [Banakar et al. 2002]. Given the power, cost, performance and real time advantages of scratch-pad, it is not surprising that scratch-pads are the most common form of SRAM in embedded CPUs today. Some examples of processors with scratch-pad memory are intel IXP network processor, ARMv6, IBM 440 and 405, Motorola's MCORE and 6812 and TI TMS-370. Trends in recent embedded designs indicate that the dominance of scratch-pad will likely consolidate further in the future [Banakar et al. 2002] [Lctes Panel 2003].

Although many embedded processors with scratch-pad exist, using the scratch-pad effectively has been a challenge. Central to the effectiveness of caches is their ability to maintain, at each time during program execution, the subset of data that is frequently used *at that time* in the fast memory. The contents of cache constantly change during runtime to reflect the changing working set of data across time. Unfortunately, two of the existing allocation approaches for scratch-pad – program annotations and the recent compiler-driven approaches [Avissar et al. 2001; 2002; Sjodin and Platen 2001] – are static allocators, *i.e.* they do not change the contents of scratch-pad at runtime. This is a serious limitation. For example, consider the following thought experiment. Let a program consist of three successive loops, the first of which makes repeated references to array A; the second to B; and the third to C. If only one of the three arrays can fit within the scratch-pad, then any static allocation suffers DRAM accesses for two out of three loops. In contrast, a dynamic strategy can fit all three arrays in the scratch-pad at different times. Although this example is oversimplified, it intuitively illustrates the benefits of dynamic allocation.

This research presents a new compiler method for allocating three types of program ob-

[1]We use the terms data and program objects to broadly refer to both program code and program data

jects – global variables, stack variables and program code – to scratch-pad that is able to change the allocation at runtime and avoid the overheads of runtime methods. A preliminary version of our method published in [Udayakumaran and Barua 2003] was the first such method to allocate global and stack data using whole program analysis. Our method (i) accounts for changing program requirements at runtime; (ii) has no tags like used by runtime methods; (iii) requires no run-time checks per load/store; (iv) has extremely low overheads; and (v) yields 100% predictable memory access times.

Our method is outlined as follows. The compiler analyzes the program to identify locations we call *program points* where it may be beneficial to insert code to copy a variable from DRAM into the scratch-pad. It is beneficial to copy a variable into scratch-pad if the latency gain from having it in scratch-pad rather than DRAM is greater than the cost of its transfer. A profile-driven cost model estimates these benefits and costs. The compiler ensures that the program data allocated to scratch-pad fits at all times by occasionally evicting existing variables in scratch-pad to make space for incoming variables. In other words, just like in a cache, data is moved back and forth between DRAM and scratch-pad, but under compiler control, and with no additional overhead.

Key components of our method are as follows. (i) To reason about the contents of scratch-pad across time, it helps to attach a concept of time to the above-defined program points. Towards this end, we introduce a new data structure called the *Data-Program Relationship Graph (DPRG)* which associates a *timestamp* with each program point. As far as we know, this is the first time that a static data structure to represent program execution time has been defined. (ii) A detailed cost model is presented to estimate the runtime cost of any proposed data transfer at a program point. (iii) A compile-time heuristic is presented that uses the cost model to decide which transfers minimize the runtime. The well-known dataflow concept of liveness analysis [Appel and Ginsburg 1998] is used to eliminate unnecessary transfers – provably dead variables [2] are not copied back to DRAM; nor are newly alive variables in this region copied in from DRAM to SRAM [3]. In programs, where the final results (only global) need to be left in the memory itself, this optimization can be turned off in which case the benefits would be reduced. [4] This optimizations also needs to be turned off for segments shared between tasks.

We observe three desirable features of our algorithm. (i) No additional transfers beyond those required by a caching strategy are done. (ii) Data that is accessed only once is not brought into the scratch-pad, unlike in caches, where the data is cached and potentially useful data evicted. This is particularly beneficial for streaming multimedia codes where use-once data is common. (iii) Data that the compiler knows to be dead is not written out to DRAM upon eviction, unlike in a cache, where the caching mechanism writes out all evicted data.

---

[2] In compiler terminology a variable is dead at a point in the program if the *value* in it is not used beyond this point, although the space could be. A dead variable becomes live later if it is written to with subsequently used data. As a special case it is worth noting that every un-initialized variable is dead at the beginning of the program. It becomes live only when written to first. Further, a variable may have more than one live range separated by times when it is dead.

[3] Our current implementation only does dataflow analysis for scalars and a simple form of array dataflow analysis that can prove arrays to be dead only if they are never used again. If more-complex array dataflow analysis is included then our results can only get better.

[4] Such programs are likely to be rare. Typically data in embedded systems is used in a time critical manner. If persistent data is required, it is usually written into files or logging devices.

Our method is clearly profile-dependent; that is, its improvements are dependent upon how representative the profile data set really is. *Indeed, all existing scratch-pad allocation methods, whether compiler-derived or programmer-specified, are inherently profile-dependent.* This cannot be avoided since they all need to predict which data will be frequently used. Further our method does not require the profile data to be like the actual data in all respects – so long as the relative re-use trends between variables are similar in the profile and actual data, good allocation decisions will be made, even if the re-use factors are not identical. A regions gain may even be higher with non-profile data if its data re-use is more than in the profile data.

Real-time guarantees Not only does our method improve run-time and energy consumption, it also improves the real-time guarantees of embedded code. To understand why, consider that the worst-case memory latency is a component of the worst-case execution time. Our method like all compiler-decided allocation methods, guarantees that the latency of each memory instruction is known for sure. This translates into total predictability of memory system behavior, thus immensely aiding in obtaining a better WCET. Such real time benefits of scratch-pad have been observed before too. [Wehmeyer and Marwedel 2004]

Program Code We also separately show how our method can also easily be extended to allocate program code objects. Although, code objects are accessed more heavily than data objects (one fetch per instruction), dynamic schemes like ours are not likely to be applicable in all cases. One, compared to data caches use of instruction caches is much more feasible due to their effectiveness with much smaller sizes. So it is not uncommon to find use of instruction caches (but not data caches) especially in high end embedded systems like motorola's STARCORE, MFC5xx and 68HC. Two, for low and medium end embedded systems code is typically stored in ROM/flash. An example of such a system is motorola's MPC500. Unlike DRAM devices, ROM/flash devices have lower seek times (in the order of 75ns-120ns, 20 ns in burst/page mode) and power consumption. For low end embedded systems, this would mean an access latency of a cycle or two. For such low end embedded systems using ROM/Flash where cost is also a lot more important factor, speeding up accesses to code objects as compared to accesses to data objects in dram is not very attractive. Nevertheless, for high end systems which store code in ROM/flash such as the motorola MCORE and motorola 6812, methods to speed up accesses to code can improve performance immensely. Our proposed extension for handling code would thus enable our dynamic method to be used for speeding up code accesses in such systems.

Heap data Our method does not allocate heap data in the program to the scratch-pad. Programs with heap data still work, however – all heap data is allocated to DRAM and the global stack and program code can still use the scratch-pad using our method, but no SRAM acceleration is obtained for heaps. Heap data is difficult to allocate to the scratch-pad at compile-time because the total amount and lifetime of heap data is often data-dependent and therefore unknowable at compile-time. Software caching strategies [Hallnor and Reinhardt 2000; Moritz et al. 2000] can be used for heap, but they have significant overheads. We have separately developed the first compile-time method for allocating heap data to scratch pad [Dominguez et al. 2005]. For this reason, we do not consider heap data any further in this paper.

Impact If adopted, the impact of this work will be a significant improvement in the cost, energy consumption, and runtime of embedded systems. Our results show up to 39.8%

reduction in run-time for our method for global and stack data and code vs. the optimal static allocation method in [Avissar et al. 2002] also extended for code. With hardware support for DMA, present in some commercial systems, the runtime gain increases to up to 42.3% respectively. *The actual gain depends on the SRAM size, but our results show that close to the maximum benefit in run-time and energy is achieved for a substantial range of small SRAM sizes commonly found in embedded systems.* Using an accurate power simulator, our method also shows up to 31.3% reduction in energy consumption vs. an optimal static allocation. The details of our power simulator and results are provided in the section 7. Our method does incur some code-size increase due to the inserted transfers; the code size increase averages a modest 1.9% for our benchmarks compared to the unmodified original code for a uniform memory abstraction; such as for a machine without scratch-pad memory.

The rest of the paper is organized as follows. Section 2 overviews the method for global and stack data, and proposes a new DPRG data structure. Section 3 describes the precise method used to determine the memory transfers of global and stack variables at each program point. Section 4 extends the algorithm for allocating procedure objects. Section 5 describes some details needed to make the algorithm work correctly in all cases. Section 6 describes the layout of variables in scratch-pad and the process of code generation. Section 7 presents an evaluation of our methodology. Section 8 overviews related work. Section 9 concludes.

2. METHOD OVERVIEW AND DATA STRUCTURES USED

Our dynamic memory allocation method for program objects such as code, stack and global variables takes the following approach. At compile-time, the method inserts code into the application to copy program objects from DRAM into the scratch-pad whenever it expects them to be used frequently thereafter, as predicted by previously collected profile data. Program objects in the scratch-pad may be evicted by copying them back to the DRAM to make room for new variables. Like in caching, all data is retained in DRAM at all times even when the latest copy is in the scratch-pad. Unlike software caching, since the compiler knows exactly where each program object is at each program point, no runtime checks are needed to find the location of any given variable. We show in [Udayakumaran and Barua 2003] that the number of possible dynamic allocations is exponential in both the number of instructions and the number of variables in the program. The problem is almost certainly NP-complete, though we have not attempted to formally prove this.

Lacking an optimal solution, a heuristic is used. Our cost-model driven greedy heuristic has three steps. First, it partitions the program into *regions* where the start of each region is a *program point*. Changes in allocation are made only at program points by compiler-inserted code that copies data between the scratch-pad and DRAM. The allocation is fixed within a region. The choice of regions is discussed in the next paragraph. Second, the method associates a *timestamp* with every program point such that (i) the timestamps form a partial order; and (ii) the program points are reached during runtime roughly in timestamp order. In general, it is not possible to assign timestamps with this property for all programs. Later in this section, however, we show a method that by restricting the set of program points and allowing multiple timestamps per program point, is able to define timestamps for all programs. Third, memory transfers are determined for each program point, in timestamp order, by using the cost-driven algorithm in section 3.

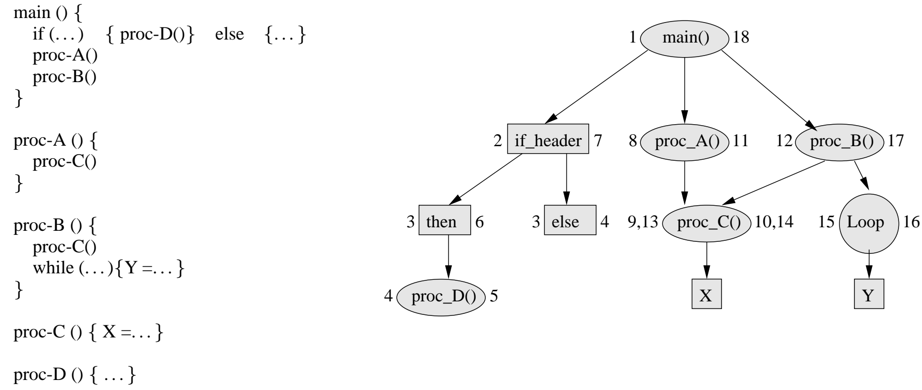

Fig. 1. Example showing (a) a program outline; and (b) its DPRG showing nodes, edges & timestamps.

2.1 Deriving regions and timestamps

The choice of program points and therefore regions, is critical to our algorithm's success. Regions are the code between successive program points. Promising program points are (i) those after which the program has a significant change in locality behavior, and (ii) those whose dynamic frequency is less than the frequency of its following region, so that the cost of copying into the scratch-pad can be recouped by data re-use from scratch-pad 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 scratch-pad. With the above two criteria in mind, we define program points as (i) the start and end of each procedure; (ii) just before and just after each loop (even inner loops of nested loops); (iii) the start and end of each **if** statement's **then** part and **else** part as well as the start and end of the entire **if** statement; and (iv) the start and end of each **case** in all **switch** statements in the program as well as the start and end of the entire **switch** statement. In this way, program points track most major control-flow constructs in the program. Program points are merely candidate sites for copying to and from the scratch-pad – whether any copying code is actually inserted at those points is determined by a cost-model driven approach, described in section 3.

