Compiler-Decided Dynamic Memory Allocation for Scratch-Pad Based Embedded Systems

Sumesh Udayakumaran
skumaran@glue.umd.edu

Rajeev Barua
barua@eng.umd.edu

Department of Electrical & Computer Engineering
University of Maryland
College Park, MD 20742, U.S.A

ABSTRACT

This paper presents 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 [4].

Existing scratch-pad allocation 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. For example, our previous work in [3] derives a provably optimal static allocation for global and stack variables and achieves a speedup over all earlier methods. 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.

In this paper we present a dynamic allocation method for global and stack data that for the first time, (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 SRAM 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 our results show runtime reductions ranging from 11% to 38%, averaging 31.2%, using no additional hardware support. With hardware support for pseudo-DMA and full DMA, which is already provided in some commercial systems, the runtime reductions increase to 33.4% and 34.2% respectively.

CASES'03, Oct. 30–Nov. 2, 2003, San Jose, California, USA.

Categories and Subject Descriptors

C.3 [**Special-Purpose And Application-Based Systems**]: Real-time and embedded systems

General Terms

Performance

Keywords

Memory Allocation, Scratch-Pad, Compiler, Embedded Systems

1. INTRODUCTION

In both desktops and embedded systems many different kinds of memory are available, such as SRAM, DRAM, ROM, EPROM, Flash, non-volatile RAM and so on. Among the write-able memories – SRAM and DRAM – SRAM is fast but expensive while DRAM is slower (often by a factor of 10 or more) but less expensive (by a factor of 20 or more). To combine their advantages, the usual approach is to use a large amount of DRAM to build capacity at low expense and then to speed up the program add a small amount of SRAM to store frequently used data. Using SRAM is critical to performance; for example [3] shows that adding a small SRAM results in an average of 2.5x lowering of runtime in a typical embedded configuration as compared with using DRAM only. This gain from SRAM is likely to increase in the future since the speed of SRAM is increasing by 60% a year versus only 7% a year for DRAM [12].

In desktops, the usual approach to adding SRAM is to configure it as a hardware cache. The caching mechanism stores a subset of the frequently used memory in the cache. Caches have been a big success for desktops, a trend that is likely to continue in the foreseeable future. The other alternative of using the SRAM as a scratch-pad under software control, is not a serious competitor.

For embedded systems however, the overhead of caches comes with a more serious price. Caches incur a significant penalty in aspects like area cost, energy, hit latency and real-time guarantees. All these criteria other than hit latency are more important for embedded systems than desktops. A detailed recent study [4] compares the tradeoffs of a cache as compared to a scratch pad. Their results are startling: a scratch pad memory has 34% smaller area and 40% lower power consumption than a cache memory 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 [4]. Even more surprisingly, the runtime they measured in cycles was 18%

better with a scratch pad using a simple static knapsack-based [4] allocation algorithm, compared to a cache. Thus, defying conventional wisdom, they found absolutely no advantage to using a cache, even in high-end embedded systems in which performance is important. With the superior dynamic allocation schemes proposed here, the runtime improvement will be significantly larger. Given the power, cost, performance andreal time advantages of scratch-pad, and no advantages of cache, we expect that systems without caches will continue to dominate embedded systems in the future.

Although many scratch-pad based embedded processors exist such as [21, 26, 20], utilizing them 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 fast memory. The contents of cache constantly change during runtime to reflect the changing working set of data across time. Unfortunately, both existing approaches for scratch-pad allocation – program annotations and the recent compiler-driven approaches [3, 2, 24] – are static data allocations. In other words, they are incapable of changing the contents of scratch-pad at runtime. This problem is a serious limitation for existing approaches. As an 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 SRAM, any static allocation suffers DRAM accesses in two out of three arrays. In contrast, a dynamic strategy can fit all three arrays in SRAM at different times. Although this example is oversimplified, it intuitively illustrates the benefits of dynamic allocation.

Attempts so far to capture dynamic behavior in scratch-pad based systems have focused on algorithms for *software caching* [19, 11]. This class of methods emulate the behavior of a hardware cache in software. In particular, a tag consisting of the high-order bits of the address is stored along with each cache line. Before each load/store, additional instructions are inserted by the compiler to mask out the high-order bits of the address, access the tag, compare the tag with the high-order bits and then branch conditionally to hit or miss code. Some methods are able to reduce the number of such inserted overhead instructions [19], but much of it remains, especially for non-scientific programs. Needless to say, the inserted code adds significant overhead, including (i) additional runtime(ii) higher code size, increasing dollar cost (iii) higher data size from tags, also increasing cost (iv) higher power consumption and (v) memory latency that is just as unpredictable as hardware caches.

In this paper we present a compiler algorithm for managing scratch-pad based systems that for the first time is able to change the allocation at runtime and avoid the overheads of software caching. In particular, it (i) accounts for changing program requirements at runtime (ii) has no software-caching tags (iii) requires no run-time checks per load/store (iv) has extremely low overheads and (v) yields 100% predictable memory access times. The outline of the method is as follows. The compiler analyzes the program to identify program points where it may be beneficial to insert code to bring in a variable, or parts of a variable, from DRAM into SRAM. It is beneficial to copy a variable into SRAM if it is repeatedly accessed thereafter and the benefit of it being in SRAM outweighs the cost of transfer. A profile-driven cost model is presented to estimate these benefits and costs. Since the compiler must ensure that at all times all the data allocated to SRAM fit in SRAM, occasionally variables must be evicted when new ones are brought in. Which variables to evict and when to evict them is also decided by the compiler. In other words, just like in a cache, data is moved back and forth between DRAM and SRAM, but under compiler control, and with hardly any additional overhead.

