# An Array Allocation Scheme for Energy Reduction in Partitioned Memory Architectures

K. Shyam[1] and R. Govindarajan[2]

[1] Sasken Comunication Technologies Limited
Bangalore, India
kshyam@sasken.com

[2] SuperComputer Education and Research Center
Indian Institute of Science, Bangalore 560 012, India
govind@serc.iisc.ernet.in

**Abstract.** This paper presents a compiler technique that reduces the energy consumption of the memory subsystem, for an off-chip partitioned memory architecture having multiple memory banks and various low-power operating modes for each of these banks. More specifically, we propose an efficient array allocation scheme to reduce the number of simultaneously active memory banks, so that the other memory banks that are inactive can be put to low power modes to reduce the energy. We model this problem as a graph partitioning problem, and use well known heuristics to solve the same. We also propose a simple Integer Linear Programming (ILP) formulation for the above problem. Our approach achieves, on an average, 20% energy reduction over the base scheme, and 8% to 10% energy reduction over previously suggested methods. Further, the results obtained using our heuristic are within 1% of optimal results obtained by using our ILP method.

## 1 Introduction

The use of portable hand-held devices like PDAs mobile phones, laptops, palmtops, etc., is on the increase. Further, portable devices of today are becoming functionally more and more sophisticated. As the functionalities of these devices increase, it places a huge demand on the power source. Since most of these devices rely on internal sources of power, i.e., batteries and are hand-held, it is important to make these devices as energy efficient as possible. Reducing the energy consumption is important as it improves the lifetime, and cost of the battery. Further, as it reduces the heat dissipated by the system, it increases the reliability of the device.

A majority of embedded applications are data intensive and access a large number of arrays in deeply nested loops. It has been observed that a major portion of the energy expended by the programs is in the memory subsystem [3]. In light of these observations, this paper presents a technique to minimize the energy consumed by off chip memory modules, which are divided into banks.

S. Krishnamurthi and M. Odersky (Eds.): CC 2007, LNCS 4420, pp. 32–47, 2007.
© Springer-Verlag Berlin Heidelberg 2007

Each of these banks can operate at various low-power operating modes. In such an architecture, if the data segments of an application are allocated to memory banks such that, a majority of the memory banks can be placed in a low-power mode, for large parts of the duration of execution of a program, it leads to a reduction in the energy consumed by the memory subsystem. Thus in this paper we try to present a technique for such a data segment (array) placement for energy reduction.

Techniques for allocating arrays to memory banks have been proposed earlier. Earlier approaches [6,7] either model the array allocation problem as a maximum weight path cover problem or use a set of heuristics and certain subgroup ordering. As will be observed in Section 3, neither of these approaches is akin to the array allocation problem and results in inferior solution. We model this problem as a graph partitioning problem which is a natural way of formulating the array allocation problem. The arrays in a single partition are allocated to a single memory bank. During this partitioning process, we try to minimize the sum of the weights of the edges that are being cut. We use existing well-known heuristics to solve the graph partitioning problem. Lastly formulating the array allocation problem as a graph partitioning problem has also led us to develop an Integer Linear Programming formulation(ILP) for it.

Initial experiments on array intensive benchmarks show that, on an average our approach obtains around 21% reduction over energy-unaware allocation and 8% to 10% improvement over the method proposed in [6]. In comparison to the optimal solution obtained from the ILP formulation, our heuristic approach produces near optimal allocation in most of the cases and is within 1% of the energy consumption values obtained by using ILP techniques.

Section 2 presents a brief introduction to partitioned memory architectures and low-power operating modes. Section 3 motivates the problem addressed and the issues involved with earlier approaches using an example. In Section 4, we discuss our problem formulation techniques and heuristics that we have used to solve them. We present experimental results in Section 5. Section 6 compares our work with related work. Finally, we conclude the paper in Section 7.

## 2 Background

In this section we give a brief background about the partitioned memory architecture and various low-power operating modes. The memory is divided into banks and each of these banks can be placed into one of the following low-power modes, *Standby, Napping, Power-Down* and *Disabled*, depending on the access patterns. A memory bank is in active mode when it is processing read and write requests [14]. Each low-power operating mode is characterized by the energy consumed in that mode and the resynchronization time. Resynchronization time is the time that is needed for the bank to move from the low-power mode it is currently in to the active mode. The resynchronization times (in cycles), and the energy consumption (in nJ), of various low power operating modes are: 9000 cycles and 0.00875 nJ for Power-Down mode, 30 cycles and 0.0206 nJ for Napping

mode, 2 cycles and 0.468 nJ for Stand-By mode and zero cycles and 0.718 nJ for Active mode respectively. These memory bank energy values and resynchronization times are obtained from the current values of a 2.5V, 1.25nS cycle time, 4MB memory bank [14]. Since resynchronization times are high for those modes which consume the least energy we must choose a low-power mode carefully.

