# Locality Aware Memory Assignment and Tiling

Samuel Rogers
University of North Carolina at Charlotte
Department of Electrical and Computer Engineering
Charlotte, NC, USA
sroger48@uncc.edu

Hamed Tabkhi
University of North Carolina at Charlotte
Department of Electrical and Computer Engineering
Charlotte, NC, USA
htabkhiv@uncc.edu

## ABSTRACT

With the trend toward specialization, an efficient memory-path design is vital to capitalize customization in data-path. A monolithic memory hierarchy is often highly inefficient for irregular applications, traditionally targeted for CPUs. New approaches and tools are required to offer application-specific memory customization combining the benefits of cache and scratchpad memory simultaneously.

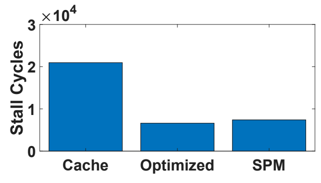
This paper introduces a novel approach for automated application-specific on-chip memory assignment and tiling. The approach offers two major tools: (1) static memory access analysis and (2) variable-level memory assignment. Static memory analysis performs at the LLVM abstraction. It extracts target-independent pointer behaviors, measures the access strides and analyze the prefetchability of variables. (2) variable-level memory assignment creates a memory allocation graph for memory assignment (cache vs. scratchpad) based on the variables size and their estimated locality. It also explores the opportunity for tiling memory access. For the exploration and results, this paper uses Machsuite benchmarks (with both regular & irregular memory access behaviors), and gem5-Aladdin tool for performance & power evaluation. The proposed approach optimizes the memory hierarchy by automatically combining the benefits of cache, (tiled-) scratchpad at variable level granularity per individual applications. The results demonstrate more than 45% improvement in our power-stall product, on average, over the monolithic cache or scratchpad design.

**ACM Reference Format:**
Samuel Rogers and Hamed Tabkhi. 2018. Locality Aware Memory Assignment and Tiling. In *DAC '18: DAC '18: The 55th Annual Design Automation Conference 2018, June 24–29, 2018, San Francisco, CA, USA.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3195970.3196070

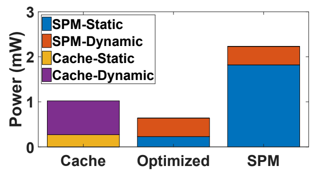
## 1 INTRODUCTION

Customization has emerged as a promising approach for high-performance, low power execution of many challenging applications [1–3]. Despite the trend toward customization, the main focus has been on application-specific data-path customization. The application-specific memory-path design has been not explored in depth. An efficient memory-path design is vital to capitalize customization in data-path. For custom hardware/specialized accelerators, Scratchpad memory (SPM) has been the predominant memory choice to hide long access latency to the main memory. The scratchpad often performs well for regular streaming memory access patterns (with short strides). However many irregular applications, traditionally targeted for CPUs, monolithic SPM design is highly inefficient, diminishing the benefits of data-path customization.

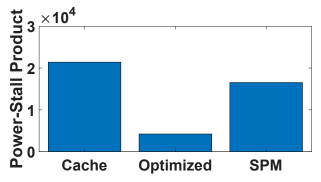
Much like the design of the data-path, the memory-path design for HWACCs is highly application-specific. As an example, Fig. 1 compares the memory performance (evaluated by measuring the memory stalls) and power across different memory assignments for the MD-KNN application. An optimized memory design (combining the benefits of cache, SPM and tiled-SPM selectively across variables) achieves far better memory performance per watt compared to monolithic SPM or cache. Fig. 1 presents that application-specific memory fine-tuning can reconcile the conflicting access pattern of an application (appeaser across the variables) resulting in an optimized power-stall product (PSP) (Fig. 1c) with much lower physical memory requirements (Fig. 1d).



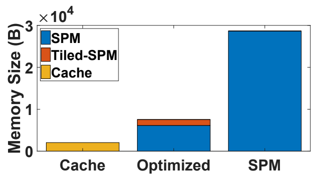
(a) Memory Stalls



(b) Memory Power



(c) Power-Stall Product



(d) Memory Assignment

**Figure 1: MD-Grid Exploration**

Studying and exploring cache for HWACCs is at early stages, and there is not much research exploring the memory-path (cache vs. SPM) for HWACCs. Determining which hierarchy is best has until now been largely ad-hoc, requiring many simulations to identify a Pareto-optimal solution. One of the early studies is gem5-Aladdin [4] which integrated the cache memory modeling present in gem5 [5] into the accelerator model provided by Aladdin [6]. However, approaches such as [5, 7], leave the memory design space exploration and optimization to the designer. Overall, there is lack of tools that can look at an application and allocate the correct type of memory. Novel tools are required to streamline the design of an efficient application-specific memory-path.

