

Application-Specific Memory Partitioning for Low Power Consumption

Sumesh Udayakumaran, Bhagi Narahari, Rahul Simha*

ABSTRACT

This paper presents an algorithm, and a design tool that incorporates the algorithm, to address the following memory design question: given a collection of memories with power-consumption and size data, and an application with particular execution behavior, which memory devices should be selected for the memory bank and how should the application's data be allocated so that overall memory power consumption is minimized? Experimental results with some benchmark applications indicate the algorithm often results in 30-50% power reduction and near-optimal performance in comparison with exhaustive search.

1. INTRODUCTION

In designing the memory sub-system of an embedded system to use power efficiently, a common precept is to use a bank of smaller, individually controllable memory devices (or cores, in the case of an IP selection process [8, 9, 35]) so that devices are powered only during use and are only as large as required to store the data allocated to them. However, the power thus saved will depend on how the application's data has been allocated to the different devices, on how the overall memory address space has been partitioned amongst the devices, and which particular device is used for each partition.

The purpose of this paper is to address the design choices in the space of memory partitioning: the allocation of data, the partitioning of memory and the choice of particular memory chips for the partitions. The contributions of the paper are:

- *Formulation of optimization problem.* We formulate a non-linear integer-programming optimization problem that relates the overall energy consumed with the

following input: the execution profile of the application, the code structure and a database of memory chips each with their size and power characteristics. The particular form of the objective (cost) function itself can be modified as desired; for our experimentation we use a particular objective function that incorporates memory power consumption during operation (in active, sleep or wake-up modes), the added cost of excessive partitioning and the wake-up time required.

- *Optimization algorithm.* We present a fast optimization heuristic that exploits aspects of the problem structure. Our experimentation with benchmarks used in [12] shows that it produces results within 10-15% of the optimal value found by (exponential-time) exhaustive search, with up to 30-50% power savings over the case that no memory partitioning is used.
- *Back-end compiler tool.* We have integrated our algorithm into a back-end compiler tool written for the ARM family of processors. The tool includes an ARM simulator that creates the execution profile of an application and that uses the database of memory chip characteristics to provide a cycle-accurate measurement of power consumption. The tool takes ARM assembly language as input and is thus complementary to several higher-level compiler optimizations such as, for example, splitting arrays [12]. In these cases, our tool would simply use the results of front-end optimization as manifested in the resulting assembly code. Furthermore, by working directly with assembly, the tool can accommodate low-level programmer optimizations at the assembly level.

Our approach lies in the area of software-hardware co-design: both the data allocation and the memory partitioning are performed *jointly* by our tool. The key hardware/software characteristics that appear to play an important role include the application's variables (their number and size), the access patterns to the data (spatial and temporal locality) and the memory characteristics (size, time to switch from sleep mode to active mode). There is often a somewhat complex interplay between these factors. For example, two large variables might be stored in two separate memories; however, if they are repeatedly accessed in sequence it might be more effective to employ a single large memory. At the same time, if the time between accessing the variables is sufficiently long, the two-memory solution might be better.

*Department of Computer Science, The George Washington University, Washington, DC 20052. Email: {sumesh,narahari,simha}@seas.gwu.edu. This work was partially supported by NSF award 0216137.

The remainder of the paper is organized as follows. Related work is reviewed next in Section 2. The problem is described and mathematically formulated in Section 3, following which our algorithm is described in Section 4. We present experimental results in Section 5 before concluding in Section 6.

2. RELATED WORK

Extensive surveys of past research on power minimization can be found in [20, 27, 17]. Recent research on software strategies to save power include operating system level strategies [22, 31, 34] and compiler techniques [15, 7, 10, 16, 18, 3, 30].

A review of power-related memory issues can be found in [19]. Note that several papers have studied the problem of data partitioning from the perspective of latency [25, 2]. The idea of exploiting memory sleep modes appears to have been first presented in [13]. They suggested partitioning of the memory into two segments that could dynamically be put into “sleep” mode to save energy and suggested heuristics to solve the optimization problem, which was shown to be NP-complete. Shiue and Chakrabarti [29] and Catthoor et al [6] focused on cache design and custom memory hierarchy respectively to minimize power consumption. Delaluz et al [12] also consider the memory partitioning problem for low power. Their approach is to study high-level program information such as the number of loop iterations to allocate array variables to different partitions. They do not consider the problem of designing memory sizes in accordance with program structure.

