PROJECT FILES:

There are 10 files in this repository:

- ***Area_costs.h, energy_costs.h*** *and* ***performance_costs.h:*** These contain data gathered from the memory simulator CACTI(See the github repo at [1] for more info). The data in these header files contain the needed information for the cost function that will evaluate the cost of the bankings in the algorithm.
- ***Trace.h:*** The array in this file contains the memory access trace to the main signal in the "embedded" program after running it. This array is used to determine the cost of a banking too, depending on how many times the element is to be accessed.
- ***Dyn_ours_area.cpp, dyn_ours_energy.cpp*** **and** ***dyn_ours_perf.cpp:*** Each of these files is the implementation of the algorithm with a different cost objective, to either optimize the chip area, the energy consumption or the performance, respectively. So the only difference between these three files is the cost function.
- ***Area_optimization_results.cpp, energy_optimization_results.cpp*** **and** ***performance_optimization_results.cpp*** **in the** ***results*** **folder*:*** These files contain the results of running the program with area, energy and performance optimization, respectively.

To run the algorithm, download the repo to your machine then compile run any of the *.cpp* programs to get the resulting files with the banking positions and the cost of each banking configuration.

A DYNAMIC PROGRAMMING APPROACH

This project explores the dynamic programming technique to solve the scratchpad banking problem. It exhibits a novel algorithm to solve the scratchpad partitioning problem. Generally tuned toward data-intensive applications, the dynamic programming algorithm discussed in this chapter conquers many of the shortcomings associated with previous algorithms that tackled the same problems. In this file, I will explain how it works and why I believe it is a major enhancement over previous techniques. For more details on the problem background and previous attempts to solve is visit: http://dar.aucegypt.edu/handle/10526/3779?show=full

So, here, I will first, study the feasibility of using a dynamic programming approach to solve the partitioning problem. If you are not interested in the theoretical proof of the optimality and applicability of the technique, you can skip over to part 22 of this file where the program is explained.

1.1 About Dynamic Programming

This section verifies the optimality and applicability of a dynamic programming method to our problem. At the end of this section, a brief comparison between dynamic and linear programming is made to justify the selection of the former over the later.

1.1.1 Proof of Optimality

In dynamic programming approaches, all possible cases are evaluated just like a brute force algorithm. It is essentially a *"smart"* recursion. Often extra work doesn't have to be repeated if solutions to subproblems are cached after they are solved. Therefore, the only difference from exhaustive algorithms which makes dynamic programming significantly more efficient is the reuse of solutions to subproblems that have already been computed. That is why dynamic programming is guaranteed to always achieve the global optimal solution to any problem that has proven to be applicable to this method of solution.

1.1.2 Feasibility of Applying Dynamic Programming to the Partitioning Problem

Dynamic programming solves problems by combining the solutions to sub-problems using a tabular method. Unlike the divide-and-conquer method, dynamic programming solves problems where sub-problems are not independent, that is, subproblems share subproblems [5]. It solves a subproblem only once and saves its answer in a table, thereby avoiding the work of re-computing the answer every time the sub-problem is encountered.

In general, dynamic programming algorithms are typically applied to optimization problems; which is, in fact, the case in our scratchpad partitioning problem. In the scratchpad partitioning problem, each solution has "a value" representing the energy, performance and/or area costs of the partitioning represented in that solution. We wish to find a solution with the optimal value, that is, minimum cost. It is important to notice that we are interested in finding *an* optimal solution to the problem, since it is possible to have several solutions that achieve the optimal value.

It seems from this introduction that our scratchpad partitioning problem is a perfect fit for a dynamic programming approach. However, let us prove it from an engineering perspective before making a hasty judgment.

1.1.3 Does the scratchpad partitioning problem have the elements of dynamic programming?

According to T. Cormen *et al. [5*] , there are two key ingredients that an optimization problem must have in order for dynamic programming to be applicable: optimal substructure and overlapping subproblem.

1.1.3.1 Optimal Substructure

A problem is said to have an optimal substructure if an optimal solution to the problem contains within it optimal solutions to subproblems. Whenever a problem exhibits optimal substructure, it is a positive sign that dynamic programming might apply. In dynamic programming, we build an optimal solution from optimal solutions to subproblems.