This paper introduces a novel approach for automated application-specific on-chip memory assignment and tiling. The approach offers

*DAC '18, June 24–29, 2018, San Francisco, CA, USA*
© 2018 Association for Computing Machinery.
ACM ISBN 978-1-4503-5700-5/18/06…$15.00
https://doi.org/10.1145/3195970.3196070

an integrated tool-chain with two major phases: (1) static memory access analysis and (2) variable-level memory assignment. It works at LLVM abstraction and statically extracts and analyzes variables and the memory footprint (job size). At the memory assignment, the proposed approach assigns on-chip memory and design memory-path (combining cache, SPM, tiled-SPM) intrinsic to the application. Our results, based on gem5-Aladdin [4] running the Machsuite benchmark suite [8], reveals significant benefits. On average, the results demonstrate 45% improvement in the power-stall product, with significant saving in on-chip memory demand, over the monolithic cache or scratchpad design.

This paper is organized as follows: Section 2 briefly overviews related work. Section 3 overviews our proposed approach. Section 4 presents the static memory access analysis. Section 4 explains selective memory prefetching with static binary analysis. Section 5 expands the variable-level memory assignment. Section 6 presents the simulation results and finally Section 7 concludes this paper.

## 2 RELATED WORK

The memory hierarchy has been studied extensively in general-purpose CPUs. Many innovative solutions have been developed for memory-path optimization (cache vs. SPM) on embedded CPUs [9–12]. Extensive work has been done in that portion of the field to optimize hybrid memory structures for domain specific CPUs. Such optimizations have shown to lead to significant improvements (20% or more) in both raw performance and overall power consumption within a specific domain [10, 12].

In contrast to general-purpose CPUs, memory exploration and memory-path optimization for hardware accelerators are at early stages. Accelerator-centric approaches, such as [6, 7, 13], have assumed that accelerated applications demonstrate highly regular streaming data access patterns. As a result, the majority of the research has centered on improving the performance and power consumption of data-path with the assumption of direct memory access (DMA) and local scratchpad memories (SPM) for the memory-path [4, 14, 15]. Some researchers have proposed hand-crafted optimized memory-path for particular applications [2, 16–18]. However, their narrow scope, as well as their manual design, limits their usefulness. When focusing on highly regular applications, as was done in Chen et al [17] and Cong et al [18] it is possible to create highly efficient data tiling schemes that reduce memory stalls and power consumption at the cost of programming complexity. However as development shifts toward the creation of accelerators for more irregular applications, traditional optimization are often impractical to implement.

To address this problem, the Aladdin simulator[6] was extended and integrated it into the memory system provided by gem5[5] to create gem5-Aladdin[4]. This integration brought a more advanced memory model to Aladdin allowing for exploration of hybrid SPM-cache models accelerators. The capacity to sweep across a wide range of memory configurations can be valuable for memory optimization. The challenge in HWACC design then becomes a matter of identifying the optimal use of SPMs and/or caches for a given application. Tools like PARADE [7] and gem5-Aladdin [4] place this entire burden in the hands of the user, offering no insight as to where this optimization point rests without doing exhaustive simulation.

## 3 APPROACH OVERVIEW

This section overviews our proposed approach in an integrated scheme. It performs at the LLVM abstraction, an intermediate representation between high-level languages and assembly. LLVM provides an architecture-independent flat representation of all variables appearing in an application. It captures the raw semantic of execution without the effects of target-specific constraints or optimizations. Fig. 2 presents an abstract high-level model of proposed approach. Internally, the proposed approach consists of two major tools: (1) static memory access analysis and (2) variable-level memory assignment. It receives the target-independent but optimized LLVM model of the application as an input, and make a fully automatic memory allocation decision (cache, SPM, SPM-tiled) as the design decision output. For creating the LLVM, we use standard CLang tool (a compiler front-end) to translate high-level C/C++ code to an optimized LLVM.

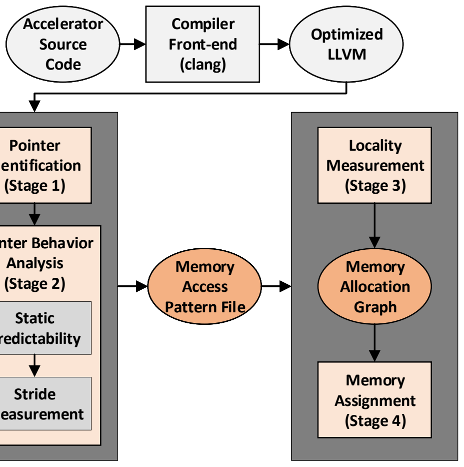

Accelerator Source Code
Compiler Front-end (clang)
Optimized LLVM
Pointer Identification (Stage 1)
Pointer Behavior Analysis (Stage 2)
Static Predictability
Stride Measurement
Memory Access Pattern File
Locality Measurement (Stage 3)
Memory Allocation Graph
Memory Assignment (Stage 4)

**Figure 2: Memory Allocation Tool Chain**

## 4 STATIC MEMORY ACCESS ANALYSIS

This section presents the details of our proposed static memory access analysis.