Although we also develop graphs which is similar to the interference graph used for register allocation [24], the construction of the graph varies remarkably. The work that comes closest to ours is described in a seminal paper by Benini et al [5] using the memory modeling technique in [11]. In their approach, using the default data allocation given by the compiler, they focus on determining partitions of the address space to minimize a power-related objective function. They adopt a recursive approach that may result in non-standard memory sizes (arbitrary integers, rather than powers-of-2). This approach is sometimes feasible when the memory core is being designed from scratch. In contrast, we use a database of of the shelf memory devices with their unique size and power characteristics and, most importantly, we allow the application’s data to be re-allocated (re-ordered) for greater efficiency in co-locating variables that tend to be used together.

3. PROBLEM DESCRIPTION

3.1 System architecture

We begin with a description of a sample system architecture. As mentioned earlier, our approach and algorithm can be applied to a variety of architectures. For concreteness, we use the particular architecture shown in Figure 1. The memory sub-system consists of a partitionable scratchpad and a partitionable off-chip memory. Typically, data is first allocated to either the scratchpad and main memory; the most frequently accessed items are placed in the scratchpad. Although both types of memory may be partitioned, we only consider partitioning the scratchpad SRAM in our

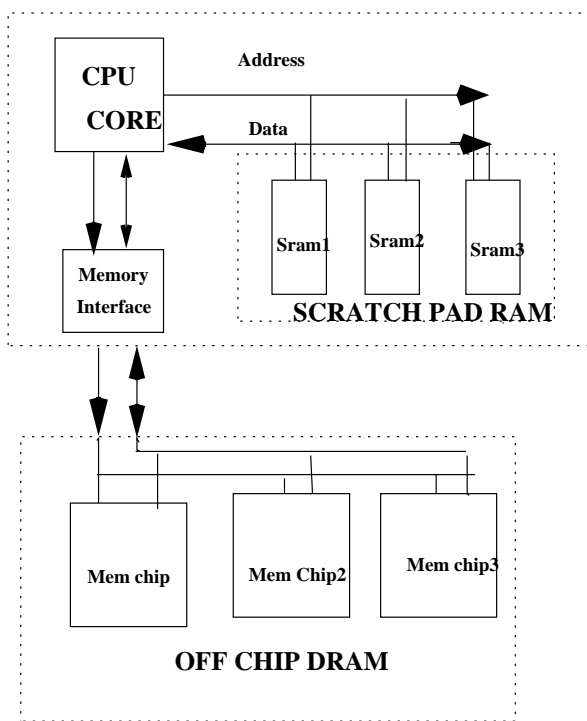


Figure 1: The architecture

experiments. Also, we assume that the code itself (instructions) and associated stack are placed in a separate partition that we do not consider for allocation.

3.2 Memory devices

Memory devices differ in power consumed during active, sleep and wake-up modes, and in the time spent during wake-up. Generally, a larger memory uses more power and therefore once data is clustered into different partitions, it is most often optimal to use the smallest memory that fits a given partition. Thus our algorithms are governed by this assumption that energy per access, energy consumed in different modes by a device monotonically increases with size. While this suggests using as fine a partition as the data allows, there is an energy cost to excessive partitioning that arises from the additional controls required to separately control the device assigned to each partition. We allow the number of partitions to be limited in our approach. This limit can also arise from board-level and other architectural considerations.

3.3 Notation and problem formulation

To better explain the problem, it will help to first describe the design optimization process. Figure 3 depicts components in our design optimization tool. In essence, the instruction-level simulator first creates a behavior profile from a sample execution of the application. This profile together with the assembly code for the application, and the memory device database, is input to the optimizing algorithm. The algorithm first clusters variables and then assigns the best possible memory device to each cluster. The result is then fed back into the simulator that can evaluate the allocation and

memory organization. The simulator can output both an energy and execution profile for the result.

The following notation will be useful in the problem formulation and discussion.

- Let $V = \{v_1, v_2, \dots, v_g\}$ be the collection of *static* global variables in the application (We do not consider dynamic memory). The size of each variable v_i is denoted by $|v_i|$.
- If we opt to use m memories (potentially of different sizes) then the address space is partitioned among these m memories. A data allocation of these variables to m different memories (or partitions) can be described as follows:

$$x_{ij} = \begin{cases} 1, & \text{if } v_i \text{ is assigned to partition } j \\ 0, & \text{otherwise.} \end{cases}$$

Note that the value of m will be an output of the optimization process.