In our study, initially we assume an oracle memory bank activation, i.e., the memory bank $m$ that would be required at time $t$, is transitioned exactly at time $(t - r_t)$, where $r_t$ is the resynchronization time from the low-power mode the bank is currently in, to the active mode. We also study the effects of waking up the memory bank at time $t$, when the actual memory request is made. This incurs a penalty of $r_t$ cycles as the memory bank becomes available only at time $(t + r_t)$. We refer to this scheme as *on-demand activation.*

If a memory bank $m$ is accessed at times $t_1$ and $t_2$, for the purpose of our experiments, the low power mode that the bank can be put into for the duration $[t_1 - t_2]$ is calculated as follows. We consider only those low power modes for which the resynchronization time is less than $(t_2 - t_1)$. Let $(E_{pd}, R_{pd})$, $(E_{np}, R_{np})$, $(E_{sb}, R_{sb})$ be the energy consumption and resynchronization times of the memory bank in Power-Down, Napping, Standby modes and $E_{act}$ the energy consumption in the Active mode. If $R_{sb} < (t_2 - t_1)$, then the energy that would have been expended, if a bank is in a particular low-power mode $E_{lp}$ is given by $min((E_{pd} * (t_2 - t_1 - R_{pd}) + E_{act} * R_{pd}), (E_{np} * (t_2 - t_1 - R_{np}) + E_{act} * R_{np}), (E_{sb} * (t_2 - t_1 - R_{sb}) + E_{act} * R_{sb}))$. Thus for the duration $[t_2 - t_1]$ the bank is put into that mode which consumes the minimum energy. This paper, however, does not deal with how the appropriate low-power mode is identified and the memory bank is transitioned into that mode. This requires an estimation of the duration $(t_2 - t_1)$ which can be obtained either through compile time analysis or through profile runs.

# 3 Motivation

## 3.1 Motivating Example

In this section we describe the problem formulation with the help of the example. Consider the example code given in Figure 1. In loops L1, L2, L3, L4, and L5, the pairs of arrays accessed are $a$ and $d$, $a$ and $b$, $c$ and $d$, $b$ and $c$, and $b$ and $d$. Let us assume loops L1, L2, L3, L4 and L5 take $N$, $2N$, $4N$, $8N$ and $N$ cycles respectively. Further let us assume that arrays $a$ and $d$ occupy 1 MB each, while arrays $b$ and $c$ each occupy 2 MB. Last, let there be two memory banks in the architecture, each of size 4MB. For simplicity, in this example, we assume that

```
float a[N]; double b[N], c[N]; float d[N];
  L1: f or(i = 0; i < N ; i + +)
        {d[i], a[i]}
  L2: f or(i = 0; i < N ; i + +)
        {a[i], b[i]}
  L3: f or(i = 0; i < N ; i + +)
        {c[i], d[i]}
  L4: f or(i = 0; i < N ; i + +)
        {b[i], c[i]}
  L5: f or(i = 0; i < N ; i + +)
        {b[i], d[i]}
```

**Fig. 1.** Motivating Example

the memory bank can be in either active or power-down mode and the resynchronization time is zero.

In an energy unaware allocation, the arrays are allocated in the order in which they are declared. In this example, arrays $a$ and $b$ will reside in memory bank $M_1$, while array $c$ will partially reside in both banks. Array $d$ will reside in bank $M_2$. For this allocation, since arrays $a$ and $d$ are accessed in loop L1, both memory banks need to be in the *active* mode during its execution. Since 2 memory banks are active for $N$ cycles, we say that for $2N$ bank-cycles[1] the memory is *active*. Similarly for loops L2, L3, L4, and L5, the memory is *active* for $2N$, $8N$, $16N$ and $2N$ bank-cycles respectively. Table 1 shows that in the energy unaware allocation, the memory banks are active for a total of $30N$ bank-cycles.

**Table 1.** Memory Banks Active under Various Methods for the Example Code

| Loop | No. of Exec.Cycles | Arrays Accessed | Energy-Unaware | | MWPC Method | | Graph Partitioning | |
|---|---|---|---|---|---|---|---|---|
| | | | Banks Active | Bank-Cycles | Banks Active | Bank-Cycles | Banks Active | Bank-Cycles |
| L1 | $N$ | $a, d$ | 1, 2 | $2N$ | 1, 2 | $2N$ | 1 | $N$ |
| L2 | $2N$ | $a, b$ | 1 | $2N$ | 1 | $2N$ | 1, 2 | $4N$ |
| L3 | $4N$ | $c, d$ | 1, 2 | $8N$ | 2 | $4N$ | 1, 2 | $8N$ |
| L4 | $8N$ | $b, c$ | 1, 2 | $16N$ | 1, 2 | $16N$ | 2 | $8N$ |
| L5 | $N$ | $b, d$ | 1, 2 | $2N$ | 1, 2 | $2N$ | 1, 2 | $2N$ |
| Total | | | | $30N$ | | $26N$ | | $23N$ |

### 3.2 Problems with Existing Approaches

The Maximum Weight Path Cover (MWPC) method proposed by Delaluz et.al in [6] uses the Array Relation Graph (ARG). The ARG for our motivating example is shown in Figure 2. The maximum weight cover of a graph is a path which includes all the nodes in the graph (but not necessarily all the edges) such that the sum of the weights of all edges in the path is the maximum among all covers. A MWPC for the ARG is $a$—$b$—$c$—$d$, which is depicted in the figure using thick edges. The method proposed in [6] suggests that the nodes are traversed in the order in which they appear in the MWPC and are allocated to various memory banks, subject to availability of space in each memory bank. We will assume an array is allocated fully to a single memory bank, whenever the size of the array is less than that of any memory bank. According to the above

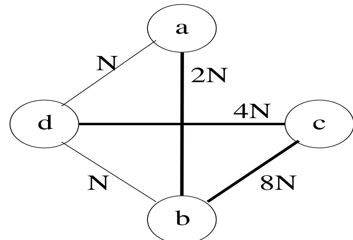



**Fig. 2.** Array Relation Graph and its Maximum Weight Path Cover

[1] We introduce the metric bank-cycle (similar to man-months) to collectively represent the number of memory banks and the cycles for which they remain active.