### 4.1 Pointer Identification

*Pointer Identification* parses the LLVM file. It searches for all functions, appearing in the optimized LLVM, and then lists all pointers within every function. It also identifies the scope of each pointer (global or local). All pointer analysis occurs at the function level granularity to avoid control behaviors that are not statically identifiable. Fig. 3 presents the flow-chart graph of *Pointer Identification*. Within the scope of a function run-time dependent pointer indexes can be broken down into (1) loop indexes, (2) internal variable indexes, and (3) external variable indexes. Loop indexes are easily predictable. Internal variable indexes are predictable if the controlling variable is itself predictable. The only type of index that causes issues in static analysis is the external variable index, which is often governed by run-time conditions.

*Pointer Identification* calls *Pointer Behavioral Analysis* (Stage 2) whenever a new pointer address is found. Once all functions and pointers in the LLVM file have been identified, the parser will move onto Stage 3 where it measures the static locality of each pointer. In

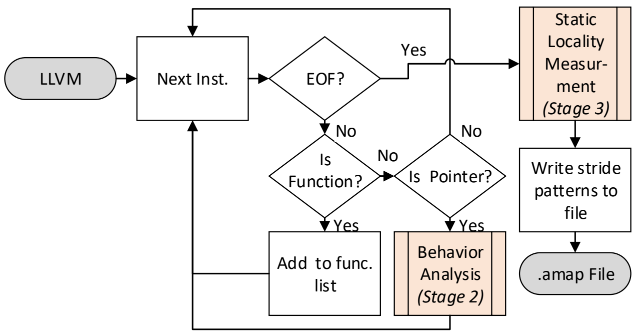

LLVM
Next Inst.
EOF?
Yes
Static Locality Measur-ment (Stage 3)
No
No
Is Function?
No
Is Pointer?
Write stride patterns to file
Yes
Yes
Add to func. list
Behavior Analysis (Stage 2)
.amap File

**Figure 3: *Pointer Identification* (Stage 1)**

the end, it is also able to calculate the job size per entire application by statically extracting the volume of Bytes per variable array access for each iteration of execution. For variables with runtime dependent strides, it considers the worst-case access boundaries.

## 4.2 Pointer Behavioral Analysis

*Pointer Behavioral Analysis* extracts the nature of each pointer (static vs. dynamic). Fig. 4 presents the flow-chart graph of *Pointer Behavioral Analysis*. With every call, it first checks the pointer list to determine if the referenced pointer already exists or not. If not, the new pointer will be added to the pointer lists of the current function. Otherwise, the existing pointer entry will be updated with the destination register of the operation and the source register(s)/index(es) of the address calculation. It internally detects the prefetchability of variables, and their access strides.

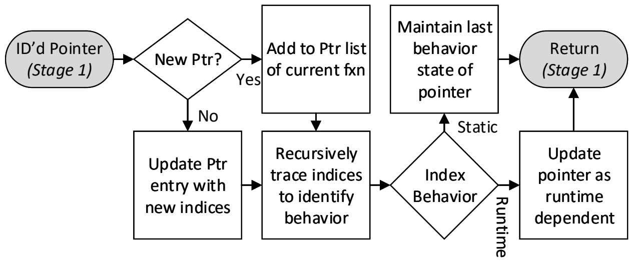

ID'd Pointer (Stage 1)
New Ptr?
Yes
Add to Ptr list of current fxn
Maintain last behavior state of pointer
Return (Stage 1)
No
Static
Update Ptr entry with new indices
Recursively trace indices to identify behavior
Index Behavior
Runtime
Update pointer as runtime dependent

**Figure 4: *Pointer Behavioral Analysis* (Stage 2)**

*4.2.1 Predictability Analysis.* Before any static analysis of stride can be performed, we must first separate the pointers with runtime dependencies from those with purely static dependencies. To determine the static/run-time nature of the address calculation, each index is recursively traced. If the recursive trace of an index concludes with an unpredictable variable, the pointer access will be marked as a run-time dependent variable. Otherwise, the pointer will retain its current classification of static/run-time. A key LLVM optimization that aids in this process is the separation of loop control variables from other conditional variables. This leads to unique loop index variables, even when a variable name is re-used in the original host code, and helps us to extract independent loop behaviors which are always statically predictable. When all index traces are completed, and the pointer behavior is updated, the parser transitions back to Stage 1.

