Application Specific Memory Access, Reuse and Reordering for SDRAM

Samuel Bayliss and George A. Constantinides

Department of Electrical and Electronic Engineering,
Imperial College London, South Kensington Campus,
London SW7 2AZ, United Kingdom.
s.bayliss08@imperial.ac.uk, g.constantinides@imperial.ac.uk

Abstract. The efficient use of bandwidth available on an external SDRAM interface is strongly dependent on the sequence of addresses requested. On-chip memory buffers can make possible data *reuse* and request *reordering* which together ensure bandwidth on an SDRAM interface is used efficiently. This paper outlines an automated procedure for generating an application-specific memory hierarchy which exploits reuse and reordering and quantifies the impact this has on memory bandwidth over a range of representative benchmarks. Considering a range of parameterized designs, we observe up to 50x reduction in the quantity of data fetched from external memory. This, combined with reordering of the transactions, allows up to 128x reduction in the memory access time of certain memory-intensive benchmarks.

1 Introduction

On-chip memory hierarchies improve the performance of the memory system by exploiting *reuse* and *reordering*. Reusing data retained in an on-chip memory hierarchy reduces the number of requests made to the external memory interface. Where those data requests can be safely reordered, often the control overhead of servicing the requests can be reduced and hence bandwidth on the external interface is used more efficiently.

On general purpose computers, the sequence of memory requests generated by the CPU is generally unknown at design time, and the cache is able to reuse and reorder data without a static compile time analysis of the program to be executed. Complex dynamic memory controllers in modern CPUs also buffer and dynamically reorder requests to external memory. Both the cache and memory controllers therefore contain memory and associative logic to buffer and dynamically select and service memory requests.

However, when a specific application is targeted for hardware acceleration on an FPGA, it can often be broken into computation kernels for which static analysis is tractable. More specifically, for such kernels, the sequence of memory requests can often be determined at compile time. This makes possible the development of a custom application-specific memory system which exploits reuse and reordering of data requests to maximize bandwidth and reduce the average

request latency. This work describes a methodology for developing such a system and benchmarks demonstrating its efficacy. The key contributions are :

- A representation of the memory accesses contained within a loop body as a parameterized set.
- A methodology which uses parametric integer programming techniques to find a monotonic function which loads each set member only once.
- An implementation of that function as a state machine within an FPGA.
- An evaluation of the SDRAM bandwidth efficiency improvements achieved by such an approach.

2 Background

2.1 SDRAM Memory

Synchronous Dynamic Random Access Memories (SDRAMs) are used as the external off-chip memory in most general purpose computers. They are designed for high yield and low cost in manufacturing and have densities which exceed competing technologies. However, the design trade-offs needed to achieve these characteristics means much of the burden of controlling SDRAM memories falls on an external memory controller.

SDRAM memories store data as charged nodes in a dense array of memory cells. An explicit 'activate' command sent to the memory selects a row within the array before any reads or writes can take place. This is followed by the assertion of a sequence of memory 'read' or 'write' commands to columns within the selected row followed by a 'precharge' command which must be asserted before any further rows can be activated. The physical structure of the memory device determines the minimum time that must elapse between the issuing of 'precharge', 'activate' and 'read' / 'write' commands.

The memory controller is responsible for ensuring these minimum constraints are met. When reading consecutive columns within a row, SDRAM memories can sustain two word per clock cycle data-rates. With a 200MHz DDR2 Device, this means 400Mbytes/s. However the overhead of 'activate' and 'precharge' commands and the delays associated with their respective timing constraints means that achieving this peak bandwidth is difficult. In the worst case where a single 32-byte word is requested from memory from different alternating rows within the same bank, memory bandwidth is reduced to 72Mbytes/s.

2.2 Previous Work

High performance memory systems are a goal across the spectrum of computing equipment and a large body of work exists which seeks to improve cache performance, for which [8] provides a basic review. Most of the work in the literature assumes the sequence of data is randomly (but not necessarily uniformly) distributed, and describes optimizations of the dynamic on-chip structures which

exploit data locality. Where scratchpad memories have been used within a memory hierarchy, there are examples of static analysis to determine which specific memory elements are reused. Of particular note is the work of Darte *et al.* [5] and Liu *et al.* [12]. These two works both explore data-reuse using a polytope model. One develops a mathematical framework to study the storage reuse-problem, and the other is an application of the techniques within the context of designing a custom memory system implemented on an FPGA.