- Let $P_j = \{i : x_{ij} = 1\}$ denote the group of variables assigned to partition j . Define the size of each group of variables as $|P_j| = \sum_i x_{ij}|v_i|$.
- Let $\mathcal{M} = \{M_1, \dots, M_k\}$ denote the database of memory devices. $|M_j|$ denotes the size of memory device M_j .
- Define the function $\beta(j)$ that assigns a memory device to a partition: $\beta(j) = \min_{|M_q|} \{M_q : |M_q| \geq |P_j|\}$. Thus, $\beta(j)$ is the smallest memory that fits the group of variables in partition j . We will assume that the database produces a unique memory for this size. If that is not the case, equivalent memory devices (for this size) can be prioritized by cost or vendor preference, and the optimization process can be repeated for different such selections.
- We next define the characteristics of each memory device. Each memory device is assumed to consume different levels of energy during active, sleep or wake-up modes. Furthermore, we allow that reads and writes consume different levels of energy per unit time.
 $L(M_j)$ – the latency of device j .
 $E_A(M_j)$ – the energy device j consumes in an active state.
 $E_U(M_j)$ – the energy device j consumes in wake-up mode.
 $E_I(M_j)$ – the energy device j consumes in the idle (sleep) mode.
 $E_R(M_j)$ – the energy consumed by a read operation on device j .
 $E_W(M_j)$ – the energy consumed by a write operation on device j .
 $W(M_j)$ – the wake-up time required for device j .

The last value indicates the lead time required to transition from the sleep mode to active mode. During this time, the energy consumed is $E_U(M_j)$ per unit time.

Up to this point, we have defined device characteristics and introduced notation to describe a partition. However, the exact energy consumed depends on the particular execution behavior. Accordingly, the next few definitions introduce

the notion of time, discretized into small time-steps (a cycle is one unit of time). Define the indicator function

$$y_i(t) = \begin{cases} 1, & \text{if a read on } v_i \text{ occurs at time } t \\ 0, & \text{otherwise.} \end{cases}$$

Likewise let $z_i(t) = 1$ if and only if a write to v_i occurs at time t . Once a partition (grouping of variables) is created, this results in an access pattern to the memory devices. Define $U_j(t) = 1$ if and only if $\exists v_i : y_i(t) = 1$ and $x_{ij} = 1$. Thus, $U_j(t) = 1$ when there is an access to partition j at time t and $U_j(t) = 0$ otherwise.

The access patterns to the devices then induce a pattern of energy usage for each device. Unfortunately, this pattern is somewhat complex to describe for the following reason. A device must be active at the time of access and therefore, if it was asleep a wake-up instruction must be executed in sufficient time for the access. Furthermore, if two successive accesses for memory device j occur in a timespan less than $W(M_j)$ then the memory must remain active during this timespan. This is because the control state of a memory device is unique at any given time, and cannot store a sequence of commands related to state transitions. Thus, to compute the times at which a device is active, sleeping or in wake-up mode will require careful consideration of the access pattern. For completeness, we describe this next.

Define the following indicator functions that will identify the state of a device j at time $t \in [1, T]$ where T is the total cycle length of the execution: $\gamma_j(t) = 1$ when $\beta(j)$ is active; $\delta_j(t) = 1$ when $\beta(j)$ is in wake-up mode; $\epsilon_j(t) = 1$ when $\beta(j)$ is in sleep mode. We next show how these can be computed given the execution behavior:

1. Set $\gamma_j(t) = 1$ when $U_j(t) = 1$.
2. Set $\gamma_j(t + t') = 1$ when $U_j(t) = 1$, for all $t' \in [t, t + L(\beta(j))]$. Thus, the memory remains active during the latency period.
3. Set $\delta_j(t') = 1$ for all $t' \in [t - L, t]$ where $L = W(\beta(j))$ and $\gamma_j(t) = 1$. Here, we identify the wake-up mode time prior to each access. Note that this is not yet complete since we have to account for short access intervals.
4. Set $\gamma_j(t') = 1$ for all $t' \in [t_1, t_2]$ where $t_2 - t_1 < W(\beta(j))$ and $\gamma_j(t_1) = \gamma_j(t_2) = 1$. Thus, for all short access times when a device cannot be powered down, we must leave it active. Since we earlier set $\delta_j(t) = 1$ for some of these periods, we must now set $\delta_j(t) = 0$ whenever $\gamma_j(t) = 1$.
5. Set $\epsilon_j(t) = 1$ for all t such that $\gamma_j(t) = \delta_j(t) = 0$.

