Mempack: An Order of Magnitude Reduction in the Cost, Risk, and Time for Memory Compiler Certification

Kartik Mohanram†, Matthew Wartell†, and Sundar Iyer‡

†Department of Electrical and Computer Engineering, University of Pittsburgh, PA

‡Memoir Systems, Santa Clara, CA

kmram@pitt.edu maw192@pitt.edu sundaes@memoir-systems.com

Abstract

Advances in memory compiler technology have helped accelerate the integration of hundreds of unique embedded memory macros in contemporary low-power, high-speed SoCs. The heavy use of compiled memories poses multiple challenges on the characterization, validation, and reliability fronts. This motivates solutions that can reduce overall cost, time, and risk to certify memories through the identification of a reduced set of 'fundamental' memory macros that can be used one or more times to realize all memory instances in the design. This paper describes MemPack, a fast, general method based upon the classical change-making algorithm for the identification of such fundamental memory macros. By relaxing the need for exact realization of memories and tolerating wastage within the context of change-making, MemPack enables tradeoffs between memory capacity and reduction in the number of fundamental macros. It also controls multiplexing and instantiation costs, minimizing the impact on critical path delay and address line loading. Results on industrial and synthetic benchmarks for three different optimization objectives (performance, balance, and minimization) show that MemPack is effective in identifying fundamental sets that are as much as 16× smaller than the original set for 0.8–4.7% wasted bits.

1. Introduction

The number of on-die memory instances per design is increasing rapidly, and it is projected that embedded memories will occupy over 70% of the die-area in future SoCs [1]. Memory compilers are CAD tools that can quickly generate hundreds or even thousands of unique memories differing in depth and width, organization (banks, ports, etc.), density, performance, and power across SRAM and embedded DRAM technologies [2, 3].

As the number of unique memory instances per chip increases, it becomes prohibitively expensive to fabricate and validate each memory instance over a range of process, voltage, and temperature corners (PVTs). Furthermore, specifications can change throughout the design cycle, and it is often impractical to validate critical instances that are liable to change later in the design cycle. In practice, memory compilers create models by fitting data derived from a small sample of validated memory instances. These models are used to interpolate and characterize each of the instances that they compile for a design. Even with careful guardbanding, such memory characterization approaches are ad-hoc and cannot accurately model the detailed timing, noise, and power data necessary for electrical sign-off, forcing re-spins, delaying schedules, and increasing the total cost of design. Furthermore, the system-level effects of online errors in these memories can be far reaching, especially given the recent trend toward highly integrated hardware platforms with multi-threaded, parallel execution environments [4, 5].

This research was supported by grants CCF-1211099 and CCF-1208933 from the National Science Foundation.

In this paper, we describe MemPack, a methodology to systematically reduce the number of distinct memory macros that are used by a memory compiler to realize all the memory instances in a design. It is motivated by the premise that if all the memory instances in an SoC design can be realized from a reduced set of 'fundamental' blocks, the characterization and validation costs can be reduced significantly. Such an approach will add to improved timing convergence and design validation, as well as positively reduce effort at all stages of the pre- and post-silicon characterization and validation flow. Furthermore, by relying on a small number of unique memory blocks that can be intensively characterized, a corresponding increase in field reliability can be achieved. To the best of our knowledge, this is the first attempt to propose a systematic scalable solution to this increasingly important problem.

For a set of memory instances M, each instance $m \in M$ is given by the product $d \times w$ where d is its depth and w is its width. MemPack seeks a reduced set M' (note that it is not necessary that $M' \subset M$) from which all $m \in M$ can be realized through two fundamental compiler operations: multiplexing and instantiation. Multiplexing is the operation of compiling larger memory depths using one or more macros of equal or smaller depth, whereas instantiation is the operation of compiling wider macros using one or more macros of equal or smaller width. Typically, compilers generate the m_i by interpolating across a range of pre-characterized memory macros, and adopt a greedy strategy (e.g., largest macro first) during compilation. More generally, memory compilation is a two-dimensional combinatorial problem within the general class of $\mathcal{NP}$-hard knapsack and bin-packing problems. When the depth and width dimensions are decoupled, it can be shown that optimal solutions in each dimension can be obtained through the change-making ($\mathcal{CM}$) algorithm, which is a knapsack-type problem.