The above compiler algorithm for global and stack data has several innovative features which include the following. (i) To reason about the contents of SRAM across time, it is helpful to associate a concept of time with particular program points. To this end, we introduce a new data structure called the *Data Program Relationship Graph (DPRG)* which associates a *timestamp* for several key program points of interest. As far as we know, this is the first time that a data structure to represent time during program execution has been defined. (ii) A cost model determines the runtime cost of possible transfers at each program point. (iii) A greedy compile-time heuristic at each point determines, using the cost model, which transfers should be selected to maximize the overall runtime benefit. (iv) Compile-time optimizations are done to reduce the cost of data transfer. For example, if dataflow analysis reveals that a variable is not live at the point it should be transferred back to DRAM, the transfer can be deleted. We observe three desirable features of our algorithm. (a) No additional transfers beyond those required by a caching strategy are done. (b) Data that is accessed only once is not brought into SRAM, 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. (c) Data known to the compiler to be dead is not written out to DRAM upon eviction, unlike in a cache, where the caching mechanism writes out all evicted data.

A decrease in energy consumption is another likely benefit from our algorithm. Energy consumption is an important criteria for embedded systems, especially for portable devices. We have not measured the impact of our algorithm on energy consumption. However, energy consumption is known to be roughly proportional to runtime when the architecture is unchanged [16]. Since our runtime improves by 31.2% vs an optimal static allocation, it is likely that the energy consumption also decreases by a similar fraction, but we have not measured the exact amount.

Our method does not allocate heap data in the program to SRAM. Programs with heap data still work, however – all heap data is allocated to DRAM and the global and stack data can still use the SRAM using our method, but no SRAM acceleration is obtained for heaps. Heap data is difficult to allocate to SRAM 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 can be used for heap, but they have significant overheads. Another possibility for speeding up heap accesses is to use an embedded processor with both a scratch pad and a cache, such as the ARMv6 [8], and to allocate the heap data to the cache, thereby making heap accesses faster. Since the cache stores only heap data, it can be smaller than without the scratch-pad, and thus the disadvantages of the cache are smaller too. Further, tasks not accessing the heap can still benefit from the better real-time guarantees of the scratch-pad. We do not consider heap data any further in this paper.

The rest of paper is organized as follows. Section 2 overviews related work. Section 3 overviews the method for global and stack data, and proposes a new DPRG data structure. Section 4 describes the precise method used to determine the memory transfers at each program point. Section 5 describes some details needed to make the algorithm work correctly in all cases. Section 6 presents evaluation results. Section 7 concludes.

2. RELATED WORK

Methods for static allocation of data among different non-cached memory banks include our earlier work in [3, 2], the work by

Sjodin et. al [23] and the one by Sjodin and Von Platen [24]. Our earlier method in [3, 2] 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 matlab [17] or CPlex [14]. Sjodin et. al's method in [23] uses a greedy heuristic, while their method in [24] uses an ILP formulation; both papers handle global variables only, and not stack variables. We demonstrate that our method for stacks in [3, 2] yields a 44% improvement over not using it, demonstrating its value. Unfortunately static methods cannot adapt their allocations to changing demands at runtime which is a serious drawback compared to caching strategies.

Other dynamic memory allocation strategies proposed so far have been mostly restricted to Software Caching [19, 11]. Software caching methods emulate a cache in fast memory using software. The tag, data and valid bits are all managed by compiler-inserted software at each program data access. Software overhead is incurred to manage these fields, though compiler optimizes away the overhead in some cases [19]. In [19] they target the primary cache, while [11] is intended for managing the secondary cache in desktop systems. All software caching techniques suffer from significant overheads in runtime, code size, data size, energy consumption, and result in unpredictable runtimes. Our dynamic allocation technique overcomes all of these disadvantages.

Our work is in some aspects also analogous to the offline paging problem first formulated by [5]. 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. But there are some key differences between the two ideas. The algorithm by Belady essentially chooses the page occuring furthest in the dynamic trace. In contrast our approach associates memory allocation decisions with static points in the program. This ensures better predictability. Also, our memory allocation decision is based on a cost model which takes into account both reuse and distance of occurence of the variable. Such an approach is likely to avoid unnecessary transfers between the memory banks.

Some software caching schemes use *dynamic compilation* [13]. The improvements in [13] are small, but more importantly, in dynamic compilation the program is in RAM and is changed at runtime. In most embedded systems however, since the program is in fixed-size and unchangeable ROM, dynamic compilation schemes cannot be used. There are other software caching schemes that have been proposed with different goals and/or non-applicable platforms [25, 6, 9, 22, 7, 15]. None of these apply to our objectives; for lack of space, we do not discuss these further.

The only work that takes a strategy similar to ours is [18]. It moves data back and forth between DRAM and scratch pad like we do under compiler control. To understand their method, it is easiest to see how it compares to ours. We identify two advantages of our approach and one disadvantage. These are discussed in the next three paragraphs.

The first advantage of our method over [18] is that it is general and applies to all global and stack variables, and all access patterns to those variables. In contrast, their method [18] applies to global and stack *array* variables only with the following three additional restrictions. (i) The programs should primarily access arrays through affine (linear) functions of enclosing loop induction variables. (ii) The loops must be well-structured and must not have any other control flow such as **if-else**, **break** and **continue** statements. (iii) The codes must contain these constructs in a clean way without hand-optimizations often found in many such codes, such as common-subexpression eliminations and array accesses through pointer indirections; since with these features the needed affine analysis cannot succeed. Combining these three restrictions, their method applies to well-structured scientific and multimedia codes. Unfortunately, most programs in embedded systems including many of those in the control, automotive, network, communication and even DSP domains do not fit within these strict restrictions. We have observed that even many regular array based codes in embedded systems violate the above restrictions, especially (ii) and (iii). In contrast, our method is completely general and is able to exploit locality for all codes, including those with irregular accesses patterns, variables other than arrays, code with pointers and irregular control flow.