Prior work focused more specifically on SDRAM controllers can also be divided into work which seeks to optimize the performance of random data streams using run-time on-chip structures and those which use static analysis techniques on kernels of application code. A review of different dynamic scheduling policies for SDRAM controllers can be found in [15] showing that none of the fixed scheduling policies is optimal across all benchmarks. This is justification for the pursuit of an application-specific approach. The approach presented in [1] describes a memory controller designed to guarantee a minimum net bandwidth and maximum latency to many requestors in a complex SoC. A set of short templates are defined which optimize bursts of data and time-slots are allocated using a credit-controlled static priority arbiter to guarantee allocated bandwidth and bounded latency.

Static compile time approaches to improving SDRAM efficiency can be found in [10] where different data-layouts are used to exploit efficiency in a image-processing application. They propose a block-based layout of data within a data array rather than more traditional row-major or column-major layouts and develop a Presberger arithmetic based model to estimate the number of 'precharge' and 'activate' commands required in the execution of a video benchmark under different parameterizations. Their results show a 70-80% accuracy compared to simulation results and achieve up to 50% energy savings. Energy savings are considered specifically in [14] where an algorithm is developed for maximizing the number of possible concurrent memory accesses through a careful allocation of arrays to different banks.

Some work exists which considers the fine grained sequence of SDRAM commands. Dutt *et al.* [9] break with the traditional approach of treating loads and stores as atomic operations in a high level synthesis system and introduce fine grained SDRAM control nodes into their CDFG based synthesis flow.

None of the work mentioned above tackles the issue of data reuse (reducing the number of transactions issued to the external memory system) *and* reordering (reducing the latency associated with each transaction) in a comprehensive framework. The approach taken in this paper seeks to address this. We propose using on-chip memory to reuse data accessed within the innermost loops of a loop nest. Our methodology ensures that only the referenced data items are fetched into on-chip memory buffers and that the sequence of addresses on the external memory interface increases monotonically whilst filling those buffers. This final point ensures that the minimum number of 'precharge' and 'activate' commands are required, ensuring optimal SDRAM bandwidth efficiency.

3 Example

As a 'toy' illustrative example, consider the code shown in Figure 1(a). Two dimensional iteration vectors $[i, j]$ represent each iteration of the statement within the loop nest. Figure 1(c) shows how each of these vectors generates a memory address. The iterations $[0, 1]$ and $[2, 0]$ both access the same memory address. This data reuse by different iterations can be exploited by only loading the data at address 4 once. Furthermore the example contains 'holes'; the items $1, 3, 5$ and 7 are never accessed and need not be loaded. Finally, the access order implied by the ordering of the loop iterations implies that a non-monotonic sequence of addresses is generated by this code. Such a sequence implies that in the presence of page breaks, unnecessary 'precharge' and 'activate' commands are generated to swap between rows. Enforcing monotonicity on the address generating function minimizes the number of 'precharge' and 'activate' commands needed, improving overall bandwidth efficiency.

4 Problem Description

4.1 Polytope Model

We will restrict our scope to considering memory access sequences which are generated by references within a nested loop body.

A nested loop body such as the example in Figure 1(a) is made up of a number of different levels, each with an associated induction variable. The 2-level nested loop body shown in Figure 1(a) contains two induction variables (i and j). In the general case, an n-level nested loop can be represented by induction variables $x_1, x_2 \ldots x_n$ with the inner-most loop arbitrarily labelled with the highest index. Each execution of the statement(s) within the innermost loop is associated with a unique vector $x \in \mathbb{Z}^n$. If the loop nest is restricted further by the requirement that the bounds of each loop are affine functions of the induction variables which exist higher in the loop nest, the set of vectors, which we refer to as an iteration space is given by the set of integral points in a convex polytope. The loop bounds define this polytope and can be expressed in the form $Ax \leq b$. The loop bounds in Figure 1(a) can be expressed as $A = \begin{bmatrix} 1 & 0 \\ -1 & 0 \\ 0 & 1 \\ 0 & -1 \end{bmatrix}$ and $b = \begin{bmatrix} 2 & 0 & 1 & 0 \end{bmatrix}^T$.

