Energy-aware Allocation of Dynamic Variables in Partitioned Memory Architectures

Renato Levy
rlevy@i-a-i.com
Intelligent Automation, Inc.
Rockville, MD, USA

Bhagirath Narahari
narahari@gwu.edu

Rahul Simha
simha@gwu.edu

Department of Computer Science
George Washington University
Washington, DC, USA

Abstract

This paper addresses the problem of minimizing the energy consumption by the memory subsystem in an embedded system. While significant work in the literature has addressed compiler driven energy aware allocation of global static variables, no general solutions have been proposed for the corresponding problem of allocating run-time dynamic variables. This paper proposes an approach, and the corresponding framework, to compiler driven energy aware allocation of dynamic variables. The approach is driven by the use of execution profile information and the concept of heap segmentation. Our framework and solution have been integrated into a compiler backend tool. Experimental results with some benchmark applications indicate that our techniques result in up to 40% memory energy reduction and 20-25% overall system energy reduction.

1 Introduction

A major constraint in embedded systems is energy and power consumption since many of these systems rely on a limited source of energy in the form of a battery. Energy consumption in the system has direct implication on the lifetime of the system, in addition to other issues such as heat dissipation, size and weight of the system. As a result a number of researchers have proposed both hardware and software techniques for energy optimization in embedded systems, including compiler driven optimizations for energy conservation [1,2,3,4]. Software techniques, including compiler techniques, have addressed various aspects such as processor energy consumption, peripherals, cache memories, and memory sub-system [1,2,3]. It has been observed that the portion of energy consumed by the memory sub-system (memory budget) can be as much as 90% of the total energy required by the system [1,2,3], thus making it a good candidate to target energy optimization techniques.

In designing the memory subsystem of an embedded system to use power efficiently, a common approach is to use *partitioned memory architectures* which use a bank of smaller, individually controllable memory devices so that devices are powered only during use [5,6]. However, the energy thus saved will depend on how the application's data has been allocated to the different memories, on how the overall memory address space has been partitioned amongst the devices, and which particular device is used for each partition. The focus of this work is on compiler directed approaches for data allocation. Energy savings can be obtained through compiler directed approaches by exploiting memory reference behavior to dynamically control power (on or off) to the memory modules. This depends on how the data has been allocated and therefore the need for compiler directed energy aware data allocation methods, a number of which have been designed. Past work in this area has focused on static analysis and on allocation of static global variables such as arrays. This paper considers the problem of compiler directed energy aware allocation of dynamic variables.

The purpose of this paper is to address the problem of energy aware allocation of dynamic, run-time heap allocated, variables and present an approach, and framework, to solving this problem in an effective manner. Our solution is based on using dynamic profile information to model the optimization problem and proposes segmenting the runtime heap to perform an energy aware allocation. We target our optimization technique to optimize energy savings in the memory sub-system by customizing the allocation of dynamic variables and by dynamically controlling the power states of memory banks. The contributions of this paper are

- A *framework* for solving the dynamic variables allocation problem. The framework suggests the use of execution, and power, profile information and proposes segmenting the heap to provide energy aware allocation.

- *Formulation* of the optimization problem. We provide a graph theoretic formulation for the energy aware allocation of dynamic variables.
- *Heuristic algorithm* for energy aware allocation of dynamic variables. This algorithm extends our recent work [17] on allocation of static variables using profile information and the concept of segmenting the heap.
- *Back-end compiler tool*. We have integrated our techniques and algorithms into a back-end compiler tool written for the ARM family of processors. The tool takes ARM assembly language as input and is thus complementary to several higher level compiler optimizations such as loop optimizations [6] and can be integrated into the back-end of most compilers.

The rest of this paper is organized as follows. Related work is reviewed next in Section 2. The problem is described, mathematically formulated, and framework implementation and assumptions stated in Section 3, following which our solution and implementation is described in Section 4. We present experimental results in section 5 and conclusions in section 6.

2 Related Work

A number of energy optimization techniques, using software approach, that explore the memory sub-system have been proposed in the literature. These studies have considered embedded systems where the memory system is assumed to have no cache. Designing partitioned memory systems was explored initially in [5] and revisited in [8]. In their work, the authors pursue the problem of identifying the best block partitioning for scratch-pad memory once the expected access profile and memory power features are known. Intractability of the problem is shown and heuristics are proposed.

Compiler directed energy aware data allocation techniques, such as in [3,6,11], most closely resemble the focus of our research. They consider the same target platform as this paper but focus only on global scalar arrays variables and thus their solution cannot be applied to dynamic variables. They used the lifetime of variables to determine the layout of arrays in memory. The techniques rely on loop optimizations to increase the independency between variables within a control loop and to break big arrays in order to fit each variable within a singular memory block. In recent work we have suggested improvement to their algorithms for global variables [17]. However, both approaches focused only on allocating global static variables and do not consider allocation of dynamic variables. Run-time recompilation to address the dynamic nature of the energy consumption model, such as available battery energy, has been discussed in [11]. Dynamic profiling approaches to determine lifetime of variables was also presented in [4,8]. Our work differs from previous work in power/energy optimizations in partitioned memory systems, because we consider direct memory control for heap allocated (dynamic) variables. The overhead impact of our power-aware technique is a consequence of the control instructions introduced in the code by the optimization, and varies only as a function of the program's path of execution decided at runtime.