The definitions above completely capture the energy profile of the application. What remains is to define the total energy consumed during this entire time. For this purpose, we will define two additional variables for the read and writes to a partition: $Y_j(t) = \sum_i y_i(t)x_{ij}$ (the total number of reads to device j) and $Z_j(t) = \sum_i z_i(t)x_{ij}$ (the total number of writes to device j). Then, the overall energy consumed is

given by:

$$\begin{aligned}
 E = & \sum_{j=1}^m \sum_{t=1}^T (\gamma_j(t) E_A(\beta(j)) \\
 & + \delta_j(t) E_U(\beta(j)) + \epsilon_j(t) E_I(\beta(j))) \\
 & + \sum_{i=1}^g (E_R \beta(j) Y_i(t) + E_W \beta(j) Z_i(t))
 \end{aligned}$$

The first three terms are the active, wake-up, idle mode energy consumptions whereas the last is the (separately counted) energy for reads and writes.

Lastly, we observe that excessive partitioning brings increased overhead in decoding and addressing complexity. This cost generally increases monotonically with the number of partitions. Although not shown above, this cost is easily added to the cost function.

We are now in a position to state the problem: *Given (1) an application in assembly, (2) a memory device database, (3) the desired scratchpad size, and (4) the maximum number of partitions, find a partition and allocation of variables to minimize the total energy consumed.*

At first, the use of 0-1 variables suggests an integer programming formulation. Unfortunately, the computation of the overlapped active intervals results in a complicated non-linear cost function (that must be evaluated programmatically); current integer-programming packages are not adequate to this task. However, we are able to generate all possible partitions exhaustively to compute the optimal solution for use in comparison with our direct heuristic algorithm. The timing complexity of the direct heuristic algorithm is $O(n)$ when compared to the exponential $O(2^n)$ complexity of the exhaustive search.

The power model that we use for the algorithms is thus implicit in the cost functions which is based on components like energy per read/write access, energy per cycle in each of the different modes of the device. These numbers are extracted from the device database. Note our solution approach does not depend on the precise form of the cost function above. Both constants and the functional form can be varied if desired.

4. A GRAPH-BASED HEURISTIC

4.1 Solution overview

The solution process is organized into four distinct phases: (1) the application is run on the simulator to obtain an execution profile that is analysed together with the code to produce variable usage and lifetime information; (2) variables are then selected for the scratchpad according to the well-known greedy knapsack algorithm [23]; (3) the memory to be partitioned is then partitioned by the algorithm; (4) the resulting memory organization is then fed back to the simulator to evaluate the partition.

Note that the number of groups of variables is not a fixed number. For example, consider three variables A, B, C . There are five partitions: a partition of size one (ABC), three partitions of size two ($A|BC$, $B|AC$ and $C|AB$) and a partition of size three ($A|B|C$). The number of candidate partitions

possible grows very quickly with the number of items [28]. Therefore, instead of searching the space of partitions exhaustively, we must create partitions through a faster heuristic. This heuristic uses information from the program execution and the structure of the code to preferentially group variables. We transform part of this problem into an instance of the well-known graph partitioning problem that captures the relationship between variables.

The traditional definition of variable lifetime as used in compiler optimization is a collection of definition-use intervals that start with a variable's first definition and its last use. This definition, as mentioned earlier, is usually independent of the particular hardware upon which the code executes, especially if the optimizations are performed on intermediate code.

We now introduce a different definition of lifetime that explicitly uses the wake-up time of a memory device. The purpose is to allow the device characteristics to influence the choice of variables that are to be allocated the same device. Consider a particular memory device. If variable v_i is being considered for allocation to that device, we transitively coalesce every pair of intervals where their separation is less than the wake-up time of the device. Thus, our definition transforms the lifetime set with this filter. The intuition is that if the wake-up time is short, then energy can be saved if the device is placed in sleep mode between definition and use.

The following points summarize our approach towards determining whether two variables are good candidates for being placed in the same memory device:

- First, a variable's lifetime data is extracted and the intervals coalesced according to the device's wake-up time.
- For every two variables, a *co-allocation index* is computed in the following manner. For every pair of lifetime-intervals, one from the first variable and one from the second, if the intervals overlap, the co-allocation index is increased by the length of the intervals' union; if the intervals do not overlap but are separated by a time less than the device's wake-up time, then the length of the union of the intervals and the period between is added to the co-allocation index; otherwise, the co-allocation index is taken to be infinity. Intuitively, if the intervals are distinct but close, the memory device should not be powered down if the spacing between the intervals is less than the wake-up time. Thus, over all the intervals, the co-allocation index is an inverse indication of the desirability of co-locating two variables. If the overlap is perfect, the index is minimal; otherwise it is high.
- Whenever a particular device is being considered for two variables, the co-allocation index is computed for that device.
- If a decision is made to co-locate two variables, the two variables are temporarily combined into a "compound variable" for the purpose of optimization. The lifetime

of this compound variable is computed based on the individual lifetimes of the two variables and coalesced according to the device lifetime. Compound variables may then be combined with others, leading to a data allocation when the combining process stops. It is the decision-making in combining that forms the key to the algorithm; we use a graph-theoretic approach for this part of the problem.