A second advantage of our scheme over [18] is that our method is a whole-program analysis across all control structures while their method considers each loop nest independently. This has several consequences. One is that their method is locally optimized for each loop, while our method is globally optimized for the entire program. Another is that they make available the entire scratch pad for each loop nest. Our method might choose to do this, but is not constrained to do so. It may choose to use part of the scratch pad for data that is shared between successive control constructs thus saving on transfer time to DRAM.

The method in [18] does have one advantage over ours, though. For regular affine-function codes matching their restrictions [18], their method, unlike ours, allows 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 plan to investigate such a strategy in future work that is able to transfer parts of an array into SRAM 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.

3. METHOD OVERVIEW AND DATA STRUCTURES USED

Our method for implementing dynamic memory allocation of global and stack variables takes the following approach. The compiler-driven method inserts code into the application to copy variables from slow memory into SRAM whenever it expects them to be used frequently thereafter, as predicted by previously collected profile data. Variables placed in SRAM earlier may be evicted by copying them back to slow memory to make room for new variables. Like in caching, the data is retained in DRAM at all times even when the latest copy is in SRAM. Unlike software caching, since the compiler knows exactly where each variable is at each point in the program, no runtime checks are necessary to find the location of any given variable. Consequently the overheads and unpredictable latencies of software caching are avoided.

Let us consider how the compiler may determine the optimal allocation solution for globals and stacks. Each solution is defined by the set of program points where transfers can take place and what variables to transfer at these points. Computing the theoretically optimal solution can be done in an exponential amount of time and like a host of other compiler problems, is almost certainly NP-complete, though we have not attempted to formally prove this. It can certainly be solved in an exponential amount of time by evaluating the cost of all possible solutions and choosing the solution with minimum cost. To see how, let the number of instructions, and therefore the number of program points, be n. Let the total number of global and stack variables be m. Then it can be shown that the

number of solutions is $O(2^{2^{mn}})$, a large exponential. The proof for this is not shown for lack of space, but it is not difficult.

In the absence of an optimal solution, a heuristic must be used. Our cost-model driven greedy algorithm has the following four steps. First, it partitions the program code into *regions* where the start of each region is a *program point*. Regions correspond to the granularity at which allocation decisions are made, *i.e.,* the allocation stays the same within a region. Changes in allocation are made only at program points by compiler-inserted code that copies data back and forth between SRAM and DRAM. 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 total order among themselves; and (ii) the program points are reached during runtime 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 almost all programs of interest. Third, the greedy method steps through the program points in timestamp order and at each point by using a detailed cost model determines the set of variables in slow memory to bring into SRAM and the set in fast memory to evict. The cost model estimates the benefit or loss from every candidate variable to bring in or evict. Then among candidate re-allocations with positive estimated benefit, the re-allocation with greatest benefit is chosen. Since each program point might have multiple timestamps, it might be assigned multiple re-allocations corresponding to each timestamp. The result is a program that implements dynamic memory allocation.

3.1 Deriving regions and timestamps

The choice of program points and therefore regions, is critical to the success of the algorithm. Regions are defined as the code between successive program points. We observe that promising program points are those after which the program has a significant change in locality behavior. Further, the dynamic frequency of program points should be less than the frequency of regions, so that the cost of copying data into SRAM can be recouped by data re-use from SRAM in the following region. Thus, sites just before the start of loops are especially promising program points since they are infrequently executed compared to the insides of loops. Moreover, the subsequent loop often re-uses data, justifying the cost of data copying into SRAM. With these considerations, we define program points for our base algorithm as *(i) the start of each procedure; and (ii) just before the start of every loop* (even inner loops of nested loops). Program points are merely candidate sites for copying to and from SRAM – whether any copying code is actually inserted at those points is determined by a cost-model driven approach, described in section 4.

Figure 1 shows an example illustrating how a program is divided into regions and then marked with timestamps. Figure 1(a) shows the outline of an example program. It consists of four procedures, namely *main(), proc-A(),proc-B()* and *proc-C()*, two loops that we name *Loop 1* and *Loop2*, and accesses to two variables *X* and *Y*. The program may contain other instructions that are not shown. Only loops, procedure declarations and procedure calls are 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 the identification of regions and marking of timestamps. The DPRG is essentially the program's call graph appended with new nodes for loops and variables. In the DPRG shown in figure 1(b), there are four procedures, two loops and two variables represented by nodes. We see that oval nodes represent procedures, circular nodes represent loops and square nodes represent variables. Edges to procedure nodes represent calls; edges to loop nodes shows that the loop is nested in its parent; and edges to variable nodes represent memory accesses to that variable in its parent. The DPRG is usually a directed acyclic graph (DAG), except for recursive programs, where cycles occur. The *program points* – namely the starts of procedures and loops – are represented by the start of the code in each oval or circular node. In case of a loop, its program point is outside the loop at its start. In case of a procedure, its program point is inside its body at its start.

Figure 1(b) also shows the one or more timestamps for each node in the DPRG. Since the start of each node is a program point, this timestamps the program program points as well. Timestamps are derived using the following rule: the timestamp for each node is the timestamp of its parent appended by a '.' followed by a number representing which child it is in a left to right order. In this way if the *main()* function is assigned a timestamp of 1, the timestamps of all nodes can be computed by a simple variant of the well-known breadth-first-search graph traversal method. The figure shows the results. A node may get more than one timestamp if it has more than one parent – an example of this is the node for *proc-c()*, which is marked with two timestamps: 1.1.1 and 1.2.1. An ordering on timestamps is their dictionary order. In other words, timestamps are compared according to the following rule: find their longest common prefix ending by a '.'; the larger timestamp is the one with the larger subsequent number. For example, $1.2.1 < 1.3$ since their longest common prefix ending with a '.' is "1.", and the subsequent number (2) for the first timestamp is less than that of the second timestamp (3). With such an ordering, the timestamps always form a total order among themselves.