To discover the optimal substructure property, we need to prove that a given solution to a problem cannot be optimal unless it uses optimal solutions to the contained subproblems. Applying this method of proof to our scratchpad partitioning problem, we let P_{ij} denote the optimal partitioning solution to the address range from i to j. Therefore, for any partitioning for the address range Q_{ij}, the following is true:

$cost(Q_{ij}) >= cost(P_{ij})$ where $Q_{ij} \neq P_{ij}$

Assume that P_{ij} optimal solution divides the range i to j at address k, and according to the optimal substructure property, solutions to the subproblems must be optimal as well, hence:

$Cost(P_{ij}) = cost(P_{ik}) + cost(P_{kj})$[1]

where P_{ik} and P_{kj} are both optimal solutions for address ranges i to k and k to j, respectively.

Now, using proof by contradiction, we try to prove that we can get an optimal solution that costs less than—or equal to—P_{ij} without using optimal solutions to the subproblems.

Assume that there exists another optimal solution Q_{ij} that makes a partition at k but does not take the optimal solutions for the subproblems i to k and k to j. That is:

$Cost(Q_{ij}) = cost(Q_{ik}) + cost(Q_{kj})$

where Q_{ik} and Q_{kj} are not optimal solutions for address ranges i to k and k to j, respectively. That is, $cost(Qik) > cost(Pik)$ and $cost(Qik) > cost(Pkj)$. Consequently:

$cost(Q_{ik}) + cost(Q_{kj}) > cost(P_{ik}) + cost(P_{kj})$

$cost(Q_{ij}) > cost(P_{ij})$ --Contradiction!

which contradicts our previous conjecture of Q_{ij} optimality. Thus, an optimal partitioning for any address range must use optimal partitioning for sub-ranges, too. We have now proved the optimal structure property for the scratchpad partitioning problem.

1.1.3.2 Overlapping Subproblems

The second ingredient that an optimization problem must have for dynamic programming to apply is that the space of subproblems must be “small” in the sense that a recursive algorithm for the problem solves the same subproblems over and over, rather than always generating new subproblems.

Typically, the total number of distinct subproblems is a polynomial in the input size. When a recursive algorithm revisits the same problem repeatedly, we say that the optimization problem has overlapping subproblems. In contrast, a

[1] In this calculation of partitioning cost, we are ignoring the cost of the overhead produced by adding an extra partitioning to the solution since, even if this overhead is not to be neglected, it does not harm our argument here since it will always be added for any partitioning whether it is optimal or not.

problem for which a divide and conquer approach is suitable usually generates brand-new problems at each step of the recursion.

Dynamic-programming algorithms typically take advantage of overlapping subproblems by solving each subproblem once and then storing the solution in a table where it can be looked up when needed, using constant time per lookup.

It is obvious that the scratchpad partitioning problem has overlapping subproblems. Figure 1 shows a trivial example of a memory space of only 4 addresses and the space subproblems under it. The dimmed subproblems denote an overlap with one of the undimmed subproblems; the overlapping of subproblems can be clearly noticed.

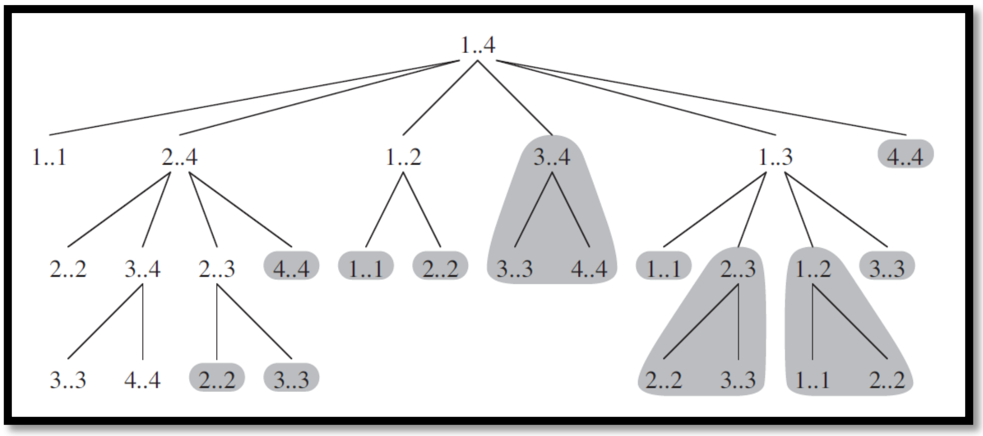

Figure 1 The recursion tree for computing scratchpad partitioning solution for an address range 1 to 4. The darkened subtrees are overlapping subproblems. Figure from [5].

1.1.4 Dynamic Programming versus Linear Programming

A main advantage of dynamic programming is that it allows for any cost function to be used in optimization. Linear programming, on the other hand, restricts the optimization problem to the case where the optimization function is subject to only linear constraints. Linear programming becomes applicable only if we can specify the objective as a linear function of certain variables, and if we can specify the constraints on resources as equalities or inequalities on those variables [6]. Since our optimization problem is subject to non-linear constraints—namely, energy consumption, access time delay and chip area—considering linear programming for our problem is therefore not a feasible option.