4.2 Algorithm

We construct a graph by representing each variable with a graph vertex. the weight of an edge between two nodes is given by the *co-allocation index*, as described earlier.

Intuitively, pairs of nodes whose joining edges have a low edge-weight are good candidates for clustering; that is, the corresponding variables should be placed in the same partition. The partitioning process ($O(n)$ algorithm) is an iterative node clustering process that is carried on until either no further clustering is possible, no gain is achieved from additional clustering, or the number of allowable partitions is exhausted. After each step of clustering the lifetime is constructed for each resulting compound variable. This process is illustrated in Figure 2 and the pseudocode given below.

```

FIND-MEMORY-CONFIGURATION ()
1. Saved power = 0
2. Increment = 1
3. Initialize configuration with each
   node in a separate bank
4. for each node in  $G$ 
5.   Extract clustered access times using threshold
6. endfor
7. Estimate power consumed by initial configuration
8. while numEdges  $\neq 0$ 
9.   Select edge with minimum weight
10.  Cluster the endpoints as a compound node
11.  Select best fit memory device for
     this compound node
12.  Recalculate lifetime for new node
13.  Evaluate new energy consumption
14.  Compute increment over previous energy
15.  if increment  $\leq 0$ 
16.    break
17. endwhile
18. return configuration.

```

The output at the end of clustering is an allocation of data variables to partitions: each cluster represents a partition. Following this step, memory devices are selected as described earlier. The result is a combination of data allocation and memory device selection. This result is then evaluated in the simulator for performance.

5. EXPERIMENTAL SETUP AND RESULTS

We present our experimental results and analyze them in this section. To evaluate the effectiveness of our heuristics and the rest of the framework we use seven array-dominated benchmarks used by Delaluz et al [12]. To illustrate the

strength of our results we use these benchmarks without any locality enhancing transformation on them.

We used the gcc compiler to generate the assembly code for the instruction level simulator. The load/store trace of the application generated by the simulator is used by the algorithms. The instruction level simulator that we built also has facilities for fine grained configuration of the memory subsystem. This is useful for the validation of the results from the algorithm. The power model used by the simulator to validate is an instruction level power analysis using the cost functions described earlier. As mentioned earlier, our architecture consists of scratchpad and main memory. For ease of experimentation, we use a large scratchpad and consider the problem of applying our technique to partitioning the scratchpad. As discussed in an earlier section, our techniques are applicable to both scratchpad and main memory; for illustration, we choose to use scratchpad partitioning. The size of our scratch pad is taken to be 1 MB. The device database used is made up of devices from two different vendors. The device sets vary in the energy consumption figures and wake up times. To represent the two extreme cases device set1 has lower energy consumption figures but higher wake up times. Device set2 represents the other extreme. It has to be noted that the search results would improve if the database were richer.

We identify an energy cost to excessive partitioning. This cost is attributed to the extra control circuitry and address lines required for each device. We structure this cost to reflect both these attributes. Consider the case when we have m devices (an m -way partition). In this case, about $\log_2(m)$ device selection lines are required, and each device needs selection circuitry. Thus, one part of the cost depends on the number of lines and the other is approximately linear in m . Let $E_P(m)$ be the energy cost representing an m -way partition. Then, $E_P(m)$ can be written recursively as follows:

$$\begin{aligned}
 E_P(m) &= E_P(2^{\lfloor \log(m-1) \rfloor}) + c_1(m - 2^{\lfloor \log(m-1) \rfloor} - 1) + c_2 \\
 E_P(2) &= c_1(m - 2^{\lfloor \log(m-1) \rfloor} - 1) + c_2
 \end{aligned}$$

where c_1 is the added cost for each device and c_2 is the added cost for each new address line. The boundary condition is: $E_P(1)$ is taken to be the constant c_2 . Figure 4 shows a plot of the addressing cost versus the number of partitions taking c_2 as .2 and c_1 as .1.

The input to our experiments is assembly for the ARM 7 processor as generated by the gcc compiler.

