Optimizing SDRAM Bandwidth for Custom FPGA Loop Accelerators

Samuel Bayliss and George A. Constantinides
Department of Electrical and Electronic Engineering
Imperial College London
London SW7 2AZ, United Kingdom
{s.bayliss08, g.constantinides}@imperial.ac.uk

ABSTRACT

Memory bandwidth is critical to achieving high performance in many FPGA applications. The bandwidth of SDRAM memories is, however, highly dependent upon the order in which addresses are presented on the SDRAM interface. We present an automated tool for constructing an application specific on-chip memory address sequencer which presents requests to the external memory with an ordering that optimizes off-chip memory bandwidth for fixed on-chip memory resource. Within a class of algorithms described by affine loop nests, this approach can be shown to reduce both the number of requests made to external memory and the overhead associated with those requests. Data presented shows a trade off between the use of on-chip resources and achievable off-chip memory bandwidth where a range of improvements from 3.6× to 4× gain in efficiency on the external memory interface can be gained at a cost of up to a 1.4× increase in the ALUTs dedicated to address generation circuits in an Altera Stratix III device.

Categories and Subject Descriptors

B.5.2 [**Design Aids**]: Automatic Synthesis

Keywords

FPGA, Memory, SDRAM, Loop Transformations

1. INTRODUCTION

The number of pins available on semiconductor devices, and the data rate available on such pins, has not scaled with transistor density [1]. This, among other factors has contributed to a 'memory wall' in which the parallelism achievable with computing devices is limited by the speed in which large off-chip memories can be accessed. 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 on 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 more efficiently used.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
FPGA'12, February 22–24, 2012, Monterey, California, USA.
Copyright 2012 ACM 978-1-4503-1155-7/12/02 ...$10.00.

The impact of this reordering can be seen when Dynamic Random Access Memories (DRAMs) are used as the external off-chip memory. DRAM devices have a hierarchical structure in which a small number of independent memory arrays (banks) are themselves composed of rows and columns. A row within a bank must be explicitly opened ('activated') before columns within it can be accessed, and closed ('precharged') before another row can be selected. While DRAM capacity has scaled in line with Moore's observations [19], increased cell density has been used to deliver increased capacity for a given silicon die area. This means the wire-lengths upon which 'precharge' and 'activation' times are dependent are not significantly shorter than 10 years ago. With smaller transistors delivering shorter clock periods, the overall impact is that timing delays associated with 'precharge' and 'activation' *cycles* take up an ever increasing proportion of clock cycles on the external memory interface. The net effect is that the bandwidth efficiency of external memory interfaces has reduced with each generation of SDRAM device.

On general purpose computers, the sequence of memory requests generated by the CPU is typically unknown at design time, so the processor cache is designed to be 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 buffer and dynamically reorder cache line-fill 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 [15]. As well as the direct area cost this imposes, caches and dynamic memory controllers make it very difficult to predict memory performance at compile-time.

In this paper we ask the question: "What can be done in the memory controller to improve bandwidth efficiency without sacrificing predictability?". In essence, we aim to trade logic resources (whose availability increases with process scaling in line with Moore's observations [19]) for increased memory interface bandwidth (which does not). While this area has been explored in the context of dynamic memory controllers and caches which operate on random streams of data, we focus instead on opportunities for a *statically* scheduled reordering of memory requests. Such an approach means we can exactly determine memory access latency and bandwidth at compile time for each memory access. This knowledge might be exploited in a high level synthesis flow to improve resource scheduling algorithms.

For a given application targeted for hardware acceleration on an FPGA, the structure of computation kernels often makes static analysis tractable. More specifically, for such kernels, the sequence of memory requests can often be determined at compile time. In this paper, we give a methodology grounded in a formal computation framework (the Polyhedral Model [16, 20]) for reordering

memory requests for such kernels. We present results which are specific to hardware implementation, in that they exploit the reconfigurability of the device by creating a custom memory architecture for a specific application, and also take advantage of the abundance of arithmetic logic components available on modern FPGAs to build sophisticated controller structures.

The key contributions are :

- A parametric representation of the set of SDRAM rows accessed by a memory reference within a loop nest as the integer points enclosed in a convex polytope, avoiding the need for enumeration.
- An efficient procedure to optimize the size of such a parametric representation, corresponding to optimizing the FPGA resources dedicated to constructing an address generator.
- A method of code generation from the optimized set description which minimizes the number of row swaps performed by, and the number of commands issued to, the SDRAM memory controller.
- A method for producing an efficient and flexible pipelined memory address sequencer.
- An evaluation of the SDRAM bandwidth efficiency improvements achieved by such an approach, considering separately the improvements attributable to the reuse of data and the reordering of data requests to external memory.
- An evaluation of the cost of that transformation in terms of additional logic required in address generation and on-chip data storage.

We have implemented our method as a complete design flow that reads in a subset of C, extracts a polyhedral representation and generates a Verilog implementations of an optimized SDRAM address sequencer. We would encourage readers to try the flow via the web interface, available at `http://cas.ee.ic.ac.uk/AddrGen`.

2. BACKGROUND

2.1 SDRAM Memory

DRAMs 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 DRAM memory access 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 which must elapse between the issuing of 'precharge', 'activate' and 'read'/'write' operations. The on-chip memory controller is responsible for ensuring these timing constraints are met. When reading consecutive columns within a row, DDR SDRAM memories can sustain two words-per-clock-cycle data-rates. With a 200MHz DDR2 device (with 8-bit wide data bus), this means 400MBytes/s. However the overhead of 'precharge' and 'activate' commands and the delays associated with their respective timing constraints means that achieving this peak bandwidth is rare. In the worst case, where single bursts are requested from different rows within the same bank, memory bandwidth is reduced to 20% of its peak rate.

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 ([13] provides a comprehensive review). Most of this work assumes the sequence of addresses is randomly (but not necessarily uniformly) distributed and describes optimizations of dynamic on-chip structures to 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.* [11] and Liu *et al.* [18]. 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 technique within the context of designing a custom memory system implemented on an FPGA, but without considering the impact that ordering has on performance.

The 'Connected RAM' (CoRAM) methodology presented in [10] is notable in that, in common with our approach, it seeks to decouple communication and computation threads within an application. Their methodology lacks a compilation framework to extract communication threads from high level descriptions and optimally schedule those threads to access external memory, and our work presented here is a step towards the realization of that goal.

Prior work focused more specifically on SDRAM controllers can be divided into work that seeks to optimize the performance of random data streams using runtime on-chip structures, and that which uses static analysis techniques on kernels of application code. A review of different dynamic scheduling policies for SDRAM controllers can be found in [23]; the results show that none of the fixed scheduling policies are optimal across all benchmarks, providing a motivation for our approach of using FPGA programmability to pursue an application-specific approach. The method presented in [2] describes a memory controller that guarantees an allocation of bandwidth and bounded access 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 dynamic credit-controlled priority arbiter. The static approach we demonstrate in this paper enables fine-grained scheduling of datapath operations that is difficult using a dynamic approach.