After proving that dynamic programming can actually find optimal solutions without restricting the optimization functions to be linear and that our scratchpad partitioning problem has both the optimal substructure and overlapping subproblems features, we can now confidently consider a dynamic programming approach to solve it. .

2 THE PROPOSED DYNAMIC PROGRAMMING APPROACH

The major accomplishments of this algorithm are mainly related to cost evaluation and overall efficiency. Cost evaluation in this approach is built upon memory simulation results of energy and time obtained from CACTI [3] as described in [1]. The drastic improvement of time and space complexity in this approach are a result of eliminating the need to compute the area at every step in the previous approach by, instead, translating a given area constraint to its equivalent SPM size (in words), as will be seen shortly. Also, the genuine transformation of the main processing matrix to contain drastically less number of elements cuts down the time needed to process each element.

2.1 Inputs

The algorithm takes as input the number N of addresses in the sorted execution trace, along with the actual address $addr_i$ and the number of accesses rw_i associated with the ith sorted trace entry and the maximum area allocated for the scratchpad in total, denoted θ. It is assumed that α_i is expressed in 32-bit words, while θ is given in nm^2.

It is possible to determine the maximum number of partitions possible by finding the maximum value of M that satisfies the formula:

$$\sum_{i=1}^{M} \left[SPM\_area(\frac{\delta . N}{M}) + \Delta A_{i,i+1} \right] \leq \theta \tag{1}$$

where:

M	is the number of maximum number of partitions; desired to maximize its value.
$SPM\_area(n)$	is the area cost of a partition of size n bytes. This function obtains its values from Cacti; the memory simulation tool.
δ	the word size in bytes, assumed to be 4 bytes per word.
N	is the total number of addresses in the input trace.
ΔA	is the area overhead associated with an extra partition; obtained from Cacti.

θ is the given area constrained for the whole scratchpad memory module.

After finding the maximum number of partitions (M) possible under the given area constraint, the algorithm calculates the minimum partition size possible(denoted Φ and measured in 32-bit words) under the given area constraint by a simple division:

$$\Phi = \left\lfloor \frac{N}{M} \right\rfloor \qquad where\ M, N, \Phi \in Z^{+} \tag{2}$$

In other words, the algorithm translates the given area constraint (in *nm*^{2}) to a simpler, *more quantized* constraint to easily apply on the dynamic table calculations. This simpler constraint is the size of the partition that cannot be further partitioned. By doing that, we simplified the search process by pruning all inconvenient partitioning possibilities that yield a partition size smaller than Φ—and that are most likely to violate the given area constraint.

Starting from this point, the given area will not be needed since we now know the smallest possible partition size in words (Φ) beyond which the area constraint is violated. Therefore, the complications created by the area cost function in Angiolini *et al.*'s algorithm are now avoided.

Even though constraining the partition size to match the maximum area allowable for the scratchpad module may give the impression that the resulting area cost of the generated solution leaves a tiny margin—if at all--before exceeding the given area limit, this does not necessarily imply that optimization for area is not possible. On the contrary, the idea of having the area constraint as a hard limit for partition size gives a chance to best utilize the available area when area optimization is not the target, while still allowing for area optimization, if required, in cost calculation.

2.2 The Algorithm

After refining our inputs, our objective now is to select a set of mutually disjoint address ranges(each range forming a partition), the overall cost of which is maximized, and any partition size is not less than Φ. The algorithm below shows a pseudo code of the proposed dynamic programming algorithm.

```
1  for $j := \Phi$ to $N$ do
2  for $i := 1$ to $N$ and $k := j$ to $N$ do
3      $P[i][k] = rw(i,k) \,.\, cost(i,k)$
4      if $k-i < 2 \,.\, \Phi$ then
5          continue
6      for $l := i + \Phi$ to $k - \Phi$ do
7          if $P[i][l] + P[l+1][k] < P[i][k]$ then
8              $P[i][k] := P[i][l] + P[l+1][k]$
9              $S[i][k] := l$
```

Pseudo code for the novel dynamic programming algorithm.

For each pair $[i, j]$, $1 \leq i \leq k \leq N$, let $rw(i, k)$ be the total number of accesses to the memory range starting from location *i* and ending at location *k*, and let $cost(i, k)$ be the cost associated with a single access to a memory location in the range of addresses from the i^{th} to the k^{th}. See the next section for more details on cost calculation.