In contrast to allocation by the compiler, techniques [9,10,11] have also been designed to control the memory power state at the operating system level by providing a modification to the virtual page control algorithm. Because the system relies on the final physical allocation done when a page fault occurs, there is no power control directly exercised over the memory by the application.

The behavior of dynamically allocated data objects has been studied previously with the objective of increasing cache results and reducing paging in virtual memory systems to improve system performance. The use of profiling to trace lifetime of dynamic objects to reorganize the heap was studied in [12]. The concept of fragmenting the heap to improve virtual memory allocation was proposed in [13]. In [14] Lattner and Adve presented a link time pointer analysis to isolate dynamic allocated data structures that were locally used. The strategies developed in previous work cannot be directly applied to our energy optimization problem since their objective was performance and not energy conservation, but they did lead to some of the insights used in our solution and in particular the concept of profile driven optimizations and segmenting the heap. In relation with previous dynamic allocation techniques our work differs in providing a heap segmentation technique that generates a variable number of segments and in the manner in which the dynamic allocated data is assigned to these segments. The organization of dynamic allocated data into the segments is determined by the affinity groups to which they belong (retrieved from the input code), and the usage patterns of these groups rather then the details of the individual data object.

3 Problem Description

The objective of this research is to examine the problem of energy aware allocation of dynamic variables to partitioned memory architectures. Static global variables are un-initialized global variables

defined outside the scope of a function and visible across all sections of the code, and are defined with a fixed size by the compiler. Dynamic variables are data objects allocated and de-allocated at run-time, usually referred only by usage of pointers. In data intensive systems, which handle large amounts of data and act based on results obtained in processing of the data, the size of dynamic variables can form a substantial part of the overall memory requirements. Thus, in such systems, it is imperative to address energy optimizations of dynamic variables. Since precise information of dynamic variables, such the precise access pattern and sizes, is known only at run-time this is an especially challenging problem in the context of compiler driven techniques.

3.1 Proposed Approach

Our research is motivated by the question: if information, such as access patterns and sizes, about dynamic variables can be obtained can an energy aware allocation of these variables be accomplished by the back-end of a compiler. This paper proposes a framework for allocating dynamic variables at the back-end of the compilation process. Our framework is based on two key aspects: (1) use of profiling to derive information about the dynamic variables and (2) segmenting the heap thereby reducing the problem to allocating data objects to discrete segments. By obtaining the information using profiling and viewing the heap as a collection of segments consisting of a number of memory modules, we formulate the dynamic allocation as a variation of the static allocation problem.

Our framework, and solutions, are targeted to work at the assembly level and thus based entirely on information present at the assembly code level. While working at this level has its challenges and disadvantages it also allows our techniques to be incorporated into the backend of any compiler thereby providing a general solution framework which makes no assumptions on properties of the high level language. In addition, it enables existing code to be readily processed for energy optimization even if the source code is not available thereby enabling conversion of existing code to optimized code. Our techniques are thus applied directly at the basic block level of the code, without regard to any high structures that may exist in the original source.

The profile information needed for our approach should capture both the execution profile and the power consumption profile. Our energy-aware allocation is based on creating affinity groups of dynamic allocated data based on the location in the input code where data is allocated (allocation point). The affinity groups are clustered based on their utilization pattern which is obtained by the dynamic profiler, during code execution. Clusters are assigned to heap segments in a manner to maximize the energy gains that can be obtained by keeping the memory blocks that belong to unneeded segments in a "sleeping mode". The number of heap segments is determined by heuristics. Due to the dynamic nature of heap operation, the partitioning of a heap has profound side effects, such as increasing the memory requirements, and altering the speed of access to heap allocated objects. In order to reduce this effect we must try to limit the number of segments in which we partition the heap, and we provide control parameters for the user that can be used to shape the results of the heuristics. Finally, to accommodate for the imprecise nature of profiling in predicting size and patterns, we suggest a modification to the runtime allocation process (such as *malloc*) by providing an "overflow" segment to the heap to handle cases where a segment runs out of space. Power control instructions are then applied for all memory blocks belonging to a segment. A segment is kept powered during a basic block if any of the affinity groups placed within the segment may be used during this basic block execution. The segment is kept in "sleeping mode" in other condition.