Other static compile time approaches to improving SDRAM efficiency can be found in [17] where different data layouts are used to improve efficiency in a image processing application. A block-based layout of image data is proposed rather than a traditional row-major or column-major layouts and a Presberger arithmetic model is used estimate the number of 'precharge' / 'activate' operations required in the execution of a video benchmark. Their results show a 70-80% accuracy compared to simulation results and achieve up to 50% energy savings. In [9], a strategy is proposed for allocating arrays to different memory banks to hide the latency of row activation. Their heuristic approach assumes each logical row of an allocated array fits within an SDRAM row; an assumption is likely to be restrictive in handling large data-sets. While our proposed methodology does not consider bank allocation directly, we believe it complimentary to the concept demonstrated in [9], since by reordering memory accesses to cluster together accesses to the same row, SDRAM rows which are accessed consecutively can be allocated to different banks with a simple permutation of address bits.

To the best of our knowledge, our work is the first to propose static analysis of loop nests for developing hardware address generators for application-specific SDRAM-optimized memory reordering *and* data reuse.

```
char A[56];
for (x1 = 0 ; x1 <= 2 ; x1++) {
  for (x2 = 2 - x1 ; x2 <= 2 ; x2++) {
    for (x3 = x1; x3 <= x2 ; x3++) {
      A[7*x1 + 8*x2 + 9*x3] = func(...);
    }
  }
}
```

Figure 1: C source code for 3-level nested loop example.

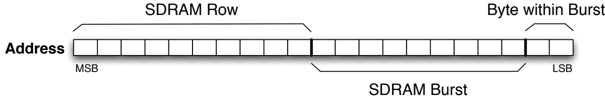

Figure 2: Memory address and associated fields.

3. MOTIVATING EXAMPLE

The example shown in Figure 1 provides a motivation for our methodology, and will be used as a running example throughout the paper to illustrate the algorithms. This is a 'toy' loop nest with three levels ($n = 3$) and a memory array within the innermost loop. The array A is assumed to reside in external SDRAM memory and, for didactic reasons, we assume that each row in the memory has length of 16 bytes and each individual memory request is made of bursts of 4 bytes. In real SDRAMs, of course the the size of each row is much greater (as is the number of accesses made in a real application), but we have tried to keep this example as simple as possible to illustrate the key features and novelties of our approach.

For simplicity of exposition, we assume the array A originates at address 0 in memory. Thus element $A[i]$ of the array resides at memory address i. Every memory request to DDR2 SDRAM is made with the selection of a unique bank and row within the memory followed by a burst request (of 4 or 8 words). Considering only a single bank within the device, each memory address within that bank can be divided into three bit fields, corresponding to SDRAM Row, Burst, and Byte Within Burst, as shown in Fig. 2. These three fields can be represented as vectors in $\mathbb{Z}^3$ where the three dimensions denote the Row, Burst and Byte Within Burst respectively.

When running the code in Figure 1 without transformation, a sequence of seven memory requests is generated, as shown in order in Table 1. This sequence exhibits several features. Firstly, the ordering of requests means that both the first and second rows of the SDRAM are opened more than once. SDRAM timing constraints mean a significant penalty is incurred when 'activating' and 'precharging' SDRAM rows, hence this is an inefficient order to access the memory; a more efficient order would activate each row only once. Secondly, in some rows, there are multiple accesses to the same burst, for example Row 2 Burst 0 is accessed both by $(x_1, x_2, x_3) = (0, 2, 2)$ and by $(x_1, x_2, x_3) = (1, 2, 1)$. If we can store the data from the burst in on-chip memory and reuse it later in the computation, we can reduce the number of external memory transactions. The final important feature of this sequence is the presence of 'holes': not all bursts are accessed within each row; indeed, burst number 1 is never accessed for any row. Careful attention to these holes is important in ensuring code is correct (since spurious write operations corrupt data) and efficient (since non-essential read operations reduce bandwidth efficiency).

Our aim, therefore, is to establish an automatic methodology for deriving an efficient memory subsystem capable of addressing these three features, by reordering external memory accesses when appropriate, by storing reused data on-chip when possible, and by ensuring only those memory locations accessed by the original code are accessed by the derived memory subsystem.

Table 1: Sequence of memory accesses generated by example in Figure 1.

Order	x_1	x_2	x_3	Array Index	Row	Burst
1	0	2	0	16	1	0
2	0	2	1	25	1	2
3	0	2	2	34	2	0
4	1	1	1	24	1	2
5	1	2	1	32	2	0
6	1	2	2	41	2	2
7	2	2	2	48	3	0

4. METHODOLOGY

In our methodology, we restrict ourselves to kernels of code for which the set of associated memory accesses can be determined at compile-time and are independent of any inputs to the program. This is referred to as the static control portion of a program. We also restrict ourselves to code that can be expressed as a perfect loop nest using normalised loop indices. Loops in this form are easily expressed in a mathematical form referred to as the Polyhedral Model, which enables reasoning about code transformations. Examples naturally arise in most DSP and Scientific Computing applications [20, 21, 22]. However, we are not restricted to these domains, recent advances show that we can convert imperfect loop nests into canonical forms [8] and demonstrate that static control code kernels can be automatically extracted from intermediate compiler representations [12] further broadening the applicability of our technique.

The steps in our compilation flow are described in the flowchart in Figure 3, with each step enumerated below.

1. Parse the 'C' kernel code and construct a polytope representation.
2. For each memory reference, augment the polytope description with a variable representing the SDRAM row (r) and burst (u) accessed.
3. For each memory reference, find a unimodular matrix and change of variables such that the maximum number of variables can be eliminated from the polytope by the sufficient conditions in [24].
4. Check the necessary conditions for elimination in [24] for the remaining variables and use code generation tools to generate transformed code with a reordered loop structure.
5. Generate pipelined hardware which implements the loop indexing function

In the sections which follow, we formulate an initial problem description and describe each of these steps, demonstrating each transformation using our example code from Figure 1.

4.1 Decoupling Memory Access from Execution using On-Chip Memory

In any useful program, there is a mix of read accesses and write accesses. In general, freedom to reorder the statements executing in a program is restricted by data dependencies within the program: the true read-after-write dependencies within the code prevent arbitrary reordering of loop structures. We tackle this problem by

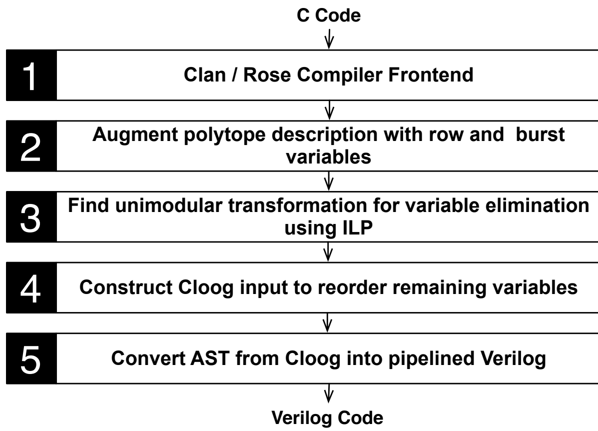

Figure 3: Flowchart showing steps in methodology.

separating a program into separate communication and execution threads. The simplest legal schedule sees all the data required for program execution loaded into on-chip memory before execution begins, and then the data items written during execution are buffered and written back after execution completes.