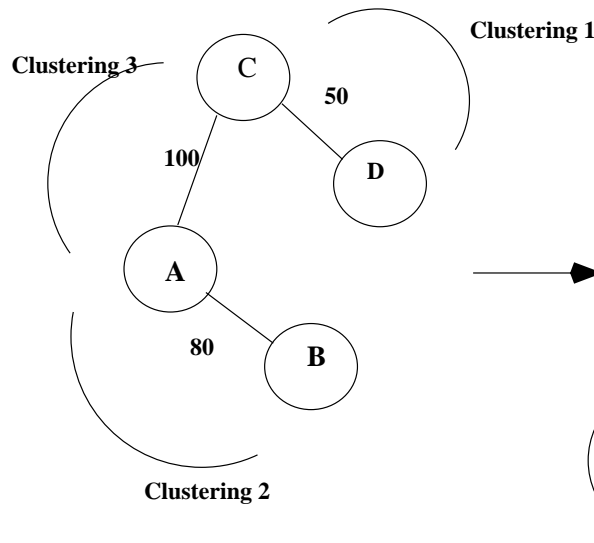
Table 1 shows the savings comparison between exhaustive and greedy heuristics for two different device sets. The energy savings are with respect to a single monolithic memory with no power management. The greedy heuristics results for both the device sets is always only marginally less than the results of the exhaustive search. An interesting point to note is that any one device set does not give the best results for all the benchmarks. The partitionings are also different for the device sets for a few benchmarks. This illustrates the need for the choice of device set which is most suited to the application. The benchmarks *phods*, *matvec*,

```

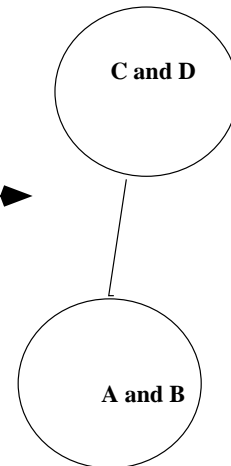
for(int i=0 ; i < limit ; i++)
{
    Access A,B
}
.
.
for(int i = 0 ; i < limit/2 ; i ++ )
{
    Access A,C
    .
}
.
for(int i = 0 ; i < limit ; i ++ )
{
    Access C,D
    .
}

```

(a) Sample code



(b) Clustering process



(c) Possible clustering outcome

Figure 2: The algorithm

`eflux` and `tomcatv` give as much as 47% savings whereas the benchmarks `apsi` and `mxm` show no power savings. The latter could be attributed to the way these benchmarks exhibit locality. Benchmarks like `phods` which are made up of loops accessing a few variables present a collection of clearly distinguishable lifetimes. This allows for efficient partitioning. The benchmarks `apsi` and `mxm` use multiple variables in the same loop. Consequently the lifetime interval due to the loop cannot be split further so as to favor any two variables. As partitioning directly depends on using these lifetime intervals to enhance spatial locality, the opportunity to partition is severely restricted. As the initial configuration is one with the maximum partitions, a negative cost results due to the unnecessary partitioning without any useful clustering of variables. We conjecture that using locality enhancing transformations such as array renaming, interleaving as proposed by Delaluz et al [12] can improve the energy consumption of these benchmarks.

6. CONCLUSIONS

This paper proposed a compiler-based technique for synthesizing the organization of an application-specific partitioning of memory. The partitioning is based on the intuition of clustering data that exhibit temporal nearness in access patterns. The strengths of the approach are summarized as follows: (1) we use a given database of memory devices; (2) since our input is assembly, our approach is complementary to several high-level compiler-based optimizations and it also allows programmer input in assembly; (3) we account

for the cost of excessive partitioning; (4) we combine data allocation and partitioning; (5) our program analysis takes memory characteristics into account. Our results show that our algorithm runs quickly and produces results with 10-15% of the optimal solution found by exhaustive search.

7. REFERENCES

- [1] S.G.Abraham and B.R.Rau. Efficient design space exploration in PICO. *CASES*, 2000.
- [2] O.Avissar, R.Barua, D.Stewart. Heterogeneous memory management for embedded systems. *Proceedings of the ACM 2nd Int'l Conf. on Compilers, Architectures, and Synthesis for Embedded Systems (CASES)*, Atlanta, GA, November 2001.
- [3] A.Azevedo, R.Cornea, I.Issenin, R.Gupta, N.Dutt, A.Nicolau, A.Veidenbaum. Architectural and compiler strategies for dynamic power management in the copper project. *IWIA International Workshop on Innovative Architecture*, Maui, Hawaii, January 2001.
- [4] R.Banakar, S.Steinke, B-S Lee, M.Balakrishnan, P.Marwedel. Comparison of cache- and scratch-Pad based memory systems with respect to performance, area and energy consumption. *Technical Report No. 762*, University of Dortmund, Dept. of CS.
- [5] L.Benini, A.Macil and M.Poncino. A recursive algorithm for low-power memory partitioning. *ISSLPED proceedings*, pp.78-83, 2001.