The loops will contain memory addressing functions ($f : \mathbb{Z}^n \rightarrow \mathbb{Z}$) mapping induction variables to memory locations, *e.g.* in Figure 1(a), the function $f(i, j) = 2i + 4j$ describes memory accesses which load elements of the A array. In the case of an affine function $f(x)$, we may write $f(x) = Fx + c$ and describe the set of memory addresses accessed as (1).

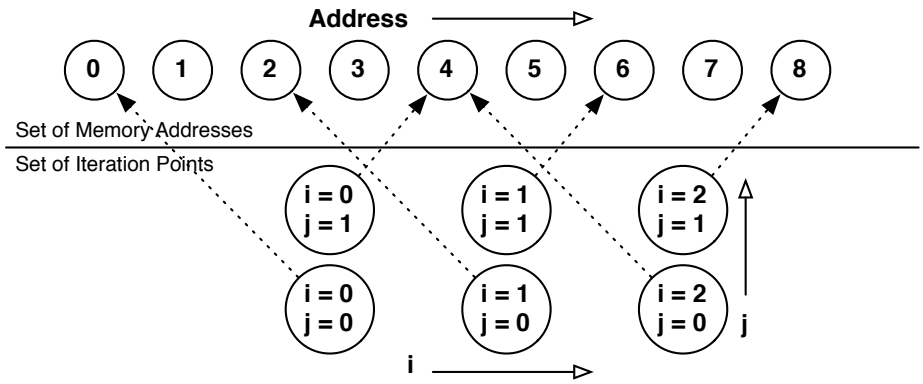
$$S = \{Fx + c \mid Ax \leq b\}. \tag{1}$$

```
int A[9];
for  (i = 0 ; i <= 2 ; i++) {
  for  (j = 0 ; j <= 1 ; j++) {
      func(A[2i + 4j]);
  }
}
```

(a) Source Code

```
if (-i - 2*j + 1 >= 0) {
  k = (2*i + 3) / 4;      /* Note: integer division */
  if (j + k - 1 >= 0) {
    R' = 2*i + 4*j + 2;
    i' = i - 2*k + 1;
    j' = j + k;
  }
  else {
    R' = 2*i + 2*j - 2*k + 2;
    i' = i + j - k + 1;
    j' = 0;
  }
}
else {
  if (-i - j + 3 >= 0) {
    R' = 2*i + 4*j + 2;
    i' = i + 2*j - 1;
    j' = 1;
  }
  else {
    // No Solution.
  }
}
```

(b) Output Code



(c) Mapping Diagram for source code

Fig. 1. (a) Source code for example (b) Output code listing solution for example (c) Mapping from (i, j) iteration space to memory addresses for code (a).

The memory referencing function ($f : \mathbb{Z}^n \rightarrow \mathbb{Z}$) can be non-injective, which means different iterations within the loop nest can access the same item in memory.

Many common image processing and linear algebra problems can be expressed in this polytope framework. Scheduling and loop transformations which alter the execution order within this polytope framework have been extensively studied [11, 7, 4] either to expose parallelism by eliminating loop carried dependencies, or to reduce the communications cost when partitioning code across many processors in a HPC system.

4.2 Memory Buffering and Reordering

While the work in [11, 7, 4] considers reordering the statements within a loop nest, we propose a different approach; preserving the original execution order, we propose the introduction of memory buffers (reuse buffers) at levels within the loop body. Unlike caches, which are filled by an implicit hardware mechanism, these memories are explicitly populated with data using code derived from the original problem description. External memory references in the innermost loop are replaced by references into the reuse buffers. The memory buffer can be introduced at any level within the loop nest, we define a parameter $1 \leq t \leq n$ which describes the placement of the buffer within a n-level loop nest. A buffer introduced outside the outermost loop and explicitly populated with all the data referenced within the loop-nest is referred to as $t = 1$ while a buffer inserted at $t = 2$ must be refilled every time the outermost loop index is incremented and contains only data accessed within that iteration of the outer loop.

Because such an approach does not alter the *execution* order of statements, introduction of memory buffers can be done without specifically considering data dependencies within the code. Prior work using explicitly populated memory buffers in this way can be found in [13] and [2].