In many cases, such a simple communication/execution split is infeasible, because the on-chip memory is not large enough to simultaneously store all the data read and written within a loop nest. In such a case, the set of memory items corresponding to some specific iteration of an outer loop (or outer loops) can be loaded into on-chip memory, execution of the inner loops can progress and when complete, data written during execution of those inner loops can be written back to memory before repeating for the next iteration of the outer loops.

We can introduce on-chip memory buffers for decoupling memory access from execution at any level of the loop nest and use the parameter t to denote the level at which a buffer is introduced. Where $t = 1$, this denotes we indicate the introduction of a buffer at the outermost level of the loop nest and the requirement to prefetch all the data required for execution of the loop nest before execution begins. At the opposite extreme, parameterisation where $(t = n + 1)$ indicates introduction of a buffer large enough to contain just the elements accessed in a single iteration of the innermost loop. While not considered within this paper, standard loop-tiling transformations can be used to give the user even finer control over the size of the required on-chip data buffer.

For some chosen level of parameterisation, the set of read accesses generated by the inner loop nest for a specific outer loop iteration vector can be arbitrarily reordered, as can the set of write accesses, and the data-dependencies are guaranteed to be satisfied under the condition that all the read accesses required to fill the data buffer occur before execution and the write accesses required to commit results are scheduled after execution and before data is fetched for the next iteration.

In the sections that follow, we present a methodology for exploiting data reuse and reordering memory transactions using the abstract notation of the Polyhedral Model, for a fixed value of t.

4.2 Representing Memory Accesses in the Polyhedral Model

The Polyhedral Model is a mathematical description of a sequence of computations. For code which can be expressed within its restrictions, the Polyhedral Model provides a basis for reasoning about loop transformations, data dependencies and memory access patterns.

Consider the example in Figure 1 which contains three nested loops ($n = 3$). There is an loop variable for each level in the loop (labelled x_1, x_2 and x_3). In the general case, these loop variables are $x_1, x_2, \ldots, x_n$ 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 iteration vector $x \in \mathbb{Z}^n$. The bounds of each loop are affine functions of the loop (induction) variables of outer levels in the loop nest. The set S_E of integer vectors executed by a given loop nest, which we refer to as the *iteration space*, can be represented parametrically as the integer points contained within a polytope described as a set of linear inequalities $Ax \leq b$, where A is an $m \times n$ matrix, b is an m-vector, and the vector inequality is interpreted as $x \leq y$ iff $x_i \leq y_i$ for all i.

For the loop in Figure 1, the polytope representing the specified loop bounds is given in (1) :

$$\begin{pmatrix} -1 & 0 & 0 \\ 1 & 0 & 0 \\ -1 & -1 & 0 \\ 0 & 1 & 0 \\ 1 & 0 & -1 \\ 0 & -1 & 1 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} \leq \begin{pmatrix} 0 \\ 2 \\ -2 \\ 2 \\ 0 \\ 0 \end{pmatrix} \tag{1}$$

The loops contain memory references within the innermost loop, where the array indexing functions are themselves affine functions of the loop variables, *i.e.* of the form $A[fx + h]$ where f is an n-dimensional row vector and h is a scalar. For the example code, $f = [7\ 8\ 9]$, $h = 0$. We can describe the set S_M of memory addresses accessed within a loop nest as in (2). For our example code, if we were to enumerate the elements of this set, we would obtain $S_M = \{16, 24, 25, 32, 34, 41, 48\}$, as illustrated in Table 1. Crucially, for each memory reference, the number of elements in the set S_M is always less than or equal to the number of iteration vectors in S_E. This is because, while each memory reference accesses only a single memory element, multiple iteration vectors can access the same memory element. Exploitation of this is referred to as data *reuse* since elements in S_M could be stored in on-chip memory and reused on more than one iteration in S_E.

$$S_M = \{fx + h \mid \exists x \in S_E\} \tag{2}$$

Each of the memory accesses in S_M corresponds to a specific row and aligned burst in external SDRAM memory. Beyond data-reuse, we can achieve higher off-chip memory bandwidth by reordering accesses so that accesses to the same row in external memory are grouped together and thus the number of row-swaps (and associated 'precharge' and 'activate' commands) is minimized. We represent SDRAM rows and bursts *explictly* in the Polytope Model to help us reorder accesses for improved bandwidth efficiency.

4.3 Explicit Representation of SDRAM rows in the Polytope Model

The first step of our procedure is to explicitly represent the rows and bursts of SDRAM access by introducing new variables into the polytope representing the iteration space.

If the size of each SDRAM row is R words, the row accessed by memory address fx is given by $r = fx \operatorname{div} R = \lfloor fx/R \rfloor$, where $\lfloor\cdot\rfloor$ represents the *floor* function. If the size of each SDRAM burst is B words, then the burst number is similarly given by $u = \lfloor (fx - rR)/B \rfloor$. Unfortunately, neither of these representations is amenable to linear algebraic manipulation, due to the floor functions.

However, we may note that from the properties of the floor function:

$$\lfloor \frac{fx}{R} \rfloor - 1 < r \leq \lfloor \frac{fx}{R} \rfloor \quad (3)$$

and

$$\lfloor \frac{fx - rR}{B} \rfloor - 1 < u \leq \lfloor \frac{fx - rR}{B} \rfloor \quad (4)$$

which we can write as the linear equalities below, without loss of information

$$fx - R + 1 \leq Rr \leq fx \quad (5)$$

$$fx - rR - B + 1 \leq Bu \leq fx - rR \quad (6)$$

We then add these 4 extra inequalities to those already present defining the loop bounds, to form an augmented system of linear inequalities that completely describe not only the iteration space, but the SDRAM rows and bursts accessed within the innermost loop:

$$\begin{pmatrix} A & 0 & 0 \\ f & -R & 0 \\ -f & R & 0 \\ f & -R & -B \\ -f & R & B \end{pmatrix} \begin{pmatrix} x \\ r \\ u \end{pmatrix} \leq \begin{pmatrix} b \\ R-1 \\ 0 \\ B-1 \\ 0 \end{pmatrix}. \quad (7)$$

The corresponding augmented system for our Figure 1 is shown below

$$\begin{pmatrix} -1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 \\ -1 & -1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & -1 & 0 & 0 \\ 0 & -1 & 1 & 0 & 0 \\ 7 & 8 & 9 & -16 & 0 \\ -7 & -8 & -9 & 16 & 0 \\ 7 & 8 & 9 & -16 & -4 \\ -7 & -8 & -9 & 16 & 4 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ r \\ u \end{pmatrix} \leq \begin{pmatrix} 0 \\ 2 \\ -2 \\ 2 \\ 0 \\ 0 \\ 15 \\ 0 \\ 3 \\ 0 \end{pmatrix} \quad (8)$$

In principle, we can use this augmented definition of the polyhedral loop bounds to rearrange the loops in our loop nest to move the variables r and u which iterate over SDRAM rows and burst accesses respectively to the outer levels of the loop body. This transformation gathers together the memory accesses to a specific SDRAM row, and reduces the number of row swaps ('activation' and 'precharging' of rows) incurred.