*4.2.2 Stride Measurement.* Once we have identified the variables that can be statically analyzed we move on to measuring their strides. Fig. 5 shows the general flow of the of *Stride Measurement* stage. For variable indices we do a quick recursive trace covering its last three operational dependencies. This is deep enough to capture the loop behaviors of 2-D and 3-D arrays that are accessed liked 1-D arrays (ex: A[i*rowlength+j]). Otherwise the stride is marked as variable and assumed to have a long or irregular stride. In the case of variables with constant indices, this is simply a matter of checking the distance between successive indices. If there is irregularity in the stride, the pointer is marked as having a variable stride. For loop indices our tool retraces the code around which the loop index was identified, looking for the .next variable associated with that loop index. The .next statement will contain the value by which the loop index is incremented or decremented, or in other words its stride.

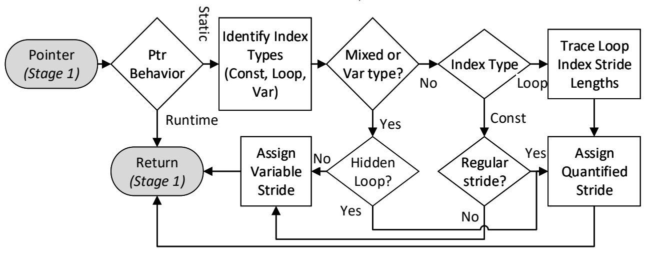


**Figure 5: *Stride Measurement* (Stage 3)**

*4.2.3 Memory Access Pattern (MAP)-File.* The Memory Access Pattern (MAP) file provides a brief summary of the identified pointers and their static behaviors extracted by the first part of our tool chain. It serves as the foundation on which we identify optimal memory assignments, but also offers a key for hand tracing of LLVM if a designer wishes to further code optimization at the LLVM abstraction.

```
Function: TYPE @FUNC_NAME
Parameters: (PARAMS)
        Variable: %PTR_NAME   STRIDE_LENGTH
        Type: PTR_TYPE
        Accesses:
                %REG_OP -> INDEX_LIST
        Variable: %PTR_NAME2  STRIDE_LENGTH
        Type: PTR_TYPE
        Accesses:
                %REG_OP -> INDEX_LIST
                %REG_OP -> INDEX_LIST
```

**Figure 6: Memory Access Pattern-file (*MAP-file*)**

The MAP file outlined in Fig. 6 groups pointer information at the function level and displays information for every global variable captured within that function. The MAP file contains the data type, stride length as captured in the *Stride Measurement* phase of *Pointer Behavior Analysis*, and a list of all accesses associated with each variable. Accesses are indexed by their associated register operation as captured in LLVM. This provides a quick reference used for spatial and temporal locality measurements.

## 5 VARIABLE-LEVEL ASSIGNMENT

This section presents the details of our proposed variable-level memory assignment.

## 5.1 Locality Measurement

*Locality Measurement* (Stage 4) is responsible for assessing the inherent static locality of each unique variable array in the program (identifiable by a unique pointer). It assesses the spatial locality based on the length of the stride that appears for each variable. It also assesses the temporal locality by statically measuring the reuse potential of each variable index. We use Eq. 1, originally described by Weinburg et al. in [19], to assess the static locality of identified pointers. For loop and constant indexed pointers the locality calculated in *Stride Measurement* is used. For pointers previously marked as having variable strides, we perform one last trace of the LLVM to identify the stride between each access.

$$L_{spatial} = \sum_{stride=1}^{\infty} \frac{P(stride)}{stride} \tag{1}$$

To capture the temporal locality of each variable, we use a modified version of Eq. 2, sourced from Weinburg et al.[19]. Since the raw addresses of memory access are not exposed in LLVM, we instead define our reuse distance by the number of memory operations before a non-loop index is repeated.

$$L_{temporal} = \sum_{i=0}^{log_2(N)} \frac{(dist_{2^{(i+1)}} - dist_{2^i}) * (log_2(N) - i)}{log_2(N)} \tag{2}$$

The spatial and temporal locality measurements are combined using Eq. 3 and adjusted by a spatial locality weight ($w_s$) and a temporal locality weight ($w_t$). Since data tiling and vectorization optimization require high spatial locality, more so than temporal locality, we place a higher emphasis on spatial locality. The temporal locality is still an important factor in the performance of cache, so we still consider its effects even when spatial locality is lower.

$$L_{total} = \left(w_s * L_{spatial} + w_t * L_{temporal}\right) * 100 \tag{3}$$

## 5.2 Memory Assignment

Once a locality level has been assigned to each pointer (variable index), the *Memory Assignment* (Stage 4) assigns memory types with respect to the conceptual graph in Fig. 7. Variable-level memory assignment provides the opportunity for dedicated memory allocation with respect to the behavior of individual variables in the application. It often results in a hybrid memory model combining the benefits of cache and SPM.