Figure 1 shows an example illustrating how a program is marked with timestamps at each program point. Figure 1(a) shows the program outline. It consists of five procedures, namely *main()*, *proc-A()*,*proc-B()*, *proc-C()* and *proc-D()*, one loop and one **if-then-else** construct. The only program constructs shown are loops, procedure declarations and calls, and **if** statements – other instructions are not. Accesses to two selected variables X and Y are also shown.

Figure 1(b) shows the *Data-Program Relationship Graph* (DPRG) for the program in figure 1(a). The DPRG is a new data structure we introduce that helps in representing regions and reasoning about their time order. The DPRG is the program's call graph appended with new nodes for loops, if-then's and variables. In the DPRG shown in figure 1(b), there are five procedures, one loop, one if statement, and two variables represented by nodes. Separate nodes are shown for the entire **if** statement (called if-header) and for its **then** and **else** parts. On the figure oval nodes represent procedures, circular nodes represent loops, rectangular nodes represent **if** statement nodes, and square nodes represent variables. Edges to procedure nodes represent calls; edges to loop and if nodes shows that the child is in its parent; and edges to program object nodes represent memory

accesses to that program object from its parent. No additional edges exist to model continue and break statements. The DPRG is usually a directed acyclic graph (DAG), except for recursive programs, where cycles occur.

Figure 1(b) also shows the timestamps (1-18) for all program points, namely the beginnings (shown on left of nodes) and ends (shown on right) of every *procedure*, *loop*, *if-header*, *then* and *else* 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, in two ways. First, our DFS is modified to number *then* and *else* nodes of **if** statements starting with the same number since only one part is executed per invocation. For example, the start of the *then* and *else* nodes shown in the figure both are marked with timestamp 3. The numbering of the end of *if-header* node (marked 7 in the figure) follows the numbering of either the *then* and *else* parts, whichever consumes more timestamps. Second, it traverses and timestamps 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 9 & 13 at its beginning, and 10 & 14 at its end.

Now we can see that the timestamps are a partial order rather than a total order. This is because timestamps should not be used to derive an order between two nodes such that one is a child of the *then* part of some *if-header* node, and the other is a child of the *else* part of the same *if-header*. Such nodes have no relative order.

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. This is a useful property, and we speculate that timestamps may be useful for other compiler optimizations as well that need to reason about execution order, such as compiler-controlled prefetching [Luk and Mowry 1998], value prediction [Larson and Austin 2000] and speculation [Bringmann 1995].

Timestamps have their limitations in that they do not directly work for **goto** statements or the insides of recursive cycles; but we have work-arounds for both which are mentioned in section 5.

3. ALGORITHM FOR DETERMINING MEMORY TRANSFERS OF GLOBAL AND STACK VARIABLES

This section describes the proposed algorithm for determining the memory transfers of global and stack variables at each program point. Before running this algorithm, the DPRG is built to identify program points and mark the timestamps. Next, profile data is dynamically collected to measure the frequency of access to each variable separately for each region. This frequency represents the weight of the edge from a parent node to a child variable. Profiling also measures the average number of times a region is entered from a

parent region. This represents the edge weight between two non variable nodes. The total frequency of access of a variable is the product of all the edge weights along the execution path from the *main()* node to the variable.

We first discuss the part of our method that allocates global and stack variables. In the next section, we show how this method can be extended with some minor modifications to also allocate program code. An overview of the first part of our memory transfer algorithm is as follows. At each program point, the algorithm determines the following memory transfers: (i) the set of variables to copy from DRAM into the scratch-pad and (ii) the set of variables to evict from DRAM to the scratch-pad to make way for incoming variables. The algorithm computes the transfers by visiting each program point (and hence each region) once in an order that respects the partial order of the timestamps. For the first region in the program, variables are brought into the scratch-pad in decreasing order of frequency-per-byte of access. Thereafter for subsequent regions, variables currently in DRAM are considered for bringing into the scratch-pad in decreasing order of frequency-per-byte of access, but only if a *cost model* predicts that it is profitable to do so. Variables are preferentially brought into empty space if available, else into space evicted by variables that the compiler has proven to be dead at this point, or else by evicting live variables. Completing this process for all variables at all timestamps yields the complete set of all memory transfers.

The cost model works as follows. Given a proposed incoming variable and one-or-more variables to evict for the incoming variable, the cost model determines if this proposed swap should actually take place. In particular, copying a variable into the scratch-pad may not be worthwhile unless the cost of the copying and the lost locality of evicted variables is overcome by its subsequent reuse from scratch-pad of the brought-in variable. The cost model we use models each of these components to derive if the swap should occur.

Detailed algorithm Figure 2 describes the above algorithm in pseudo-code form. A line-by-line description follows in the rest of this section.

Figure 2 begins by declaring several compiler variables. These include V-fast and V-slow to keep track of the set of application variables allocated to the scratch-pad and DRAM, respectively, at the current program point. Bring-in-set, Swap-out-set and Retain-in-fast-set store their obvious meaning at each program point. Dead-set refers to the set of variables in V-fast in the previous region whose lifetime has ended. The frequency-per-byte of access of a variable in a region, collected from the profile data, is stored in freq-per-byte[variable, region].

The algorithm is as follows. Lines 1-2 computes the allocation for the first region in the application program. For the first region, variables are greedily brought into the scratch-pad in decreasing order of frequency-per-byte of access. Line 3 is the main **for** loop that steps through all the subsequent program points in timestamp order. At each program point, line 7 steps through all the variables, giving preference to frequently accessed variables in the next region. For each variable V in DRAM (line 8), it tries to see if it is worthwhile to bring it into the scratch-pad (lines 9-21). If the amount of free space in the scratch-pad is enough to bring in V, V is brought in if the cost of the incoming transfer is recovered by the benefit (lines 10-13). Otherwise, if variables need to be evicted to make space for V, the best set of variables to evict is computed by procedure Find-swapout-set() called on line 15 and the swap is made (lines 16-20). If the variable V is in the scratch-pad (line 22), then it is retained in the scratch-pad provided it has not already been swapped out so

```
Define                      /* The values of all of the quantities defined below change at each program point */
   Set V-slow                                  /* Set of variables in DRAM at this point */
   Set V-fast                                  /* Set of variables in the scratch-pad at this point */
   Set Bring-in-set                            /* Variables to bring into the scratch-pad at this program point */
   Set Swapout-set                             /* Set of variables for eviction to DRAM */
   Set Retain-in-fast-set                      /* Set of variables to retain in the scratch-pad */
   Set Dead-set                                /* Set of variables in V-fast whose lifetimes have ended */
   float freq-per-byte[variable,region]        /* Access frequency per byte of variable in region in profile data*/
void       Memory-allocator()
1.  initial-candidate-list ← Sort variables accessed in first region in decreasing order of freq-per-byte[variable, first region].
2.  Assign V-fast and V-slow for first region by filling the scratch-pad in greedy order from initial-candidate-list.
3.  for all timestamped program points in the application visited in partial order of their timestamps, starting at second region
4.        Swapout-set ← NULL_SET;    Bring-in-set ← NULL_SET;    Retain-in-fast-set ← NULL_SET
5.        Dead-set = Variables which are no longer alive at this program point
6.        Free-space = Free-space + sizeof(Dead-set)
7.        for all variables V accessed in this region in decreasing order of frequency-per-byte
                    /* In rest of code 'this region' ≡ region after current program point */
8.             if V ∈ V-slow
9.                  if (sizeof(V) ≤ Free-space)                        /* V fits; no need to swap out variables */
10.                      Benefit-of-bring-in-V ← Find-benefit(V, NULL_SET)
11                       if (Benefit-of-bring-in-V > 0)
12.                           Bring-in-set ← Bring-in-set ⋃ {V}
13.                      endif
14.                 else                                              /* V does not fit; try to swap out variables */
15.                      Swapout-set-for-V ← Find-swapout-set(V)
16.                      if (Swapout-set-for-V ≠ NULL)
17.                           Swapout-set ← Swapout-set ⋃ Swapout-set-for-V
18.                           Bring-in-set ← Bring-in-set ⋃ {V}
19.                           Free-space ← Free-space + sizeof(Swapout-set-for-V) − sizeof(V)
20.                      endif
21.                 endif
22.            else /* V ∈ V-fast */
23.                 if (V not in Swapout-set)                          /* Has not been swapped out so far */
24.                      Retain-in-fast-set ← Retain-in-fast-set ⋃ {V}
25.                 endif
26.            endif
27.       endfor
28.       V-fast ← V-fast ⋃ Bring-in-set − Swapout-set − Dead-set
29.       V-slow ← V-slow ⋃ Swapout-set − Bring-in-set − Dead-set
30.       Store V-fast and V-slow for this region.
31.endfor                                                              /* For all program points */
32.return

Set       Find-swapout-set(V)
33.Swapout-set-for-V ← NULL_SET
34.Swapout-candidate-list ← Sort variables in the scratch-pad in ascending order of size. In case of a tie, choose variable with
   the higher next-timestamp-of-access. Exclude variables that have become dead in this region (already removed in line 6).
35.Swapout-candidate-list ←
       Swapout-candidate-list − (Swapout-set ⋃ Bring-in-set ⋃ Retain-in-fast-set)    /* Update candidate-list*/
36.Size-required ← sizeof(V) - Free space
37.while (((Swapout-candidate ← next-element(Swapout-candidate-list)) ≠ NULL) and (Size-required > 0))
38.       Benefit-of-swap ← Find-Benefit(V, Swapout-candidate)
39.       if (Benefit-of-swap > 0)
40.            Swapout-set-for-V ← Swapout-set-for-V ⋃ {Swapout-candidate}
41.            Size-required ← Size-required − sizeof(Swapout-candidate)
42.       endif
43.end-while
44.if (Size-required > 0)                          /* Could not find required space by swapping out */
45.       return (NULL)                            /* Do not swap */
46.endif
47.return (Swapout-set-for-V)                     /* Found required space by swapping out */

int       Find-benefit(V,Swapout-candidate)
48.Latency-gain ← freq-per-byte[V, this region] ⋆ size(V) ⋆ (Latency_slow_mem − Latency_fast_mem)
49.Latency-loss ←
       freq-per-byte[Swapout-candidate, this region] ⋆ size(Swapout-candidate)* (Latency_slow_mem − Latency_fast_mem)
50.Migration-overhead ← Time for copying Swapout-candidate (if modified) to DRAM + Time for copying V to the SRAM
51.Benefit-of-swap ← latency-gain − latency-loss − Migration-overhead
52.return Benefit-of-swap
```

Fig. 2. Algorithm for determining dynamic memory allocation

far by a higher frequency-per-byte variable (line 23-25). Finally, after looping through all the variables, lines 28 and 29 update, for the next program point, the set of variables in scratch-pad and DRAM, respectively. Line 30 stores this resulting new memory map for the region after the program point.

Next we explain **Find-swapout-set()** (lines 33-47) called in line 15. It calculates and returns the best set of variables to copy out to DRAM when its argument V is brought in. Possible candidates to swap out are those in scratch-pad, ordered in ascending order of size (line 34); but variables that have already been decided to be swapped out, brought in, or retained are not considered for swapping out (line 35). Thus variables with higher frequency-per-byte of access are not considered since they have already been retained in scratch-pad in line 24. Among the remaining variables of lower frequency-per-byte, as a simple heuristic small variables are considered for eviction first since they cost the least to evict. Better ways of evaluating the swapout set of least cost by evaluating all possible swapout sets are avoided to avoid an increase in compile-time; moreover we found these to be unnecessary since only variables with lower frequency-per-byte than the current variable are considered for eviction. The **while** loop on line 37 looks at candidates to swap out one at a time until the space needed for V has been obtained. A cost model is used to see if the swap is actually beneficial (line 38); if it is the swapout set is stored (lines 40-41). More variables may be evicted in future iterations of the **while** loop on line 37 if the space recovered by a single variable is not enough. If swapping out variables that are eligible and beneficial to swap out did not recover enough space (line 44), then the swap is not made (line 45). Otherwise procedure Find-swapout-set() returns the set of variables to be swapped out.