Classical $\mathcal{CM}$ requires a set of denominations and provides perfect packings. Although this is useful when perfect packings are desirable, it can result in excessive fragmentation unless a large range of denominations (i.e., fundamental memory macros) are available. However, since our objective is to reduce the number of fundamental memory macros, we relax the need for exact realization of memories in this paper. For example, along the depth dimension, such a relaxation can yield packings of the form 63K = [32K,32K] where an exact packer would yield 63K = [7K,8K,16K,32K]. The core algorithms in Mempack extend classical $\mathcal{CM}$ to explicitly tradeoff area, i.e., memory capacity for reduction in the number of fundamental memory macros required by the memory compiler. Mempack also controls the extent of multiplexing and instantiation during the composition process to ensure that the critical path delay is not impacted by multiplexing during depth packing and that loading on the address lines is not increased due to memory fragmentation during width packing.

The core methods in MemPack decouple the depth and width dimensions, i.e., MemPack first processes the set $D = \{d \mid d \times w \in M\}$ to determine a set of fundamental depth macros D' that can be used to realize all depths (irrespective of width) in M. In the second phase, for each $d' \in D'$, Mempack consolidates the unique widths across all depths from D and invokes a pass of the core packing algorithm. Thus, for every $d' \in D'$, MemPack obtains a reduced set of widths $W'_{d'}$ by applying the $\mathcal{CM}$-based algorithm. The final set of fundamental macros M' is given by $\bigcup_{d' \in D'} \{d'\} \times W'_{d'}$.

We validate the effectiveness of MemPack on data from real-world designs, and also on synthetic data generated to model this real-world data from small, medium, and large designs. We present optimization results on these datasets for three different optimization objectives: (i) performance, that gives priority to richer fundamental sets with low muxing and instantiation costs, (ii) balance, that attempts to reconcile performance against minimization of the fundamental macro set, and (iii) minimization, which prioritizes fundamental set minimization at the expense of wasted bits. Our results show that MemPack is capable of identifying fundamental sets that are up to 16× smaller for 0.8–4.7% wasted bits, providing a rich space of alternatives.

2. Change-making and Memory Compilation

In this section, we motivate the problem of reducing the set of memory instances in a design into a smaller set of fundamental macros that can fulfill the requirements while keeping compilation costs within limits. In the most general sense, we seek solutions that optimize across the width and depth axes simultaneously. Since this is a combinatorially expensive search space, memory compilers interpolate across a large space of depth and width combinations and use lookup table techniques to converge to the best configurations. However, even such approaches cannot handle both dimensions simultaneously and it common practice to identify the best depths first since the range of memory depth instances is considerably larger (bits↔Mbits) in comparison to the range of memory width instances (1↔2048). Once optimal depths are identified, the appropriate widths are realized by searching the restricted set of widths available at these optimal depths. Note that unless stated otherwise, we adopt the standard practice of decoupling the depth and width axes for the rest of this paper.

Packing a container of fixed size with one or more objects is a computational problem that appears frequently in a large number of practical problems. The term that describes the general class of problem is the Knapsack Problem [6]. Of the many variations of this class of problems, it is possible to reduce the search for optimal packings in the depth and width dimensions to the classical problem of change-making ($\mathcal{CM}$). Put simply, $\mathcal{CM}$ is how a cashier selects coins in order to efficiently yield a given amount of change due a customer. Depending on the set of coin denominations to choose from, there is sometimes a simple solution to $\mathcal{CM}$ known as the 'greedy' approach: take the largest coin denomination that is less than the amount of change remaining to make and repeat until the entire amount is satisfied. In the general case, the greedy approach does not work efficiently nor exactly: for example, given coins of denominations $\{1, 3{,}4\}$, the greedy approach does not satisfy 6 minimally ([4, 1, 1] instead of [3, 3]) and with denominations of $\{3, 4\}$ cannot satisfy 5 exactly ([4, 3]). MemPack uses a standard non-greedy approach to $\mathcal{CM}$ that avoids these pitfalls but with increased combinatorial complexity.