Fig. 7 summarizes the memory assignment decision with respect to job size and locality. As the job size and locality of the data increases, cache becomes more efficient due to its selective on-demand data access. As job size and locality decrease, cache loses its advantage over SPM and suffer more memory stalls due to cache-trashing and the high number of misses. There is a conceptual threshold in job size and locality which make the cache more preferable over SPM. Once the locality hits a certain threshold associated with data having a strong streaming behavior, the assignment changes over to a tiled SPM as shown in the rightmost sector of Fig. 7. This is because tiling and double buffering optimizations enable a programmer to significantly reduce the size of their SPM without exposing stalls to the datapath.

The greatest advantage of SPM is its capacity to capture the spatial and temporal locality of data within the granularity of its job size. One significant drawback of SPM is the power cost of maintaining a large on-chip memory. The power costs can be mitigated if the data can be further divided into tiles, but this optimization is limited to streaming data. Cache, in contrast, can bypass this limitation entirely and offer a transparent interface by directly requesting data as needed from memory at a higher power cost per Byte of memory. While this means that a cache will be far less efficient than a similarly sized SPM, a small cache can achieve parity in efficiency for data that requires a very large SPM.

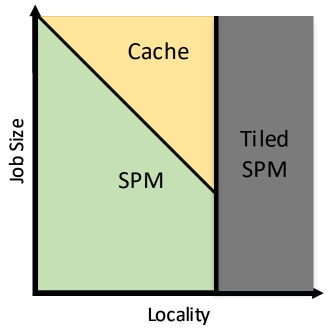


**Figure 7: Memory Allocation Graph**

*5.2.1 Tiling Eligibility.* Using the locality approximation, it is possible to determine the eligibility of a variable for tiling. Variables with high spatial locality are immediately marked as eligible for tiling. It is also possible, however, that a variable with lower locality may still be eligible for tiling to some degree. To identify this potential, we look at the list of indices associated with the variable. In some cases, a strided access behavior may be obfuscated by intermediate calculations, such as when a 2-dimensional matrix is exposed as a 1-dimensional array. In this case, the indices of the pointer will appear variable in LLVM, but actually point to a loop index modulated by some intermediate operations. In this case the the pointer is marked as a potential variable for tiling, however tiling potential may be severely limited due to low spatial locality.

## 6 EVALUATION

## 6.1 Simulation Setup

For the simulation results, we use the Machsuite benchmark suite [8], dedicated for HWACC research, and gem5-Aladdin [4] for performance and power estimation. Our tool (captured in C++) runs alongside gem5-Aladdin to identify memory access patterns and make an automatic memory assignment decision. It will be released to the public upon publication of this work.

To evaluate our tool, we compare our suggested optimization to a job-sized SPM and a variable size cache (sized for an optimal power-stall product). We evaluate the performance of our memory assignment by analyzing the memory stalls introduced to the accelerator data path. The overall memory efficiency is measured by multiplying stalls by the power consumed by memory. A lower power-stall product (PSP) is representative of a more efficient memory design. When tiling is suggested by our tool, we chose the minimum tile size achievable without significant algorithm modifications. The DMA engine is assumed to handle scheduling for tiled data in to avoid unnecessary CPU overheads. In the case of a cache assignment, we sweep the size of the cache to find the optimal PSP.

## 6.2 Performance and Power Comparison

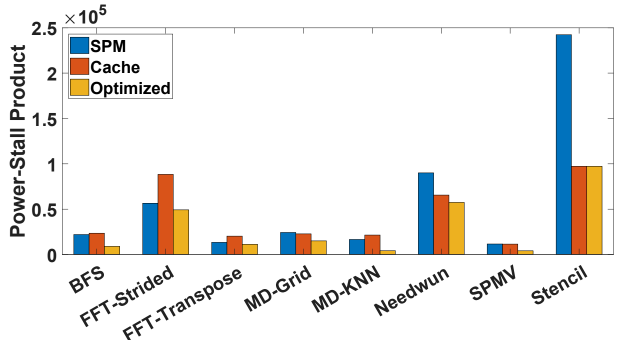
Analyzing eight of the MachSuite benchmarks, our tool identified one benchmark in which all variables could be tiled (MD-Grid), two benchmarks in which some variables could be tiled (SPMV-CRS, MD-KNN), and five benchmarks in which no variable could be effectively tiled (BFS, FFT-Strided, FFT-Transpose, Needwun, 3D Stencil). A common characteristic of the benchmarks lacking variables suitable for tiling is irregular array index patterns (BFS) or alternatively, multidimensional arrays indexed such that locality is destroyed (3D Stencil). This is not to say that all variants of these applications share the same issue. In fact, as we discuss in our case study later on, algortihmic changes can have a significant impact on the optimal memory configuration.