An example code where this transformation results in an optimal ordering of accesses is shown in Fig. 4. The original code for this example is shown in Fig. 4(a). Interpreting the addition of explicit row and burst variables as the introduction of new loop iteration variables, the augmented polytope description corresponds to Fig. 4(b), where each of the two innermost loops iterates exactly once, by construction. By itself, this transformation has only made the loop body more complex, however, it now allows us to move the r and u variables to the outermost loops using standard loop transformation techniques [5, 6], and add a buffer following [18] resulting in Fig. 4(c). Note now that the x_2 loop only iterates once and can thus be eliminated giving the end result shown in Fig. 4(d). This code is far preferable, as it accesses each row only once, streaming the data into a buffer, coalescing data reads into bursts where possible.

```
char A[256];
for(x1 = 0; x1 <= 15; x1++)  {
  for(x2 = 0; x2 <= 15; x2++) {
    .. = f( A[x1 + 16*x2] );
  }
}
```

(a) Original Code

```
char A[256];
for(x1 = 0; x1 <= 15; x1++) {
  for(x2 = 0; x2 <= 15; x2++) {
  // Note : / is integer division.
    for( r = (x1+16*x2)/16; r <= (x1+16*x2)/16;
          r++ ) {
      for( u = (x1 + 16*x2 - 16*r)/4;
             u <= (x1+16*x2-16*r)/4; u++ ) {
        ... = f( A[x1 + 16*x2] )
      }
    }
  }
}
```

(b) Augmented Code

```
char A[256];
char buff[16][4][4];
for(r = 0; r <= 15; r++)  {
  for(u = 0; u <= 4; u++) {
    buff[r][u][0..3] = burstread(r,u);
    for( x2 = r; x2<=r; x2++ ) {
      for( x1 = 4*u; x1<=4*u+3; x1++ ) {
        ... = f( buff[x2][x1/4][x1%4] );
      }
    }
  }
}
```

(c) Intermediate Code

```
char A[256];
char buff[16][4][4];
for(r = 0; r <= 15 ; r++) {
  for(u = 0; u <= 4; u++) {
    buff[r][u][0..3] = burstread(r,u);
    for(x1 = 4*u; x1 <= 4*u+3; x1++ ) {
      ... = f( buff[r][u][x1-4*u] );
    }
  }
}
```

(d) Transformed Code

Figure 4: C source code for 2-level nested loop example.

In general, however, such a direct transformation may not be possible. As already noted, the earlier example in Fig. 1 contains 'holes'. Moving row and column accesses to the outermost loop levels will, in this case, fill in the holes, introducing superfluous reads/writes and/or requiring complex guard statements to skip the holes. Thus our transformation engine aims to determine when such transformations can be safely applied, and manipulates the loop structure to allow their application. The first question to address, therefore, is when a loop variable, *e.g.* x_2 in Fig. 4, can be eliminated from the augmented code without changing the set of memory locations accessed.

4.4 Variable Elimination

We may formalise the question: Is the set $\{fx \mid \exists x \in \mathbb{Z}^n, Ax \leq b\}$ equal to another set $\{f'y \mid \exists y \in \mathbb{Z}^m, A'y \leq b'\}$ for some choice of f' (representing the new array indexing function), A' and b' (representing the new loop bounds), with $m < n$? If the answer is 'yes', this tells us that we may eliminate a variable, resulting in a lower complexity addressing sequencer.

A related problem has been studied in the context of operational research by Williams [24], who looked at the specific case $y = (x_1\ x_2\ \ldots x_{q-1}\ x_{q+1}\ x_n)^T$, *i.e.* the loop iterators are kept the same, but one variable is deleted (as in Fig. 4). Williams gives the following sufficient conditions for this special case:

- The q th column in matrix A has at least one entry with the value +1 with corresponding entries in all other rows being 0, negative or +1 *or*
- The q th column in matrix A has at least one entry with the value -1 with corresponding entries in all other rows being 0, positive or -1

We generalise Williams' result by trying to transform the loop body such that the above conditions are satisfied. We draw on the theory of *unimodular loop transformations* [5, 20, 25] to write $\{fx \mid \exists x \in \mathbb{Z}^n, Ax \leq b\} = \{fUz \mid \exists z \in \mathbb{Z}^n, AUz \leq b\}$ for an arbitrary unimodular matrix U, allowing us to apply Williams' elimination procedure to the matrix AU rather than to the original matrix A. We may therefore expose further opportunities for variable elimination.

For our specific example from Fig. 1, we have added dimensions r and u to our polytope description, alongside the original loop variables $[x_1, x_2, x_3]$. Adding r and u does not change the number of items in the set S_M, since each memory reference addresses data in exactly one row and one burst. Applying the unimodular transformation in (9), we can transform our matrix describing loop bounds into those shown in (10).

$$U = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 \\ -1 & 1 & 0 & 0 & 0 \\ 0 & -1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{pmatrix}, \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ r \\ u \end{pmatrix} = Uz\ ,\ AUz \leq b \quad (9)$$

$$AU = \begin{pmatrix} -1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 \\ 0 & -1 & 0 & 0 & 0 \\ -1 & 1 & 0 & 0 & 0 \\ 1 & 1 & -1 & 0 & 0 \\ 1 & -2 & 1 & 0 & 0 \\ -1 & -1 & 9 & -16 & 0 \\ 1 & 1 & -9 & 16 & 0 \\ -1 & -1 & 9 & -16 & -4 \\ 1 & 1 & -9 & 16 & 4 \end{pmatrix} \quad (10)$$

From (10) , we can see that Williams' sufficient conditions can be applied to eliminate the z_1 and z_2 variables. The set of integer points enclosed by the polytope with the z_1 and z_2 projected out has been reduced (from 7 to 5) by the transformation, but crucially, the number of unique rows and bursts accessed is the same. The difference here is that we have exploited the explicit representation of rows and bursts to enable data reuse. Where a burst within a specific row was activated more than once in the original code, in the transformed code, here it is only accessed once. The problem remaining is to find an appropriate matrix U to enable variable elimination, which we address below.