With this background, we introduce an example drawn from a real world design with over a hundred unique memory instances:

Depth	Widths
16	8, 16
24	24
32	8
48	15

Fundamental Depths: The choice of denominations plays a significant role in determining not just the quality of the packings, but also the size of the fundamental macro set in the end. Although it is intuitive to select a subset of the macros from the memory instances themselves, such an approach does not necessarily yield the best denominations (i.e., memory depths and widths) to realize other memory instances in the design. The practical, scalable approach is to use a set of denominations that are independent/agnostic of the problem, but that are known to be optimal for arbitrary amounts (i.e., memory depths and widths) that may occur within the dataset. It is well established that the powers of 2 constitute one such fundamental denomination set and that for any n, one would need no more than $\lceil \log_2 n \rceil$ coins in order to optimally realize n. Such a set of denominations can be potentially augmented with the most frequently occurring instances in order to provide a 'richer' starting set for minimization.

For illustrative purposes, let us consider a starting set of macros given by $\{1,2,4,8,16,\ 32,...,256K\}$. Acceptable packing solutions are further constrained by the cost of multiplexing (die area, propagation delay, power dissipation, etc.); multiplexing depth must therefore be limited. For example, given only a denomination of 1, every integer depth could be fulfilled, but a block of depth 31 would require muxing 31 instances of depth 1. Using this rationale, it is possible to eliminate certain denominations such as 1, 2, and 4 from consideration. Since the smallest block in the dataset is 16, one could argue that 8 is also redundant. However, the next larger block is of depth 24 with an optimal packing [16,8]. Since we seek to reduce the number of unique macros, one potential solution would be to accept waste on the block of depth 24 and to realize it using two blocks of depth 16. In practice, mux cost and waste are tradeoff parameters within MemPack.

Depth	Packing	Waste
16	16	
24	16, 16	8
32	32	
48	16, 32	

In section 3, we describe how it is possible to use a hill-climbing approach within MemPack to explore sub-sets of the current fundamental set to selectively drop denominations, i.e, depths/widths from contention to further reduce the size of the fundamental set.

Fundamental Widths: We have only reduced one dimension of the problem, depths, with the memory block widths ignored until now. Although depth reduction reduces the set of fundamental depths from 4 to 2, we have yet to consider the memory block widths. Our method could simply attach the required widths to the reduced set of depths, but this would still be far from our goal of minimizing the number of fundamental units. Fortunately, the memory widths are amenable to the same packing method used for the depths, resulting in yet further reductions in fundamental units.

As shown below there are 4 unique depths and 5 unique widths. The reduced set of fundamental depths is paired back up with the memory unit that they realize. For this example, this yields the cross-tabulation shown in the table below.

Depth	Widths			
	8	15	16	24
16	16 × 8		16 × 8, 8	
24				16, 16 × 8, 8, 8
32	32 × 8			
48		32, 16 × 8, 8		

This example illustrates that only 2 fundamental units are needed to realize all 5 design memory instances. Such a reduction in is offset by memory cell waste on macros 24×24 and 48×15.

3. MemPack

This section elaborates the $\mathcal{CM}$-based algorithms embedded in MemPack for the determination of the fundamental memory sets. As motivated in the previous section, MemPack decouples the depth and width dimensions, i.e., it first processes all depths irrespective of widths to identify a set of fundamental depth macros. For each fundamental depth macro, it processes all unique widths to determine a set of fundamental widths. Note that the $\mathcal{CM}$-based framework is reused in both phases, but that the full algorithm is described in the context of determining the set of fundamental depths in the next sub-section.