MWPC, arrays $a$ and $b$ will be allocated to memory bank $M1$ and $c$ and $d$ to memory bank $M2$. For this allocation, the memory banks that are *active* in each loop and the bank-cycles for which the memory is *active* are shown in Table 1. We see that memory is *active* for a total of $26N$ bank-cycles.

Although the MWPC method correctly identifies that edge $(b, c)$ has a large weight, the requirement to allocate arrays to memory banks in the order in which they appear in the MWPC causes the bad decision in this example. Further, MWPC does not take into account the set of nodes that are already allocated to a partition. More specifically, if nodes $v_1$, $v_2$, $v_3$ are already allocated, in that order to partition $P_1$, and in choosing between $v_4$ and $v_5$ that can also be allocated to the same partition $P_1$, it only considers the weight of edges $(v_3, v_4)$ or $(v_3, v_5)$, and not the cumulative benefits due to edges from $v_1$ and $v_2$ to $v_4$ or $v_5$. This is a basic limitation of formulating the array allocation problem as a Maximum Weight Path Cover problem.

We now visit the heuristic proposed in [7] and show that it has a few ambiguities. The authors propose the use of compiler-directed clustering, where the objective is to group array variables with similar lifetime access patterns, so that they can be placed in the same memory module. This method uses three heuristics namely *last-use*, *first-use*, and *same-use* pattern to divide the arrays into subgroups and then using the fourth heuristic, reorder the array variables across two neighboring subgroups which have similar access patterns. However the ordering of the subgroups in the first 3 steps (sub-grouping steps) is arbitrary and is not akin to the underlying problem. As a consequence, the heuristic may or may not result in a good partition depending on the subgroup order generated by the implementation. Further, the sub-grouping may result in a degenerated case where each array is in a subgroup by itself. In fact, for our motivating example this degenerated situation arises after applying the *first-use* and *last-use* heuristics. This prevents an efficient allocation of arrays to memory banks.

### 3.3 Overview of Our Approach

From the discussion in the previous sections, we observe that, given an ARG, we need to partition it into a number of sub-graphs such that the sum of the sizes of the arrays corresponding to the nodes in each sub-graph is less than that of a memory bank size. The edges across the sub-graphs correspond to the cost of keeping multiple memory banks simultaneously active. The objective of the graph partitioning problem is to minimize the sum of the weights of the edges across two partitions.

Let us partition the example ARG into two sub-graphs, one containing nodes $a$ and $d$ and another containing nodes $b$ and $c$. The sum of the sizes of the array corresponding to these sub-graphs is less than 4MB, the size of a memory bank. The edges that are across the two sub-graphs are: $(a, b)$, $(b, d)$, and $(c, d)$. The sum of the weights of these edges is $7N$. For this allocation, the memory banks that are *active* in each loop and the bank-cycles for which the memory is *active* are shown in Table 1. We see that memory is *active* for a total of $23N$ bank-cycles.

## 4 Our Approach

In this section we formulate the array allocation problem as a graph partitioning problem, which, in turn, leads to an Integer Linear Programming formulation.

### 4.1 Graph Partition Formulation

We now give a formal definition of this problem. Let $G = \{V, E, w, c\}$ be the array relation graph which is an undirected graph where each vertex $v$ represents an array. We use the same symbol $v$ to denote the node as well as the array it represents. An edge $(u, v)$ represents that the arrays corresponding to $u$ and $v$ are accessed together in same region of program execution. Associated with each edge $(u, v)$ is a cost $c_{u,v}$, which represents the number of cycles for which arrays $u$ and $v$ are accessed together. Since $G$ is undirected, $c_{u,v} = c_{v,u}$. Finally we associate a weight $w_v$ with each vertex $v$ which corresponds to the size of the array $v$. Let $w$ be a positive number, such that $0 < w_v \leq w$ for all $v$. We are given a memory architecture with $k$ memory banks where the size $m$ of each memory bank is greater than $w$. We can make this assumption without loss of generality since, if for some $v$, $w_v > m$, then a number of memory banks $l = \lfloor (w_v/m) \rfloor$ can be allocated exclusively for $v$ and the remaining array locations in $v$ can be considered in our array allocation problem.

A *k-way partition* of $G$ is a set of subsets $G_i = \{V_i, E_i, w, c\}$, such that

1. Any pair of subsets $G_i$ and $G_j$ are disjoint.
2. $\bigcup_{i=1}^{k} V_i = V$ and
3. For all $(u, v) \in E$, $(u, v)$ is in $E_i$ iff $u \in V_i$ and $v \in V_i$.

A partition is *admissible* if $\sum_{v \in V_i} w_v \leq m$ for all $G_i$. An edge $(u, v) \in E$ is said to be an external edge for a partition if $u \in E_i$ and $v \in E_j$ and $i \neq j$. The *cost* of a partition is the summation of weights of all external edges. We refer to this cost as the external cost of the partition. The partitioning problem is thus to find an admissible partition of $G$ with minimal external cost.