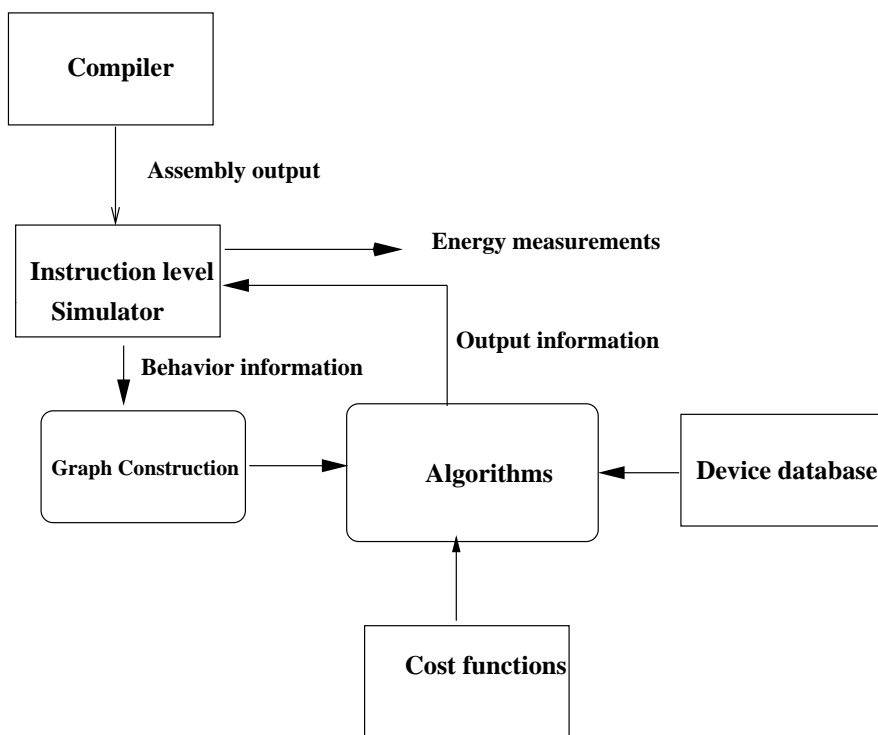


Figure 3: Experimental Setup

Benchmark	Device set1		Device set2	
	Exhaustive	Greedy	Exhaustive	Greedy
phods.c	50.7	47	22	22
adi.c	23	17	18	18
eflux.c	32	32	39	39
matvec.c	56	47	22	22
mxm.c	0	-5	0	0
apsi.c	0	-10	5	-12
tomcatv.c	21	21	32	31

Table 1: Percentage savings compared with monolithic memory

- [6] F.Catthoor, S.Wuytack, E.De Greef, F.Balasa, L Nachtergaele and A.Vandecappelle. Custom memory management methodology for memory optimization for embedded multimedia system design. *Kluwer*, 1998.
- [7] J.Chang and M.Pedram. Register allocation and binding for low power. *32nd ACM/IEEE Design Automation Conference*, 1995.
- [8] T.Chang, L.Benini and G.De Micheli. Component selection and matching for IP-based design. *DATE 2000 proceedings*, Munich, Germany, 2000.
- [9] P.Chawla, EDaptive Computing, Inc, P.Alexander and R.Vemuri, University of Cincinnati. Search and retrieval tool to enable system design through IP reuse.
- [10] K.D.Cooper and T.J.Harvey. Compiler controlled memory. *ASPLOS VIII CA*, October 1998.
- [11] S.Coumeri and D.Thomas. Memory modeling for system synthesis. *ISLPED '98*, pp.179-194, August 1998.
- [12] V.Delaluz, M.Kandemir, N.Vijaykrishnan and M.J.Irwin. Energy-oriented compiler optimizations for partitioned memory architectures. *CASES*, November 2000
- [13] A.H.Farrahi, G.E.Tellez, M.Sarrafazadeh. Memory segmentation to exploit sleep mode operation. *Proceedings of the Design Automation Conference*, pp. 36-41, June 1995
- [14] M.Garey and D.Johnson. Computers and intractability: A guide to the theory of NP-completeness. *W.H.Freeman and Co.*, New York, 1979.
- [15] C.H.Gebotys. Low Energy Memory and Register Allocation using network flow. *DAC97*, Anaheim, CA, 1997.