Cost model Finally, we look at **Find-benefit()** (lines 36-43), called in lines 10 & 38. It computes whether it is worthwhile, with respect to runtime, to copy in variable V in its first argument by copying out variable Swapout-candidate in its second argument. The net benefit of this operation is computed in line 51 as the latency-gain − latency-loss − Migration-overhead. The three terms are explained as follows. First, the latency gain is the gain from having V in the scratch-pad in the next region (line 48). Second, the latency loss is the loss from not having Swapout-candidate in the scratch-pad in the next region (line 49). Third, the migration overhead is the cost of copying itself, estimated in line 50. The overhead depends on the point at which the transfer is done. So the overhead of transfers done outside a loop is less than inside it. We conservatively choose the transfer point that is outside as many inner loops as possible. The choice is conservative in two ways. One, points outside the procedure are not considered. Two, transfers are not moved beyond points with earlier transfer decisions. An optimization done here is that if variable Swapout-candidate in scratch-pad is provably not written to in the regions since it was last copied into the scratch-pad, then it need not be written out to DRAM since it has not been modified from its DRAM copy. This optimization provides functionality to the dirty bit in cache, without needing to maintain a dirty bit since the analysis is at compile-time. The end result is an accurate cost model that estimates the benefit of any candidate allocation that the algorithm generates.

Optimization One optimization that we consider is ignoring the multiple allocation decisions inside higher level regions and instead adopting one allocation inside the particular region. The static allocation adopted is found by doing a greedy allocation based on the frequency per byte value of the variables used in the region. Such an optimization can be

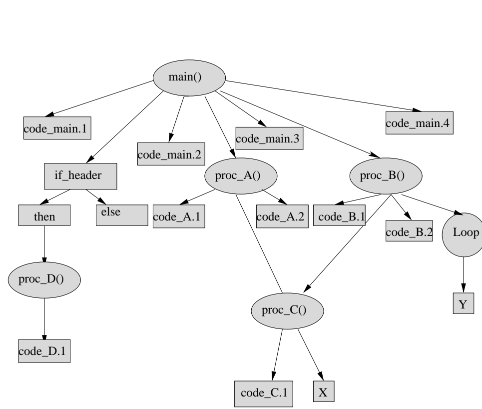

Fig. 3. Example DPRG with code nodes.

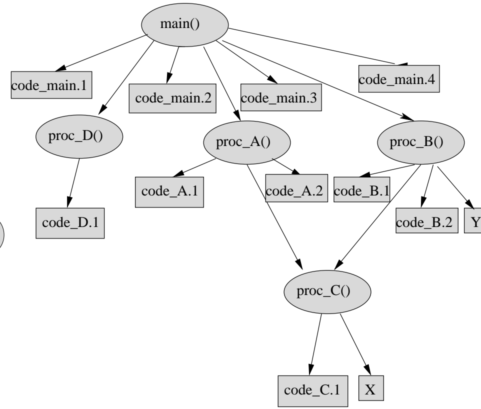

Fig. 4. An example coalesced DPRG.

useful in cases when transfers are done inside loops and the resulting transfer cost is very high. In such cases all though our method would guarantee than the high cost can be recouped, it might be beneficial to adopt a simple one allocation for the particular region. To aid in making this choice, our method compares the likely benefit from a purely dynamic allocation with a static allocation for the region. Based on the result either the dynamic allocation strategy is retained or the static allocation used for the region.

4. ALGORITHM EXTENSION FOR CODE OBJECTS

We now show how the above framework can be extended for allocating program code objects. The key questions that that need to be answered are: One, at what granularity do we allocate code objects (basic-block/procedures/files); two, how is an code object represented in the DPRG; and three, how is the algorithm and cost model modified. The first issue we look at is the granularity of the program objects. Like in the case of data objects, the smaller the size of the code objects, the larger the benefits of scratch-pad placement are likely to be. One way of achieving this is to consider code objects in units of basic blocks. However, code generation for allocations at such small granularity is likely to involve introducing too many branch instructions while also precluding the use of existing linker technology for its implementation. The other drawback is that complexity of profiling also increases. Another approach to obtaining smaller sized code objects is to selectively create procedures from nested loop structures in programs since it is profitable to place loops in scratch-pad. This optimization called outlining (inverse of inlining) is available in some commercial compilers like IBM's XLC compiler. Both methods can yield code objects of smaller size but at vastly different implementation costs. For its ease of implementation we choose outlining to provide small sized program objects.

The next issue is how to represent the code objects in the DPRG. Since our choice of program objects is at the level of procedures (native or outlined), we attach code objects to parent procedures just like variables are (henceforth called code variable nodes). Figure 3 shows an example of a DPRG which also includes code objects shown as rectangular nodes. Every procedure node has a variable child node representing the code executed before the next function call. For example in figure 3 code_A.1 represents all the instructions in proc_A executed before the procedure proc_C is called and code_A.2 represents

all the instructions in proc_A executed after return from proc_C until the end the proc_A. An advantage of such a representation is that the framework for allocating data objects can be used with little modification for allocating code objects as well. As in the case of data objects, profiling is used to find for every node the frequency of access of each child code variable accessed by that node. For a code variable its frequency is given by its corresponding number of dynamic instructions executed. The size of the code variable is the size of the portion of the procedure until the next call site in the procedure. We also create a modified DPRG structure in which non procedure DPRG nodes have been coalesced into the parent procedure node. We call this new structure the *coalesced-dprg*. Figure 4 shows the coalesced-dprg for the original DPRG in Figure 3.

Now the original algorithm described in the previous section is modified as follows. When a procedure node in the DPRG is visited, first we check if the procedure node can be allocated in the scratch-pad. Such an approach is motivated by the same considerations as for our choice of procedures over basic blocks. Using the original algorithm would have required using expensive profiling to find the frequency of the code variable in much smaller portions of code. To determine if a procedure node can be allocated to scratch-pad, it is helpful to use the coalesced-dprg. It suffices to find out if a hypothetical allocation (lines 4-30) done at the corresponding procedure node using the coalesced-dprg(using lines 7-21), would allocate the procedure node to the scratch-pad. If the procedure gets allocated to the scratch-pad then the available scratch-pad memory is decreased by the size of the procedure node. Then the algorithm proceeds with the rest of the pseudo code explained in the previous section(lines 4-30) using the original DPRG with the only other difference that the procedure node is ignored, that is it is neither considered for swapin or swap out. Thus the modified algorithm would allocate both data and code while retaining the same framework.

5. ALGORITHM MODIFICATIONS

For simplicity of presentation the algorithm in sections 2 and 3 leaves some issues unaddressed. Solutions to these issues are proposed in this section. *All the modifications proposed here are carried out by the algorithm presented before and are driven by the same cost model. They do <u>not</u> define a new algorithm. Two of the modifications– pointers and goto's extend the functionality of the algorithm*

Function pointers and pointer variables that point to global and stack variables Function pointers and pointer variables in the source code that point to global and stack variables can cause incorrect execution when the pointed-to object is moved. For example consider a pointer variable p that is assigned to the address of global variable a in a region where a is in the scratch-pad. Later if p is de-referenced in a region when a is in DRAM, then p points to the incorrect version of a. We now discuss two different alternative strategies to handle this issue. An advantage of both these schemes is that both alternatives need only basic pointer information. However, pointer analysis information can be used to optimize both the schemes and further reduce their overhead.

Alternative 1: The first alternative that we present involves using a runtime disambiguator that corrects the address of the pointer variable. The alternative involves four steps. First, pointer analysis is done to find the pointers to global/stack and code objects. Second, at statements where the address of a global/stack or a code variables is assigned including when they are passed as reference parameters, the address of the DRAM location of the

object is assigned. This is not hard since all compilers identify such statements explicitly in the intermediate code. With such a reassignment all pointer variables in the program refer to the DRAM locations of variables. The advantage of DRAM addresses of objects is that they are unique and fixed during the program object's lifetime; unlike its current address which changes every time the program object is moved between the scratch-pad and the DRAM. Note that only direct assignments of addresses need to be taken care, statements which copy address from one pointer to another do not need any special handling. The third step is that at each pointer de-reference in the program the pointer address is translated by compiler-inserted code from the DRAM address it contains to the current address of the pointed-to program object which may be in the scratch-pad or DRAM. This translation is done using a custom run-time data structure which, given the DRAM address, returns the current location of the program object. Since pointer arithmetic is allowed in C programs, the data structure must be able to look up addresses to the middle of program objects and not just to their beginnings. The data structure we use is a height-balanced tree having one node for each global/stack and code object whose address is taken in the program. Each node stores the DRAM address range of the variable and its current starting address. Since recursive procedures are not allocated to the scratch-pad, each variable has only one unique address range.[5] The tree is height-balanced with the DRAM address as the key. The tree is updated at run-time when a pointed-to variable is transferred between banks and it is accessed through pointers before its next transfer. Since n-node height-balanced trees offer $O(log_2 N)$ lookup this operation is reasonably efficient. An advantage of both these schemes if that both alternatives need only basic pointer information. However pointer analysis information can be used to immensely, simplify both the schemes.. Once the current base address of the program object in scratch-pad is obtained, the address value may need to be adjusted to account for any arithmetic that has been done on the pointer. This is done by adding the offset of the old pointer value from the base of the DRAM address of the pointed-to variable to the newly obtained current location. The final step that we do is that after the dereference the pointer is again made to point to its DRAM copy. Similar to when translating, the pointer value may need adjustments again to account for any arithmetic done on the pointer.

It appears that the scheme for handling pointers described above suffers from high runtime overhead since an address translation (and retranslation) is needed at every pointer dereference. Fortunately this overhead is actually very low for four reasons. First, pointers to global/stack and code objects are relatively rare; pointers to heap data are much more common. Only the former require translation. Second, most global/stack and code accesses in programs are not pointer de-references and thus need no translation. Third, even when translation and hence a subsequent retranslation is needed in a loop (and thus is time-consuming) it is often loop-invariant and can be placed outside the loop. The translation is loop invariant if the pointer variable or aggregate structure containing the pointer variable is never written to in the loop with a address taken expression.[6] We found this to be often the case and in almost all cases the translation can be done before the loop. Consequently the retranslation can be done after the loop. Finally, one optimization we employ is that

[5]With pointer analysis information this translation can be replaced by a simpler comparison involving only the dram addresses of variables in the pointed-to set

[6]Pointer arithmetic is not a problem since it is not supposed to change the pointed-to variable in ANSI-C semantics.

in cases where it can be conservatively shown that the variable's address does not change between the address assignment and pointer de-reference, then the current address of the program object regardless of it is in the scratch-pad or DRAM can be assigned and no translation is required. Such instances most trivially happen in cases when for optimized code generation, the address is assigned to a pointer just before the loop and then the pointer variable is used in the loop. For all these reasons, we found the run-time overhead of translation to be under 1% for most of our benchmarks. This cost is easily overcome for all our benchmarks by the much larger gain from our method in access latency. Finally, the memory footprint of the run-time structure is vanishingly small for our benchmarks.

Alternative 2: A second alternative is to use a strategy of restricting the offsets a pointed-to variable can take. The strategy proceeds in the following steps. First, a variable whose address is never taken is placed with no restrictions since no pointers can point into it. Address-taken information is readily available in most compilers; in this way, many global/stack variables are unaffected by pointers. Second, variables whose address is taken have the following allocation constraint for correctness: *for all regions where the variable's address is taken or the variable may be accessed through pointers, the variable must be allocated to the same memory location.* For example if variable a has its address taken in region R_1, and may be accessed through a pointer in region R_5, then both regions R_1 and R_5 must allocate a to the same memory location. This ensures correctness as the intended and pointed-to memory will be the same. The consensus memory bank for such regions is chosen by first finding the locally requested memory bank for each region; then the chosen bank is the frequency-weighted consensus among those requests. Regions in which a variable with address taken is accessed but not through pointers are unconstrained, and can allocate the variable anywhere.

Currently we have only explored alternative 1 for our benchmarks. Exploring alternative 2 and hybrids of alternative 1 and alternative 2 is part of our future work.