The optimal partitioning problem is NP-Complete [11]. There are a number of heuristic approaches to this problem. We used one such heuristic proposed in [11]. The heuristic proposed primarily aims to find a minimal cost partition of a set of $2n$ elements into two sets of $n$ elements each. The heuristic algorithm works by starting with a pair of initial partitions $A$ and $B$ and swapping vertices $a \in A$ and $b \in B$ to the other partition based on External Cost (ECost) and Internal Cost (ICost). We define ECost of $a$ as $E_a = \sum_{y \in B} c_{ay}$. We also define ICost $I_a$ as $I_a = \sum_{x \in A} c_{ax}$. Similarly we define ECost $E_b$ and ICost $I_b$ for each $b \in B$. Let $D_a = E_a - I_a$ for each $a \in A$ be the difference between the ECost and ICost. Now according to a lemma proved in [11], for any $a \in A$ and $b \in B$, if they are interchanged, the reduction in the partitioning cost is given by $R_{ab} = D_a + D_b - 2 * c_{ab}$. The nodes $a$ and $b$ are interchanged to partitions $B$ and $A$ respectively if $R_{ab} > 0$.

Next we generalize the heuristic algorithm for doing a *k-way partition* (refer to Algorithm 1). In Step 1 the graph is partitioned into a set of $k$ *admissible* partitions. We then proceed to make sure that they are pairwise optimal. To do that we consider a pair of such partitions. In Step 5 and Step 7 we calculate the ICost and ECost. In Step 9 we iterate through the elements of each of the pairs and calculate the reduction in partitioning costs, if they were to be interchanged. In Step 14 we choose the pair of nodes $a$ and $b$, which has the largest positive $R_{ab}$ value. We move $a$ to partition $G_j$ and $b$ to partition $G_i$ if the resulting partitions are admissible. We repeat the steps till no more such interchanges are possible. This process is performed pair-wise on the partitions till no interchange of elements occurs.

We shall illustrate the heuristic on the example graph in Figure 2. The graph is split into two sets $A$, containing the elements $\{a,\ b\}$, and $B$, containing the elements $\{c,\ d\}$. Now if we consider $a \in A$ and $c \in B$ we have $E_a = N$, $I_a = 2N$, $E_c = 8N$, $I_c = 4N$, $D_a = -N$ and $D_c = 4N$ The $R_{ac}$ value is now $3N$. We see that this is the maximum value and hence we need to interchange $a$ and $c$. We get the partition $(a, d)$ and $(b, c)$. The algorithm iterates over step 2 to step 11 and then concludes that no more interchanges are possible and hence terminates.

**Algorithm 1.** Algorithm to partition a graph

1. Partition the graph G randomly into subsets $G_1, G_2, ..., G_k$ such that $G_i = \{V_i, E_i, w, c\}$ and $\sum_{j \in V_i} w_j \leq m$. (*Admissible* Partitions)
2. Do
3. Take a pair of partitions $G_i$ and $G_j$ that are not marked as pairwise optimal.
4. Repeat
5. For each $a \in G_i$ calculate $E_a, I_a, D_a$
6. EndFor
7. For each $b \in G_j$ calculate $E_b, I_b, D_b$
8. EndFor
9. For each $a \in G_i$ do
10. For each $b \in G_j$ do
11. Calculate $R_{ab}$.
12. EndFor
13. EndFor
14. For the largest $R_{ab}$ value, $R_{ab} > 0$, interchange $a \in G_i$ and $b \in G_j$ if the resulting partitions $G_{i'}$ and $G_{j'}$ are admissible.
15. Until all $R_{ab} > 0$
16. Mark $G_i$ and $G_j$ as being optimal with respect to each other
17. While there is no interchange of elements between any two pairs $G_i$ and $G_j$

There are quite a number of implementations of the graph partitioning algorithm available. We use one such implementation described in [10]. A detailed discussion on the technique used for performing the partitioning can be found in [10].

### 4.2 Integer Linear Programming Formulation

In this section we formulate the array allocation problem as an Integer Linear Programming problem. We use the Array Relation Graph representation for this formulation too. We use a 0-1 integer variable with $x_{ij} = 1$ to denote that array $i$ is allocated to a memory bank $j$. Let $s_j$ denote the size of memory bank $j$. If all memory banks are of size $m$, then $s_j = m$ for all $j$. Once again we assume that the size of an array $w_v$ is less than that of a memory bank size $s_j$. Further, since we assume an array can be allocated to only one memory bank, we have the following constraint:

$$\sum_{j=1}^{k} x_{ij} = 1 \text{ for all } i = 1, n \tag{1}$$

Now the sum of the sizes of arrays allocated to each memory bank must be less than the size of the memory bank. This constraint can be formulated as:

$$\sum_{i=1}^{n} x_{ij} * w_i \leq s_j \text{ for all } j = 1, ..., k \tag{2}$$

Note that in the above equation $w_i$ is a constant. To model whether an edge $(i, j)$ is an external edge, i.e., spans two partitions, we use a 0-1 integer variable $e_{ij}$. If $x_{ip} = 1$ and $x_{jq} = 1$, where $p \neq q$, indicating that arrays $i$ and $j$ are placed in two different memory banks (viz. $p$ and $q$ ), then the edge $(i, j)$ is an external edge and therefore the value of $e_{ij}$ must be one. This is specified by the following logical statement $(x_{ip} \mathit{\Lambda} x_{jq}) \implies e_{ij}$. This can be written as a linear constraint as follows:

$$x_{ip} + x_{jq} - e_{ij} < 2 \text{ for all } i, j \in [1, n] \text{ and } p, q \in [1, m] \tag{3}$$