For nearly every benchmark shown in Fig. 8b the base SPM implementation outperformed the base cache implementation. In contrast the cache implementation almost universally had a lower overall power consumption than the SPM as seen in Fig. 8c. This can be attributed to the PSP limitation used to identify the ideal cache size. In Fig. 8a we see that the net PSP is similar for the base SPM and cache implementation across the benchmarks. In contrast, the tool optimized allocation was able to match the performance of DMA with the power usage of the cache for all eight benchmarks.

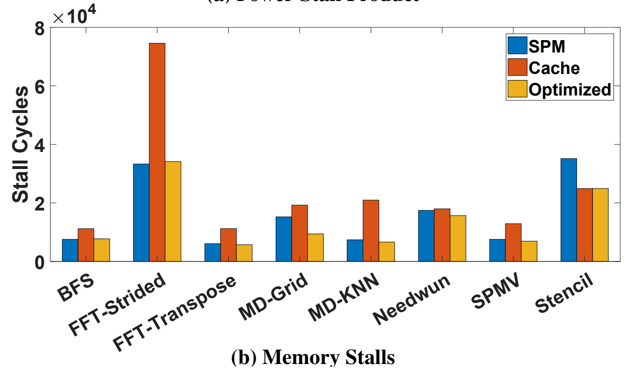
The most dramatic stall and PSP improvements can be seen in MD-Grid, MD-KNN, and SPMV-CRS. Each of these benchmarks had data that could tiled without significant changes to their algorithm. The performance increase in these benchmarks can be attributed entirely to the tiling and double buffering of large parts of their data, reducing both the delays exposed to their datapaths, as well as the power consumed by on-chip memory. While the benchmarks without tiled data did not benefit as significantly from optimization, they still saw an average PSP reduction of 37%.

## 6.3 Memory Assignment

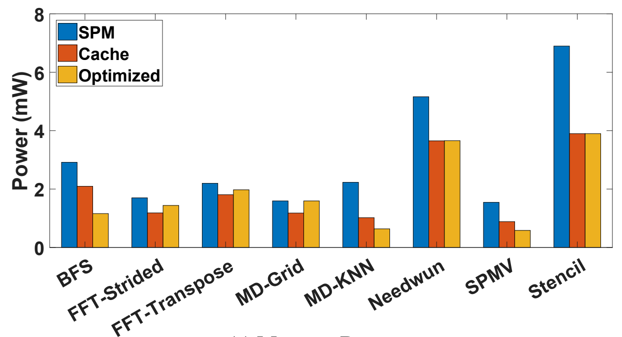
Looking at Fig. 9 we can see the final physical memory allocation for each benchmark with respect to its job size, as well as the relative allocation between each type of memory. In all but two cases the allocated memory was much smaller than the job size of the benchmark. In the case of MD-Grid, the data could only be divided into two tiles without significant changes to the benchmark. Even so, it still enabled the use of double buffering to significantly improve the performance and power usage. As mentioned above Merge Sort followed the basic SPM implementation. The two most noticeable differences in memory allocation vs. job size can be seen with the Needwun and 3D Stencil allocations. These two applications had the largest jobs sizes of any of the benchmarks tested (82KB and 128KB respectively). Without the ability to tile the data for these benchmarks due to algorithmic limitations, the only way to reduce the memory footprint of these applications was with caches. After assigning variables according to our tools suggestion, and optimizing cache size around a PSP target, we were able to reduce the memory footprints of these two jobs to 2KB or less each. Caches played a big role in reducing the memory footprints of the other benchmarks that lacked data eligible for tiling. In both FFT implementations caches reduced the memory footprint by roughly half, while accounting for less than 10% of memory allocated.