Timestamps are useful because they reveal dynamic program execution order: the order in which the program points are visited at runtime is roughly the same as the total order of their timestamps. The only exception is when a loop node has multiple timestamps as descendants. In such case the descendants are visited cyclically in every loop iteration; thus earlier timestamps are repeated when returning to the beginning of the loop in subsequent iterations, violating the timestamp order. Even then, we can probabilistically 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 – derived at compile-time – to reason about dynamic execution order in the compiler across the whole program. This is a useful property and enables us to derive a good dynamic allocation at compile-time.

Timestamps have their limitations however, in that they cannot be derived for programs with unstructured control flow, or for recursive programs. Fortunately, unstructured control flow is exceedingly rare nowadays in any domain – we only refer to arbitrary **goto** statements here; other constructs such as break and continue statements within loops, and switch statements, are okay for timestamps. Recursive programs are also rare in embedded programs, but we do consider simple extensions of our method for recursive programs later in section 5.

4. ALGORITHM FOR DETERMINING MEMORY TRANSFERS

This section describes the proposed algorithm for determining the memory transfers at each program point. Before running this algorithm, the DPRG is built to identify program points and mark the timestamps.

An overview of the memory transfer algorithm is as follows. At

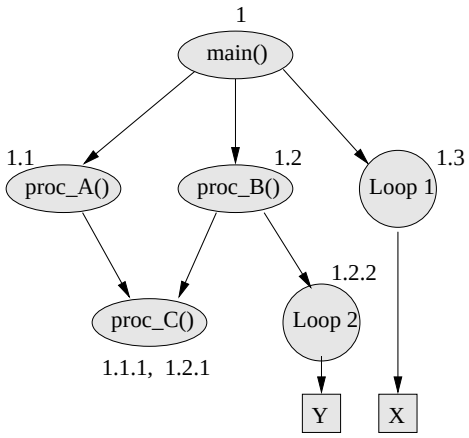
```
main () {
  proc-A()
  proc-B()
  while (...){X =...} /* Loop 1 */
}

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

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

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

(a)



(b)

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

each program point, the algorithm determines the following memory transfers: (i) the set of variables to copy from slow memory into fast memory; and (ii) the set of variables to evict from fast memory to slow memory to make way for incoming variables. Further, it inserts code at the program points to implement the transfers. The algorithm is structured as a one-pass algorithm that iterates over all the program points in timestamp order. At each program point, variables currently in slow memory are considered for bringing into fast memory in decreasing order of frequency of access per-byte. 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. The algorithm next generates a candidate best-live-variable-set to evict if such eviction is necessary to create space to bring in this variable. This set is chosen by the *promise-value heuristic*: variables are chosen to evict if they are accessed in the furthest region in the future in timestamped order, and among variables with the same next region of access, variables with lower frequencies-per-byte of access in that region are preferred for eviction. Finally, given the variable to bring in and the candidate best-live-variable-set to evict for that variable, we use a detailed cost model to determine if this proposed swap should actually take place. In particular, copying a variable into fast memory may not be worthwhile unless the cost of the copying and the lost locality of evicted variables is overcome by its subsequent reuse from fast memory of the brought-in variable. The cost model we use models all these three components of cost to derive if the swap should occur. Details of the cost model are presented later in this section. Completing this process for all variables at all timestamps yields the complete set of all memory transfers.

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 fast and slow memory, respectively, at the current program point. Bring-in-set and Swap-out-set store their obvious meaning at each program point. Promise-value is an array of structures, one array element for each program variable. For each program variable, the structure has a field to store the timestamp of the next region that accesses that variable; and another field to store the access frequency per byte of that variable in that next region.

The algorithm is as follows. Line 1 computes the Promise-values at the first timestamp. Line 2 defines a sorted order on the Promise-value structures: the intuition is that variables that are most promising to bring into fast memory are brought to the front in the sorted order. The ordering gives a higher preference to variables accessed earlier (by comparing next-timestamp first); else if accessed at the same time, the access frequency is used to determine promise. Note, however, that Promise-values are used only as a heuristic to prune the search-space; eventually transfer decisions are made on the basis of a detailed cost model, not on Promise-value.

Continuing further, Line 4 is the main **for** loop that steps through all the program points in timestamp order. At each program point, line 6 steps through all the variables, giving preference to frequently accessed variables in the next region. For each variable in slow memory (line 7), it tries to see if it is worthwhile to bring it into fast memory (lines 8-12). Line 8 calls **Find-swapout-set()** which returns the set of variables most suitable for eviction (swapping out) to create space if V is to be brought in. If it is beneficial in runtime to bring V into fast memory (line 9), then Bring-in-set and Swapout-set are updated accordingly (line 10-11). After looping through all the variables, lines 15, 16, and 17 update, for the next program point, the set of variables in fast memory, the set in slow memory and the promise values, respectively.

Next we explain **Find-swapout-set()** (lines 20-35) called in line 8. It returns the best set of variables to copy out to slow memory when its argument V is brought in. Line 22 checks to see if space is available in fast memory for V. Space may be available from (a) the set of variables in fast memory that are no longer alive (the Dead-set) found by liveness analysis as in [1]. These need not be copied out to slow memory – a worthwhile optimization, implemented by not including the Dead-set in Swapout-set; (b) variables evicted so far from earlier calls to **Find-swapout-set()** for the same program point. Combining both these sources, if space is available, a detailed cost model in **Find-benefit()** is invoked in line 23 to estimate if bringing in V is worthwhile. Even if space is available, bringing in V may not be worthwhile unless the cost of the copying is over-