It can be seen that if $x_{ip} = 1$ and $x_{jq} = 1$, then $e_{ij}$ should be equal to 1 to satisfy the above equation. Although the above constraint does not necessarily set the value of $e_{ij}$ to 0 when either of $(x_{ip} = 0)$ or $(x_{jq} = 0)$, the use of $e_{ij}$ in the objective function will ensure this. Thus the objective function of the array allocation problem is to minimize the sum of the weights on the external edges. That is

$$minimize \sum_{i=1}^{n} \sum_{j=i+1}^{n} e_{ij} * c_{ij} \tag{4}$$

subject to Equations 1, 2 and 3. Note that $c_{ij}$s in the objective function are constants.

# 5 Experimental Results

## 5.1 Implementation Details

We have used the SUIF compiler framework [18] to implement our data allocation heuristics. We first compile the given benchmark into a SUIF intermediate file. SUIF provides a framework to analyze this intermediate file on the basis of data dependence framework, live dependence analysis, etc. We use the dependency analysis framework to compute a co-access index matrix, which is the edge weight matrix $C_{uv}$. That is, for a given array $A$ in the program, this matrix is used to find out those arrays that are accessed together along with this array. The sizes of the arrays, along with co-access index matrix are used to construct the ARG which in turn is used as input to the array allocation heuristic. We have implemented Algorithm 1 for our array allocation heuristic. The output of the heuristic is the partition of arrays into different memory banks. For the Integer Linear Programming problem formulation we have used the commercial solver CPLEX® [5]. From the partition obtained from the heuristic or CPLEX solver, we derive the declaration order of the arrays (with appropriate padding) to enforce the partition to different memory banks. We also make necessary modifications to accommodate arrays whose sizes are greater than the memory banks.

We have used Simple-Scalar[16] to simulate the execution of the benchmark programs. The benchmarks with modified array declaration order, are compiled using the PISA tool-chain compiler provided along with the Simple-Scalar distribution with `-O2` optimizations. We have simulated full program execution. The energy consumption of the memory subsystem is estimated by first generating the address trace and determining the active or low-power modes in which the memory banks are in during the different regions of program execution. The energy consumption of the memory subsystem is estimated using the method outlined in Section 2.

## 5.2 Evaluation Methodology

We have used six *array-dominated* and *data-intensive* benchmarks, four from scientific applications and two applications from the embedded systems domain. *Liv8* is a part of the Livermore[12] kernel, which does 2D ADI Integration and has 6 arrays with a dataset size of 33MB. *tomcatv* having 6 arrays and a dataset of 48MB, is a part of SPEC'95 benchmark suite and is a vectorized mesh generation program. *eflux* is a part of Perfect Club benchmark suite and is widely used in image processing applications. It has 5 arrays and a data set size of 42MB. *vpenta* having 8 arrays and a dataset of size 34MB, is a part of the nasa7 kernel, a program in the SPEC'92 floating-point benchmark suite, and is a routine to invert 3 pentadiagonals. The *MPEG-4 Encoder and Decoder* is a video decoder and encoder, having 12 arrays and a dataset of 54MB. The *AMR Encoder and Decoder* is a speech encoder and decoder. It has 10 arrays and a dataset of size 57MB. These are primarily used in many multimedia applications for portable

devices like video capture etc. For the purposes of our experiments, a fifteen minute raw video sample and a fifteen minute raw audio sample were used. These samples were encoded and then decoded back to raw video and raw audio samples.

Many portable devices of today do not provide multiprogramming environment nor have a virtual memory subsystem. Hence, we have assumed a single program environment and all addresses are physical addresses. Further, for most of our experiments, we have assumed a memory system without caches. We have assumed an in-order execution processor having two memory system ports, four integer and floating point ALU's, and one integer and one floating point multiplier/divider.

### 5.3 Results

First we report the performance comparison of four different array allocation techniques. The *No-allocation* scheme refers to one in which arrays are allocated to memory banks in the order in which they are declared in the program. However, we assume that, whenever possible, inactive memory banks are put into appropriate low power mode even in this *No-Allocation* scheme. The *MWPC* technique refers to Delaluz' scheme [6] which allocates arrays to memory banks based on the Maximum Weight Path Cover approach. The *HGPS* scheme corresponds to the heuristic graph partitioning scheme discussed in Section 4.1. Finally, the *ILPS* scheme refers to the ILP formulation presented in Section 4.2. In this study we assume a memory bank size of 2MB and a memory subsystem having enough memory banks to hold all the arrays. In all these experiments, we assume oracle activation of a memory bank as discussed in Section 2. The oracle activation scheme, assumed equally for all four schemes, gives the upper bound of the energy reduction achievable by each of the schemes. We will