The algorithm's main structures are two $N \times N$ matrices, *P* and *S*, representing the costs and partitioning positions, respectively, see Figure 1. Every cell in *P* contains the overall cost according to one of the strategies mentioned above. The cell in row *i* and column *k* contains the cost of the best partitioning found so far, starting from the i^{th} address up to the k^{th} address. Initially, all entries of the matrix *S* are set to 0. If that best partitioning was achieved by making a partition rather than having it as a single bank, the matrix *S* is updated with data about the partitioning position that charges this cost.

The entries in *P* and *S* are filled out diagonally in an ascending order of range size represented by the index *j* in line 1. At any time of processing, *j* is always equal to k-i, which indicates the current range size being explored.

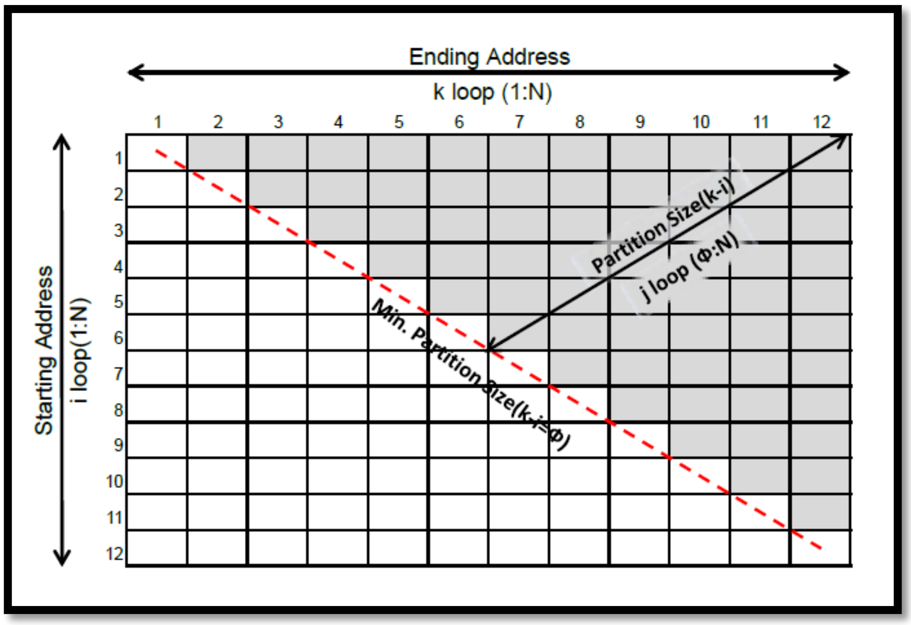

Figure 1 Principal structure of the P and S matrices presented in the dynamic programming algorithm.

In light of the pseudo-code, we can roughly summarize the algorithm as follows: it actually checks for all possible partition sizes starting from the minimum partition size, Φ, up to the maximum possible partition size, N, (line 1) and all possible ranges from *i* to *k* that give this size(line 2) whether storing this range in one partition(line 3) is more profitable than partitioning at any possible partitioning position(lines 6 through 9). Lines 4 and 5 make sure that the partition at hand can be partitioned to smaller banks without violating the constraint before entering the comparison loop.

2.3 Cost Calculation

The cost function is represented by the value returned from querying Cacti memory simulator database—see [1] https://github.com/abuaesh/CACTI-Batch-Generator . The query to Cacti's database could be for energy cost, performance cost and/or area cost per memory access. The cost function referenced in line 3 of the pseudo code deploys these value(s) to set the costs of unpartitioned ranges in the processing matrix *P* for a memory bank of the size (*k*-*i*).

The cost function gives the flexibility to set the values directly as energy, performance or area costs, or a weighted sum of any of them in case of multiple objectives, depending on the user's optimization target.

2.4 Results and Discussion

The results in this section were obtained by running a C++ implementation of the algorithm on a PC with an i5 core at a 2.5 GHz processor. The program was tested using the same benchmark used to test the previous approaches with different values of Φ and different granularities. The optimal partitioning cut sets found by this algorithm are shown in the table in

Appendix B – Testing Results, along with data captured for each generated solution; like the total energy consumption, the total time cost of the partitioning and the total area needed for the partitioning. The memory simulation data were acquired for the 32 nm^2 technology, scratchpad memory type.

The granularity (denoted G) specifies the exploration accuracy for each partition possibility. In general, specifying a granularity can be viewed as increasing the word width. Increasing the granularity of the search not only cuts down the memory requirements as mentioned above, but also significantly accelerates the algorithm's execution time.