3.1 Depth Reduction

Depth reduction refers to the problem of processing the set $D = \{d \mid d \times w \in M\}$ to determine a set of fundamental depth macros D' that can be used to realize all depths (irrespective of width) in M. Note that D' may or may not be a sub-set of D. Although the primary objective is to reduce the number of elements in D', MemPack respects the constraints of being able to realize all memory depths in the design, not exceeding multiplexing costs, and not exceeding waste tolerances. There are two phases to the depth reduction: 1) selection of a starting set of fundamental depths and 2) successive reduction of the set of fundamental depths to the minimal set D'.

It is well established that the powers of 2 constitute an optimal set of currency denominations, i.e., if $D' = \{2^k | 0 \leq k \leq n-1\}$, it is possible to realize all instances $d \in D$ over the range $1 \leq d \leq 2^n - 1$ optimally. However, although this is a good choice of starting fundamental depths such that $n = \lceil \log_2 \max(d \in D) \rceil$, it is possible in practice to enrich this starting set. One option that we have explored and integrated into MemPack is to include those depths from D where the closest power of 2 is not part of the problem set D. In this manner, we initialize a set of candidates D' to fulfill the memory depths needed by the design.

With this starting set, we invoke MemPack to pack the fundamental depths we ‘have’ into all the design depths we ‘need’. Note that this starting set sets the baseline against which all further reductions in the set will be evaluated. The optimal solution to $\mathcal{CM}$ is an $\mathcal{NP}$-hard problem which would make computation of multiple packings using tens of fundamental depths infeasible. To overcome this limitation, MemPack uses a dynamic programming implementation of $\mathcal{CM}$. Dynamic programming is a standard transformation of algorithms which exchanges increased space complexity for reduced time complexity. It is a transformation that can only be applied to problems that have a recurrence relation i.e., computations which can be decomposed into smaller constituent sub-problems. Fortunately, $\mathcal{CM}$ is in the class of computations that have an admissible recurrence relation which changes the time complexity from $\mathcal{NP}$ to $O(cN)$ where c is the cardinality of the set of denominations D' and N is the range of depths to be filled.

Once the modified $\mathcal{CM}$ recurrence is computed, determining how to fulfill each design depth is a simple matter of repeated lookup. Note that the multiplexing limitation results in wasted memory cells. If we have already rejected a $\mathcal{CM}$ solution for a given design depth d, it still must be realized somehow. For blocks that have no exact solution, a larger block is realized in its place subject to the waste tolerance w. Although MemPack allows for variable per-depth waste tolerances, it is preferable to keep w as low as possible. In practice, we have found that a waste allowance of $w \approx 1.1$ to allow all packings to be satisfied. Since w sets an upper bound on how much a packing can waste and we rarely see MemPack even approach that 10% allowance, there isn’t compelling reason to find an ideally low value of w. Each depth packing for a set of available denominations D' has a computable waste penalty. This penalty is used to guide the search for smaller sets of minimal depths.

Hill-climbing: Looking back to the original goal: how to find as few fundamental units needed to realize the designs generated by memory compilers, MemPack uses a hill-climbing approach to reduce the set even further. Our approach systematically searches through the reduced set of depths until a still smaller set of denominations can be found. For example, given a starting set of 10 depths, all subsets of 9 depths are run through $\mathcal{CM}$-based packing in MemPack. Some subsets will yield more costly packings, some will fail to yield an acceptable packing (for example, excessive mux cost), and some may be less costly. Given the best of the subsets of length 9, MemPack repeats the process for all subsets of length 8. We continue this reduction until all subsets are worse than the set we have in hand which becomes our optimal set. One of the problems inherent in such searches is that they can encounter local maxima where every next smaller set is worse (in waste penalty) than the current set. We use standard perturbation approaches to detect local maxima and circumvent them. This packing and reduction is at the core of MemPack. The inner loop of the algorithm computes the packing using $\mathcal{CM}$, and the outer loop explores further reductions searching for the smallest member of the $|D'|$ subsets of D' with one less element that can cover the needs within a given waste tolerance.