(a) Power-Stall Product



(b) Memory Stalls



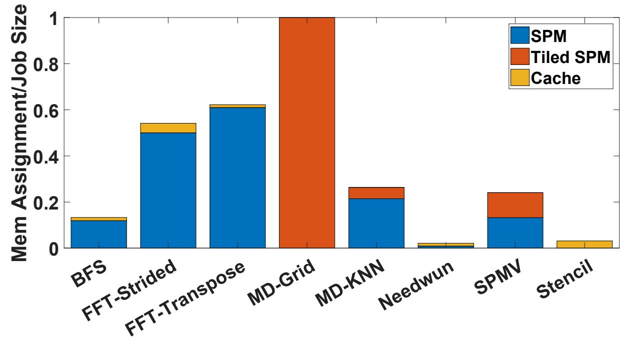
(c) Memory Power

**Figure 8: Performance and power comparison**

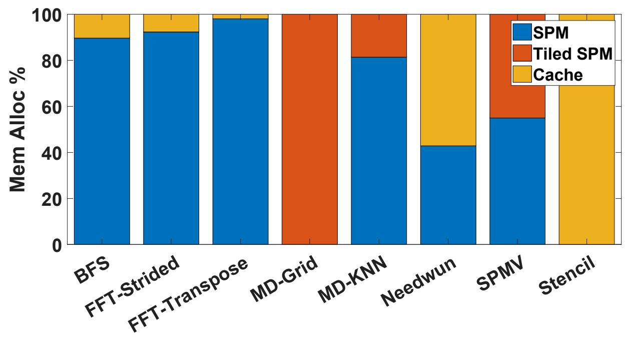
## 6.4 Case Study: Algorithm Effects on Optimal Memory Allocation

A key insight that can be explored through the use of our tool is the impact of algorithmic changes on the optimal memory allocation for a given application. Two different versions of a molecular dynamics application are implemented in MachSuite (MD-Grid and MD-KNN). Both versions of the application take the same input, perform the same basic operation, and produce the same output. The way in which they organize and access data, however, is very different. MD-Grid stores its data in a 3-dimensional data structure, and exhibits a very neat stride behavior across its entire data that is ideal for tiling. MD-KNN on the other hand stores its data in a much less dense 1-dimensional data structure and several small supporting data arrays. This leads to a more irregular access pattern in a part of its data that limits tiling.

In Fig. 10 we can see memory access graphs for MD-Grid and MD-KNN based on data generated by our tool. In Fig. 10a we can see that the three main structures of MD-Grid all fall within the tiled

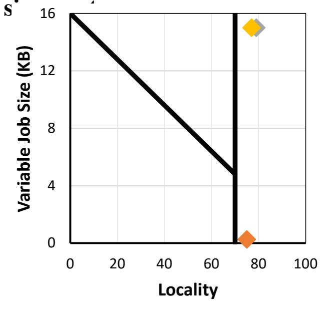


**(a) Physical memory assignment as a percent of job size**

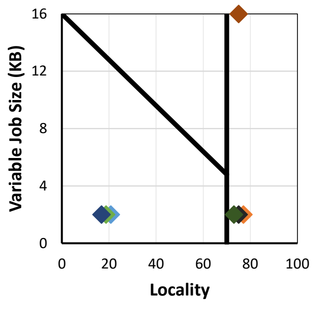


**(b) Logical memory assignment per job size**

**Figure 9: Memory type distribution for optimized memory as-
s**



**(a) MD-Grid Access Graph**



**(b) MD-KNN Access Graph**

**Figure 10: Memory Access Graphs for Alternate MD Realizations**

data range described by our conceptual memory access graph in Fig. 7. In comparison, we see in Fig. 10b that the MD-KNN algorithm has data split across both the tiled SPM and untiled SPM regions. While this kind of access may appear less than ideal at first, it actually allows for deeper tiling optimization for the bulk of the data leading to nearly 4 times reduction in the PSP.

Memory optimization isn't merely application specific, but also heavily algorithm specific. With the current shift toward domain specific computing, this algorithm-level insight can be useful for building an optimal memory hierarchy for a domain of applications.

## 7 CONCLUSIONS

This paper introduced a novel tool-chain to offer systematic application-specific memory assignment. The proposed approach is based on static application analysis. It works at the LLVM abstraction to extract memory access behaviors and enable memory allocation at an application or variable granularity. It offers an automated way for application-specific memory hierarchy design, combining the benefits of both caches and SPMs. Our results, based on gem5-Aladdin running the Machsuite benchmark suite revealed significant benefits of the proposed approach. On average, the results demonstrated 45% improvement in the power-stall product, with the significant saving in on-chip memory, over the uniform cache or scratchpad design.

## REFERENCES

[1] D. Melpignano, L. Benini, E. Flamand, B. Jego, T. Lepley, G. Haugou, F. Clermidy, and D. Dutoit, "Platform 2012, a many-core computing accelerator for embedded socs: performance evaluation of visual analytics applications," in *Proceedings of the 49th Annual Design Automation Conference*. ACM, 2012, pp. 1137–1142.