```
Define                  /* The values of all of the quantities defined below change at each program point */
    Set V-slow                                          /* Set of variables in slow memory at this point */
    Set V-fast                                          /* Set of variables in fast memory at this point */
    Set Bring-in-set                                    /* Variables to bring into fast memory at this program point */
    Set Swapout-set                                     /* Set of variables for eviction to slow memory */
    Set Dead-set                                        /* Set of variables in V-fast whose lifetimes have ended */
    int Promise-value[variable].next-timestamp          /* Timestamp at next access of variable */
    float Promise-value[variable].f-per-byte            /* Access frequency per byte of variable in next region */
    int Maximum-benefit                                 /* Computed in Find-swapout-set() as benefit of swapping */

void    Memory-allocator()
1.  For each variable, assign structure Promise-value[variable] with next-timestamp (from DPRG) and
        f-per-byte (from profile data) for first region in code.
2.  Sort the variables in descending order of their Promise-value structures; ordering defined as:
    Define Promise-value[var1] ≥ Promise-value[var2] iff
        (Promise-value[var1].next-timestamp > Promise-value[var2].next-timestamp) or
            ((Promise-value[var1].next-timestamp == Promise-value[var2].next-timestamp)
                and (Promise-value[var1].f-per-byte ≥ Promise-value[var2].f-per-byte))
3.  Assign initial V-fast and V-slow by filling fast memory in greedy order from sorted Promise-value list.
4.  for all timestamped program points in the application program, visited in timestamp order
5.      Swapout-set ← NULL_SET;   Bring-in-set ← NULL_SET;   Dead-set ← {V ∈ V-fast | V is dead}
6.      for all variables V accessed in the next region in decreasing order of f-per-byte
7.          if V ∈ V-slow
8.              Best-swapout-set ← Find-swapout-set(V)
9.              if Maximum-benefit > 0                  /* Use Best-swapout-set only if its benefit is positive */
10.                 Swapout-set ← Swapout-set ⋃ Best-swapout-set
11.                 Bring-in-set ← Bring-in-set ⋃ {V}
12.             endif
13.         endif
14.     endfor
15.     V-fast ← V-fast ⋃ Bring-in-set − Swapout-set − Dead-set
16.     V-slow ← V-slow ⋃ Swapout-set − Bring-in-set − Dead-set
17.     Repeat lines 1 and 2 to re-assign and sort Promise-value structures for next program point.
18. endfor      /* For all program points */
19. return

Set     Find-swapout-set(V)
20. int Maximum-benefit ← 0
21. Best-swapout-set ← NULL_SET
22. if (V fits in unallocated fast memory after removing Dead-set and swapped out variables so far)
23.     Maximum-benefit ← Find-benefit(V,NULL_SET)          /* V fits; no need to swap out variables */
24. else            /* Not enough space for V; need to swap out variables */
25.     Candidate-swapout-set ← Select variables in fast memory with minimum promise values such that
            size(Candidate-swapout-set) ≥ 2 ⋆ (size(V) - size(unallocated fast memory))
26.     for all sets S ⊆ Candidate-swapout-set                  /* All subsets of Candidate-swapout-set */
27.         if size(S) < (size(V) - size(unallocated fast memory)) continue endif /* Go to next for iteration */
28.         Benefit ← Find-benefit(V,S)
29.         if Benefit > Maximum-benefit
30.             Maximum-benefit ← Benefit
31.             Best-swapout-set ← S
32.         endif
33.     endfor
34. endif                                                   /* if Dead set not big enough */
35. return Best-swapout-set