3.2 Width Reduction

The second phase of the problem — reducing the elements of the set of design widths (W) needed to realize the full set of memory blocks — is amenable to the same method as the reduction of depths with some slight modification. Although width-wise block stitching is not nearly as costly as depth muxing, there is a non-zero cost associated with it; if this were not so, every width could be realized as some multiple of width 1.

There are two reasonable alternate sets of widths to seek to reduce. In the first case, just as depth reduction considers depths across the entire design, so could width reduction i.e., the entire design set of memory widths can be aggregated independent of depth. And the $\mathcal{CM}$-based framework could converge to a minimal subset $W' \in W$. However, this ignores the reductions in depth that were achieved in the first phase and requires that all fundamental depths $d' \in D'$ be characterized at all fundamental widths $w' \in W'$, regardless of whether the a particular $d' \times w'$ was used in the design. Although post-processing could be used to drop such unused fundamental blocks, this is a potentially expensive approach to the problem of fundamental width identification. The alternative is to look at each fundamental depth and to minimize the associated fundamental widths required at that depth. Thus, for each $d' \in D'$, MemPack consolidates the unique widths across all depths from D and invoke a pass of the $\mathcal{CM}$-based algorithm. Note that for widths, the waste tolerance needs to be slightly higher ($w \approx 1.25$) because the depth packing has already reduced the degrees of freedom available in the starting problem. For every $d' \in D'$, MemPack obtains a reduced set of widths W'_d by applying the $\mathcal{CM}$-based framework described above. The cross-product of the two independently minimized sets is constructed to determine the final set of fundamental macros M' given by $\bigcup_{d' \in D'} \{d'\} \times W'_d$. For every original memory block $m \in M$, MemPack computes the best composition of fundamental units needed to realize that block. Waste is calculated from this bijection for all design blocks whose size is exceeded by their composing fundamental blocks.

Table 1: Results for fundamental macro minimization

Design Specifications				-performance			-balance			-minimize		
Design	Macros	Depths	Widths	Fundamental Macros	Reduction	Waste	Fundamental Macros	Reduction	Waste	Fundamental Macros	Reduction	Waste
Industry 1	113	33	77	45	2.5×	1.4	29	3.9×	0.8	21	5.4×	2.1
Industry 2	181	58	89	80	2.3×	3.4	40	4.5×	2.9	29	6.2×	4.0
Industry 3	313	85	136	99	3.2×	3.5	51	6.1×	2.3	36	8.7×	4.7
Syn 1	305	75	66	97	3.1×	2.8	45	6.8×	2.1	31	9.8×	3.6
Syn 2	335	74	87	95	3.5×	2.2	48	7.0×	2.0	33	10.2×	4.5
Syn 3	483	83	181	104	4.6×	2.6	48	9.3×	3.1	39	11.2×	3.7
Syn 4	577	84	108	105	5.5×	3.2	48	12.8×	3.1	40	16.0×	3.6

4. Results

We present simulation results for two datasets in table 1. The first dataset comprises three cases based on sanitized industry designs from high volume and leading SoC vendors. The first two cases in this dataset are stand alone designs, and the third case was constructed by aggregating data from the first two designs and other smaller designs with a limited number of memory instances. We constructed this set to confirm results in a case with more fundamental blocks, since it captures the scenario when there are a large number of designs supported on a single design flow.

In order to demonstrate that MemPack is effective across a wide class of potential designs that it may be applied to, we constructed synthetic test cases to constitute the second dataset. These were intended to model features of the real designs we do have available to us. Statistical description of the data has been combined with some knowledge of the behavior of memory compilers to generate plausible synthetic test sets with the freedom to test the method across a design space that we expect to encounter in practice.