The novelty of our work is derived from the reordering of the memory requests used to populate the reuse buffers. The reordering considers the most efficient access pattern for SDRAM, both exploiting the reuse of data elements with a loop nest *and* rearranging the data accesses so that all the data that lies within a SDRAM row is accessed consecutively, thus minimising the overhead of 'precharge' and 'activate' commands.

The most efficient ordering for transferring data between the external memory and internal buffer is by visiting each row only once, minimizing the number of row swaps necessary. If the function describing the scheduling of loads and stores to the memory is strictly monotonic, this property will hold. The property ensures that memory reuse is exploited (strictly increasing functions are injective, therefore each data item can be loaded only once) and that the minimum overhead of row swaps is incurred.

5 Parametric Integer Linear Programming Formulation

Integer Linear Programs are optimization problems which seek to minimize an objective function subject to a set of linear constraints while forcing only integer assignments to the optimization variables. This is illustrated in the formulation in Equation 2. This concept has been extended to parametric integer programming (Equation 3) where the constraints can be described in terms of some parameter q thus producing p as an explicit *function* of q rather than as an integer vector as in ILP. We can use parametric integer linear programming to derive an address generating function for populating reuse buffers within our proposed application specific memory subsystem.

$$\min_{p} k^T p \quad s.t. \quad Ap \leq b \tag{2}$$

$$\min_{p} k^T p \quad s.t. \quad Ap \leq b + Cq. \tag{3}$$

5.1 Constraints

Our objective is to derive a function which generates a sequence of memory addresses populating a reuse buffer. We propose a method which produces an address generating function directly from the polytope problem definition using parametric integer linear programming with embedded constraints which ensure that the strict monotonicity property holds. The solution takes the form of an iterative function which generates an address (R') and $n + 1 - t$ state variables which are used to generate subsequent addresses. This is illustrated in Figure 2.

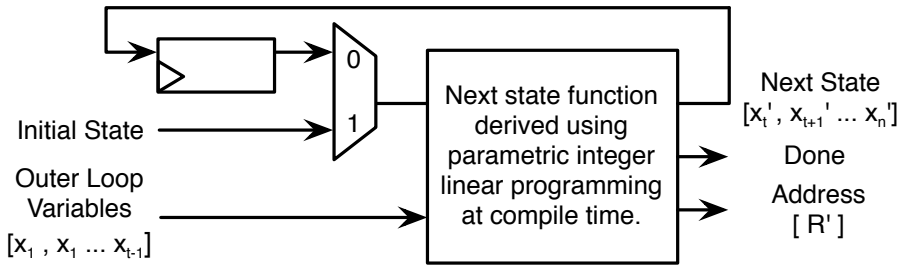


Fig. 2. Diagram showing how the Piplib derived addressing function is implemented.

For a reuse buffer introduced at a specific level t in the loop structure we define the parametric integer linear program in terms of the variable $p = [R', x'_t, x'_{t+1}, \ldots, x'_n]^T$, representing the next address and inner loop iterators and parameterized in terms of variables $q = [x_1, x_2, \ldots, x_n]^T$. Given the current value q of the loop iterators, our objective is simply to find the first address accessed in the code after $Fq + c$, the current address. This corresponds to

minimizing R' subject to the constraints we outline below. In explaining these constraints, we make reference to the example in Figure 1(a) and assume that a reuse buffer is inserted at the outermost level (outside the loop nest). Therefore for this example $p = [R', i', j']^T$, $q = [i, j]$ and $t = 1$.

The first constraint is an inequality constraint which ensures strict monotonicity, i.e. that the next address will be one more than the current address.

$$R' \geq Fq + c + 1 \tag{4}$$

In our example code, this corresponds to the constraint $R' \geq 2i + 4j + 0 + 1$.

The second constraint is an equality constraint that ensures that the new address (R') can be generated by the memory access function $f(x) = Fq + c$ through a linear combination of the domain variables $x_t, x_{t+1} \ldots x_n$. Where $F = [F_1\ F_2]$:

$$R' - (F_1[x_1 \ldots x_{t-1}]^T + F_2[x'_t, x'_t, \ldots, x'_n]^T) = c \tag{5}$$

In our example, this corresponds to the constraint $R' - 2i' - 4j' = 0$.