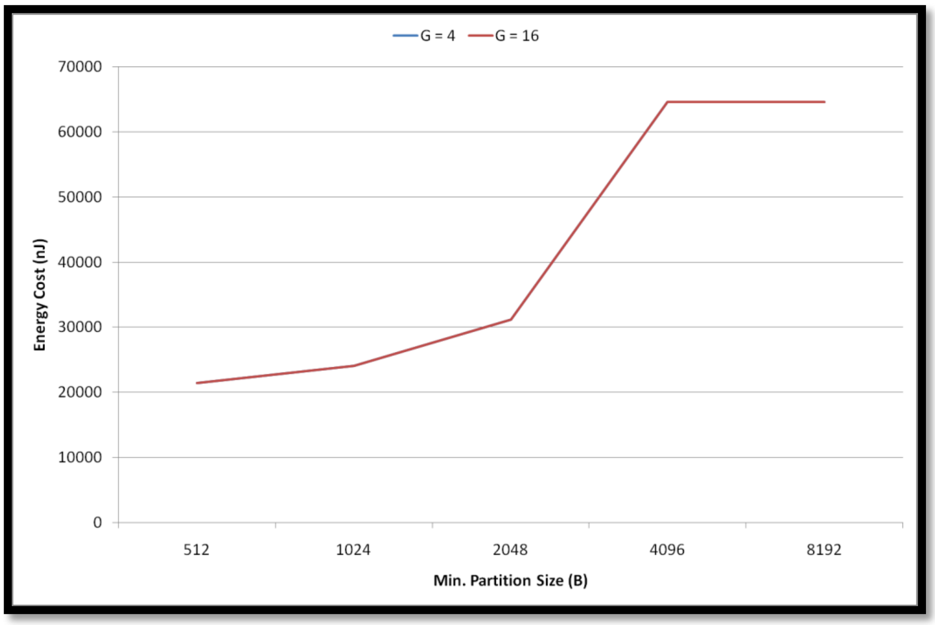

Figure 2 Energy costs obtained for different values of Φ and G. Even though energy cost is expected to increase with the partitions sizes increase; since accessing larger banks of memory consumes more energy than accessing banks of smaller sizes, the figure suggests a negligible difference between the energy results.

Even though this section discussed the results obtained while running for energy optimization since the main objective of our problem is to optimize energy consumption, our cost function can be oriented to optimize for any of the parameters of interest: energy, performance and area.

Appendix B – Testing Results shows the results obtained for running this algorithm for performance optimization. Optimizing for area alone was expected to give a monolithic solution. Tests asserted this expected conclusion because, in practice, memory modules of bigger sizes are denser than smaller modules in terms of area **Error! Reference source not found.**, so it's always better for area to use less memory banks.