int     Find-benefit(V,Swapout-set)
36. Latency-gain ← Promise-value(V).f-per-byte ⋆ size(V) ⋆ (Latency_slow_mem - Latency_fast_mem)
37. Latency-loss ← 0
38. for all (V_i ∈ Swapout-set)
39.     Latency-loss += (Promise-value[V_i].f-per-byte ⋆ size(V_i))* (Latency_slow_mem - Latency_fast_mem)
40. endfor
41. Migration-overhead ← Time for copying all non-read-only variables in Swapout-set to slow memory +
                         Time for copying V to fast memory
42. Benefit ← latency-gain − latency-loss − Migration-overhead
43. return Benefit
```

Figure 2: Algorithm for determining dynamic memory allocation

come by the subsequent reuse of V from fast memory. If space is not available, line 25 narrows down a candidate set for swapping out by using a promise-value-based heuristic. The **for** loop in line 26 exhaustively evaluates all subsets S of this small candidate set using the accurate cost model in **Find-benefit()** (line 28). Among these the best is found in Best-swapout-set (line 31), and returned (line 35).

Cost model Finally, we look at **Find-benefit()** (lines 36-43), the heart of the cost model, called in lines 23 & 28. It computes whether it is worthwhile, with respect to runtime, to copy in its argument V and copy out is argument Swapout-set. The net benefit of this operation is computed in line 42 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 fast memory in the next region, and is computed in line 36. Second, the latency loss is the loss from not having Swapout-set in fast memory in the next region; the loss from each variable in Swapout set is accumulated in lines 37-40. Third, the migration overhead is the cost of copying itself, estimated in line 41. An optimization done here is that variables that are read-only in the last region (checked in line 41) need not be written out to DRAM since they have not been modified from their DRAM copies. This optimization provides the compile-time equivalent of the functionality to the dirty bit in cache. The end result is an accurate cost model that estimates the benefit of any candidate allocation that the algorithm generates.

5. ALGORITHM MODIFICATIONS

For simplicity of explanation, the algorithm in section 4 leaves five issues unaddressed, solutions to which are presented in this section.

Offsets in SRAM The first issue in this section is deciding where in SRAM to place the variables being swapped in. The criteria for good variable placement is twofold. First, the placement process should minimize the fragmentation that might result when variables are swapped out. Second, when a memory hole of a required size cannot be found, the cost of compaction should be considered in the cost model. We implement a solution to variable placement as a separate pass (not shown) at the end of the memory allocator. Similar to the memory allocator this pass visits the nodes in the timestamped order. To guide our allocation in the first step, we use a simple heuristic based on the lifetimes of the variables. If possible, the variable being swapped in is placed contiguously with variables which are likely to be swapped out at the same future timestamp, reducing the likelihood of small, useless holes. In the event that memory holes of adequate size are not available, the next step considers if the memory can be compacted. Compaction is done only if the cost of compaction of a selected portion is amortized by the benefit gained by retaining the variable in fast memory.

Procedure join nodes A second complication with the above algorithm is that for any program point 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 which has two timestamps, 1.1.1 and 1.2.1. In general, the number of timestamps is the number of paths from *main()* to that node. We call nodes with multiple timestamps *join* nodes since they join multiple paths from *main()*. 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. For the join node itself, however, multiple recommended allocations result, one from each path to it, presenting a problem. One solution is duplicating the join node along each path to it, but the resulting code growth is unacceptable for embedded systems. Instead, we use a strategy that adopts different allocation strategies for different paths but with the same code. The procedure calls are modified to pass a unique path identifier to the child node. This is used to select the allocation strategy specific to the path. The extra parameter only occurs for join nodes and nodes downstream to them in the DPRG.

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 fast and slow 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. Unlike for procedure join nodes, our current implementation makes the decision not to maintain these multiple allocations further, but to arrive at a single consensus allocation at the join point. The consensus allocation is chosen assuming the incoming allocation from the most probable predecessor, and modifying it with memory transfers for the join node. Subsequently compensation code is added at the termination of all the less probable paths to ensure that their allocation is modified to be the same as the newly computed allocation at the join node.

Conditional paths are timestamped as follows. Different conditional paths from the same condition can be visited by our memory transfer algorithm in any order, since they are independent of each other, and the result for each path depends only on the shared predecessor's allocation. Thus our method timestamps sibling conditional paths in any arbitrary order, such as the program order.

Our different handling of procedure join nodes and conditional join nodes illustrates the two design choices for join nodes. In procedure join nodes we maintain the different incoming allocations as different allocations in the join nodes themselves and in their successors. In conditional join nodes the different incoming allocations are reconciled to a single consensus allocation at the join node. Both approaches have their advantages: maintaining different allocations in the join nodes and successors retains the best memory performance since all paths get their optimal allocation. On the other hand, having a single consensus allocation incurs less runtime overhead since, unlike with multiple allocations, no path tags or conditional memory transfers based on tags are needed. We have not done a careful quantitative study of the tradeoffs involved in choosing among the two allocation strategies – we leave such a study to future work. Our current design and implementation chooses multiple allocations for procedure join nodes and a single allocation at conditional join nodes. Our (admittedly sketchy) intuition for this choice is that that conditional paths tend to be shorter than procedure calls, so the likely deviation in allocations will be small among different paths, so a single allocation will not be too costly in memory performance. Future work will study this tradeoff further.

Fixed point iteration for loops A third modification is needed for loops containing multiple regions within them. 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 correctness condi-

```
main () {
  proc-A(X)
  while (...) {
    /* Swap in Y, Swap out X */
    proc-A(Y)
    /* Swap in Z, Swap out Y */
    proc-A(Z)
  }
}

proc-A (X) {
  while (...)
  while (...)
    X = ...
}
```

(a)

```
main () {
  proc-A(X)
  /* Swap in Z, Swap out X */
  while (...) {
    /* Swap in Y, Swap out Z */
    proc-A(Y)
    /* Swap in Z, Swap out Y */
    proc-A(Z)
  }
}