3.2 Architecture and Simulator

We assume that the embedded system memory sub-system architecture consists of a partitioned memory, with no cache, consisting of a number of memory banks. We evaluated our power-aware optimization techniques using an operation level simulator – TriSim developed during our research. Based on a given hardware configuration TriSim executes the code and calculates the power and timing information which includes the memory and processor energy (power). The input to the simulator is ARM assembly output generated by a GCC compiler after it has been processed by our optimizer. During simulation, TriSim executes the power control pseudo-instructions introduced by our optimizer, modifies the power state of the memory banks, and also implements the segmentation of the heap area in memory as specified by the pseudo-instructions in the optimized code. The most relevant output of TriSim, for the purpose of this research, is the energy profile of the code, which includes the amount of energy consumed by each memory block, the number of times each basic block in the code was executed, and the total number of cycles used by the processor to complete the program. TriSim has also other profiling outputs that were used to retrieve dynamic information about the code execution.

3.3 Notation and problem formulation

In order to formally model this problem,

- A program F is defined to be a set of basic blocks $B=\{b_0,b_1,b_2,\ldots,b_n\}$. (A basic block is

a straight line segment of code with exactly one entry point at the start and one exit point at the end of the basic block.)

- Let $D=\{d_0,d_1,d_2,\ldots d_k\}$ denote the set of dynamic data objects allocated at runtime by F.
- For each d_i there is a b_i in B that allocates the data object from the heap at runtime. The allocation of each d_i in the heap, is a function of its impact on the total power optimization of F.

The index of the basic block (allocation point) that creates d_i can be used as a differentiator of the type of information held (b_i) by the data object. Therefore we can partition the set D of F into n sub-sets, referred to as *data sets*, based on their allocation basic blocks in the form:

$D = \{D_1, D_2, \ldots, D_n\}$,where data set D_i represents the sub-set of D allocated by b_i during execution.

In normal programming practices, dynamic data objects allocated at a specific runtime call in a program tend to represent a certain type of information. One must observe that at the assembly level this is the only "type" information available. In our approach, we assume, without loss of generality, that the type of information held at d_i is consistently used in other basic blocks across the program during its execution. In other words, if a block b_k is able to manipulate a data object allocated at basic block b_j, then it should be able to handle any object allocated by b_j.

Let B^*_i denote the use set of d_i, which is composed by the set of basic blocks in F that use the data object d_i. Let B'_j denote the set of basic blocks of F that can handle data objects allocated at basic block b_j (overlap set). Assuming d_i was allocated at basic block b_j, then $d_i \in D_j$. Our assumption of consistent use implies that B'_j is the union of B^*_i for all d_i in D_j, and can be expressed by the statement below:

If $b_m \in B^*_i$ and $d_i \in D_j \rightarrow b_m \in B'_j$

Possible power savings of partitioned memory architecture is based on the pattern of utilization of a program's data objects. In our approach to dynamic data objects, we analyze the overlap sets (B'_j) of the data objects, and determine which of its basic block members are also members of overlap sets of data objects allocated elsewhere in the program.

$(b_m \in B'_j)$ and $(b_m \in B'_k)$, where $j \neq k$

In this case, there is a possible synergy/afinity between the utilization of data objects allocated at b_j and data objects allocated at b_k. This synergy may be explored by targeting the memory allocation of b_j and b_k to the same heap segment. In this manner, the exact number of segments in which the heap is divided will depend on the actual code being optimized, as well as the heuristics used to determine the segmentation of the heap.

This problem can be represented as a graph assignment problem in which the vertices of the graph correspond to the set of data objects allocated in each b_i – *i.e.,* the data sets (D_i). The arcs between vertices D_i and D_j are labelled with the set of basic blocks that can handle objects allocated on both vertices ($B'_i \cap B'_j$). If $B'_i \cap B'_j = \varnothing$ then the nodes are not connected. The basic blocks that belong to B'_i and are not shared with any other overlap set in the code, are considered a sub-set of B'_i denoted as B''_i, and represented in the graph as a self-arc on vertice D_i. Figure 1 shows a graph equivalent to the proposed problem formulation. This problem formulation has similarities with the one presented for the static variables problem in [6,17], with some important distinctions that are based on the dynamic nature of the objects. Firstly, each vertex represents a set of dynamic allocated data objects rather then a unique data object. Secondly, is the emphasis given to the weight of the basic blocks in which the data set is used by itself. Thirdly, weights are represented based on the actual utilization of the data objects across the program's execution using the profile information and not based on their static access patterns such as loop location as in [6].

An arc between two vertices is as strong as the number of basic blocks (taking into consideration their sizes and profiles) that are common to their overlap sets. If two connected data sets are assigned to the same segment h_i, then their connecting arc is represented in their assignment, because by powering h_i the basic blocks in both overlap sets can use the dynamic objects allocated in either vertex. Thus, this suggests a correspondence between the weight of the arc and importance of keeping both sets of datasets in the same segment. The allocation hypothesis of an arc to a segment (edge) has a value which is related with the original arc value, but also evaluates the impact of this allocation in the power profile of the program's. The weight (or value) of the edges is defined using multiple variable functions and are defined in detail in Section 4.

An allocation $\prod$ is an assignment of the vertices in the dependence graph to the k segments in the heap $\{h_1,h_2,\ldots,h_k\}$.