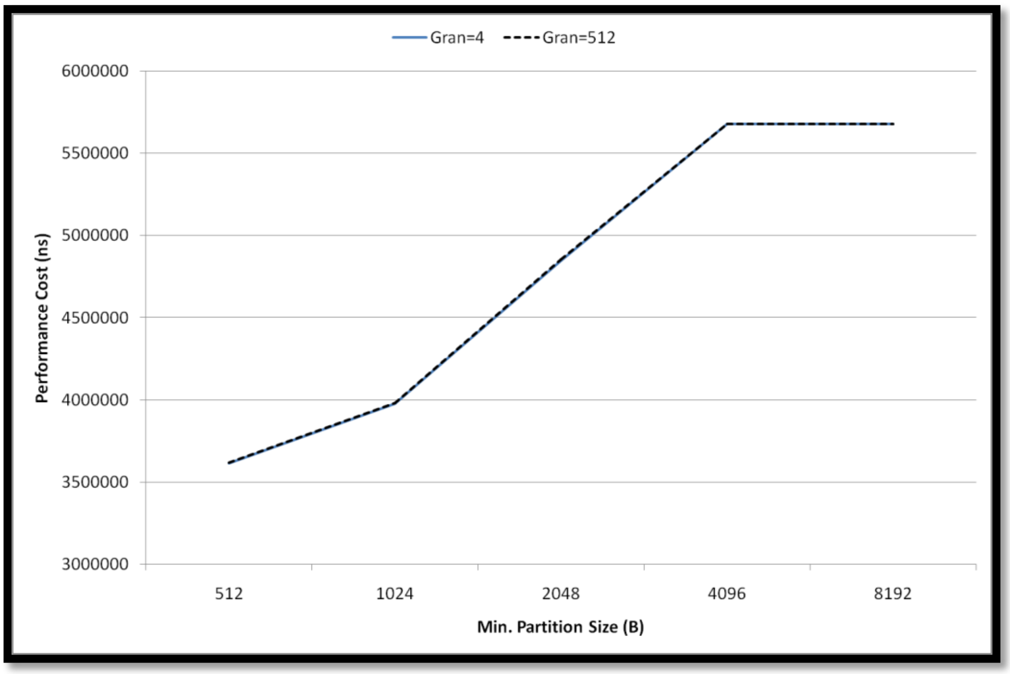

Figure 3 Performance costs obtained for different values of *Φ* and *G*. Performance exhibits a similar behavior to energy. Analogous to energy consumption, accessing larger banks of memory takes more time than accessing banks of smaller sizes. We may roughly estimate that optimizing for energy produces somewhat efficient solutions. This can also be noticed especially when compared with the results obtained from performance optimization tests; see page 19.

However, we may want to optimize for area and another parameter at the same time; for example, optimizing for area and energy at the same time. A method to account for multiple competing objectives in the cost function is discussed in [4]. The method can be mainly summarized in setting weights for each optimization parameter (energy, performance and area) according to the relative improvement required for the corresponding objective. In such a case the solution may not always give a monolithic solution—depending on the specified weights-- even though optimization for area is accounted for.

2.5 Complexity and Analysis

One of the main accomplishments we claim for this approach over the previous dynamic programming technique is the enhanced space and time complexity. The following two sections justify this claim and explain techniques that can help in further improving the algorithm's complexity.

2.5.1 Space Complexity:

The memory requirements for this algorithm are influenced by the sizes of matrices P and S. Even though **Error! Reference source not found.** shows that P and S densities are always less than 0.5, we are going to consider the memory taken by the structures as a whole, considering even the sparse elements.

Suppose we use a single matrix whose elements are data structures holding profit and address information at the same time, and suppose also that we need 4 bytes to store addressing information and 8 bytes to store the profit. A single element in this matrix, therefore, takes 12 bytes of memory. Consequently, the memory required to process an N-word trace is $12.N.N$ bytes, resulting in a space complexity of $\theta(N^2)$. Comparing this complexity with Angiolini *et al.*'s algorithm, we notice that our algorithm improves the memory requirement by an order of magnitude.

It is important to mention that the memory actually needed for processing is far less than the amount considered above, since the parts of the matrix actually used in processing highly depend on Φ, the minimum partition size. The sparse parts of the matrix may be eliminated by performing additional vectors —or pointers— manipulation that may affix simple calculations to elements dereferencing.

Another dodge that significantly reduces the memory usage is playing with the word width, or the granularity of the search. This means that instead of allocating an $N \times N$ matrix, we can allocate only $\frac{N}{G} \times \frac{N}{G}$, where G is the selected granularity. While not sensibly improving the overall complexity, this reduction does make a significant outcome in practical measurements of both memory consumption and time of execution, as can be noticed from Figure 4 and Figure 5.

Experiments showed that the granularity can have any value starting from 1(which means all partitioning possibilities are tested) up to the minimum partition size (Φ). A granularity value greater than Φ, in addition to its negative effect on the solution's quality, overrides the advantage of Φ in the first place. Figure 4 shows the memory requirements obtained from processing an 8 KB trace(N=8192) tested with different word widthes. The memory requirements are the same for different minimum partition sizes since the allocation of the matrices depend on the trace size and the granularity of the search.

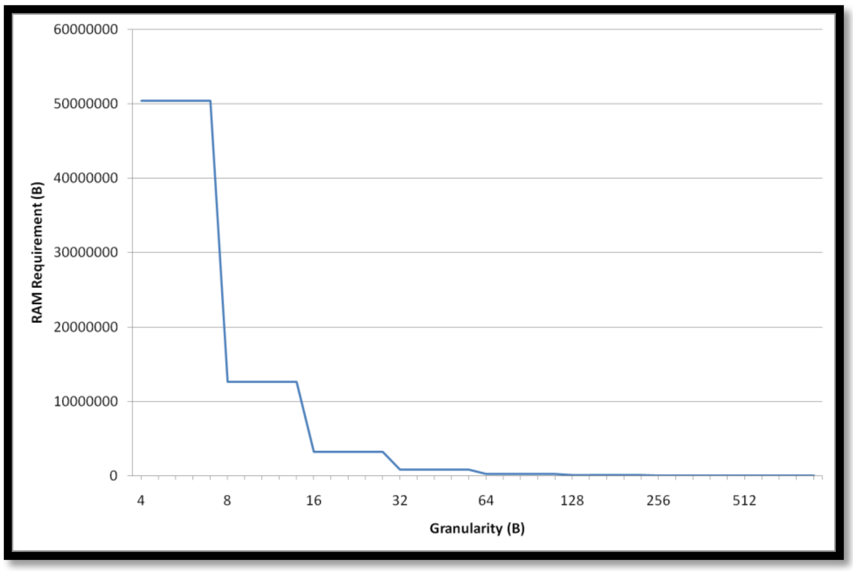

Figure 4 Memory requirements for an 8 KB trace(N=8192) tested with different granularities(or word widthes). The memory requirements are the same for different minimum partition sizes.