4.5 Integer Linear Program to Maximise Eliminable Variables

We wish to find a unimodular matrix U that enables a change of variable ($x = Uz$) such that the maximum number of variables can be eliminated from a polytope by Williams' conditions. To do this, we construct the formulation in Figure 5 and solve using CPLEX[14]. The formulation expresses a matrix multiplication of the input matrix A whose coefficients are known, with the unknown matrix U made up of decision variables. The elements of the resulting matrix AU are labelled $P_{i,j}$ in our formulation. The decision variables D_i form the diagonal elements of the unimodular matrix U we are trying to find, and $N_{i,j}$ are the lower triangular elements. The upper triangular elements of the unimodular matrix U are all zero. This ensures that the resulting matrix is unimodular since all lower triangular matrices with diagonal elements of -1 or +1 are unimodular.

Integer Linear Program for finding unimodular matrix to maximize variable elimination by Williams' conditions[24], $A_{k,m}$ is input integer matrix, Sz is a bound on size of any entry in the unimodular matrix.

max: $\sum_{i=1}^{n} sum_i$

subject to:

% *restricts lower triangular elements to [-Sz Sz]*
$1 \leq i \leq n,\ 1 \leq j < i \quad N_{i,j} \leq Sz$
$1 \leq i \leq n,\ 1 \leq j < i \quad N_{i,j} \geq -Sz$
$1 \leq i \leq n,\ i \leq j \leq n \quad N_{i,j} = 0$
% sum_i *is 0 if* pos_i *and* neg_i *are both zero*
$1 \leq i \leq n \quad pos_i + neg_i - sum_i \geq 0$

$1 \leq i \leq n,\ 1 \leq j \leq n \quad P_{i,j} = A_{i,j}D_j + \sum_{g=j+1}^{n} A_{i,g}N_{g,j}$
% $M_{i,j}$ *is precomputed value guaranteed to be larger than* $P_{i,j}$
$1 \leq i \leq n,\ 1 \leq j \leq n \quad P_{i,j} + (M_{i,j} - 1)neg_i \leq M_{i,j}$
$1 \leq i \leq n,\ 1 \leq j \leq n \quad P_{i,j} - (M_{i,j} - 1)pos_i \geq -M_{i,j}$

$1 \leq i \leq n \quad D_i \in \{-1, 1\}$
$1 \leq i \leq n \quad sum_i \in \{0, 1\}$
$1 \leq i \leq n \quad pos_i \in \{0, 1\}$
$1 \leq i \leq n \quad neg_i \in \{0, 1\}$
$1 \leq i \leq n,\ 1 \leq j \leq n \quad P_{i,j} \in \mathbb{Z}$
$1 \leq i \leq n,\ 1 \leq j \leq n \quad N_{i,j} \in \mathbb{Z}$

Figure 5: Finding a unimodular matrix which maximises the number of eliminable columns.

The formulation presented finds the optimal unimodular matrix for eliminating variables by Williams' conditions subject to the constraint that each of the lower triangular coefficients $N_{i,j}$ is bounded in the range [-Sz, Sz]. This is done to ensure that we can always calculate a constant value $M_{i,j}$ which is guaranteed to be bigger than $P_{i,j}$, as required by the constraints. The constraints containing $M_{i,j}$ are trivially satisfied if the associated binary variable (neg_i or pos_i) is zero. This binary variable is only allowed to become 1 if all the variables in a column of P are less-than-or-equal to 1 or greater-than-or-equal-to -1. If either the neg_i or the pos_i variable for a particular column is non-zero, the sum variable becomes non-zero. Since the optimization procedure is attempting to maximize the sum of the sum_i variables, the optimization

```
char buff [56];
for (r = 1 ; r <= 3 ; r++) {
   for ( z3  = ceil( (16r+2) / 9) ;
         z3 <= min (6, 2r+1) ; z3++) {
            for ( u  = -4*r+2*z3 ;
                  u <= -4*r+2*z3 ; u++) {
                  burstwrite(r,u) = buff[...];
            }
   }
}
```

Figure 6: Transformed source code for memory accesses in example code from Figure 1.

procedure effectively pulls up the sum variables, finding optimal values for the decision variables which form the unimodular matrix D_i for the diagonal elements, and $N_{i,j}$ for the lower triangular elements. The binary values sum_i declare whether under the change of variables, $x = Uz$, the variable z_i is eliminable.

Having found a unimodular function which gives a change of variables and allows elimination of variables, we apply that unimodular transformation to the original polytope and eliminate the appropriate variables by a Fourier-Motzkin projection [4]. In our example code, the unimodular matrix in (9) is generated, which when multiplied by the bounds in (8) gives (10) which allows for the elimination of the z_1 and z_2 indices by the sufficient conditions in [24]. In our experiments, all the ILP formulations generated complete within sub-second timing on a desktop PC.

After applying the optimal unimodular transformation, further necessary conditions from [24] are checked to see if those variables not identified as eliminable in the ILP formulation (which quickly checks for sufficient conditions) can be eliminated without creating holes. The interested reader is referred to [24] for further explanation of these conditions, with the note that the complexity of checking these conditions is dependent on the coefficients of the loop bounds, which increase with the size of the data-set to be processed. Our ILP approach scales instead with the number of nested loops which is independent of the size of the input data.

For our example code in Figure 1, checking these necessary conditions shows that the remaining variable (z_3) can be eliminated from the row dimension without creating holes, but cannot be eliminated from the burst dimension without creating holes. This is consistent with our sequence of memory accesses shown in Table 1. We use this information to reorder the loops.

Since all the variables can be shown to be eliminable from the 'r' dimension, that dimension is traversed at the outermost level of the generated loops, followed by the only remaining existential variable shown not to be eliminable in our ILP formulation, z_3 . This loop variable is nested inside the 'r' variable but outside the 'u' variable in the resulting code, because it can be eliminated without causing holes in the 'r' dimension, but cannot be eliminated without causing holes in the 'u' dimension. All the other dimensions can be safely projected out. When this projection and 'C' code generation is performed using Cloog [6], the data transfer code in Figure 6 is produced.

In this generated code, we observe that the sequence of rows accessed is monotonic, bursts within each row are accessed only once and the set of rows and bursts accessed is exactly the set in the original code description (*i.e.* the holes in the original set are preserved). The access pattern of this transformed code contains two fewer memory accesses and two fewer row activations, as a result, its usage of scarce external memory bandwidth is more efficient than the code in the original example. The final stage of our procedure is to generate an efficient hardware address generation function to implement this transformed loop structure.

4.6 Code Generation

The tool Cloog [6] is used to generate nested loop structures by traversing the integer points within an input polytope in a specified order. Cloog generates an abstract syntax tree which can be directly translated into 'C' code. For our example, the generated code is given in Figure 6. We choose to work with this abstract syntax tree to produce pipelined streaming hardware which implements the loop index generation.