$\prod: \{D_1, D_2, \ldots, D_n\} \rightarrow \{h_1,h_2,\ldots,h_k\}$

The value, or weight, W(∏) of the assignment indicates the amount of energy conserved by co-locating the data sets, and is defined as the sum of the weights of the edges w(i,j) that are mapped to the same segment, *i.e.,*

$W(\prod)= \sum_{(\prod(Di)=\prod(Di)} w(i,j)$

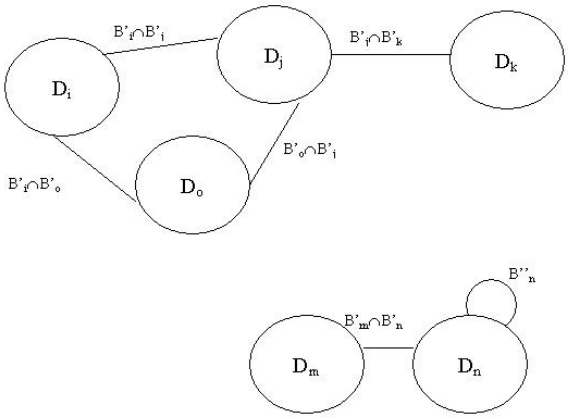

Figure 1 – Dependence Graph equivalent to the code-data dependencies of dynamic objects

Maximizing the energy savings is equivalent to finding the allocation ∏ with maximum value W(∏). For brevity we omit the formal proof of NP completeness for this optimization problem, (observe the analogy to the graph partitioning problem). The intractability suggests the need for efficient heuristics.

3.4 Framework and Assumptions

Key aspects of our solution approach are (a) the use of dynamic profiles to obtain the information used in our problem formulation and (b) segmenting the heap into k segments. We next discuss how these are implemented in our framework and the assumptions.

3.4.1 Heap segmentation

As mentioned earlier our approach is based on segmenting the heap into k segments. We assume that our heap may be composed of multiple types of memory blocks. A heap area corresponds to a homogeneous contiguous part of memory. A heap area may contain one or more segments. Each heap segment can span multiple memory blocks, and the overall size of all segments combined must not exceed the total memory available for the heap. For simplicity, two general assumptions are used for segment size and location:

1. Segment boundaries coincide with memory block boundaries.
2. The memory block structure is homogenous for any specific segment.

The two assumptions above will bound the number of segments into which the heap can be partitioned.

Since the number of segments in which the heap is partitioned can have significant impact on the total footprint and performance (speed and power) of the program, a parameter was introduced to let the system's user provide guidance on how many segments he/she believes are appropriate for the application. The number of segments suggested by the user is not necessarily adopted by the algorithm, but rather is used as an input to control the overall size of each segment.

3.4.2 Dynamic size

Utilization of dynamic data objects and global data objects have an important difference in semantics that needs to be taken into consideration in the determination of the heap segment sizes. Global objects are allocated once for the duration of the program. Dynamic objects can be de-allocated at runtime. This extra functionality allows two different data objects to be placed in the same memory area provided their lifetime is not concurrent. This flexibility is the reason why data and event driven programs use dynamic data in order to decrease their memory requirements.

This flexibility is exploited in programs through natural covariance among the sizes of two apparently distinct dynamic data sets. In order to capture the dynamic nature of this behavior, we have substituted the notion of a maximum size required by a set of dynamic objects, with the notion of a historic variation of the dynamic set size across the program's execution. Our dynamic profiler captures the total size allocated by each distributed set along the execution of the program. This size variation is defined as the dynamic size of the data object set. Using this concept, we can define the dynamic sum of the sizes of two data object sets as the space required for allocating the two data objects sets at any moment during the program execution. We denote this dynamic sum operation by the symbol ‡.

Let σ_i be the dynamic size of D_i, which is the set of sizes of D_i captured during execution. Let σ_j be the dynamic size of D_j, which is the set of sizes of D_j during execution. Then if $\sigma_i = \{s_{i1},s_{i2},s_{i3}, \ldots, s_{in}\}$ and $\sigma_j = \{s_{j1},s_{j2},s_{j3}, \ldots, s_{jn}\}$, the dynamic sum of D_i and D_j sizes is

$\sigma_i \ddagger \sigma_j = (s_{ik} + s_{jk})$, $\forall\ 1 <= k <= n$

where n is the number of size samples captured by the simulator

The segmentation algorithm must take into consideration when allocating dynamic sets to segments the natural covariance between the size variations of the dynamic sets. Ignoring this feature, which is the key reason for using dynamic data, will wrongfully increase the memory demands to execute a program. Correct capture of this dynamic behavior from the program will allow the optimization to position dynamic sets with negative correlation on the same segment, reducing the fragmentation effect introduced by the heap segmentation while reducing the impact on the size of the segment.