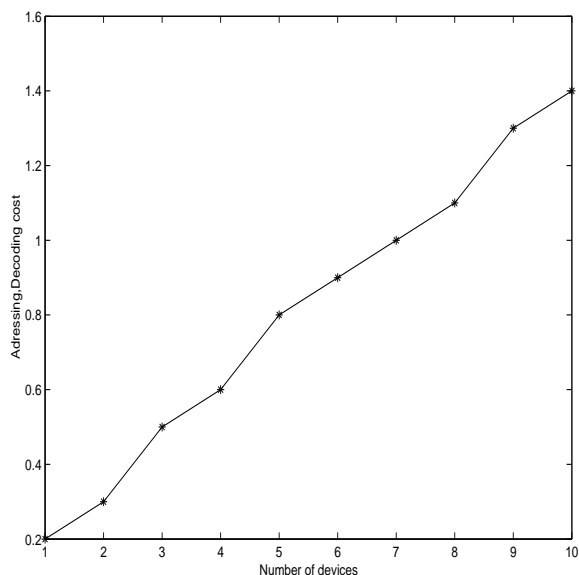


Figure 4: Plot of addressing cost versus partitions

[16] N.B.I.Hajji, C. Polychronopoulos and G.Stamoulis. Architectural and compiler support for energy reduction in the memory hierarchy of high performance microprocessors. *ISLPED98*, Monterey, CA, 1998.

[17] M.J.Irwin. Low power design for systems on a chip - a tutorial. *12th Annual IEEE ASIC/SOC workshop*, September 2000.

[18] M.Kandemir, N.Vijaykrishnan, M.J.Irwin, W.Ye and I.Demirkiran. Register relabeling: A post compilation technique for energy reduction. In *workshop on Compilers and Operating Systems for Low Power 2000 (COLP'00)*

[19] R.Levy, B.Crilly, B.Narahari and R.Simha. Memory issues in power-aware design of embedded systems: an overview. *CASES*, 1999.

[20] J.Lorch and A.J.Smith. Software strategies for portable computer energy management. *IEEE Personal Communication Magazine*.

[21] J.Lorch and A.J.Smith Reducing processor power consumption by improving processor time management in a single-user operating system. *Proc. of the 2nd ACM International Conf. on Mobile Computing (MOBICOM96)*, 143-154, Nov.1996

[22] Y-Hsiang Lu, L.Benini, G.De Micheli. Operating-system directed power reduction. *International Symposium on Low Power Electronics and Design*, Stanford University, 37-42, July 2000.

[23] S.Martello and P. Toth. Knapsack Problems. *Wiley and Sons*, 1993..

[24] S.S Muchnik. Advanced compiler design implementation. *Morgan Kaufmann*, San Francisco, California, 1997.

[25] P.R.Panda, N.D.Dutt and A.Nicolaue. On-chip vs off chip memory: the data partitioning problem in embedded processor based systems. *ACM transactions on design automation of electronic systems*, 5(3), July 2000.

[26] Rambus Inc. www.rambus.com.

[27] K.Roy and M.C.Johnson. Software design for low power. In *NATO Advanced Study Institute on Low Power Design in Deep Submicron Electronics*, Aug. 1996, NATO ASI Series.

[28] R.P.Stanley. Enumerative Combinatorics, Vol. I. *Wadsworth and Brooks*, 1986.

[29] W-T.Shieu and C.Chakrabarti. Memory design and exploration for low power embedded systems. *Design Automation Conference Design Automation Conference*, 1999.

[30] C.L.Su, C.Y.Tsui and A.M.Despain. Low power architecture design and compilation techniques for high performance processors. *IEEE COMPCON*, Feb. 1994.

[31] S-F Li, R.Sutton and J.Rabaey. Low power operating system for heterogeneous wireless communication systems. *Workshop on Compilers and Operating Systems for Low Power 2001*, September 2001.

[32] TenSilica Inc., www.tensilica.com.

[33] V.Tiwari, S.Mallik and A.Wolfe. Compilation techniques for low energy: an overview. *Symposium on Low Power Electronics*, San Diego, October 1994.

[34] A.Vahdat, A.Lebeck and C.S.Ellis. Every Joule is precious : The case for revisiting operating system design for energy efficiency. *Proc. of the 9th ACM SIGOPS European workshop*, September 2000.

[35] T.Zhang, L.Benini and G.D.Micheli. Component selection and matching for IP-based design. *DATE 2001 proceedings*, Munich, Germany, 2000.