Join nodes A second complication with the above algorithm is that for any program point visited along multiple paths (hence having multiple timestamps), the **for** loop in line 4 is visited more than once, and thus more than one allocation is made for that program point. An example is node proc C() in figure 1. We call such nodes with multiple timestamps join nodes since they join multiple paths from main(). Join nodes can arise due to many program constructs including (i)in the case of a procedure invoked at multiple call sites, (ii)at the end of conditional path or (iii) in the case of a loop. For parents of join nodes, considering the join node multiple times in our algorithm is not a problem - indeed it the right thing to do, so that the impact of the join node is considered separately for each parent. However, for the join node itself, multiple recommended allocations result, one from each path to it, presenting a problem. One solution is cloning the join node and the sub-graph below it in the DPRG along each path to the join node, but the code growth can be exponential for nested join nodes. Even selective cloning is probably unacceptable for embedded systems. Instead, our strategy avoids all cloning by choosing the allocation desired by the most frequent path to the join node for the join node. Subsequently compensation code is added on all incoming edges to the join node other than for the most frequent path. The compensation code changes the allocation on that edge to match the newly computed allocation at the join node. The number of instances of compensation code is upper-bounded by the number of incoming edges to join nodes We now consider the most common scenarios separately.

Join nodes: Procedure join nodes Our method chooses the allocation desired by the most frequent path to the procedure join node for the join node. Subsequently as discussed before, compensation code is added on all incoming edges to the join node other than for the most frequent path.

Join nodes: Conditional join nodes Join nodes can also arise due to conditional paths in the program. Examples of conditional execution include **if-then**, **if-then-else** and **switch** statements. In all cases, conditional execution consists of one or more conditional paths followed by an unconditional join point. Memory allocation for the conditional paths poses no difficulty – each conditional path modifies the incoming memory allocation in the scratch-pad and DRAM memory to optimize for its own requirements. The difficulty is at the subsequent unconditional join node. Since the join node has multiple predecessors, each with a different allocation, the allocation at the join node is not fixed at compile-time. The solution used is the same as for procedure join nodes and is used for similar reasons. Namely, the allocation desired by the most frequent path to the join node is used for the join node, just as above.

Join nodes: loops A third modification is needed for loops. A problem akin to join nodes occurs for the start of such loops. There are two paths to the start of the loop – a forward edge from before the loop and a back edge from the loop end. The incoming allocation from the two paths may not be the same, violating the desired condition that there be only one allocation at each program point. To find the allocation at the end of the backedge, Procedure **Find-swapout-set** is iterated once over all the nodes inside the loop. The allocation before entering the loop is then reconciled to obtain the allocation desired just after entering the loop – in this way, the common case of the back edge is favored for allocation over the less common forward edge.

Recursive functions Our approach discussed so far does not directly apply to stack variables in recursive or cross-recursive procedures. With recursion the call graph is cyclic and hence the total size of stack data is unknown. Hence for a compiler to guarantee that a variable in a recursive procedure fits in the scratch-pad is difficult. Our baseline technique is to collapse recursive cycles to single nodes in the DPRG, and allocate their stack data to DRAM. Edges out of the recursive cycle connect this single node to the rest of the DPRG. This provides a clean way of putting all the recursive cycles in a black box (not to be considered in the future). Our method can now handle the modified DPRG like any other DPRG without cycles. DRAM placement of stack variables is not too bad for two reasons. First, recursive procedures are relatively rare in embedded codes. Second, a nice feature of this method is that when recursion is present, all program objects other than stack frames of recursive procedures such as data in non recursive descendents and non stack data can still be placed in the scratch-pad by our method.

One optimization is for tail-recursive procedures, *i.e.* those where only the variables from the last invocation are live. Our method can re-use the same fixed-size scratch-pad space for all the dynamic invocations of a tail-recursive procedure, thus eliminating the unknown-size problem. This optimization is not currently implemented.

Goto statements Our DPRG formulation in section 2.1 does not consider arbitrary **goto** statements. Because it is widely known that goto statements are poor programming practice they are exceedingly rare in any domain nowadays. Nevertheless, it is important to handle them correctly. We only refer to goto statements here; breaks and continues in loops are fine for DPRGs.

Our solution to correctly handle goto statements involves two steps. First, the DPRG is built and the memory transfers are decided without considering goto statements. Second, the compiler detects all goto statements and inserts memory transfer code along all goto edges in the control-flow graph to maintain correctness. The fundamental condition for correctness in our overall scheme is that the memory allocation for each region is fixed at compile-time; but different regions can have different allocations. Thus for correctness, for each goto edge that goes from one region to another, memory transfers are inserted just before the goto statement to convert the contents of scratch-pad in the source region to that in the destination region. In this way goto statements are handled correctly but without specifically optimizing for their presence. Since goto statements are very rare, such an approach adds little run-time cost for most programs.

The DPRG construct along with the extensions in this section enable our method to handle all ANSI C programs. For other languages, structured control-flow constructs likely will be variants, extensions or combinations of constructs mentioned in this paper, namely procedure calls, loops, if and if-then-else statements, switch statements, recursion and goto statements.

6. LAYOUT AND CODE GENERATION

This section has three issues. First, it discusses the layout assignment of variables in scratch-pad. Second, it discusses the code generation for our scheme. Third, it discusses how the data transfer code may be optimized.

Layout assignment The first issue in this section is deciding where in the scratch-pad to place the program objects being swapped in. A good layout at a region should be able to place most or all of the program objects desired in the scratch-pad by the memory transfer algorithm in section 3. To increase the chances of finding a good layout, the layout assignment algorithm should have the following two characteristics. First, the layout should minimize fragmentation that might result when program objects are swapped out, so as to increase the chance of finding large-enough free holes in future regions. Second, when a memory hole of a required size cannot be found, compaction in scratch-pad should be considered along with its cost.

Our layout assignment algorithm runs as a separate pass after the memory transfers are decided. It visits the regions of the application in the partial order of their timestamps. At each region, it does the following four tasks. **First**, the method updates the list of free holes in the scratch-pad by de-allocating the outgoing variables from the previous region. **Second**, it attempts to allocate incoming variables to the available free holes in the decreasing order of their size. The largest variables are placed first since they are the hardest to place in available holes. When more than one hole can be used to fit a variable, the *best-fit* rule is followed: the smallest hole that is large enough to fit the incoming program object is used for allocation. The best-fit rule is commonly used for memory allocation in varying domains such as segmented memory and sector placement on disks [Tanenbaum 1998]. **Third**, when an adequate-sized hole cannot be found for a variable, compaction in the scratch-pad is considered. In general, compaction is the process of moving variables towards one end of memory so that a large hole is created at the end. However, we consider a limited form of compaction that has lower cost: only the subset of variables variables that need to be moved to create a large-enough hole for the incoming request are moved. Also, for simplicity of code generation, compaction involving blocks containing

program objects used inside a loop is not allowed inside the loop. Compaction is often more attractive than leaving the incoming program object in DRAM for lack of an adequate hole – this is because compaction only requires two scratch-pad accesses per word, which is often much lower cost than even a single DRAM access. The cost of compaction is included in our layout-phase cost model; it is done only when its cost is recovered its benefit. Compaction invalidates pointers to the compacted data and hence is handled just like a transfer in the pointer-handling phase (section 5) of our method. Pointer handling is delayed to after layout for this reason. **Fourth**, in the case that compaction is not profitable, our approach attempts to find a candidate program object to swap out to DRAM. Again, the cost is weighed against the benefit to decide if the program object should be swapped out. If no program object in the scratch-pad is profitable to swap out, our approach decides to not bring in the requested-incoming program object to the scratch-pad. In section 7 we show that this simple strategy is quite effective.

Code generation After our method decides the layout of the variables in SRAM in each region, it generates code to implement the desired memory allocation and memory transfers. Code generation for our method involves changing the original code in three ways. **First**, for each original variable in the application (Eg: *a*) which is moved to the scratch-pad at some point, the compiler declares a new variable(Eg: *a_fast*) in the application corresponding to the copy of *a* in the scratch-pad. The original variable *a* is allocated to DRAM. By doing so, the compiler can easily allocate *a* and *a_fast* to different offsets in memory. Such addition of extra symbols causes zero-to-insignificant code increase depending on whether the object formats includes symbolic information in the executable or not. **Second**, the compiler replaces occurrences of variable *a* in each region where *a* is accessed from the scratch-pad by the appropriate version of *a_fast* instead. **Third**, memory transfers are inserted at each program point to evict some variables and copy others as decided by our method. The memory transfer code is implemented by copying data between the fast and slow versions of to-be-copied variables (Eg: between *a_fast* and *a*). Data transfer code can be optimized; optimizations are described later in this section.

Since our method is dynamic, the fast versions of variables (declared above) have limited lifetimes. As a consequence different fast variables with non-overlapping lifetimes may have overlapping offsets in the scratch-pad address space. Further, if a single variable is allocated to the scratch-pad at different offsets in different regions, multiple fast versions of the variables are declared, one for each offset. The requirement of different scratch-pad variables allocated to the same or overlapping offsets in the scratch-pad in different regions is easily accomplished in the backend of the compiler.

Although creating a copy in scratch-pad for global variables is straightforward, special care must be taken for stack variables. Stack variables are usually accessed through the stack pointer which is incremented on procedure calls and decremented on returns. By default the stack pointer points to a DRAM address. This does not work to access the stack variable in scratch-pad; moreover the memory in scratch-pad is not even maintained as a stack! Allocating whole frames to scratch-pad means losing allocation flexibility. The other option of placing part of stack frame in scratch-pad and the rest in main memory requires maintaining two stack pointers which can be a lot of overhead. The easiest way to place a stack variable *a* in scratch-pad is to *declare its fast copy a_fast as a global variable but with the same limited lifetime as the stack variable*. Addressing the scratch-pad copy as a global avoids the difficulty that the scratch-pad is not maintained as a stack. Thus all

Application	Source	Description	Lines of code	Object code size in KB	Data size in bytes	Data memory instructions as % total dynamic instructions
Lpc	UTDSP	Linear predictive coding encoder	493	340	7684	29.1
Edge Detect	UTDSP	Image edge detection	368	350	196600	7.0
Gsm	MIbench	Speech compression	5473	417	20287	29.2
Spectral	UTDSP	Power spectral estimation	449	332	4356	47.5
Compress	UTDSP	Discrete cosine transform	296	324	70752	12.2
G721.Wendyfung (G721)	UTDSP	G.721 ADPCM algorithm	627	250	4148	44.0
Stringsearch	MIbench	String search	2757	242	1572	47.1
Rijndael	MIbench	AES algorithm	1142	315	22160	26.0

Table I. Application programs.

variables in scratch-pad are addressed as globals. Having globals with limited lifetimes is equivalent to globals with overlapping address ranges. The handling of overlapping variables was mentioned in the previous paragraph.

Code generation for handling code blocks involves modifying the branch instructions between the blocks. The branch at the end of the block would need to be modified to jump to the current location of the target. This is easily achieved when the unit of the code block is a procedure by leveraging current linking technology. Similar to the case of variables, the compiler inserts new procedure symbols corresponding to the different offsets taken by the procedure in the scratch-pad. Then it suffices to modify the calls to call the new procedures. The backend and the linker would (without any modifications) then generate the appropriate branches. As mentioned earlier, outlining or extracting loops into extra procedures can be used to create small sized code blocks. For this optimization to work, we promote the local variables that are shared between the loop and the rest of the code as global variables. These are given unique names prefixed with the procedure name. In our set of benchmarks, we observe the overhead due to these extra symbols to be very small.

Reducing runtime and code size of data transfer code Our method copies data back and forth between the scratch-pad and DRAM. This overhead is not unique to our approach – hardware caches also need to move data between scratch-pad and DRAM. The ode-size overhead of such copying is minimized by using a shared optimized copy function. In addition, faster copying is possible in processors with the low-cost hardware mechanisms of Direct Memory Access (DMA) such as in ARMv6, ARM7. DMA accelerates data transfers between memories and/or I/O devices.

7. RESULTS

Experimental setup This section presents results comparing our dynamic method against the earlier provably optimal static allocation described in [Avissar et al. 2002]. A summary of the previous results of the optimal static method in [Avissar et al. 2002] are as follows:

for their benchmarks, the optimal static method improves run-time by an average of 54% compared to an all-DRAM allocation; and by an average of 44% compared to an earlier static method that places only global variables in SRAM rather than both global and stack variables. For our comparison, the static method in [Avissar et al. 2002] is extended to also handle program code. In the new formulation for the static solution, procedures (the same set as used by our dynamic method) are treated as being similar to global variables.

Our methodology is as follows. We have implemented the DPRG and memory transfer algorithm in a GCC v3.2 cross-compiler targeting the Motorola M-Core embedded processor. After compilation the benchmarks are executed on the public-domain cycle-accurate simulator for the Motorola M-Core available as part of the GDB v5.3 distribution. DMA is simulated by counting the estimated costs of those mechanisms in the simulator.