3.4.2.1 Overflow heap segment

Grouping dynamic sets within a segment reduces the sensitivity of the technique to the differences between the training set data and the actual execution of the program in the field since not all paths of execution will necessary hold the same size relationships for different dynamic sets. Even with careful profiling and selection of input data, it is possible for a segment to run out of allocation space due to internal fragmentation or to a non-expected runtime data input mix. This can happen even while other segments still have room for allocation. In this case, it is important to guarantee the correctness of the program by allowing the runtime allocation routine, such as *malloc*, to allocate memory from an overflow segment.

The overflow segment can be composed either by a special memory block which was reserved for this function and kept away from the allocation process or by the assignment of memory blocks left unused after the allocation process. The overflow segment is used only in extreme conditions and therefore does not need to be energized during execution, unless an active allocation was performed.

3.4.3 Insertion of control instructions

All the memory blocks that belong to a segment change power state at the same time. Power control instructions are introduced at start and end of basic blocks in a manner to guarantee that the memory will be in full active mode when the access is performed.

If a basic block b_x belongs to the overlap set of a data set D_i, which is allocated at segment h, then a power on control instruction for the memory blocks in h is introduced as the first instruction of basic block b_x. Since we have assumed that the memory blocks are able to re-synchronize within the time that is takes to execute the power instruction then the memory will be active for any subsequent access performed during the basic block execution. The memory blocks of segment h are returned to power off mode by a power off control instruction introduced at the end of the basic block b_x. If the last instruction of b_x is a branch instruction, then the power control instruction must be inserted before the last instruction, otherwise the control instruction is inserted as the last instruction in basic block b_x.

The overflow segment is not directly controlled by inserted instructions in the code. When the runtime allocation routine is forced to allocate memory at the overflow segment, it sets the hardware support to energize the segment. The segment will be kept powered and active until all memory allocated in the overflow segment is freed, when it can be again de-energized (zero consumption). In this way, the energy consumed by the overflow segment can be computed as part of the system overhead energy. The memory budget X will be reduced during the period in which the overflow segment is activated to a new X' value. The overall savings of the system will remain between the boundaries for X and X' memory budgets.

3.4.4 Dynamic Profiler

As seen from our problem formulation, our approach to the dynamic variable allocation problem is heavily dependent on run-time profile information. The dynamic profiler used in our work traces each access to memory during the program's execution in a log file. The log file relates the usage of a dynamically allocated data object by a basic block with the point in the code where this data object was allocated. In this manner we can identify each B^*_i and construct the overlap sets for each data set. The dynamic profiler would also indicate the frequency of execution of each basic block. The frequency of execution is important to determine the weights of the arcs along with the size of each basic block. The final contribution of the dynamic profiler in extracting the features of the input code is in tracing the amount of memory allocated (and still used) by each allocation point on the code. The simulator outputs the accumulated allocated size for each allocation point before each dynamic de-allocation statement is executed, and at the end of the execution. We have chosen to use the de-allocation statement as a reference point to capture the accumulated sizes of the data sets, because only if memory is released, there is a chance of capturing the relative co-variances between data-structures in the code. If no de-allocation statement is executed the sizes of all data sets increase during the program's execution and the profile of dynamic sizes is reduced to the final value required for each set.

One weak point of our profile-based technique is the determination of the sizes of the dynamic-sets and the correct identification of the possible co-variances

between different data sets to derive the overlap sets. This limitation comes from the strong sensitivity in the data collected to the correct mix used in the data training. Unfortunately, this is a feature not from the manner in which we recovered the dependencies, but is in fact related to the high level of runtime flexibility of data intensive systems. We have also looked at composing the overlap set using pointer analysis based on allocation site [15] and shape analysis [16]. These techniques were developed towards the utilization in source code of high-level languages, where type-reach information is available. However, we observed that pointer analysis techniques would not be able to bound the size of dynamic size sets any better then dynamic profiling.

4 A Heuristic Algorithm

In order to decide which data sets should be clustered together in the same segment, we developed heuristics to evaluate the importance of co-locating two sets into the same segment by defining an edge weight function. A greedy four phase algorithm selects each data set segment based on the values of the edges related with the arcs in the graph.

4.1 Algorithm Description

Our allocation algorithm has four phases:

- Phase 1: Organization of data
- Phase 2: Allocation of connected dynamic sets
- Phase 3: Allocation of independent dynamic sets
- Phase 4: Placement of segments on the heap.

We next describe the process in each phase.

4.1.1 Phase 1: organization of data

During this phase, we extract from the input all the required information, such as the weight of each arc, the control flow graph of the input code, and the architecture information about the heap memory. The initial segment of each heap area is constructed.

4.1.2 Phase 2:allocation of connected sets

This phase of the algorithm is composed of three steps. During the first step, the data sets are grouped in connected pairs. Two data sets represent a connected pair if they are used by the same basic block of code (indicated by an arc in the dependence graph). Each arc's allocation to an existing segment is evaluated using the heuristic edge weight objective function (defined in the next subsection). The arc is also evaluated for creation of a new segment on each heap area that holds enough free memory blocks to allocate the data sets.