Statements which may occur in the abstract syntax tree include 'for' statements, 'assignment' statements and 'compound' statements describing the serial composition of more than one statement. The expressions within those statements include integer division, multiplication by a scalar, reduction of a vector using min, max and summation functions and modulus, floor and ceiling functions.

The expressions contained with the upper and lower bounds of the 'for' statements and within the right hand side of assignments statements can be quite complex, and without pipelining, negatively impact the achievable clock frequency. However, because Cloog derives nested loop structures in which the inner loop indices only depend on indices in outer loops, we can arrange the logic as a feed-forward pipeline with distributed control, adding arbitrary pipeline stages and using the auto-pipelining features of a logic synthesis tool (Altera Quartus II) to distribute them in a manner which minimizes the length of the critical timing path. This ensures that our hardware implementation of address generation is scalable to meet future requirements for high clock-speeds.

The synthesis and transformation of the Cloog AST format into hardware address generators produces Verilog code which can be synthesized for implementation in an FPGA. Results are reported in Section 5 showing the efficiency of each of the generators produced using our tool.

5. RESULTS

We show the effectiveness of our approach with three benchmarks :

Matrix-Matrix Multiply (50x50) This benchmark must access two matrices simultaneously using the columns of one and the rows of the other. The large strides in memory this implies means that row-swaps occur frequently within the inner loop of the benchmark.

Sobel Edge Detection (96x64) This benchmark reuses data as a sliding window perform a convolutions over an image. The input and output image row sizes do not align with SDRAM row boundaries which makes manual optimization of this benchmark difficult.

Triangular Backsubstitution (72x72) This benchmark demonstrates the applicability of our technique to non-rectangular loop nests, it demonstrates a non-constant stride over the blocks in the input matrix. Only the necessary upper triangular elements of the matrix are loaded into on-chip memory buffers.

Each benchmark is expressed as C code and passed through our automatic flow. Static control portions of the input code are marked with #pragma preprocessor directives and a polyhedral description is automatically extracted using [7]. After transformation using the methodology in Section 4, synthesizable address generators expressed in Verilog are generated as the tool output. The address generators produced were connected to the Altera High Performance SDRAM Controller II [3] in a testbench environment which

recorded the SDRAM interface usage at each cycle and the overall benchmark runtime. These results are reported in Table 2.

In these results we can see the total cycles required to fetch data in each benchmark decreases as we decrease the parameterisation level (t). This is in part due to data *reuse*. When data reuse buffers are inserted at the outermost levels of the loop ($t = 1$), all accessed data is preloaded into on-chip memory at the start of execution, and fetched from on-chip memory during execution. We would expect to see a significant reduction in the number of 'read' / 'write' cycles on the external interface as the parameterisation level is reduced and more data is buffered on-chip. If we compare the original code ($t = n+1$) with the ($t = 1$) parameterisation, we see results consistent with this expectation, with a 400×, 94× and 33× reduction in the number of 'read' / 'write' cycles in each respective benchmark.

Alongside this evidence of data reuse, Table 2 also shows a breakdown of the total benchmark time into the cycles in which the interface performs 'reads' and 'writes', the cycles in which it is idle due to bus turnaround time (transition from read-to-write and vice-versa) and the 'precharge' / 'activate' and 'refresh' cycles lumped together with their respective delay cycles. This information is also presented visually in Figure 7. From this we can see that the *reordering* of memory transactions through our loop transformations increases the *efficiency* of the memory interface usage. In the original code in each of the three benchmarks, ~75% of memory interface cycles are used for the control overhead of changing SDRAM rows and bus turnaround cycles. Since our static analysis approach groups together the memory requests which occur in rows and bursts, it significantly increases the efficiency of the external memory interface. If we compare the original code with the parameterisations which have reuse buffers inserted outside the innermost loop level ($t = 3$ for MMM, $t = 4$ for SOB and $t = 2$ for GBS), we see a reduction in the proportion of memory cycles used for 'precharge' and 'activate' commands from 73.24% to 21.27%, from 70.80% to 59.35% and from 57.12% to 23.45% in the MMM, SOB and GBS benchmarks respectively. The gains in efficiency (the proportion of 'read' / 'write' cycles as compared to the original code) vary from 3.6× to 4× across the benchmarks and their associated parameterisations.

In order to show that the efficiency gains arising from this reordering are achievable at a reasonable cost, we report synthesis and F_{max} results from the slow 1100mV corner of the static analysis tool in Quartus 10.1 with physical synthesis and register retiming options enabled. Registers were inserted by our code-generation flow to ensure the address generator met a 133MHz clock frequency. This corresponds to half the command frequency of our external DDR2 Memory since the minimum burst size of DDR2 memory (4 words) means two clock cycle periods are needed to process consecutive back-to-back memory requests. It should be noted that since our address sequence generators can be pipelined to an arbitrary depth, they are scalable to future memory speeds at the cost of increased register count and initial latency. The post place-and-route maximum frequency of our address generator designs and their resource requirements (ALUTs and registers) are reported in Table 3. We show the number of on-chip memory words needed to implement our three benchmarks, inserting reuse buffers at different levels in the loop nest. This is reported in words rather than an absolute number of bytes to reflect the fact that external SDRAM interfaces typically bundle together multiple parallel data-channels with commands issued by a single set of control signals (which allows scaling of our benchmark runs from 8-bit to 64-bit data types). The synthesis results show that our backend generation tool will scale to useful clock-frequencies. The logic resource utilization of the address generators at different parameterisation levels varies from a reduction of 0.5× to an increase of 1.4× when compared to the original code in the three benchmarks. It should be noted however that even the largest address generator presented uses less than 4% of the smallest available Stratix III device (EP3SL50). From the synthesis results reported, we can conclude that our address generators achieve their reordering at a very reasonable logic cost, and will scale to useful clock frequencies in modern devices.

Table 3: Synthesis results for benchmark codes.

Benchmark	Level	Req. on-chip mem. words	ALUTs	Regs	Frequency
MMM	t=1	61200	575	764	296 MHz
MMM	t=2	21200	1050	1666	174 MHz
MMM	t=3	416	1346	2098	179 MHz
MMM	Orig.	0	1003	2740	184 MHz
SOB	t=1	11411	592	717	300 MHz
SOB	t=2	579	1551	2251	182 MHz
SOB	t=3	19	1355	1907	144 MHz
SOB	t=4	7	1200	2566	153 MHz
SOB	Orig.	0	1107	3607	148 MHz
GBS	t=1	2772	833	1156	242 MHz
GBS	t=2	288	952	1366	211 MHz
GBS	Orig.	0	804	2263	186 MHz

To explore the trade-off between on-chip *memory* resources and external memory interface performance, Figure 8 shows how the overall number of memory access cycles scales with the amount of on-chip memory dedicated to buffering data for each of the benchmarks. From this we can see that if all the data in the MMM benchmark can be stored on-chip, 1500× fewer memory access cycles are needed to transfer data from external memory. What is more significant about this plot however, is that it shows that one can, using an automatic tool, explore more reasonable trade-offs such as the $t = 3$ parameterisation for the SOB benchmark, which achieves a 6.6× reduction in memory access cycles at a cost of 20 additional words of on-chip memory. In each benchmark, our pareto-optimal fronts show multiple feasible points for on-chip memory usage and automatic generation of address generators using our methodology allows evaluation of the performance trade-off each embodies early in the design cycle.