Our experimental setup for estimating the energy consumption of programs with and without our method is as follows. An M-core power simulator [Baynes et al. 2001; 2003], kindly donated by that group, is used to obtain energy estimates for instructions and SRAM. This is an instruction-level power simulator similar to [Sinha and Chandrakasan 2001; Tiwari and Lee 1998]; its instruction power numbers were measured using an ammeter connected to an M-core hardware board. DRAM power is estimated by a DRAM power simulator we built into the M-core simulator. It uses the DRAM power model described in [Janzen 2001; Micron-datasheet 2003] for the MICRON external DDR Synchronous DRAM chip. The DRAM chip size is set equal to the data size in the energy model. To store code, the experimental setup also includes a Flash device [Micron-flash data sheet]. The current and voltage values of the device are also incorporated into the power simulator.

The applications and memory characteristics are as follows. The embedded applications evaluated are shown in table I. The applications selected primarily use global and stack data, rather than heap data. In the experimental setup, an external DRAM with 20-cycle latency and an internal SRAM (scratch-pad) with 1-cycle latency is simulated in the default configuration. The Flash has a seek time of 120 ns or about 24 cycles [Micron-flash data sheet]. In the experiments below, the SRAM size is varied and for each size the run-time gain from the dynamic method in this paper vs. the best static method is measured. The DRAM and Flash sizes, of course, are assumed to be large enough to hold all program data and code respectively. Other experiments below perform more detailed studies, including varying the above parameters and others, and measure the impact of doing so.

Results on run-time improvement Before presenting results, it is important to understand that our dynamic method does not give a benefit versus a static allocation for all scratch-pad sizes. This is obvious at the extremes. For a scratch-pad size = 0% of data+ code size (or object code size), the two methods are equal since neither can put any data or code in the (absent) the scratch-pad. Similarly, at the other extreme of the scratch-pad size when scratch-pad size = 100% of program size, both methods are nearly equal [7] since they both fit all the data and code in the scratch-pad. *The benefit from any dynamic method is seen only for intermediate scratch-pad sizes which can fit some but not all of the data and code in scratch-pad.* Thus instead of presenting the benefit of the dynamic method for a fixed scratch-pad size, we measure a range of scratch-pad sizes for which our method shows an improvement and by how much.

[7] At large sizes the dynamic method degrades a little compared to the static method because dynamic transfers benefit only a little, and the dynamic method uses a polynomial-time method unlike the worst-case exponential in optimal static allocation.

Benchmark	Useful range of dynamic method				Maximum benefit vs. static	
	Minimum SRAM size (bytes)	Maximum SRAM size (bytes)	Length of range (bytes)	% Accesses to SRAM at max size (%)	SRAM size (bytes)	Run-time gain vs vs. static (%)
Lpc	210	1600	1390	86	234	23.0
Edge detect	220	950	730	96.0	500	55.0
Gsm	200	1850	1650	85.1	870	35.0
Spectral	200	2000	1800	60.1	800	15.0
Compress	40	2000	1960	96.8	490	60.0
G721	180	2500	2320	98.0	600	55.0
Stringsearch	40	400	360	72.3	280	21.2
Rijndael	7170	8000	830	66.7	7170	54.0
AVERAGE				82.7		39.8

Table II. Useful range of dynamic method and run-time gain vs. static allocation.

Table II shows, for each benchmark, the range of scratch-pad sizes for which our method yields an improvement over the static allocation method in [Avissar et al. 2002]; and the maximum improvement in that range. In particular, columns two through four show the range of scratch-pad sizes for which our method improves performance as compared to the static method by at least 1%. Columns two and three present the minimum and maximum of the useful scratch-pad sizes, respectively, for each benchmark. For example, for the *Lpc* benchmark, the dynamic method outperforms the static by at least 1% when the scratch-pad size is between 210 and 1600 bytes. Column four presents the length of the range. Columns five is discussed in the next paragraph. Finally, columns six and seven show the scratch-pad size for which the maximum improvement over the best static allocation is obtained, and the size of the improvement. [8] The average of the maximum improvements across benchmarks in the last column is 39.8%.

From table II we can derive two salient conclusions. First, our method yields a significant run-time benefit for many of the commonly occurring small scratch-pad sizes that typically appear in embedded systems (under a kilobyte for most embedded systems; a few kilobytes for some high-end embedded systems). The maximum improvement ranges from 15.0% (*Spectral*) to 60.0% (*Compress*); averaging 39.8%. Second, the reason that larger scratch-pad sizes do not yield a benefit is that larger sizes *are not needed* for our benchmarks. Hence larger sizes would not be used, and a lack of improvement for those sizes is not harmful. To see why, consider that in most programs, a small fraction of the data accounts for a large fraction of the accesses. If the scratch-pad size is large enough to accommodate this frequently used data using the static allocation, the dynamic method yields little run-time improvement for that and larger sizes. This reasoning is verified by column five of table II. Column five shows the percentage of memory accesses that go to scratch-pad for the maximum useful scratch-pad size. *The high average of 82.7% shows that the maximum useful scratch-pad size already has good performance for most benchmarks so a*

[8]The maximum improvement configuration shown is not the maximum across all sizes, but is restricted in two ways. First, only scratch-pad sizes which yield a good absolute performance, defined as sizes for which at least 60% of accesses go to the scratch-pad, are considered for finding the maximum gain. Also scratch-pad sizes greater than 20% of the total data size are not considered since they are likely to be too expensive to be used. In this way, we attempt to derive a maximum benefit across feasible scratch-pad sizes only.In the case of range of sizes, we select the smallest size.

much larger scratch-pad is not needed. This is further verified by our observation that even doubling the scratch-pad size compared to the maximum useful scratch-pad size in column three yields only a average 0.6% and a maximum 2.0 % improvement. This shows that the maximum useful range is already at the point of diminishing returns for our benchmarks; and hence *cost-effective scratch-pad sizes are in the useful range*. The maximum scratch-pad size larger than which diminishing returns are seen is highly application-dependent and is likely to be larger for applications with larger data sets.

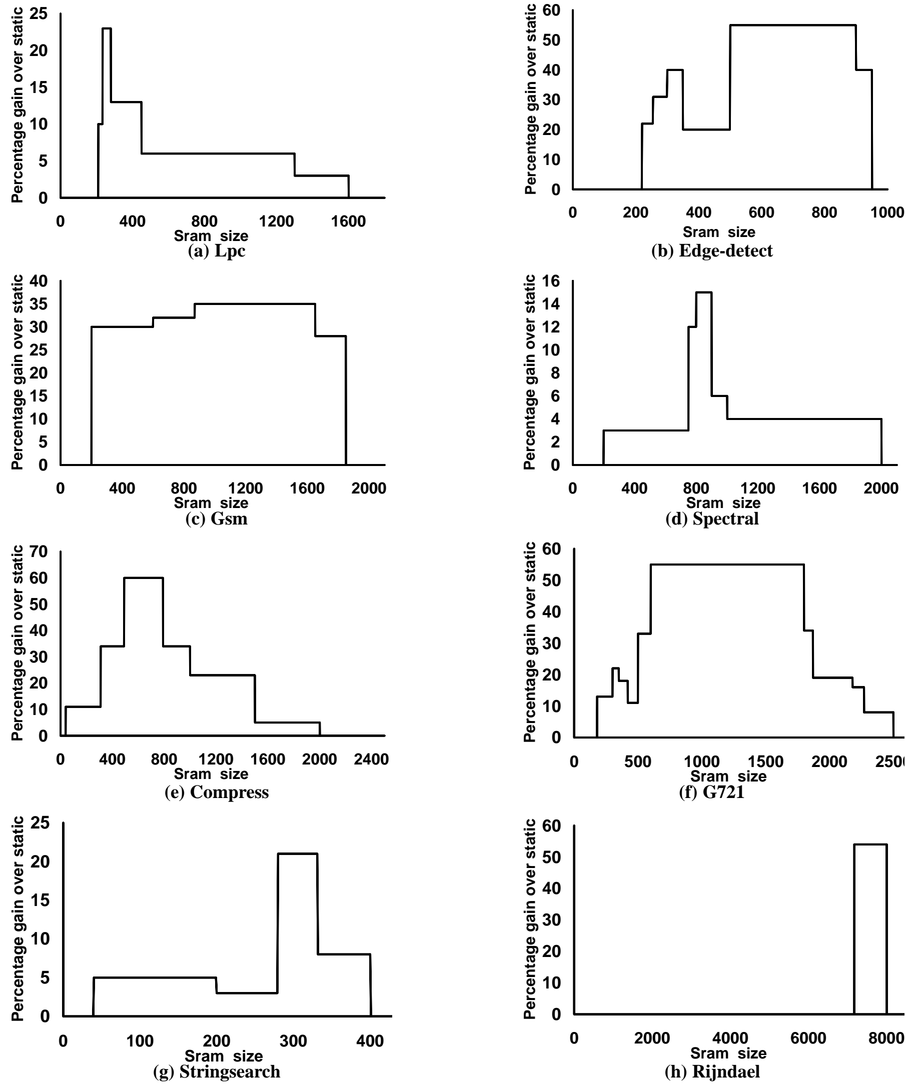

Fig. 5. Runtime gain from our dynamic method vs. optimal static method for different SRAM sizes.

Figure 5 shows the same data as in table II but in more detail. For each benchmark, the X-axis varies the scratch-pad size and the Y-axis shows the run-time gain of the dynamic method in this paper over the provably optimal static allocation method in [Avissar et al. 2002]. From the figure we see two trends. First, for any given scratch-pad size the dynamic method does at least as well as the static method and for the small sizes shown, it often does better. Second, the shapes of the curves follows steps; this is not surprising since those are the discrete points at which the allocation of individual variables in the application changes in either the static or dynamic allocations.

The shapes of the curves in figure 5 can be understood by why the up-steps, down-steps and valleys occur. First, an up-step is when the gain from the dynamic method suddenly increases beyond a certain scratch-pad size; an example is for *compress* at scratch-pad size=490 bytes. This happens when an increase in the scratch-pad size enables the dynamic method to accommodate an additional variable in the scratch-pad, perhaps by replacing a lower-frequency variable; while the static method has the same allocation since the additional space is not enough for another variable. Thus the dynamic method's gain over static increases. Second, a down-step is when the gain from the dynamic method suddenly decreases beyond a certain scratch-pad size; an example is for *compress* at scratch-pad size=790 bytes. This happens when the static method can accommodate some of the additional variables in the dynamic allocation and bridge the gap with it. Third, a valley is when the gain in a certain range is lower than either before or after it; an example is for *Edge-detect* in the range 350-500 bytes in scratch-pad size. A valley is nothing more than an down-step at its start and an up-step at its end. The down-step and up-step occur because of changes in allocations of different variables.

Experiments on maximum benefit configuration The rest of the experiments vary several architectural and method parameters to measure the impact. They are conducted for the smallest scratch-pad size which yields the maximum run-time improvement for our method versus the optimal static method. The reason for this choice is that presenting all the remaining data for all the possible scratch-pad sizes yields a volume of data that is too large to present. Thus we had to choose one scratch-pad size per benchmark to show the underlying reasons as to why our method improves performance. The point of maximum benefit is a good choice to gain such insights. Figure 6 shows the reduction in percentage

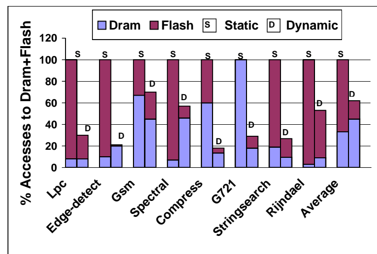

Fig. 6. Percentage of memory accesses going to the DRAM and Flash for each benchmark for the maximum benefit configuration.

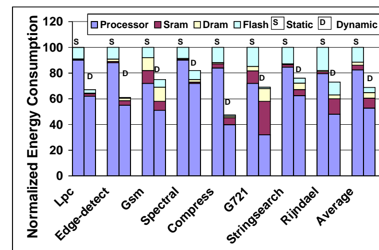

Fig. 7. Reduction in energy consumption from dynamic method for the maximum benefit configuration.

of memory accesses going to the non SRAM devices - the DRAM and Flash because of the improved locality to the scratch-pad afforded by our method. The average reduction

across benchmarks is a very significant 38% reduction in Flash+DRAM accesses versus the optimal static allocation. The total number of memory accesses actually increases in our method because of the added transfer code, but the reduced number of accesses to Flash+DRAM more than compensates for this increase by delivering an overall reduction in runtime.