2.5.2 Time Complexity:

A look at the pseudo code presented in **Error! Reference source not found.** reveals a worst case time complexity of $\theta(N^3)$. This efficiency may seem similar to that of Angiolini's algorithm discussed earlier in this chapter with the case of $C = N$. However, a more careful investigation shows that we may comfortably say that the efficiency is better than $O(N^3)$.

Let us start by examining Figure 1 of the main processing structure in our dynamic programming approach. For instance, more than half of the processing matrix is not processed-- the area below the dotted line-- due to the fact that matrix cells where $k\text{-}i < \Phi$ are not explored. In other words, a worst case scenario when the area constraint is loose enough to allow partitions as small as 1 word in size, the processing does not exceed the main diagonal ; which is less than half of the matrix. Of course, the tighter the area constraint, the larger the minimum partition size, the more sparse the matrix *P* becomes and, consequently, the faster the execution.

Now, observe the pseudo code in Pseudo code for the novel dynamic programming algorithm.. Lines 1, 2 and 6 are guaranteed to loop exactly *N-1* times in a worst case scenario with *Φ=1*. Loop *j* in line 1 loops for *N-Φ* iterations. For each iteration of *j*, loop *i* and *k* in line 2 is executed *N-j* times—that is, *N-Φ* times at first and keeps decreasing till it gets executed only once when *j* reaches *N*. Finally, the inner most loop of *l* at line 6 executes *k-i-2Φ* under the condition that *k-i > 2Φ*. This means that loop *l* is skipped for unpartitionable sizes and otherwise loops for a maximum of *N-2Φ* iterations. This gives an overall efficiency of $\theta((N-\phi)^3)$. As a result and as will be confirmed by the testing results in the coming section, a larger value of *Φ* –reflecting a tighter area constraint--speeds up the execution.

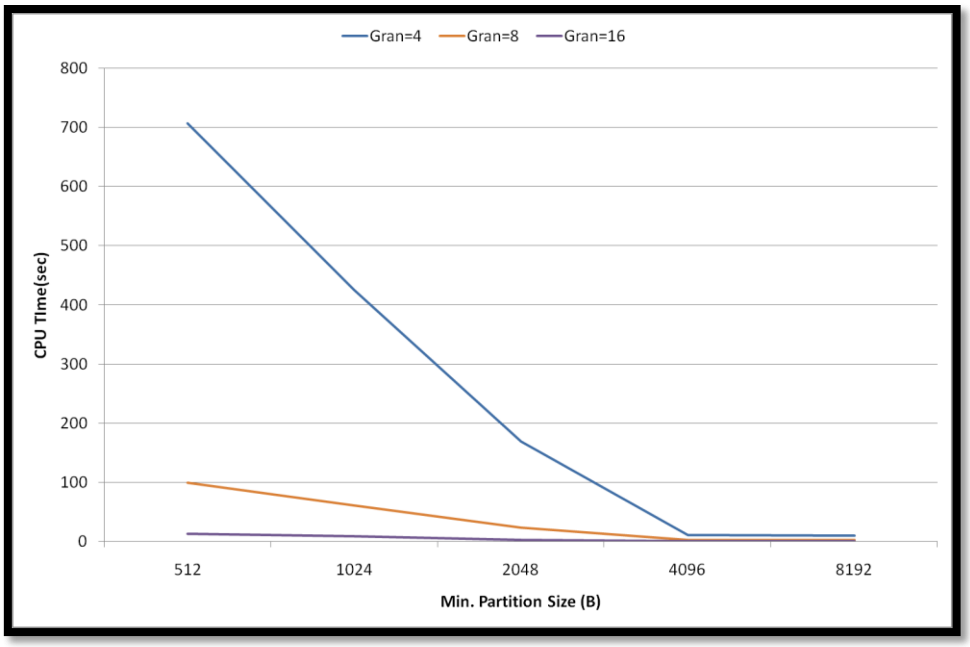

Figure 5 CPU running time for an 8 KB trace(N=8192) tested with different word width. The worst time was 10 minutes taken by word width of 4 bytes.