The result of the first step is a list of "edges" that evaluates each tuple (D_i, D_j, h_x) based on the heuristic objective function, where D_i and D_j are connected data (vertices in the arc) and h_x represents an allocation hypothesis for D_i and D_j on an existing or newly created segment.

The second step will select the edge with highest value and assign the data sets to the segment, creating a new segment in the heap if necessary. Segments will grow to the required size in order to be able to hold the additional data sets.

Since the assignment performed at step 2 may result in a change on the segment's size and composition, in step 3 each existent edge is re-evaluated. If a new segment was created during step 2, then a new assignment hypothesis is created for remaining data set pair in the list, and the proper edges to represent this assignment is generated. It is also possible that due to increased size or creation of new segments, some previous allocation hypothesis are no longer valid. In this case, such edges are removed from the list. Steps 2 and 3 are repeated until all connected data sets, *i.e.*, vertices, have been assigned to a segment.

4.1.3 Phase 3:allocation of independent sets

This phase assigns to the segments the data sets that do not share execution basic blocks with any other data set in the input code (independent sets). In graph representation the independent sets are vertices which only contain a self edge.

This phase follows the same steps as phase 2, but the function that calculates the value of an edge is simplified to handle only one data set.

4.1.4 Phase 4: placement of segments in heap

The last phase of the algorithm defines the address boundaries of each segment. The segment is placed on the heap area that corresponds to its memory type based on the number of memory blocks required by the segment and the order in which the segment was created during the allocation process.

4.2 Edge Weights Computation

The purpose of the edge weight functions is to translate into a (energy savings) number the energy-wise importance of each data set placement into the available segments or areas. During phase 2, data sets are placed in pairs, because the objective is to determine which of the possible clustering

opportunities are more relevant for the input code. In phase 3, the possible synergies between data sets that could result in power gains have already been explored, and the objective of the function is to find the placement of independent data sets that avoids reducing the potential energy gains. The weight or value of an edge (i.e., the objective function) is therefore composed of 4 separate components:

- α: Indicates how important the association of dynamic sets is in the program.
- β: Indicates the impact of allocating this/these dynamic sets to an existing segment.
- γ: Indicates the impact of allocating this/these dynamic sets to a new segment in this heap area.
- δ : Indicates the power saving potential of the memory type in this heap area.

α and δ components are features of the input code and platform respectively, and do not change during the course of the allocation process. β and γ depend on the current allocation status of the segments and their values are re-evaluated after the placement of a distributed set in a segment.

4.2.1 Edge value composition

The weight, or value, W of each edge will either consider the allocation of the data set (or pair of) in an existing segment (β-edges) or into a newly created segment (γ-edges). The value of an edge is a function of the four components as shown below.

$$W = \begin{cases} \left.\begin{array}{l} (\alpha+\beta)/\delta, \text{ if } (\alpha+\beta) > 0 \\ (\alpha+\beta)*\delta, \text{ if } (\alpha+\beta) < 0 \end{array}\right\} \text{for } \beta\text{-edges} \\ \alpha*\gamma/\delta, \text{ for } \gamma\text{-edges} \end{cases}$$

Not all allocations options are valid for all data set combinations. The edges that represent a non-valid placement option are eliminated. When the dynamic sets of a selected edge are successfully placed by the heuristics in the according with the hypothesis evaluated by the edge, we say that the heuristics have accommodated the edge.

4.2.2 Defining the four weight components

Each of the four edge weight components is computed based on a formal definition. For brevity we provide the formal definitions for computing the α component and describe what the other components model and the reader is referred to our report [18]for a detailed computation model.

The α component of an edge value is proportional to the number of instructions executed that belong to basic blocks of the program in which both data sets could be accessed. Let B'_{ij} denote the intersection set of B'_i and B'_j ($B'_{ij}=B'_i \cap B'_j$). If $B'_{ij} = \{b_1, \ldots, b_n\}$ and ω_i denotes the frequency of execution of b_i in the profile and s_i denotes its size in instructions, then the component α of an edge for distributed sets i and j, is calculated by :

$$\alpha_{ij} = \sum_{1}^{n} s_i * \omega_i , b_i \in B'_{ij}$$

The second component β of the edge value incorporates the cost of growing the size of a segment, given the previous allocations already realized by the greedy algorithm. In the calculation of β, the dynamic size of the data sets to be allocated are dynamically added to total size of the data sets already allocated in the segment using the operation for dynamic size defined in section 3.1.2. If the segment needs to be enlarged to support the newly added data sets, then the impact of this proportional increase is deducted from the value of the edge. If one of the data object sets of arc i-j has already been allocated by another edge, then only the size of the unallocated data object set is considered. In this case, only the edge that will allocate the second set in the same segment of the first set is valid. Even when just one of the data sets is allocated, an edge for every segment area in which a new segment can still be created, must be considered.

The value of γ uses the available space in the segment area, the total number of segments already created, and the user's input to evaluate the potential of allocating the arc in a new segment.