The remaining constraints ensure that the combination of the instantiation of the variables $x'_t, x'_{t+1}, \ldots x'_n$ and the parameter variables $x_1 \ldots x_{t-1}$ lie within the iteration space. Where $A = [A_1\ A_2]$:

$$A_1[x_1 \ldots x_{t-1}]^T + A_2[x'_t, x'_{t+1}, \ldots x'_n]^T \leq b \tag{6}$$

For our example, the constraints $i' \leq 2$, $-i' \leq 0$, $j' \leq 1$ and $-j' \leq 0$ ensure this.

The variables $q = [x_1, x_1 \ldots x_n]^T$ are constrained by the constraints implied by the original polytope. In our example code, this means $i \leq 2$, $-i \leq 0$, $j \leq 1$ and $-j \leq 0$.

5.2 Output

The problem is formulated at design time and used as an input to a parametric integer linear programming tool Piplib [6]. The solution returned by the tool is a piecewise linear function defined as a binary decision tree. The output from the example in Figure 1(a), when converted to an equivalent C code, is illustrated in Figure 1(b). If this code is evaluated with the initial indices ($i = 0$, $j = 0$), an address ($R' = 2$) and a further two indices $i' = 1, j' = 0$ are generated. When these are iteratively evaluated using the same function, the sequence of addresses (2,4,6,8) is generated. This is exactly the memory addresses touched by the original input code, presented in a strictly increasing order and with all repetition removed. This ensures no unnecessary row-swaps are incurred and the minimum number of 'precharge' and 'activate' commands are generated.

6 Hardware Implementation

The output from the parametric integer linear programming solver implicitly defines a state machine with state defined by $(x'_t, x'_{t+1}, \ldots x'_n)$, a single output (R') and $t-1$ inputs $(x_1 \ldots x_{t-1})$. Using Handel-C [3] we formed a direct implementation of that state machine within an FPGA and used the address generated to drive the Altera High Performance DDR2 Memory Controller. Verilog code generated from Handel-C was synthesized using Quartus II for a Stratix III target device.

7 Results and Discussion

Three benchmarks are presented to measure the effectiveness of our proposed technique in improving memory bandwidth.

Matrix-Matrix Multiply (MMM) In this benchmark two 50x50 dense matrices of 64-bit values are multiplied using a classic 3-level loop nest implementation. One matrix is accessed in a column-wise fashion and the other in a row-wise fashion.

Sobel Edge Detection (SED) This benchmark is a 2D convolution of a 512x512 matrix of 32-bit values with a 3x3 kernel. Because each iteration requires pixels from three consecutive rows of an image *and* three consecutive pixels in each row, neither row or column major storage of the input array can mitigate poor data locality.

Gaussian Back-substitution (GBS) This benchmark optimizes the memory pattern in a blocked back-substitution kernel. It illustrates that the methodology presented is equally applicable to non-rectangular loop-nests and where not all data in a (square) input array need be accessed.

Each design is parameterized by inserting reuse buffers at different levels within the loop nest (where $t = 1$ implies a reuse buffer inserted outside the outermost loop of the benchmark code). The designs were simulated using a memory controller running at 240MHz with the address generation logic running at the post-fit reported F_{max} frequency.

Synthesis results giving area and speed for the derived address generators are given in Table 1 and reflect the fact that as t is decreased, the conditions evaluated at each clock cycle require more terms (and more multipliers) and a longer critical path. The F_{max} clock speeds of each implementation fall significantly below the peak rated performance of the embedded hardware resources on which they are implemented. The long critical paths in the arithmetic functions used to generate next-state variables for our state machine implementation prevent higher clock rates. Since the parameters calculated in one cycle are used to generate those in the next, pipelining cannot be used without reducing the overall throughput of the address generator.

Yet in spite of this reduction in achievable clock frequency as t is decreased; the wall clock time taken to load data in each benchmark reduces as t is decreased. A comparison of parameterization with $t = 1$ and the original code

shows reduction of 128×, 29× and 7× in the wall clock time of the three benchmarks respectively. This significant performance improvement occurs because our methodology increases opportunities for reuse *and* reordering as t decreases (with buffers inserted at the outermost levels of the loop nest). Three mechanisms can be identified which contribute to this.