A single run with granularity of 2 bytes was performed and it took 3 hours to find a solution with almost the same energy consumption as the higher granularity runs. Based on this result, we decided to stop our tests at granularity=8 bytes since it was not reasonable to continue testing with finer granularity.

Other readings to measure the algorithm's efficiency were also collected, like the CPU time taken to find the partitioning as well as the memory space needed to complete the processing. See Figure 3 below.

2.6 Limitations and Shortcomings

Even though this algorithm has recorded significant improvements over the previously discussed algorithms in terms of solution quality with respect to time and space efficiency, we are still ajar from finding an algorithm that finds a universal optimal solution in reasonable time.

The computation time, to begin with, drastically increases when the SPM size is large relative to the minimum bank size and granularity. However, with the help of the granularity trick, we were able to find "a solution" in reasonable time. There is no guaranty to unconditional optimality unless if granularity and minimum partition size were set to 1; in which case the program's time and space requirements significantly increase. The reason is because of the constraint on partitions sizes. The preliminary step of translating the area constraint into a partition size constraint, despite its relaxing effect on the problem, prevents the algorithm from exploring possibly-optimal solutions with smaller partition sizes and thus giving a chance that optimal solution is missed.

Another drawback of the current implementation of the algorithm is limiting the granularity (G) to be a common factor between N and Φ. This is because we restrained the size of our processing matrix to be $\frac{N}{G} \times \frac{N}{G}$. As a result, the algorithm is prevented from exploring cut positions that are not divisible by G. However, while significantly improving the algorithm's performance, testing results suggest negligible energy degradation due to increasing granularity within the given value of Φ, see Figure 2.

Another limitation is the inability to determine the idleness of data stored in the banks, a technique that helps in further saving energy by putting the banks to sleep when they are not accessed to save static energy.

The question that remains open is whether it is possible to devise an algorithm that finds a universally-optimal solution to our partitioning problem without unrealistic demands of time and computing resources? The thesis in discusses an interesting algorithm that approximates solutions by means of a backtracking approach.

REFERENCES

[1] Github- CACTI Batch Generator and Simulator: https://github.com/abuaesh/CACTI-Batch-Generator

[2] F. Angiolini, L. Benini, and A. Caprara, Polynomial time algorithm for on-chip scratchpad memory partitioning, *Proceedings of the International Conference on Compilers, Architectures and Synthesis for Embedded Systems, CASES 2003*, San Jose, California, Nov., 2003.

[3] CACTI 6.5 [Online] http://www.hpl.hp.com/research/cacti/

[4] F. Balasa, I.I. Luican, N. Abuaesh, and C.V. Gingu, "Compiler-directed memory hierarchy design for low-energy embedded systems," *Proc. of the 11th ACM/IEEE Int. Conf. on Formal Methods and Models for Codesign*, Portland OR, Oct. 2013, pp. 147-156.

[5] T. H. Cormen, C. E. Leiserson, R. L. Rivest and C. Stein, "Dynamic Programming," in Introduction to Algorithms, 2nd ed, Cambridge, MA: MIT Press, 2001, ch. 15, sec. 3, pp. 339–345.

[6] T. H. Cormen, C. E. Leiserson, R. L. Rivest and C. Stein, "Linear Programming," in Introduction to Algorithms, 2nd ed, Cambridge, MA: MIT Press, 2001, ch. 29, pp. 843.

[7]

APPENDIX A – SIMULATION DATA

This appendix lists memory data concerning power, access times and area used in this research and obtained using an interface to CACTI 6.5.

The data can be found at the following folder:

Appendices\CACTI Results\CACTI Simulation Data-SPM.xlsx

The Excel file contains four tabs for data obtained with different technologies: 32nm, 45nm, 68nm and 90 nm.

APPENDIX B – TESTING RESULTS

The optimal partitioning cut sets found by the algorithm are shown in this appendix along with data captured for each generated solution; like the total energy consumption, the total time cost of the partitioning and the total area needed for the partitioning. The memory simulation data were acquired for the 32 nm^2 technology, scratchpad memory type.

The results in this appendix were obtained by running a C++ implementation of the algorithm on a PC with an i5 core at a 2.5 GHz processor. The program was tested using the same benchmark used to test the previous approaches with different values of Φ and granularity.

The data can be found at the following folder:

Appendices\DP Algorithm 5.1 Results\Algorithm 5.1 Results.xlsx

The Excel file contains three tabs for data obtained with each optimization target: energy, performance and area.