Components β and γ treat the need to allocate more memory exclusively from the point of view of memory blocks. The last component δ is responsible for capturing the power heterogeneity in memory technologies. There is a value of δ for each segment area on the heap. Since we are only interested in the relative savings between different types of memory, we normalize all δ's by the potential savings of the first memory type, hence for homogeneous memory the value of δ is one.

4.2.3 Tie-break rules

As with any multi-objective function, it is possible that edges that represent different options of allocation may result in equal value for the heuristics. Often, the impact of the selection between edges of same value can play an important role in the final energy savings obtainable. We created the following tie breaking rules, priority order, which can translate

into important differences in energy and heap organization:

1. Select edge that does not segment the heap
2. Select edge that allocates more dynamic sets
3. Select edge with highest δ component
4. Select edge that does not increase size of segment
5. Select edge which leads to a segment that uses less energy to power after allocation (in homogenous memories rule five translates to less number of memory blocks)

5 Experimental Setup and Results

5.1 Architecture and Benchmarks

The hardware configuration simulated, for our experiments, consists of an ARM7TDMI processor, with ten memory banks each consisting of four memory devices to form a 32k bank of 32 bits width. Two of these memory banks are set as non-controllable and are reserved for instruction code, static data, stack, and overflow segment since these are not targets for our optimization. The remaining eight blocks are power controlled during execution and are reserved for heap data (where dynamic objects are allocated). Energy consumption calculations were based on the energy profile of the ARM processor and the IDT71256L35Y memory device which has the power savings modes used in the optimization methods discussed in this paper. In our test platform, we have modeled the pseudo-instruction that controls the memory state as a store instruction to a privilege address. This combination of memory control and memory device allows the memory to re-synchronize (return to active state) during the time the processor takes to execute the store instruction. The memory budget (energy consumed by the controllable memory banks) of our configuration is modeled to 70% of the overall system energy budget.

5.1.1 DIS benchmarks

The set of test programs selected were four programs from the data intensive systems benchmark suite (DIS 2000) [19]. They were selected mostly because they represent a set of programs that are the prime target of this power optimization approach – namely, embedded systems which are designed to handle large amounts of data, and act based on the results obtained in the processing of this data. These programs rely on pointers, dynamic variable utilization, and data driven loop control. The four DIS benchmarks used were transitive closure, Pointer (pointer following), Update (pointer following with memory updates) and Field (collect statistics on large field of words).

5.2 Experimental Results

Tables 1 and 2 show the savings in memory energy and overall system energy obtained using our energy aware allocation of dynamic variables. The numbers, for each benchmark, are shown as percentage energy consumed when compared to the non-optimized base case. Thus, the first row in each table shows the base case energy percentages (i.e., 100% in Table 1). The first column in Table 1 shows the amount of energy originally consumed by the memory, which corresponds to 70% of the total energy required by the system. Each row indicates the memory savings associated with a specific benchmark. The first column indicates the energy consumed by the memory sub-system after the optimization process is performed. The second column indicates the overhead energy spent by the rest of the system (as a percentage of the energy consumed by the memory sub-system), due to extra cycles introduced in the execution by the optimization process. The last column shows overall savings in the energy consumed by the memory system.

	Memory Energy	System OH	Savings
NOT OPTIMIZED	100.00%	0.00%	0.00%
TRANSITIVE	55.02%	4.52%	40.46%
POINTER	57.17%	6.78%	36.05%
UPDATE	52.08%	8.60%	39.32%
FIELD	61.02%	2.75%	36.23%

Table 1 - Memory Energy for DIS benchmarks

	Total Energy	System Energy	System OH	Memory Energy	Savings
NOT OPTIMIZED	100.00%	30.00%	0.00%	70.00%	0.00%
TRANSITIVE	71.68%	30.00%	3.17%	38.51%	28.32%
POINTER	74.76%	30.00%	4.75%	40.02%	25.24%
UPDATE	72.47%	30.00%	6.02%	36.46%	27.53%
FIELD	74.64%	30.00%	1.92%	42.71%	25.36%

Table 2– System Energy for DIS benchmarks

Table 2 presents the energy savings as a function of the total energy spent by the system. The first column shows the total energy consumed by the application. The second column represents the energy consumed by the non-controlled parts of the system (processor and non-controllable memory banks), which is invariant in our optimization approach. The third column presents the overhead incurred by the non-controlled system, due to the extended period of execution of the optimized program. The fourth column shows energy used by the memory subsystem as a function of the overall energy required to execute the benchmark.

From Table 1, we see that our power optimizations techniques for the DIS benchmarks can lead to memory energy savings of up to 40%, even after compensating for the amount of system overhead

energy required by the optimization. Overall system energy savings, from Table 2, vary from 20% to 30% of the total energy requirements for the execution of the benchmark. The experimental results demonstrate that the technique is successful in obtaining significant gains in the memory energy.