proc-A (X) {
  while (...)
    while (...)
      X = ...
}
```

(b)

Figure 3: Example showing (a) an incorrect allocation; and (b) Corrected allocation using fixed point convergence

tion that there be only one allocation at each program point. This problem is illustrated by the memory allocation in the example in figure 3(a). The allocation is incorrect since from the second iteration onwards, the allocation decision "Swap out X and Swap in Y" is not correct as X is no longer in fast memory. Instead Z is in fast memory. Although a solution similar to procedure nodes can be adopted, we instead adopt a fixed point iterative approach. The motivation for doing this is that in most loops, the back edge is far more frequent than the forward edge. Procedure **Find-swapout-set** is iterated several times over all the nodes inside the loop until the allocation converges. 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. In the example, the correct allocation is shown in figure 3(b) where the modified algorithm swaps out X and swaps in Z before the loop header.

Recursive functions The method in section 4 does not apply to stack variables in recursive or cross-recursive procedures. With such procedures the call graph and DPRG is cyclic and hence the total size of stack data is unknown. Hence for a compiler to guarantee that any variable in a recursive procedure fits in SRAM is difficult. Our technique is to place stack data in all recursive call-graph cycles in DRAM. DRAM placement 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 recursive procedures are present, the globals and stack data from all non-recursive procedures in the same program can still be placed in SRAM by our method.

5.1 Reducing the runtime and code size of data transfer code

Our method needs to copy data back and forth between SRAM and DRAM. This overhead is not unique to our approach – hardware caches also need to move data between SRAM and DRAM. The simplest way to copy is a memory to memory move for scalars, a **for** loop for arrays, and a nested loop for multi-dimensional arrays. We speed up this transfer in the following four ways. First, multi-dimensional arrays are copied by a single **for** loop instead of a nested loop since they are stored contiguously. Second, the **for** loops are unrolled by a small, fixed factor to reduce the runtime overhead from end-of-loop compare and branch instructions. Third, the code size increase from **for** loops inserted in the code is almost eliminated by placing the **for** loop in a special *memory-block copy* procedure that can be reused for each array transfer. Fourth, faster copying of arrays is possible in embedded processors that provide the low-cost hardware mechanisms of Direct Memory Access (DMA) (*e.g.,* [8, 10]) or pseudo-DMA(*e.g.,* [21]). DMA accelerates data transfers within memory banks and from memory to I/O devices. Pseudo-DMA accelerates transfers from memory to CPU registers, and thus can be used to speed memory-to-memory copies via registers. Section 6 evaluates the runtime improvements from using DMA and pseudo-DMA instead of software transfers.

6. RESULTS

This section presents preliminary results by comparing our method against the provably optimal static allocation described in [3]. We have implemented the front-end allocation portion of our algorithm in the a public-domain GCC cross-compiler targeting the Motorola M-Core [21] embedded processor. The back-end code-transformation portion of the algorithm (insertion of code to copy the data between SRAM and DRAM) is not yet complete, so for now, the allocations derived automatically by our front-end are manually inserted into the code at the source level. Since the resulting executable code is the same as to what will be produced by automating the back-end, manual coding causes no error. The executables are simulated on a public-domain cycle-accurate simulator for the Motorola M-Core. The provably optimal static method in [3] has been fully implemented in the same GCC compiler for M-Core.

The memory characteristics and benchmarks are as follows. The M-Core chip simulated has a 256 Kbyte external DRAM with 10-cycle read/write latency, and an internal SRAM with 1-cycle read/write latency. The size of the SRAM is varied to be 25% of the size of the benchmark data. The names and characteristics of the benchmarks evaluated are shown in table 1. The benchmarks selected only use global and stack data, rather than heap data. Heap data is allocated to the DRAM in the current compiler.

Experiments are conducted with our pure software method, as well as with hardware acceleration from both DMA and pseudo-DMA [21]; their transfer times are estimated in the simulator as follows. First, DMA accelerates data transfers within memory banks and from memory to I/O devices. It is usually implemented by fetching multiple consecutive words from DRAM for every access, and transferring them on the memory bus one word at a time. For a 10-cycle DRAM latency and an assumed 4-word wide DRAM, this results in 4 words transferred in the first four cycles of every

Benchmark	Source	Description	Total Data Size (in bytes)
MXM	Spec92	Matrix multiply	280024
HISTOGRAM	UTDSP	Image enhancing application	1056816
EDGE DETECT	UTDSP	Edge detection in an image	3145856
BMCM	Perfect club	Molecular dynamics of water	199536
D-FFT	UTDSP	Application with two FFT kernels	32816

Table 1: Benchmark programs and characteristics.

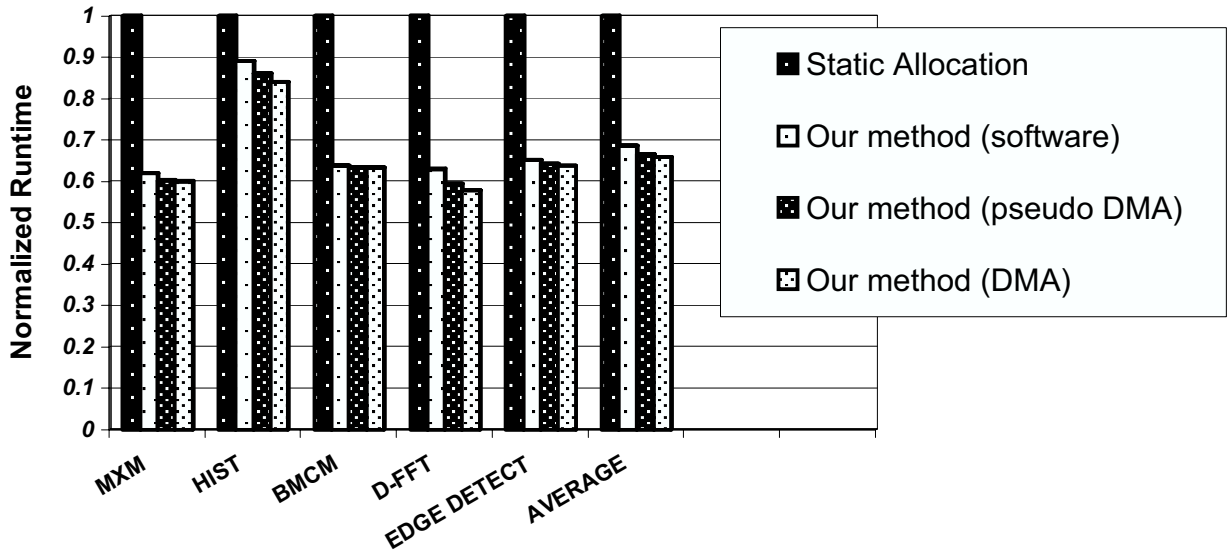


Figure 4: Normalized simulated runtimes for each benchmark for four different allocation methods.

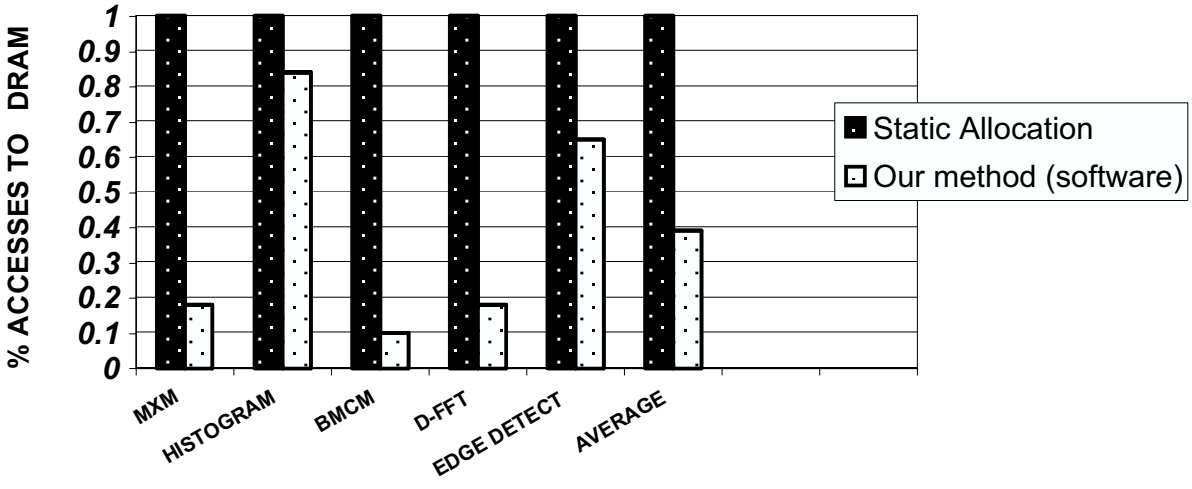


Figure 5: Percentage of memory accesses going to DRAM for each benchmark.

10 cycles, which implies that $(10/4) * N = 2.5N$ cycles are needed to transfer N words. Second, pseudo-DMA accelerates transfers from memory to CPU registers, and thus can be used to speed memory-to-memory copies via registers. Pseudo-DMA functions are reported to obtain speeds of up to 53.6 Mbyte/s for a M-CORE processor with a 50MHz speed [21]. This is nearly four times the speed that can be achieved using conventional single load/store instructions. We conservatively use a factor of two in our simulator estimates.