1. The first mechanism we can observe is the reduction in the number of read requests through the exploitation of data reuse. Data reuse means some data held in the on-chip reuse buffer is accessed by more than one different iteration within the loop nest. On-chip buffering means that only a single request to external memory is made for each reused item. Table 1 shows how the number of read cycles to external memory decreases as the reuse buffer is instantiated in the outer levels of the loop nest (*i.e.* as t decreases).
2. The second mechanism for the reduction in overall wall clock time is better locality within each memory reference. Figure 3 shows a proportional breakdown of the SDRAM commands generated by a single 'read' reference within the GBS benchmark. When parameterized at $t = 1$ and $t = 3$, the number of memory accesses is constant, however the monotonic order in which they are accessed when $t = 1$ reorders those accesses. This eliminates the large overhead of 'precharge' and 'activate' cycles which in turn means a greater than 2× reduction in total loading time is seen over parameterization at the the innermost level ($t = 3$).
3. The third observable mechanism is the reduction in the interleaving of accesses to different arrays. In general, when bursts of memory access to different arrays are serialized, the inevitable interleaving of accesses to different arrays introduces row-swaps. Under our approach, when parameterization is at the outer levels of the loop nest, the bursts of addresses are longer and there are fewer total interleavings thus the overhead of 'precharge' and 'activate' commands is reduced.

8 Conclusions and future work

In this paper, we have presented a novel methodology for generating application-specific address generators which exploit data reuse and reorder data accesses. In selected benchmarks, data reuse allows a 50x reduction in the number of memory accesses. The constraints which ensure the sequence of memory addresses requested by each access function is strictly increasing ensure an efficient use of SDRAM memory bandwidth. When exploited together, these two aspects provide up to 128× reduction in overall wall clock time in memory intensive benchmarks.

We are actively investigating methods for compactly representing the solutions produced by the parametric integer programming tool with the aim of increasing the achievable F_{max} clock speed. There remains scope for even more efficient bandwidth utilization by considering application specific *command* generation within an SDRAM controller alongside the proposed address generation scheme and extending our formulation to optimize the scheduling of memory refresh accesses.

Table 1. Area Consumed by address generation logic (post synthesis and P&R), breakdown of memory cycles and total wall clock time for memory accesses in each benchmark.

Benchmark	Reuse Level	LUTs	18-bit DSPs	Clock Speed	Read / Write Cycles	'Pre'&'Act' Cycles	Other Cycles	Wall Clock Time
MMM	t = 1	1504	0	55MHz	29992	878	22597	229 μs
MMM	t = 2	831	0	56MHz	520000	15802	394988	2874 μs
MMM	t = 3	310	0	145MHz	1009954	571448	59946	6952 μs
MMM	Orig.	242	0	145MHz	1499952	3789438	1649683	29365 μs
SED	t = 1	1951	112	40MHz	2088988	46378	2288487	18440 μs
SED	t = 2	1314	80	43MHz	4192180	106258	4550859	36880 μs
SED	t = 3	930	68	52MHz	19767560	13202772	1741317	144639 μs
SED	t = 4	482	28	107MHz	21848356	16772328	1182698	174148 μs
SED	Orig.	462	24	117MHz	28090752	70959548	30795889	535623 μs
GBS	t=1	2948	128	35MHz	35048	5178	32881	318 μs
GBS	t=2	1501	76	46MHz	35812	9038	28046	309 μs
GBS	t=3	526	36	120MHz	67536	168992	45364	1175 μs
GBS	Orig.	356	16	75MHz	165078	310758	98622	2341 μs

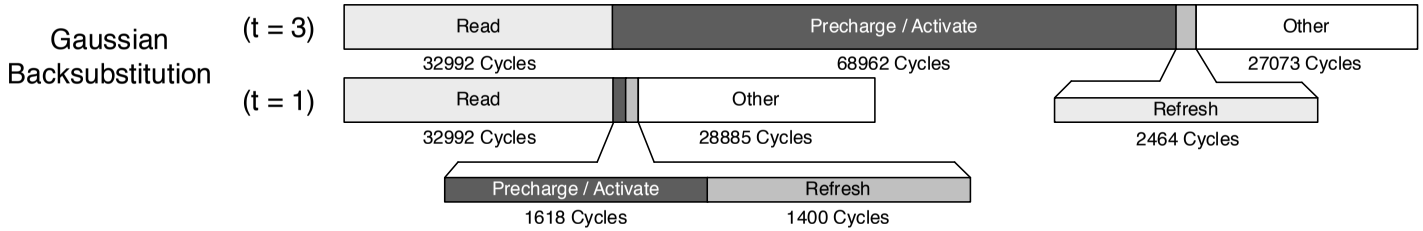


Fig. 3. Figure showing SDRAM bandwidth allocation by command type within a single reference in the Gaussian Backsubstitution benchmark.