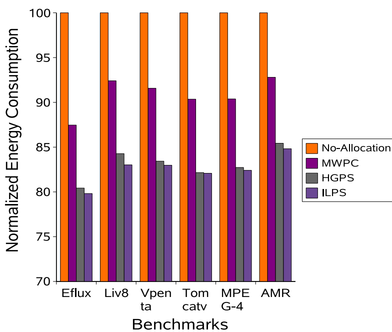



**Fig. 3.** Energy Comparisons for a 2MB Memory Bank

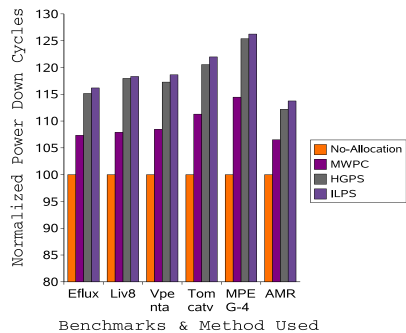



**Fig. 4.** *Power-Down* Cycle Comparisons for a 8MB Memory Bank

evaluate our schemes under a more realistic on-demand activation scheme later in this section.

In Figure 3 we plot the energy consumption of the memory subsystem, for all the benchmarks under various allocation schemes normalized to the *No-Allocation* scheme, which is treated as the base case. We observe that the the *MWPC* scheme achieves an energy reduction of 8% to 12% in various benchmarks programs. Whereas, *HGPS* and *ILPS* achieve a reduction of 18% to 20% in comparison to the *No-Allocation* scheme. Thus the *HGPS* and *ILPS* schemes achieve a further reduction in energy of 8% to 10% over MWPC. This clearly demonstrates the limitation of formulating the array allocation problem as a Maximum Weight Path Cover problem and also highlights the benefits of the graph partitioning approach.

Further, we observe that our heuristic graph partitioning method performs as well as the optimal solution given by the ILPS solver. This is encouraging, given that the heuristic approach takes only 0.1 seconds on the average to solve an average graph partitioning problem, while the ILPS solver could take hundreds of seconds for the same problem. However, in many cases, when the number of arrays and/or memory banks is small (less than 20), the ILPS solver was also able to obtain the optimal partition within 2 seconds.

Much of the effectiveness of the heuristic in trying to reduce the energy consumed by the memory subsystem comes from placing a memory bank in the lowest power mode possible viz., *Power-Down* mode for the largest number of cycles. Hence an increase in number of Power-Down cycles would mean that it is able to find large intervals of idle time for a particular memory bank. Figure 4 plots the power-down mode cycles for all the benchmarks running on a system which has memory banks of size 8MB, normalized to the base case i.e., the *No-Allocation* scheme. As can be observed from Figure 4, the *HGPS* heuristic scheme proposed in this paper is able to place a memory bank in Power-Down mode for as much as 25% more cycles when compared to the base case and upto 12% when compared to using *MWPC* heuristic. Further, the number of cycles in the power-down mode for the *HGPS* is within 1% of that for *ILPS*.

Next we study the impact of memory bank size on energy reduction. In Figure 5 we plot the actual energy consumed (in micro-Joules) by the memory system, with memory bank sizes of 2MB, 4MB, or 8MB, for MPEG-4 and AMR benchmarks under various array allocation schemes. The results for other benchmarks are similar and are not included here due to space constraints. We observe that even under various memory bank sizes *HGPS* and ILPS perform significantly better than *No-Allocation* and *MWPC* schemes. An average improvement of 8% over *MWPC* and 18% over *No-Allocation* is seen in all cases. Also we observe that the difference in the energy consumed by the *HGPS* and *ILPS* array layouts is within 1%. Next, as the memory bank size is increased from 2MB to 4MB and 8MB, the energy consumed by the memory subsystem increases by 14% in case of AMR and 105% in case of MPEG-4. However this increase is seen uniformly across all allocation schemes.

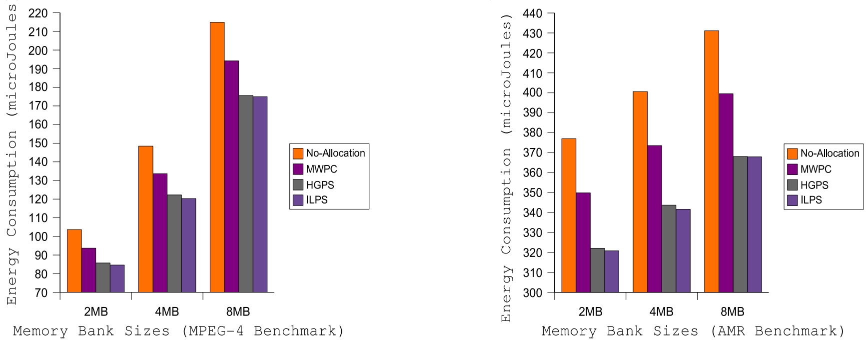



**Fig. 5.** Energy Comparisons for MPEG-4 and AMR Benchmarks

This is due to fewer memory banks and hence fewer opportunities available for the memory allocation scheme to put them to low power modes.