Figure 4 compares the runtimes for different allocation methods. On the X-axis are the benchmarks; for each benchmark, runtimes for four configurations are shown, which are, from left to right, (i) the provably optimal static allocation derived by [3]; (ii) our method implemented by software alone; (iii) our method accelerated by pseudo-DMA; and (iv) our method accelerated by DMA. The runtimes are normalized to 1.0 for the static allocation. Comparing the first bar with the second for each benchmark, we see that the benchmarks achieve runtimes gains ranging from 11% to 38%, averaging 31.2%, from using our method as compared to the optimal static allocation. The average gains increase to 33.4% and 34.2% with pseudo-DMA and DMA, respectively. The impressive speedups show that our method is able to use dynamic data movement to more fully exploit the potential of scratch-pad memory. Further, we believe that the current numbers under-estimate the runtime improvement in the following way. They are on small programs necessitated because of the hand-coding in the code generation step. For larger programs the improvement is likely to be larger, as they tend to have more phases of computation, and thus less likelihood that a static allocation will do well.

Figure 4 also reveals that not every program benefits equally from our method. In fact the gains range widely from 11% to 38%. This is not surprising since the gain depends on whether there are a significant number of regions in the program where data is re-used within that region or across the next few regions, before the data needs to be evicted because of the working set change. Without re-use, the cost of bringing in and swapping out the variable would not be recouped by reduction in latency in the region, and our method will correctly decide not to bring in data. No benefit without re-use is the case in caches too – a cache is also not beneficial without data re-use. It is important to note that for programs for which a low benefit is obtained, it is because of the program's intrinsic property of low data re-use, and not because of deficiencies in our method of exploiting re-use.

Figure 5 shows the reduction in percentage of memory accesses going to DRAM because of the improved locality to SRAM afforded by our method. The average reduction across benchmarks is a very significant 61% reduction in DRAM accesses. Note that the total number of memory accesses actually increases in our method because of the added transfer code, but the reduced number of accesses to DRAM more than compensates for this increase, delivering an overall reduction in runtime.

In our evaluation we use an SRAM of size equal to 25% of the size of the program data. An alternative would have been to use a fixed size SRAM for all programs. We believe a fixed SRAM size would be a poor choice to evaluate the benefits of our method since it would yield results that are non-intuitive and extremely data-dependent. In particular, programs with small data set sizes would yield small benefits since most data would fit in SRAM, but the very same program with a larger data set size would get larger benefits. To get fair numbers, we would need a "typical" data set size, but often there is no such typical size, and the results could be misleading. Our use of an SRAM size that varies with the data set size as a fixed fraction of it avoids such dependence with data set size and yields more meaningful results.

7. CONCLUSION

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 SRAM 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. When compared to a provably optimal static allocation our results show an average of 31.2% reduction in runtime using no additional hardware support. With hardware support for pseudo-DMA and full DMA, already provided in some commercial embedded systems, the average runtime reduction is 33.4% and 34.2% respectively.

8. REFERENCES

[1] Andrew W. Appel and Maia Ginsburg. *Modern Compiler Implementation in C*. Cambridge University Press, January 1998.

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

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

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

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

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

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

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

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

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

[11] G. Hallnor and S. K. Reinhardt. A fully associative software-managed cache design. In *Proc. of the 27th Int'l Symp. on Computer Architecture (ISCA)*, Vancouver, British Columbia, Canada, June 2000.

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

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

[14] ILOG Corporation. *The CPLEX optimization suite*. http://www.ilog.com/products/cplex/, 2001.

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

[16] T-C. Lee, V. Tiwari, S. Malik, and M.Fujita. Power Analysis and Minimization Techniques for Embedded DSP Software. *IEEE Transactions on VLSI Systems*, March 1997.

[17] *Matlab 6.1*. The Math Works, Inc., 2001. http://www.mathworks.com/products/matlab/.

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

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

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

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

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

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

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

[25] Osman S.Unsal, Rakshit Ashok, Israel Koren, C. Manik Krishna, and Csaba Andras Moritz. Cool-cache for hot multimedia. In *Proceedings of the International Symposium on Microarchitecture*, pages 274–283, 2001.

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