While other work, such as [18], demonstrates similar trade-offs between on-chip memory usage and performance due to data *reuse*, our *explicit* representation of SDRAM rows and bursts and static reordering of memory transactions achieves additional performance gains from the efficient utilization of the memory interface; the SOB $t = 3$ parameterisation achieves 6.6× better performance than the original code, despite having only a 4.7× reduction in 'read' / 'write' cycles due to data reuse. This is because by *reordering* transactions we have reduced the proportion of 'precharge' / 'activate' cycles in that benchmark from 70.8% of the total interface cycles to 52.5%. Hence we can conclude that both the data reuse uncovered using our methodology *and* the transaction reordering achieved through loop transformations contribute to the memory performance improvements we demonstrate.

In Table 4, we report the tool runtime. The runtimes are aggregated across all parameterisations of the benchmarks. We report separately the time spent checking *sufficient* conditions for variable elimination using our ILP formulation and the time spent exhaustively checking the *necessary* conditions for elimination of a variable in the event that the sufficient conditions are not met. From Table 4, we note that for each benchmark, the mean time for checking the sufficient conditions for variable elimination using our ILP formulation is less than a second, with a narrow standard deviation. In

Table 2: Simulation results for benchmark codes.

Benchmark	Level	Read/Write Cycles	Bus Turnaround Cycles	Precharge/Activate Cycles	Refresh Cycles	Total Cycles
MMM	t=1	5012 (94.00%)	8 (0.15%)	202 (3.79%)	110 (2.06%)	5332
MMM	t=2	66804 (91.54%)	976 (1.34%)	3638 (4.98%)	1560 (2.14%)	72978
MMM	t=3	590000 (73.86%)	21428 (2.68%)	169870 (21.27%)	17506 (2.19%)	798804
MMM	Orig.	2000004 (23.78%)	66904 (0.80%)	6159756 (73.24%))	183596 (2.18%)	8410260
SOB	t=1	8920 (94.93%)	8 (0.09%)	300 (3.19%)	168 (1.79%)	9396
SOB	t=2	77328 (71.18%)	1868 (1.72%)	27136 (24.98%)	2300 (2.12%)	108632
SOB	t=3	180300 (32.43%)	57188 (10.92%)	274766 (52.47%)	11392 (2.18%)	523646
SOB	t=4	442932 (32.58%)	80348 (5.91%)	806860 (59.35%)	29334 (2.16%)	1359474
SOB	Orig.	839236 (24.18%)	100768 (2.90%)	2457648 (70.80%)	73632 (2.12%)	3471284
GBS	t=1	1832 (91.88%)	8 (0.40%)	154 (7.72%)	0 (0.0%)	1994
GBS	t=2	13888 (67.90%)	1368 (6.69%)	4798 (23.45%)	400 (1.96%)	20454
GBS	Orig.	61348 (25.55%)	36640 (15.26%)	137146 (57.12%)	4972 (2.07%)	240106

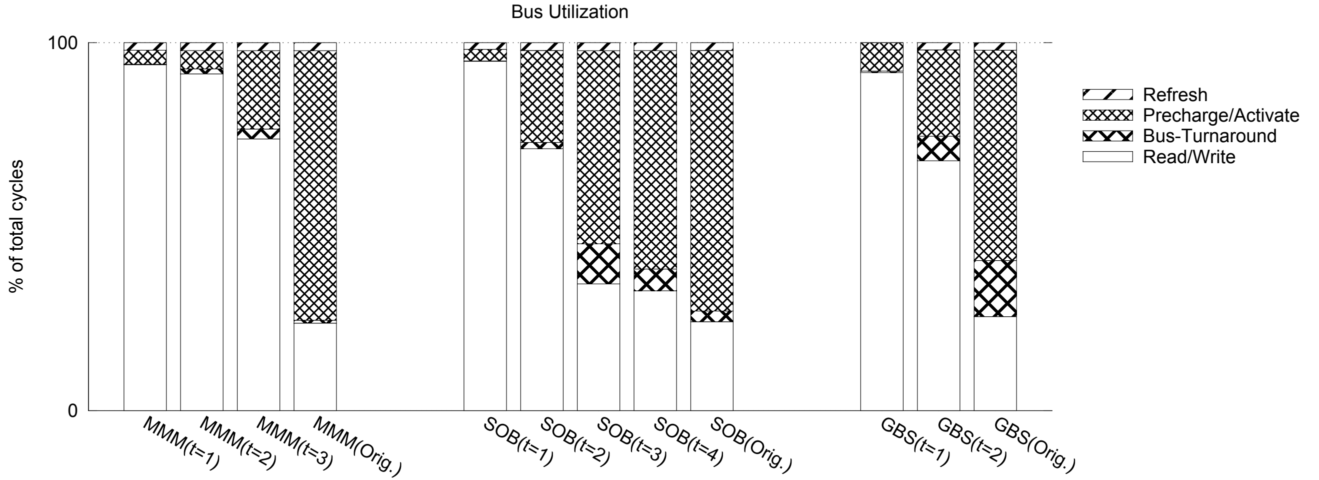

Figure 7: SDRAM Memory Interface Utilization : Breakdown by Command Type.

comparison, the mean time taken to check the *necessary* conditions for variable elimination (for those variables which cannot be eliminated by the sufficient conditions) is much greater, and varies much more significantly between the different parameterisations of each benchmark. This is because the runtime of our ILP formulation depends on the number of loop variables (n) present in the source code while checking the necessary conditions takes time proportional to the number of points in the iteration space S_E. In practice this means the time taken to check the *necessary* conditions for variable elimination scales poorly with large benchmarks, but runtimes are greatly improved if we first eliminate the variables which meet the sufficient conditions using our ILP formulation. Together

Table 4: Tool Runtime.