Figure 7 compares the total system-wide energy consumption of application programs with our method versus with the optimal static allocation in [Avissar et al. 2002]. Each bar is further divided into the different energy components namely the DRAM, Flash, SRAM and processor energy consumption. *We measured an average reduction of 31.3% in total energy consumption* for our applications by using our method vs. the best static allocation. This number is noteworthy since it refers to total system energy and not just memory system energy. The savings in energy are because our method reduces the number the number of Flash and DRAM accesses in the program by converting them to SRAM accesses. Flash and DRAM accesses cost more energy than SRAM accesses for two reasons. First, these devices take more energy to access than SRAM banks in our platform. The ratio of DRAM bank energy to SRAM bank energy for a single access is about 5.7:1 in our energy model; but this number is highly implementation-dependent. From data sheets [Micron-datasheet 2003;], we found that the energy cost of our Flash device to be similar to the DRAM device. So for simplicty, we assume that the ratio of Flash bank energy to SRAM bank energy for a single read access is also about 5.7:1. Second, when a DRAM access occurs in an in-order processor such as our Motorola MCore, the processor is idle while waiting for the DRAM access to complete, but it still dissipates substantial amounts of energy (although slightly less than when instructions are executing). Most embedded processors are in-order; superscalars are rare in embedded systems. Current-day technologies to turn down the processor to a low-energy energy-saving state typically take thousands of cycles to complete. This is infeasible during a DRAM access which typically only takes 10-100 cycles. Thus the processor burns substantial amount of energy while waiting for a DRAM access. These reasons can be also verified from the figure. From the figure it can be seen that on average the 82.5% of the energy consumption in the static case and 52% of the energy consumption in the dynamic case is consumed by the processor. Consequently the energy savings in the processor portion by eliminating unnecessary stalls contributes the most to the total savings. Energy reduction in smaller measure is also contributed by reduction in DRAM or Flash accesses or in some cases both. This is accompanied by an increase in the SRAM portion.

Table III shows some whole program and some region statistics for our benchmarks. Columns two and three show the number of global and stack variables per benchmark. There are a substantial number of them in our benchmarks. Column four shows the code growth (in bytes of code portion) from our method, primarily because of the inserted transfer code, as a percentage of the original code size. The average code growth is a modest 1.8% versus the un-modified original code for a uniform memory abstraction; such as for a machine without scratch-pad memory. Column five shows the run time spent in the copy procedure. Column six shows the number of static regions in each benchmark. Column seven shows the average static size of regions in instructions. We see that regions contain about 57 static instructions on average. (In columns six and seven only the regions that are visited at least once during run-time are counted.) Column eight shows the average *turnover fraction* across regions, where the turnover fraction for a region is defined as the

amount of *new* data allocated in the scratch-pad for that region expressed as a percentage of the SRAM size. The average turnover fraction is 11.0%; thus on average 11.0% of the scratch pad data is new per region. The relatively low turnover fraction shows that the method is careful not to unnecessarily transfer data: it does so only when beneficial and when it does bring something in scratch-pad, it retains it for several regions before eviction. The turnover is higher for benchmarks such as Rijndael where transfers are carried out inside loops.

Benchmark	**Program statistics**				**Region statistics**		
	# of global variables	# of stack variables	Code growth vs original (%)	% Runtime in copy block	# of regions (instr)	Ave. static size	Turnover fraction (%)
Lpc	17	36	1.0	0.1	44	57.3	9.1
Edge Detect	10	9.0	4.1	0.1	24	64.0	0.6
Gsm	39	382	0.2	4.4	156	58.2	6.3
Spectral	12	33	2.4	1.2	28	45.2	5.6
Compress	8	11	3.2	1.9	39	18.5	1.1
G721.Wendyfung	25	57	2.6	2.0	21	49.1	0.5
Stringsearch	6	9	0.5	16.8	13	13.4	16.0
Rijndael	11	58	0.1	4.9	26	134.8	49.0
AVERAGE	15.5	83.3	1.8	3.9	43.9	57.0	11.0

Table III. Program and region statistics.

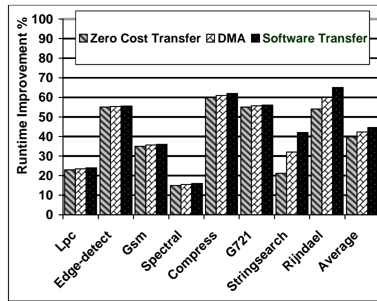

Fig. 8. Run-time gain for different data-transfer methods for the maximum benefit configuration.

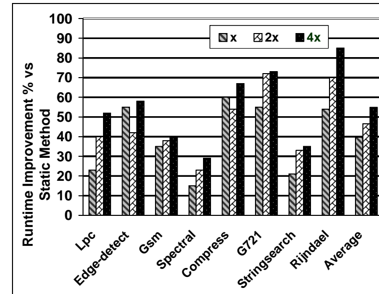

Fig. 9. Effect of varying DRAM and Flash latencies on run-time gain from our method for the maximum benefit configuration.

Figure 8 shows the run-time gain for different data-transfer strategies between the scratch-pad and DRAM. Note that a data transfer involves a call to a copy procedure(13 lines). Also at a program point there may be multiple calls to the copy procedure for different address ranges. The data transfer strategies that are shown for each benchmark are (i) all-software transfer (used in all experiments so far in this paper); (ii) transfers accelerated by DMA; and (iii) a hypothetical zero-run-time (free!) data transfer mechanism. DMA is a hardware mechanism available in some embedded processors and is discussed in section 6. Faster transfers can provide additional benefit in two ways. One, due to lower cost of transfers the

allocation method might be able to bring in more variables and in some cases choose a totally different allocation. Two, faster transfers also means that the cost of transfers is lower. But comparing the first and third bar, we see that the run-time gain suffers only by 44.6%-39.8%=4.8% because of transfers. This shows that in general software transfers can deliver good performance. This is because (as we observed) allocations do not change much with faster transfers for our benchmarks. Also, our method is largely successful in transferring data only when doing so yields a benefit; at the same time unnecessary transfers of dead data and non-dirty scratch-pad data are not done. Given these reasons, not surprisingly, the additional run-time gain from using faster transfer mechanisms is small: only an additional 2.5% with DMA transfers. While the runtime increase is modest, another benefit results because of faster transfers. We observed that faster transfers can result in increase in the useful range of SRAM sizes for a benchmark: for example *G721.Wendyfung* benchmark the useful range of the dynamic method is increased from 180 to 2500 bytes (all-software) to 180 to 2700 bytes (DMA).

Figure 9 shows the effect of increasing Flash and DRAM latency on the run-time gain from our dynamic method versus the best static allocation. Apart from the original latencies labeled as *x*, the gain is shown for two other latencies namely twice the original termed *2x* and latency four times the original termed *4x*. Recall that the original DRAM and Flash latency is assumed 20 cycles. Since our method reduces the number of accesses to Flash and DRAM, the gain from our method is greater with higher latencies for most benchmarks. The figure shows that the run-time gain from our method increases from 39.8% with the original DRAM and Flash latencies to 54.8% with latency 4 times the original latencies.

Area benefits A different perspective on the impact of our method can be seen by considering the reduction in area that it can offer to a embedded system designer, who has a desired performance requirement in mind. To measure the reduction in area, we performed a study on the area benefits. For lack of a better heuristic, we measured the area benefits at three different SRAM sizes specified as a fraction of the useful range. These sizes are 25%, 50%, 75% of the useful range. Now we ask, how much additional SRAM size would be needed by the static method to obtain the same runtime as the dynamic method at these sizes. For some of these benchmarks like Rijndael, Spectral the static method already does as well as the dynamic method at these sizes. This can also be seen in figure 5 where for these benchmarks the runtime gain at these points is small. But for the other benchmarks, we measured a decent reduction in area ranging from 5% to a high of 220%. On an average, at these three different sizes of 25%, 50% and 75% we obtain 75%, 43% and 33% reduction in area respectively. None of these are insignificant when considering that on chip memory uses upto 50% of the total chip area [Keitel-Sculz and Wehn 2001]

Efficacy of offset assignment Here we look at the efficacy of our simple offset assignment pass. As discussed before, our offset assignment pass is made up of two parts – a best fit memory management along with a limited compaction when an appropriate size hole cannot be found. To study how well this simple strategy performs, we compare it with a perfect address assignment method. The perfect address assignment method is assumed to magically fit all the variables at different program points without requiring to sacrifice any variable to fit another variable or using compaction. Further, at every program point it needs only one copy procedure and hence the minimum call overhead. Figure 10 shows the degradation of our method compared to a hypothetical method with perfect address

assignment pass. Column two of the table shows the run time degradation when compared to the perfect assignment and column three shows the runtime spent in compaction as a percentage of the total runtime. The results show that that our address assignment pass suffers negligible degradation for almost all the benchmarks and on an average suffers a 1.1% degradation. This happens primarily because, program objects do not live in the scratch-pad for too long and hence free holes get easily created. Secondly, fragmentation is almost totally overcome with the help of compaction which at a negligible cost delivers a huge benefit.

Benchmark	% Runtime Degradation Vs Perfect Assignment	% Runtime in Compaction
Lpc	0.0	0
Edge Detect	8.1	0
Gsm	0.01	0.00002
Spectral	0.04	0.00001
Compress	0.02	0.00002
G721	0	0
Stringsearch	0.5	0
Rijndael	0.0	0
AVERAGE	1.1	

Fig. 10. Comparison of our address assignment with perfect address assignment.

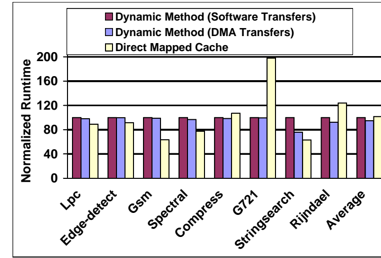

Fig. 11. Normalized run time for a cache only and scratch-pad only architecture measured for maximum benefit configuration

Comparison with caches Here we compare our method with an architecture that uses a cache. 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 scratch-pad and DRAM directly accessed by the CPU, but have no data cache or I-cache. Examples of such architectures include low-end chips such as the Motorola MPC500, Analog Devices ADSP-21XX, Motorola Coldfire 5206E; mid-grade chips such as the Analog Devices ADSP-21160m, Atmel AT91-C140, ARM 968E-S, Hitachi M32R-32192, Infineon XC166 and high-end chips such as Analog Devices ADSP-TS201S, Hitachi SuperH-SH7050, and Motorola Dragonball. 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 scratch-pad is simple to design and verify, and provide better real-time guarantees for global and stack data [Wehmeyer and Marwedel 2004], power consumption, and cost [Angiolini et al. 2004; Banakar et al. 2002; Steinke et al. 2002; Verma et al. 2004a] compared to caches.

Nevertheless, it is interesting to see how our method compares against processors containing caches. We choose our desired data capacity as the SRAM size at which the dynamic method obtains maximum benefit compared to the static method. Note that this SRAM size is only an indication of relative gain versus static method. So for purposes of comparing with caches, it is a fairly unbiased choice. To ensure a fair comparison the total silicon area of fast memory (scratch-pad or cache) is equal in both the architectures and roughly equal to the silicon area of the scratch-pad. For our experiments we choose an cache architecture similar to the intel IXP network processor which has an Icache and a Dcache of equal sizes. The goal of our experiment is to compare the performance of cache

and SPM of equal silicon area.[9] So the desired data capacity is divided equally between the Icache and Dcache. For a scratch-pad and cache of equal area the cache has lower data capacity because of the area overhead of tags and other control circuitry. Area estimates for cache and scratch-pad are obtained from Cacti [Wilton and Jouppi 1996]. The cache simulated is direct-mapped, has a line size of 8 bytes, and is in 0.5 micron technology. On a cache miss, we assume the first word incurs the full DRAM latency of 20 cycles and 1 cycle for each byte thereafter. The scratch-pad 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 the scratch-pad. The Dinero cache simulator [Dinero Cache Simulator 2004] is used to obtain run-time results; it is combined with Cacti's energy estimates per access to yield the energy results.