Next we compare the oracle memory bank activation and *on-demand* memory bank activation schemes. As explained in Section 2, the *on-demand* activation scheme results in increased execution time due to resynchronization time. Further, the low-power mode *(Standby, Napping, Power-down)* to which a memory bank is put into is determined assuming oracle knowledge. Figure 6 plots the energy consumed by *MWPC* and *HGPS* methods with *oracle* and *on-demand* memory bank activation, normalized to *No-Allocation* method. In this graph *MWPC(OD)* and *HGPS(OD)* refer to *MWPC* method and *HGPS* schemes with on-demand memory bank activation, while *MWPC* and *HGPS* refer to the respective schemes with oracle activation. The graphs in Figure 6 clearly shows that HGPS with on-demand activation performs better than *MWPC* and *MWPC(OD)* by 5% to 8%. Further, for *HGPS* method it can be observed that the energy consumption difference is only around 3% between oracle and *on-demand* activation, while this difference is upto 6% for *MWPC* method. This could be due to the fact that *HGPS* method is able to place a large number of memory banks in optimum low-power mode, which reduces the need to perform frequent resynchronization.

Figure 7 plots the execution cycles for *on-demand* memory bank activation for *MWPC* and *HGPS* methods normalized to *No-Allocation* method. We have not plotted the execution cycles for oracle memory bank activation schemes as they remain the same. Here we observe that the increase in execution cycles for *HGPS* method is within 3% while it goes upto 5% for *MWPC* method. Thus we conclude that even when we use an *on-demand* activation scheme, we are still able to obtain sufficient energy reduction with little increase in execution time of the program.

In order to study the effect of our array allocation scheme in the presence of caches, we have performed experiments assuming two different L1 cache configurations, a 4K 2-way set associative cache and a 4k 4-way set associative cache. We assumed a memory bank size of 2MB in this experiment. In all these

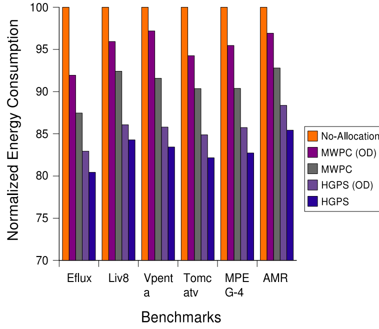



**Fig. 6.** Energy Comparisons between *oracle* and *on-demand* memory bank activation for a 2MB memory bank

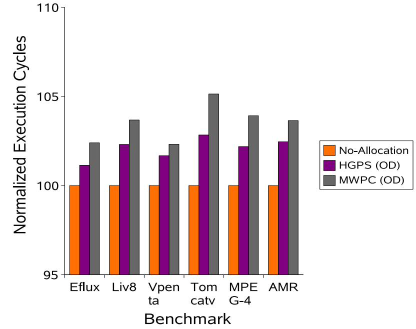
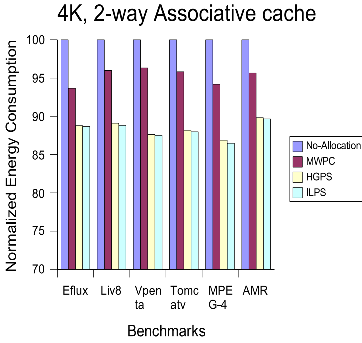



**Fig. 7.** Execution Cycle Comparisons for *on-demand* memory bank activation for 2MB memory bank



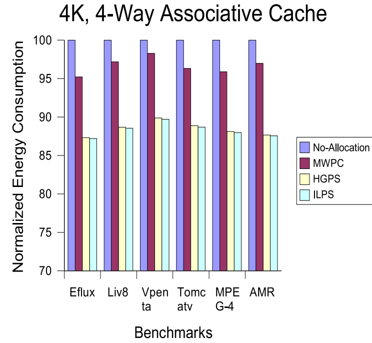




**Fig. 8.** Energy Comparisons for a 2MB Memory Bank with caches

experiments, we assume oracle activation of a memory bank. Figure 8 gives details of energy consumption of the memory subsystem with caches, for all the benchmarks under various allocation schemes normalized to the *No-Allocation* scheme. We observe that the energy reduction due to various array allocation schemes decreases when the memory subsystem consists of a cache. This is due to the fact that the cache filters many of the memory accesses (due to locality), which, in turn, enables the memory banks to be put into low-power modes for longer duration even in *No-Allocation* scheme. However, we also make an important observation that the *HGPS* and *ILPS* schemes are able to obtain a reduction of about 8% in the energy consumption when compared to *MWPC* scheme, even when a cache is present. The results for remaining memory bank configurations (4MB, 8MB) are along similar lines and have not been included due to space constraints.

## 6 Related Work

The problem of minimizing the energy consumption of the memory subsystem is dealt with in [6]. They have also proposed loop optimizations such as tiling for reducing the energy consumption which are orthogonal to the array placement technique considered in this paper. Also the loop optimization considered in their paper might introduce control overheads, which may lead to increase in execution time, which in turn may increase the energy consumed by the system. Their work does not model these appropriately.

Array allocation techniques to minimize the energy consumption of the memory subsystem is dealt with in [2]. They have proposed Array-interleaving and memory layout modifications for identifying memory banks which can be transitioned into low power modes. However they focus mainly on optimizations meant for the Java Virtual Machine environment. The array allocation we have proposed however does not limit itself to any particular run-time environment.

A memory bank assignment algorithm for retargetable compilers is proposed in [9]. They profile the program to obtain data access patterns of variables and use this information to place the variables in such a way that the memory banks can be transitioned to a low power mode. However, unlike the various low-power operating modes for memory banks considered in this paper, they assume that each memory bank is either kept in active state or is switched off.