The first four columns of the table give details about the test cases that were considered for the experiments. The first column specifies the source of the design (industry or synthetic). The second column enumerates the total number of unique memory macros that were used in the design. Note that each macro may have more than one instance, but that data is not reported here. The third and fourth columns report the number of unique memory depths and the number of unique memory widths in the design, respectively.

The proposed algorithms were run on these test cases and the results are reported for these datasets for three different optimization objectives in MemPack: (i) performance, that gives priority to richer fundamental sets with low muxing and instantiation, (ii) balance, that attempts to reconcile performance against minimization of the fundamental macro set, and (iii) minimization, which prioritizes fundamental set minimization over wasted bits.

Columns 5–7 report results for the `-performance` option, when muxing limit was set to 2/4 and replication limit also set to 2/4. Note that 2/4 means that the first pass for each packing is with a limit of 2; if the penalties are excessive, a second pass is made with a limit of 4. These options ensure that MemPack determines the fundamental macro sets without resulting in excessive overhead by way of muxing on the critical paths or instantiation to achieve larger widths. Column 5 reports the size of the fundamental set M'; column 6 reports the reduction in terms of the size of the original set M; column 7 reports the waste as a %age of the total memory when the fundamental set is used to realize all memory instances in that test case.

Columns 8–10 report results for the `-balance` option, when muxing limit set to 2/4/8 and replication limit also set to 2/4/8. These options ensure that MemPack determines the fundamental macro sets that tradeoff fundamental set minimization against waste, muxing, and instantiation costs. As the name suggests, this option strikes a balance between the two more extreme approaches of optimizing for performance or minimum fundamental sets. Although it results in increased wastage over the `-performance` option, the size of the fundamental set is reduced by a factor of 2× on average over the `-performance` option.

Columns 11–13 report results for the `-minimize` option, where muxing limit set to 4/8 and replication limit also set to 4/8. These options ensure that MemPack determines the smallest possible fundamental macro sets. As would be expected, this option results in increased wastage over the prior options while also resulting in the smallest fundamental sets.

5. Conclusions

We have addressed a problem inherent to memory compilers that has not yet been addressed: the proliferation of embedded memory geometries and the resulting characterization, validation, and reliability challenges. Our proposed framework MemPack demonstrates how hundreds of memory instances in a design can be realized from a much smaller set of 'fundamental' memory macros, reducing characterization, silicon validation, and reliability costs.

MemPack's minimization algorithms generate a continuum of fundamental sets, which allow engineering design choices between minimization and memory capacity waste. Most importantly, MemPack realizes these optimizations in reasonable execution time (tens of seconds on an average desktop computer) and leverages existing memory compilers to build all the memory instances based on the fundamental sets. MemPack is currently under integration into a memory compiler platform, where it is expected to provide significant reductions in design effort and enhance product reliability.

References

[1] K. Darbinyan, G. Harutyunyan, S. Shoukourian, V. Vardanian, and Y. Zorian, "A robust solution for embedded memory test and repair," in *Proc. Asian Test Symposium*, pp. 461–462, 2011.

[2] R. E. Matick and S. E. Schuster, "Logic-based edram: Origins and rationale for use," *IBM Journal of Research and Development*, vol. 49, pp. 145–165, January 2005.

[3] K. Zhang, *Embedded Memories for Nano-Scale VLSIs*. Series on Integrated Circuits and Systems, Springer, 2009.

[4] J. Srinivasan, S. Adve, P. Bose, and J. Rivers, "Lifetime reliability: Toward an architectural solution," *IEEE Micro*, vol. 25, no. 3, pp. 70–80, 2005.

[5] M. Horiguchi and K. Itoh, *Nanoscale memory repair*. Springer Verlag, 2011.

[6] S. Skiena, *The Algorithm Design Manual (2. ed.)*. Springer, 2008.