Figure 11 shows the normalized run time for cached and non-cached architectures. The first bar represents our method with software transfers. The third bar represents the runtime with cached architecture. As caches have the advantage of hardware mechanisms for fast transfer, we thought it would be interesting to compare when our method also can use some faster transfer mechanism. So the second bar represents our method with faster transfers using hardware like DMA. Our method does better than cached architecture for 3 of the benchmarks. On the average our method on scratch-pad has 1.7% less runtime when compared to cached architecture. This increases to 6.7% savings with the help of faster transfers. Of course, the improvement in real-time guarantees from the scratch-pad is much larger.

The reason we believe our method does better for 3 benchmarks is that it can correctly identify the more reused data/code and retains it in the scratch-pad whereas cache is likely to evict it even when fetching less used data/code. This is especially significant in case of programs with large loops and loops with procedure calls in them. In such cases a cached architecture might do worse because of large transfers. For two benchmarks – Lpc and Edge-detect the performance of the dynamic method is very close to the cached architecture. But for rest of the three benchmarks namely Gsm, Spectral and Stringsearch cached architectures do better than our method. These were benchmarks which had some very large variables which do not fit in the scratch-pad. So while our method cannot fit these variables into the scratch-pad, the cached architecture which only deals with cache lines can bring in data belonging to these variables. This is a general advantage cached architectures would have over any compile time scratch-pad method. To some extent at least for regular programs, this gap can be bridged with the help of more aggressive outlining and advanced array optimization techniques like array blocking [Ahmed et al. 2000; Lim et al. 2001],structure splitting [T. M. Chilimbi and Larus 1999] etc. Nevertheless, we believe it is remarkable for a compile-time method to do so well compared to hardware cache.

8. RELATED WORK

Earlier methods by others on allocating a portion of the data to the scratch-pad include [Banakar et al. 2002; Hiser and Davidson 2004; Panda et al. 2000; Sjodin et al. 1998; Sjodin and Platen 2001; Steinke et al. 2002; Wehmeyer et al. 2004]. All of these methods are

[9]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 adjustable. To overcome this difficulty we first fix the sizes of the Icache and Dcache to the nearest possible SRAM size. Then a scratch-pad of the same total area is chosen; this is easier since scratch-pad sizes are less constrained.

limited in two ways. First, all of them are static methods in that the scratch-pad allocation does not change across run-time. In contrast our method is a dynamic method which can change the contents of the scratch-pad during the run of the application. Second, all of these methods are able to place only global variables in the scratch-pad. Our earlier method on scratch pad allocation [Avissar et al. 2001; 2002] improved upon the previous work in being the first method published anywhere that is able to place a portion of the stack data in scratch-pad. We demonstrated large improvements by doing so [Avissar et al. 2002]; this is not surprising since some of the stack data is frequently used. Further, our earlier method in [Avissar et al. 2001; 2002] is provably optimal for global and stack data, and thus supersedes all other static methods. It models the static allocation problem as a 0/1 integer linear programming (ILP) problem which is solved optimally by a solver such as CPlex [ILOG Corporation 2001]. However, our earlier method is static. This paper carries our earlier work to the next level in making the allocation dynamic, while retaining the ability to place stack variables in scratch-pad.

Some schemes for scratch-pad allocation have had different objectives from ours; three such objectives are as follows. First, the primary goal of the method in [Hiser and Davidson 2004] is to provide an easily re-targetable compiler method for allocation of data across many different types of memories. Second, the goal of the work in [Angiolini et al. 2004; Verma et al. 2004a] has been to map portions of the *code* of the application to the scratch-pad memory; our goal is to map both data and code. Some methods such as [Steinke et al. 2002; Wehmeyer et al. 2004] can place both code and data in scratch-pad; however, their allocation is static and does not handle stack variables. Third, the goal of the work in [Angiolini et al. 2003; Udayakumaran et al. 2002] is 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.

Schemes also have been proposed which use a hardware approach for managing the scratch-pad. [Angiolini et al. 2003]. Angiolini et al select a set of memory address ranges based on their access frequency and map them to the scratch-pad. A special decoder is then used to translate the addresses to locations in the scratchpad. The above approach is based on hardware customization instead of software/compiler customization like our method. Although, hardware customization has the advantage that approaches based on it do not require application source to be available, the applicability of the approach is limited to only architectures which have the required special hardware. An extension of the above approach was proposed by Francesco et al. in [Francesco et al. 2004] who used a combination of hardware and software techniques to manage the scratch-pad at runtime. Special hardware needed included a DMA engine and a configurable dynamic memory management. Software support is mainly in the form of high-level API's. Apart from the limitation of special hardware the other drawback of the approach is that it being a runtime approach, it would not be well suited for real time applications with high predictability requirements. One advantage though is that it can adapt to dynamic applications better.

Purely dynamic memory allocation strategies to date are mostly restricted to Software Caching [Hallnor and Reinhardt 2000; Moritz et al. 2000]. Software caching emulates 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 [Moritz et al. 2000]. [Moritz et al. 2000] targets the primary cache; [Hallnor and Reinhardt 2000] manages the secondary

cache in desktops. The only software caching scheme that is a competitor is [Moritz et al. 2000]; it however does not measure speedup vs. an all-DRAM allocation, and does not quantify its overheads. All software caching techniques suffer from significant overheads in runtime, code size, data size, energy consumption, and result in unpredictable runtimes. Our dynamic allocation method overcomes all of these drawbacks.

Many studies [M.Kandemir et al. 2001; Steinke et al. 2002; Verma et al. 2004b; C. Eisenbeis and Fran 1990; Anantharaman and Pande 1998; R. Schreiber 2004] have been proposed that consider dynamic management of the scratch-pad. However, only the strategy by [Verma et al. 2004b] is a generally applicable solution like ours. To better understand the merits and demerits of these various approaches, these can be divided into two categories based on if scope of the method is at the level of a loop or the whole program. The approaches in [Steinke et al. 2002] and [Verma et al. 2004b] are whole program solutions and are based on ILP formulations with the primary objective of minimizing energy consumption. While Steinke et al in [Steinke et al. 2002] only consider program objects, Verma et al's scheme in [Verma et al. 2004b] is a comprehensive ILP formulation and considers both data and program objects. So for a detailed comparision we do not discuss the approach in [Steinke et al. 2002] further. The second category of approaches [M.Kandemir et al. 2001; C. Eisenbeis and Fran 1990; Anantharaman and Pande 1998; R. Schreiber 2004]. consider each loop nest separately. The next few paragraphs will make a detailed comparison of our approach with [R. Schreiber 2004; M.Kandemir et al. 2001; C. Eisenbeis and Fran 1990; Anantharaman and Pande 1998] and [Verma et al. 2004b].

The methods in [R. Schreiber 2004; M.Kandemir et al. 2001; C. Eisenbeis and Fran 1990; Anantharaman and Pande 1998] only allocate global arrays unlike our method which is comprehensive and can allocate global, stack and program code. To compare the global data allocation part of our method with theirs, like our method, these schemes also move data (only global) between DRAM and the scratch pad under compiler control. The primary distinguishing feature of these dynamic methods is that they are focussed on optimizing perfectly nested loop nests for scratch-pad accesses. So these methods while considering each loop independently allocate parts or whole variables accessed in the loop nest into the scratch-pad. The local analysis makes available the entire scratch pad for each loop nest. In contrast, our method is globally optimized for the entire program. Our method is also not constrained to make the entire scratch-pad available for each loop nest; instead it may choose to retain data in the scratch-pad between successive regions thus saving on transfer time to DRAM.

Another limitation of these methods is their limited applicability which stems from these method only targeting arrays accessed through affine(linear) functions of enclosing loop induction variable. Thus these cannot handle non-affine accesses such as accesses using pointers or indexed array expressions. These methods are also restricted in their use because the required affine analysis for these methods can be only done for only well structured loop without any other control flow such as **if-else**, **break** and **continue** statements.In contrast, our method is completely general and is able to exploit locality for all codes, including unstructured code, code with irregular accesses patterns, variables other than arrays and code with pointers.

These method do have one advantage over our data allocation scheme, though. For regular affine-function codes matching their restrictions, they can, unlike ours, allow for the

possibility of bringing in parts of an array instead of the whole array. This is an important advantage for their method for such regular codes. This aspect of their work is complementary to our work, and some approach based on affine functions will benefit our method for this reason. We have investigated such a strategy that is able to transfer parts of an array into the scratch-pad based on affine analysis, but takes a whole program view, and is integrated with the rest of our method so as to not sacrifice its generality [Udayakumaran and Barua 2006]. Nevertheless the method in this paper is useful by itself as well because *affine-specialized methods are not a substitute for this method* – our method handles non-affine codes and affine-specialized one's do not. Further our results are also valid since they are an under-estimate: our run-time gain against static allocations can only improve when combined with affine-specialized method.

The recent paper by Verma et al [Verma et al. 2004b] also implements a dynamic strategy similar to ours. Their approach is motivated by the ILP formulation for global register allocation [Goodwin and Wilken 1996]. Both parts of the problem - finding what variables should be in the scratch-pad at different points in the program and what addresses they should be at -are solved using ILP formulations. Then like our method code is inserted to transfer the variables between the scratch-pad and DRAM.

Being ILP based, their solution is likely to be optimal in the solution space they have defined; however, ILP based solutions have some fundamental issues that limit their usefulness. One, ILP formulations have been known to be undecidable in the worst case and in many practical situations are NP hard. Their solution times, especially for large programs can be exponential. Using ILP solutions is also constrained by issues like intellectual property of source code from different vendors, maintenance of the resulting large combined code and the financial cost of ILP solvers. Due to these practical difficulties of ILP solvers, it is very rare to find ILP solvers as part of commercial compiler infrastructures despite many papers being published that use ILP techniques.

One another drawback of their approach is that like global register allocation methods, the solution though optimal is per procedure. In other words the formulation does not attempt to exploit for reuse across procedures. This might lead to some data being needlessly swapped out even if retaining it in the scratch-pad might be more beneficial across two procedures. Now even if a variable remains in the scratch-pad in both the procedures and can be identified thus, its offsets in both the procedures could be different. So code between two procedures would need to copy data from old offsets to new offsets for all the variables that remain in the scratch-pad. Similar costs exist for register allocation as well, but in the case of register allocation spilling some registers and reloading them again at the end of the procedure is not as expensive. Spilling contents of scratch-pad even if selectively and reloading at the end of procedure is likely to much more expensive. One, latency to access the next level of storage the DRAM is many orders higher (10-100 times). Two, the size of the scratch-pad is likely to be larger. And finally, selective spilling of only used offsets is not easy because of aggregate variables like arrays. Hence its much more important that the copying cost be minimized in the case of scratch-pad memory allocation. In other words, for scratch-pad memory allocation interprocedural approaches are important.

One possible remedy to the problem might be to consider extending the formulation to work with an interprocedural interference graph. This is not practically feasible for the same reasons as why interprocedural register allocation schemes based on interprocedural interference graphs would not work. One, the interference graph is likely to be very large

and two, the ILP solution which grows linearly with the number of edges$\times$variables would only become more difficult to solve. Hence interprocedural approaches have to be different from per procedure approaches.

In contrast our scheme also considers interactions across different regions including procedures. This becomes possible because of our interprocedural approach. Our method's algorithmic complexity is polynomial even in the worst case and hence can find an interprocedural solution efficiently. Consequently our approach may choose to retain some data between procedures and thus minimize the copying overhead.

An issue that is not addressed in [Verma et al. 2004b] is the issue of correctness in the presence of pointers. To address this, the ILP formulation would need to be extended to include one more extra constraint that the offsets be the same in both the locations namely when the address is taken and when the pointer is dereferenced. The drawback of this is that it would make the formulation still harder to solve. In our scheme instead of making our offset assignment phase more constrained we adopt a dynamic approach. We use a runtime structure to maintain the current location of the variable pointed to by a pointer. Before a dereference the pointer is updated with the new address from this data structure. The details can be found in the section 5. We find that the dynamic scheme ensures correctness with very little overhead. A final drawback of the scheme in [Verma et al. 2004b] is that it does not discuss how the compiler generates code to make use of the allocation decisions.

Our work is in some aspects also analogous to the offline paging problem first formulated by [Belady. 1966]. The offline paging problem deals with deriving an optimal page replacement strategy when future page references are known in advance. Analogously we look at finding a memory allocation strategy when program behavior is known in advance. Offine paging, however, cannot be used for our purposes since it makes its page transfer decisions at runtime (address translation done by virtual memory), while we need to associate memory transfers with static program points.