Energy aware variable partitioning along with instruction scheduling for multi-bank architectures has also been dealt with in [17]. An optimal assignment of variables to memory banks is obtained through effective scheduling of memory intensive instructions. The heuristic we have proposed focuses directly on allocating variables to memory banks by making use of the features provided by the underlying hardware.

Assignment of variables to memory banks is also dealt with in [4]. Their work tries to optimize the assignment mainly for Digital Signal Processors. Although they have also reduced the assignment problem to a graph partitioning problem, they use the idea of Maximum Spanning Tree to partition their graph. Array allocation to memory banks that provide various low power operating modes was also done in [15]. The arrays that are accessed together for a large number of times are allocated to a single memory bank by a greedy heuristic which evaluates a trade-off between size of array and access with other arrays currently in the memory banks.

In [1] they consider Page allocation policies, controlled by operating system, that can take advantage of the various low power modes of the memory banks are considered in [1]. They try to reassign frequently used pages to common memory banks, so that the remaining memory banks can be switched to a low power mode without impacting program performance. While their study concentrates on pages and interaction between the operating system and memory banks, our study focuses on arrays and how compilers can make use of the hardware features.

## 7 Conclusions

In this paper we formulate the array allocation problem as a graph partitioning problem. We observe that this is a more natural formulation for the problem that the earlier approaches [6], [7]. We have used existing heuristic approaches for the graph partitioning problem as a solution to the array allocation problem. We have shown that the array allocation problem can also be formulated as an Integer Linear Programming problem. Our heuristic approaches obtain, on an average, 20% reduction in memory subsystem energy over energy unaware array allocation methods, and 8% to 10% reduction over other competitive methods. Further the heuristic solution performs as well as the optimal solution obtained from the ILP solution, and results in energy consumption that is within 1% of the optimal solution. As future work, we are investigating on methods to determine the appropriate low-power mode for the memory banks.

## References

1. A. R. Lebeck, X. Fan, H. Zeng, and C. S. Ellis. Power aware page allocation. In Proc. of the *Ninth International Conference on Architectural support for programming languages and operating systems.* pp.105-116, 2000.
2. R. Athavale, N. Vijaykrishnan, M. Kandemir, M.J. Irwin. Influence of array allocation mechanisms on memory system energy. In Proc. of the *15th International Parallel and Distributed Processing Symposium*, 2001.
3. F. Cathoor, S. Wuytack, E. De Greef, F. Fransen L.Nachtergaele, and H. De Man. System-level transformations for low-power data transfer and storage. *In Low-Power CMOS Design*, R. Chandrakasan and R. Brodersen, Eds. IEEEPress, Piscataway, NJ.
4. Jeonghun Cho, Yunheung Paek, David Whalley, Fast memory bank assignment for fixed-point digital signal processors, *ACM Transactions on Design Automation of Electronic Systems* (TODAES), pp:52-74, 2004
5. CPLEX. http://www.ilog.com/products/cplex/
6. V. Delaluz, M. Kandemir, N. Vijaykrishnan, and M. J. Irwin. Energy-Oriented Compiler Optimizations for Partitioned Memory Architectures. In Proc. of *International conference on Compilers, architecture, and synthesis for embedded systems*, pp.138-147, 2000.
7. V. Delaluz, M. Kandemir, N. Vijaykrishnan, A. Sivasubramniam, and M. J. Irwin. DRAM Energy Management Using Software and Hardware Directed Power Mode Control. In Proc. of *The Seventh International Symposium on High-Performance Computer Architecture* January 2001.
8. L.R. Ford and D.R. Fulkerson, *Flows in networks,* Princeton, New Jersey: Princeton University Press, p.11, 1962.
9. J. D. Hiser and J. W. Davidson. EMBARC: an efficient memory bank assignment algorithm for retargetable compilers. *ACM SIGPLAN Notices*, v.39 n.7, July 2004
10. G. Karypis and V. Kumar. Multilevel k-way hypergraph partitioning. Technical Report TR 98-036, Department of Computer Science , University of Minnesota, 1998.
11. B. W. Kernighan and S. Lin. An efficient heuristic procedure for partitioning graphs. *The Bell System Technical Journal*, 49(2):291-307, 1970.

12. Livermore Kernels. http://www.netlib.org/benchmark/
13. lp_solve. http://groups.yahoo.com/group/lp_solve/
14. Samsung® Electronics - Direct RDRAM DataSheet 2005.
15. V.V.N.S. Sarvani. Compiler Techniques for Code Size and Power Reduction for Embedded Processors, M.Sc[Engg] Thesis, Department of Computer Science and Automation, Indian Institute of Science, Bangalore, India, 2003.
16. D. Burger and T. Austin. The SimpleScalar Tool Set, Version 3.0. Technical report, Department of Computer Science, University of Wisconsin, Madison, 1999.
17. Z. Wang and X. S. Huw Energy-aware variable partitioning and instruction scheduling for multibank memory architectures. *ACM Transactions on Design Automation of Electronic Systems* (TODAES), pp:369-388, 2005.
18. R. Wilson, et al. SUIF: An infrastructure for research on parallelizing and optimizing compilers. *ACM SIGPLAN Notices*, 29(12):31-37, December 1994.