Benchmark	Time Taken (Suf. Cond.)		Time Taken (Nec. Cond.)	
MMM	μ(0.24s)	σ(0.03s)	μ(25.62s)	σ(30.36s)
SOB	μ(0.44s)	σ(0.20s)	μ(518.63s)	σ(1009.05s)
GBS	μ(0.26s)	σ(0.01s)	μ(0.41s)	σ(0.57s)

these results show that our ILP formulation, and the checking of sufficient conditions for variable elimination using Williams' results [24] allow us to produce safe performance enhancing loop transformations with reasonable compile-time. The methodology and tool built around these insights allows the automatic production of address generators whose logic cost is reasonable in modern devices and whose frequency scales to useful clock speeds. Parameterisation allows the trade-off of on-chip memory resources for performance by both a reduction in the amount of data transferred on the external memory interface and an improvement in the efficiency of that interface through reduction in the proportion of interface cycles required for 'precharge' and 'activation'.

6. CONCLUSION

In this work, we have described an analytical framework for hardware compilation which allows a parameterised trade-off between the usage of on-chip memory and arithmetic resources and external memory bandwidth. We show the applicability of our technique to a range of benchmarks and demonstrate scalability to support high clock frequencies demanded by future memory technologies.

Furthermore the tool (available at `http://cas.ee.ac.uk/AddrGen`) provides a starting point for those wishing to experiment with polyhedral compilation techniques within a hardware compilation flow.

In future work, we propose applying recent advances in integer point counting techniques to produce optimized static schedules for SDRAM memory access which allow exact calculation of execution time at compile time and safe overlapping of parallel execution threads and serialized external memory accesses. Such an approach will allow us to use external SDRAM memory within a general synthesis flow, exploiting knowledge of memory delays to enable efficient hardware implementations through multi-cycle logic evaluation and resource sharing.

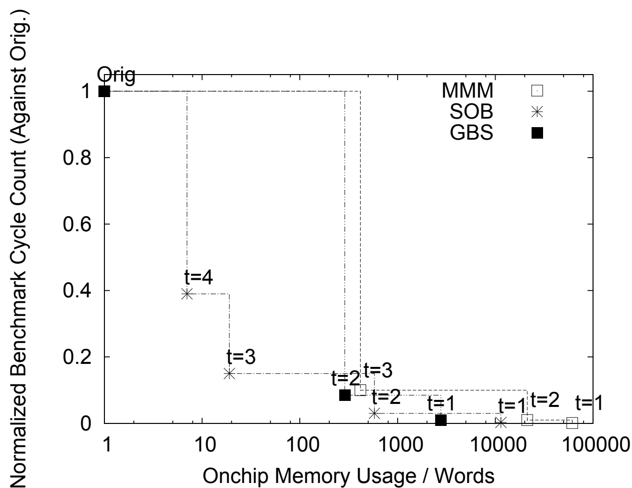

Figure 8: Pareto-optimal fronts showing designs parameterised at different levels.

7. REFERENCES

[1] Assembly and Packaging. *International Technology Roadmap for Semiconductors*, 2010.

[2] 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.

[3] Altera. DDR2 and DDR3 SDRAM Controller with UniPHY User Guide. `http://www.altera.com/literature/hb/external-memory/emi_ddr3up_ug.pdf`, June 2011.

[4] C. Ancourt and F. Irigoin. Scanning Polyhedra with DO Loops. In *PPOPP '91 : Proceedings of the 3rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 39–50, Williamsburg, United States, 1991.

[5] U. K. Banerjee. *Loop Transformations for Restructuring Compilers: The Foundations*. Kluwer Academic Publishers, Norwell, MA, USA, 1993.

[6] C. Bastoul. Code Generation in the Polyhedral Model is Easier than you Think. In *PACT '13 : IEEE International Conference on Parallel Architecture and Compilation Techniques*, pages 7–16, Juan-les-Pins, France, 2004.

[7] C. Bastoul. Extracting Polyhedral Representation from High Level Languages. Technical report, Paris-Sud University, 2008.

[8] M. Benabderrahmane, L. Pouchet, A. Cohen, and C. Bastoul. The Polyhedral Model is More Widely Applicable than you Think. In *ETAPS CC'10 : Proceedings of the International Conference on Compiler Construction*, pages 283–303, Paphos, Cyprus, 2010.

[9] H.-K. Chang and Y.-L. Lin. Array Allocation Taking into Account SDRAM Characteristics. In *ASP-DAC '00: Proceedings of the 2000 Asia and South Pacific Design Automation Conference*, pages 497–502, New York, NY, USA, 2000.

[10] E. S. Chung, J. C. Hoe, and K. Mai. CoRAM: An In-Fabric Memory Architecture for FPGA-based computing. In *FPGA '11 : Proceedings of the 19th Annual International Symposium on Field Programmable Gate Arrays*, pages 97–106, Monterey, CA, USA, 2011.

[11] A. Darte, R. Schreiber, and G. Villard. Lattice-Based Memory Allocation. *IEEE Transactions on Computers*, 54(10):1242–1257, 2005.

[12] T. Grosser, H. Zheng, R. A, A. Simbürger, A. Grosslinger, and L.-N. Pouchet. Polly - Polyhedral Optimization in LLVM. In *IMPACT'11 : First International Workshop on Polyhedral Compilation Techniques*, Chamonix, France, 2011.

[13] J. Hennessey and D. Patterson. *Computer Architecture : A Quantitative Approach*. Morgan Kaufmann, 6th edition, 2006.

[14] IBM. Introduction to CPLEX Optimization Studio. `http://www-01.ibm.com/software/integration/optimization/cplex-optimizer/`, June 2010.

[15] L. Johnson. Improving DDR SDRAM Efficiency with a Reordering Controller. *Xcell Journal*, 3:38–41, 2009.

[16] W. Kelly and W. Pugh. A Framework for Unifying Reordering Transformations. Technical Report CS-TR-3193, Dept. Of Computer Science, University of Maryland, 1993.

[17] 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.

[18] 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.

[19] G. E. Moore. Cramming More Components onto Integrated Circuits. *Proceedings of the IEEE*, 86(1):82–85, 1998.

[20] L.-N. Pouchet, C. Bastoul, A. Cohen, and N. Vasilache. Iterative optimization in the Polyhedral Model: Part I, One-Dimensional Time. In *Proceedings of the International Symposium on Code Generation and Optimization*, CGO '07, pages 144–156, Washington, DC, USA, 2007. IEEE Computer Society.

[21] F. Quilleré, S. Rajopadhye, and D. Wilde. Generation of Efficient Nested Loops from Polyhedra. *International Journal of Parallel Programming*, 28:469–498, 2000.

[22] P. Quinton and V. V. Dongen. The Mapping of Linear Recurrence Equations on Regular Arrays. *Journal of VLSI Signal Processing*, 1(2):95–113, 1989.

[23] 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, 2000.

[24] H. P. Williams. The Elimination of Integer Variables. *The Journal of the Operational Research Society*, 43(5):pp. 387–393, 1992.

[25] M. E. Wolf and M. S. Lam. A Data Locality Optimizing Algorithm. In *Proceedings of the ACM SIGPLAN 1991 conference on Programming language design and implementation*, PLDI '91, pages 30–44, New York, NY, USA, 1991. ACM.