References

1. B. Akesson, K. Goossens, and M. Ringhofer. Predator : A Predictable SDRAM Memory Controller. In *CODES+ISSS '07 : Proceedings of the 5th IEEE/ACM International Conference on Hardware/Software Codesign and System Synthesis*, pages 251–256, Salzburg, Austria, 2007.
2. N. Baradaran, P. C. Diniz, A. Way, and M. Rey. Compiler-Directed Design Space Exploration for Caching and Prefetching Data in High-level Synthesis. In *FPT 05 : Proceedings of the IEEE International Conference on Field Programmable Technology*, pages 233–240, Singapore, 2005.
3. Celoxica Ltd. DK4 : Handel-C Language Reference Manual, 2005.
4. M. Claßen and M. Griebl. Automatic code generation for distributed memory architectures in the polytope model. In *IPDPS '06 : 20th International Parallel and Distributed Processing Symposium*, pages 243–250, Rhodes, Greece, 2006.
5. A. Darte, R. Schreiber, and G. Villard. Lattice-Based Memory Allocation. *IEEE Transactions on Computers*, 54(10):1242–1257, 2005.
6. P. Feautrier. Parametric Integer Programming. *RAIRO Recherche Opérationnelle*, 22(3):243–268, 1988.
7. P. Feautrier. Automatic Parallelization in the Polytope Model. *The Data Parallel Programming Model : Foundations, HPF Realization, and Scientific Application*, pages 79–103, 1996.
8. J. Hennesey and D. Patterson. *Computer Architecture : A Quantitative Approach.* Morgan Kaufmann, 6th editio edition, 2006.
9. A. Khare, P. R. Panda, N. D. Dutt, and A. Nicolau. High-Level Synthesis with SDRAMs and RAMBUS DRAMs. *IEICE Transactions on Fundamentals of Electronics, Communications, and Computer Sciences*, E82A(11):2347–2355, 1999.
10. H. S. Kim, N. Vijaykrishnan, M. Kandemir, E. Brockmeyer, F. Catthoor, and M. J. Irwin. Estimating Influence of Data Layout Optimizations on SDRAM Energy Consumption. In *ISLPED '03 : Proceedings of the 2003 International Symposium on Low Power Electronics and Design*, pages 40–43, Seoul, South Korea, 2003.
11. C. Lengauer. Loop Parallelization in the Polytope Model. In *CONCUR '93 : Proceedings of the 4th International Conference on Concurrency Theory*, pages 398–417, Hildesheim, Germany, 1993.
12. Q. Liu, G. A. Constantinides, K. Masselos, and P. Y. K. Cheung. Automatic On-chip Memory Minimization for Data Reuse. In *FCCM '07 : Proceedings of the 15th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, pages 251–260, Napa Valley, CA, USA, 2007.
13. Q. Liu, K. Masselos, and G. A. Constantinides. Data Reuse Exploration for FPGA based Platforms Applied to the Full Search Motion Estimation Algorithm. In *FPL '06 : 16th International Conference on Field Programmable Logic and Applications*, 2006.
14. P. Marchal, D. Bruni, J. Gomez, L. Benini, L. Pinuel, F. Catthoor, and H. Corporaal. SDRAM-Energy-Aware Memory Allocation for Dynamic Multi-Media Applications on Multi-Processor Platforms. In *DATE '03 : Proceedings of the Design, Automation and Test in Europe Conference and Exhibition*, pages 516–521, Munich, Germany, 2003.
15. S. Rixner, W. J. Dally, U. J. Kapasi, P. Mattson, and J. D. Owens. Memory Access Scheduling. In *ISCA '00 : Proceedings of the 27th Annual International Symposium on Computer Architecture*, volume 28, pages 128–138, Vancouver, BC, Canada, May 2000.