The speed overhead impact resulting from the aggressive control of memory power states by the technique can be significant as shown in Figure 2. While this increased overhead is always outweighed by the energy saved by the memory, in time critical systems it is possible for the performance impact to outside the system's specifications. This is the subject of our ongoing study and we have developed a smoothing technique that balances performance with energy conservation [18].

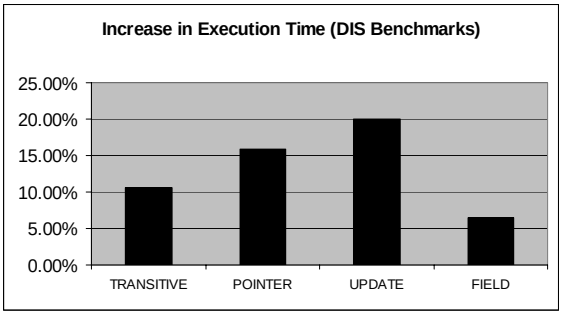

Figure 2 - Increased execution time

6 Conclusion

This paper suggested a framework and solution for compiler directed energy aware allocation of dynamic variables. Our approach is based on using information from the execution profile and proposes heap segmentation, and the concept of an overflow segment to ensure correctness, to solve the allocation problem. The technique presented is applied at the assembly level and thus can be incorporated into any compiler back-end. The techniques and framework were implemented as a backend compiler tool and experimental results were presented to evaluate our solution. The experimental results, on data intensive system benchmarks, demonstrate potential savings of 25% in overall system energy and up to 40% in memory energy for some data intensive benchmarks. Our ongoing efforts include reducing the overhead to provide a balance between impact on execution time and the energy conserved. This paper considered only sleeping and active modes, and future work must address usage of multiple power modes which are available in the current memory devices.

References

1. M. Kandemir, N. Vijaykrishnan, M. J. Irwin and W. Ye. "Influence of compiler optimizations on system power", *IEEE Transactions on VLSI Systems,9(6),801-804,2001.*
2. Vijaykrishnan, N., et al. Energy-driven integrated hardware-software optimizations using SimplePower. in *International Symposium on Computer Architecture*. 2000.
3. Delaluz, V., et al., Hardware and Software Techniques for Controlling DRAM Power Modes*. IEEE Trans. on Computers*, 2001.
4. Udayakumaran, S. and R. Barua. Compiler-decided dynamic memory allocation for scratch-pad based embedded systems. in *International Conference on Compilers, Architectures and Synthesis for Embedded Systems (CASES)*. 2003.
5. Benini, L., A. Macii. A Recursive Algorithm for Low-Power Memory Partitioning. in *Int. Symposium on Low power Electronics and Design*. 2000.
6. Delaluz, V., et al. Energy-Oriented Compiler Optimizations for Partitioned Memory Architectures.*Proc. CASES* 2000.
7. Hajj, N.B.I., C. Polychronopoulos, and G. Stamoulis. Architectural and Compiler Support for Energy Reduction in the Memory Hierarchy of High Performance Microprocessors. *ISLPED98*. 1998.
8. Udayakumaran, S., B. Narahari, and R. Simha. Application Specific Memory Partitioning for Low Power. *Compiler and Oper. Syst for Low Power (COLP)*, 2002.
9. Lebeck, A.R., et al. Power aware page allocation. *9th Int Conf. on Architectural support for prog. languages and operating systems(ASPLOS)*. 2000.
10. Delaluz, V., et al. Scheduler based DRAM Energy Management. *39th Design Automation Conference*. 2002.
11. DeLaLuz, V., et al. Energy-Conscious Memory Allocation and Deallocation for Pointer-Intensive Applications. in *Third International Conference on Embedded Software*. 2003.
12. Barrett, D.A. and B.G. Zorn. Using Lifetime Predictors to Improve Memory Allocation Performance. *Conf on Prog. Language Design and Implementation (PLDI)*. 1993.
13. Seidl, M.L. and B.G. Zorn. Segregating Heap Objects by Reference Behavior and Lifetime, *Proc. ASPLOS-1998*.
14. Lattner, C. and V. Adve. Automatic pool allocation for disjoint data structures, *Workshop on Memory system performance*. 2002. ACM Press.
15. Chase, D.R., M. Wegman, and F.K. Zadeck. Analysis of pointers and structures, in *PLDI*, 1990.
16. Sagiv, M., T. Reps, and R. Wilhelm, Parametric shape analysis via 3-valued logic, *ACM Trans. Prog. Languages and Systems (TOPLAS)*, 2002. 24(3): p. 217 - 298.
17. Levy, R, Narahari, B, and Simha, R, Assembly code level power optimization for partitioned memory architectures, to appear at *IASTED International Conference on Advances in Computer Science and technology (ACST 2004)*, Nov 2004.
18. R. Levy, B. Narahari, R. Simha, "Compiler directed Energy Aware allocation algorithms for partitioned memory systems", Technical Report, Department of Computer Science, GWU, 2004.
19. *Data Intensive Systems Benchmark Suite*. 2000, Atlantic Aerospace Electronics Corporation.