[2] H. Tabkhi, R. Bushey, and G. Schirner, "Function-level processor (flp): A novel processor class for efficient processing of streaming applications," *Journal of Signal Processing Systems*, vol. 85, no. 3, pp. 287–306, 2016.

[3] J. Cong, M. Ghodrat, M. Gill, B. Grigorian, and G. Reinman, "Architecture support for accelerator-rich CMPs," in *Design Automation Conference (DAC)*, 2012, pp. 843–849.

[4] Y. S. Shao, S. Xi, V. Srinivasan, G.-Y. Wei, and D. Brooks, "Co-Designing Accelerators and SoC Interfaces using gem5-Aladdin," in *The 49th IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2016.

[5] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood, "The gem5 simulator," *SIGARCH Comput. Archit. News*, vol. 39, no. 2, pp. 1–7, Aug. 2011. [Online]. Available: http://doi.acm.org/10.1145/2024716.2024718

[6] Y. S. Shao, B. Reagan, G.-Y. Wei, and D. Brooks, "Aladdin: A pre-rtl, power-performance accelerator simulator enabling large design space exploration of customized architectures," in *ACM/IEEE 41st International Symposium on Computer Architecture (ISCA)*, 2014.

[7] J. Cong, Z. Fang, M. Gill, and G. Reinman, "Parade: A cycle-accurate full-system simulation platform for accelerator-rich architectural design and exploration," in *2015 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, 2015.

[8] B. Reagen, R. Adolf, Y. S. Shao, G.-Y. Wei, and D. Brooks, "MachSuite: Benchmarks for accelerator design and customized architectures," in *Proceedings of the IEEE International Symposium on Workload Characterization*, Raleigh, North Carolina, October 2014.

[9] F. Piovezan, T. E. M. Crocomo, and L. C. V. dos Santos, "Cache sizing for low-energy elliptic curve cryptography," in *29th Symposium on Integrated Circuits and Systems Design (SBCCI)*, 2016.

[10] G. Wang, L. Ju, Z. Jia, and X. Li, "Data allocation for embedded systems with hybrid on-chip scratchpad and caches," in *IEEE International Conference on High Performance Computing and Communications*, 2013, pp. 366–373.

[11] J. Sancho and D. Kerbyson, "Analysis of double buffering on two different multicore architectures: Quad-core Opteron and the Cell-BE," in *International Symposium on Parallel and Distributed Processing (ISPDP)*, 2008, pp. 1–12.

[12] L. Wu and W. Zhang, "Cache-aware spm allocation algorithms for hybrid spm-cache architectures," in *Sixteenth International Symposium on Quality Electronic Design*, March 2015, pp. 123–129.

[13] R. Hou, L. Zhang, M. Huang, K. Wang, H. Franke, Y. Ge, and X. Chang, "Efficient data streaming with on-chip accelerators: Opportunities and challenges," in *High Performance Computer Architecture (HPCA)*, 2011, pp. 312–320.

[14] B. Reagan, P. Whatmough, R. Adolf, S. Rama, H. Lee, S. K. Lee, J. M. Hernandez-Lobato, G.-Y. Wei, and D. Brooks, "Minerva: Enabling low-power, highly-accurate deep neural network accelerators," in *ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*, 2016.

[15] M. Qiu, Z. Chen, Z. Ming, and J. Niu, "Energy-Aware Data Allocation With Hybrid Memory for Mobile Cloud Systems," in *IEEE SYSTEMS JOURNAL, VOL. 11, NO. 2*, 2017, pp. 813–822.

[16] C. Song, L. Ju, and Z. Jia, "Hybrid scratchpad and cache memory management for energy-efficient parallel hevc encoding," in *33rd IEEE International Conference on Computer Design (ICCD)*, 2015, pp. 712–719.

[17] J. Cong, P. Li, B. Xiao, and P. Zhang, "An optimal microarchitecture for stencil computation acceleration based on non-uniform partitioning of data reuse buffers," in *2014 51st ACM/EDAC/IEEE Design Automation Conference (DAC)*, June 2014, pp. 1–6.

[18] Y. T. Chen, J. Cong, J. Lei, and P. Wei, "A novel high-throughput acceleration engine for read alignment," in *2015 IEEE 23rd Annual International Symposium on Field-Programmable Custom Computing Machines*, May 2015, pp. 199–202.

[19] J. Weinberg, M. O. McCracken, E. Strohmaier, and A. Snavely, "Quantifying locality in the memory access patterns of hpc applications," in *Supercomputing, 2005. Proceedings of the ACM/IEEE SC 2005 Conference*, Nov 2005, pp. 50–50.