Comparison with caches Other researchers have repeatedly demonstrated [Angiolini et al. 2004; Banakar et al. 2002] the power, area and run-time advantages of scratch-pad over caches, even with simple static allocation schemes such as the knapsack scheme used in [Banakar et al. 2002]. Further, scratch-pads deliver better real-time guarantees than caches. In addition, our method is useful regardless of caches since our goal is to more effectively use the scratch-pad memory already present in a large number of embedded systems today such as the intel IXP network processor, ARMv6, IBM's 405 and 440 processors, motorola's 6812 and MCORE and TI TMS 370. It is nevertheless interesting to see a quantitative comparison of our method for scratch-pad memory against a cache. Section 7 presents such a comparison. Overall our method does slightly better than caches, but for some benchmarks which deal with large program object that do not fit into the scratch-pad, caches give better results.

Some embedded systems allow both a scratch-pad and a cache to be present. Examples of such processor are Intel IXP and IBM 405 processors. For such processors our method is best applied by placing the data as dictated by our method in scratch-pad, *and placing all the remaining data, assumed to be in DRAM in our method, in cached (DRAM-backed) address space instead*. In this way, the real-time improvements from scratch-pad allocation are retained for all frequently used variables. This is not the case with previous methods for cache-aware scratch-pad placement such as [Panda et al. 2000] and [Verma et al. 2004a] where frequently used variables are sometimes placed in cache; leading to poor real-time

bounds for their access. Further both the methods are static and apply to only some kind of variables. While the method in [Panda et al. 2000] is limited to global variables, the method in [Verma et al. 2004a] presents an approach for placing instruction traces with the objective of energy minimization. In contrast our method for runtime reduction is dynamic and applicable for both instruction objects (procedures) and data(global/stack) variables.

9. CONCLUSION AND FUTURE WORK

This paper presents compiler-driven memory allocation scheme for embedded systems that have SRAM organized as a scratch-pad memory instead of a hardware cache. Most existing schemes for scratch-pad rely on static data assignments that never change at runtime, and thus fail to follow changing working sets; or use software caching schemes which follow changing working sets but have high overheads in runtime, code size memory consumption and real-time guarantees. We present a scheme that follows changing working sets by moving data from scratch-pad to DRAM, but under compiler control, unlike in a software cache, where the data movement is not predictable. Predictable movement implies that with our method the location of each variable is known to the compiler at each point in the program, and hence the translation code before each load/store needed by software caching is not needed. The benefit of our method depend on the scratch size used. When compared to a provably optimal static allocation, results show that our scheme reduces runtime by up to 39.8% and overall energy consumption by up to 31.3% on average for our benchmarks, depending on the scratch-pad size used.

A direction for future research can be to extend the scheme for multi-tasking environment. Our future work will explore different strategies which have different varying needs for performance and predictability.

REFERENCES

AHMED, N., MATEEV, N., AND PINGALI, K. 2000. Tiling imperfectly-nested loop nests. In *Proceedings of the 2000 ACM/IEEE conference on Supercomputing (CDROM)*. IEEE Computer Society, 31.

ANANTHARAMAN, S. AND PANDE, S. 1998. Compiler optimization for real time execution of loops on limited memory embedded systems. In *Proc. of the 19th IEEE Real-Time Systems Symposium*.

ANGIOLINI, F., BENINI, L., AND CAPRARA, A. 2003. Polynomial-time algorithm for on-chip scratchpad memory partitioning. In *Proceedings of the 2003 international conference on Compilers, architectures and synthesis for embedded systems*. ACM Press, 318–326.

ANGIOLINI, F., MENICHELLI, F., FERRERO, A., BENINI, L., AND OLIVIERI, M. 2004. A post-compiler approach to scratchpad mapping of code. In *Proceedings of the 2004 international conference on Compilers, architecture, and synthesis for embedded systems*. ACM Press, 259–267.

APPEL, A. W. AND GINSBURG, M. 1998. *Modern Compiler Implementation in C*. Cambridge University Press.

AVISSAR, O., BARUA, R., AND STEWART, D. 2001. Heterogeneous Memory Management for Embedded Systems. In *Proceedings of the ACM 2nd International Conference on Compilers, Architectures, and Synthesis for Embedded Systems (CASES)* (Atlanta, GA). Also at http://www.ece.umd.edu/∼barua.

AVISSAR, O., BARUA, R., AND STEWART, D. 2002. An Optimal Memory Allocation Scheme for Scratch-Pad Based Embedded Systems. *ACM Transactions on Embedded Systems (TECS) 1,* 1 (September).

BANAKAR, R., STEINKE, S., LEE, B.-S., BALAKRISHNAN, M., AND MARWEDEL, P. 2002. Scratchpad Memory: A Design Alternative for Cache On-chip memory in Embedded Systems. In *Tenth International Symposium on Hardware/Software Codesign (CODES)*. ACM, Estes Park, Colorado.

BAYNES, K., COLLINS, C., FITERMAN, E., GANESH, B., KOHOUT, P., SMIT, C., ZHANG, T., AND JACOB, B. 2001. 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)*. Atlanta GA, 203–210.

BAYNES, K., COLLINS, C., FITERMAN, E., GANESH, B., KOHOUT, P., SMIT, C., ZHANG, T., AND JACOB, B. 2003. The performance and energy consumption of embedded real-time operating systems. *IEEE Transactions on Computers 52,* 11 (Nov.), 1454–1469.

BELADY., L. 1966. A study of replacement algorithms for virtual storage. In *IBM Systems Journal*. 5:78–101.

BRINGMANN, R. A. 1995. Compiler-Controlled Speculation. Ph.D. thesis, University of Illinois, Urbana, IL, Department of Computer Science.

C. EISENBEIS, W. JALBY, D. AND FRAN, C. 1990. A strategy for array management in local memory. In *Technical Report 1262, INRIA, Domaine de Voluceau, France*.

Dinero Cache Simulator Revised 2004. *DineroIV Cache simulator*. J. Edler and M.D. Hill. http://www.cs.wisc.edu/ markhill/DineroIV/.

DOMINGUEZ, A., UDAYAKUMARAN, S., AND BARUA, R. 2005. Heap Data Allocation to Scratch-Pad Memory in Embedded Systems. *To appear in Journal of Embedded Computing(JEC), Issue 4*. IOS Press, Amsterdam, Netherlands.

FRANCESCO, P., MARCHAL, P., ATIENZA, D., LUCA BENINI, F. C., AND MENDIAS, J. M. June,2004. An integrated hardware/software approach for run-time scratchpad management. In *In Proceedings of the Design Automation Conference*. ACM Press, 238–243.

GOODWIN, D. W. AND WILKEN, K. D. 1996. Optimal and near-optimal global register allocation using 0-1 integer programming. In *Software-Practice and Experience*. 929–965.

HALLNOR, G. AND REINHARDT, S. K. 2000. A fully associative software-managed cache design. In *Proc. of the 27th Int'l Symp. on Computer Architecture (ISCA)*. Vancouver, British Columbia, Canada.

HISER, J. D. AND DAVIDSON, J. W. 2004. 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*. ACM Press, 182–191.

ILOG CORPORATION. 2001. *The CPLEX optimization suite*. http://www.ilog.com/products/cplex/.

JANZEN, J. 2001. Calculating Memory System Power for DDR SDRAM. In *DesignLine Journal*. Vol. 10(2). Micron Technology Inc. http://www.micron.com/publications/designline.html.

KEITEL-SCULZ, D. AND WEHN, N. June 2001. Embedded dram development technology, physical design, and applicationissues. In *IEEE Design and Test of Computers*.

LARSON, E. AND AUSTIN, T. 2000. Compiler controlled value prediction using branch predictor based confidence. In *Proceedings of the 33th Annual International Symposium on Microarchitecture (MICRO-33)*. IEEE Computer Society.

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

LIM, A. W., LIAO, S.-W., AND LAM, M. S. 2001. Blocking and array contraction across arbitrarily nested loops using affine partitioning. In *Proceedings of the eighth ACM SIGPLAN symposium on Principles and practices of parallel programming*. ACM Press, 103–112.

LUK, C.-K. AND MOWRY, T. C. 1998. Cooperative instruction prefetching in modern processors. In *Proceedings of the 31st annual ACM/IEEE international symposium on Microarchitecture*. 182–194.

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

Micron-flash data sheet. *128Mb Q-Flash memory*. Micron technology Inc. http://www.micron.com/products/nor/qflash/partlist.aspx.

M.KANDEMIR, J.RAMANUJAM, M.J.IRWIN, N.VIJAYKRISHNAN, I.KADAYIF, AND A.PARIKH. 2001. Dynamic Management of Scratch-Pad Memory Space. In *Design Automation Conference*. 690–695.

MORITZ, C. A., FRANK, M., AND AMARASINGHE, S. 2000. FlexCache: A Framework for Flexible Compiler Generated Data Caching. In *The 2nd Workshop on Intelligent Memory Systems*. Boston, MA.

PANDA, P. R., DUTT, N. D., AND NICOLAU, A. 2000. 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).

R. SCHREIBER, D. C. 2004. Near-optimal allocation of local memory arrays. In *HPL-2004-24*.

SINHA, A. AND CHANDRAKASAN, A. 2001. JouleTrack - A Web Based Tool for Software Energy Profiling. In *Design Automation Conference*. 220–225.

SJODIN, J., FRODERBERG, B., AND LINDGREN, T. 1998. Allocation of Global Data Objects in On-Chip RAM. *Compiler and Architecture Support for Embedded Computing Systems*.

SJODIN, J. AND PLATEN, C. V. 2001. Storage Allocation for Embedded Processors. *Compiler and Architecture Support for Embedded Computing Systems*.

STEINKE, S., GRUNWALD, N., WEHMEYER, L., BANAKAR, R., BALAKRISHNAN, M., AND MARWEDEL, P. 2002. Reducing energy consumption by dynamic copying of instructions onto onchip memory. In *Proceedings of the 15th International Symposium on System Synthesis (ISSS)* (Kyoto, Japan). ACM.

STEINKE, S., WEHMEYER, L., LEE, B., AND MARWEDEL, P. 2002. Assigning program and data objects to scratchpad for energy reduction. In *Proceedings of the conference on Design, automation and test in Europe*. IEEE Computer Society, 409.

T. M. CHILIMBI, B. D. AND LARUS, J. 1999. Cache-conscious structure definition. In *ACM SIGPLAN Notices*. ACM, 13–24.

TANENBAUM, A. S. 1998. *Structured Computer Organization (4th Edition)*. Prentice Hall.

TIWARI, V. AND LEE, M. T.-C. 1998. Power Analysis of a 32-bit embedded microcontroller. *VLSI Design Journal 7,* 3.

UDAYAKUMARAN, S. AND BARUA, R. 2003. 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)*. ACM Press, 276–286.

UDAYAKUMARAN, S. AND BARUA, R. 2006. An integrated scratch-pad allocator for affine and non-affine code. In *Design, Automation and test in Europe (DATE)*. IEEE Computer Society.

UDAYAKUMARAN, S., NARAHARI, B., AND SIMHA, R. 2002. Application specific memory partitioning for low power. In *Proceedings of ACM COLP 2002 (Compiler and Operating Systems for Low Power*. ACM Press.

VERMA, M., WEHMEYER, L., AND MARWEDEL, P. 2004a. Cache-aware scratchpad allocation algorithm. In *Proceedings of the conference on Design, Automation and Test in Europe*. IEEE Computer Society.

VERMA, M., WEHMEYER, L., AND MARWEDEL, P. 2004b. Dynamic overlay of scratchpad memory for energy minimization. In *International conference on Hardware/Software Codesign and System Synthesis(CODES+ISSS)* (Stockholm, Sweden). ACM.

WEHMEYER, L., HELMIG, U., AND MARWEDEL, P. 2004. Compiler-optimized usage of partitioned memories. In *Proceedings of the 3rd Workshop on Memory Performance Issues (WMPI2004)*.

WEHMEYER, L. AND MARWEDEL, P. 2004. Influence of onchip scratchpad memories on wcet prediction. In *Proceedings of the 4th International Workshop on Worst-Case Execution Time (WCET) Analysis*.

WILTON, S. AND JOUPPI, N. 1996. Cacti: An enhanced cache access and cycle time model. In *IEEE Journal of Solid-State Circuits*.