The Implementation and Evaluation of a Compiler-directed Memory Interface

Dominic McCarthy, Stuart Quick

Hewlett-Packard Laboratories,
Filton Road, Stoke Gifford, Bristol. BS12. 6QZ. UK.
{dom|svq}@hplb.hpl.hp.com

Abstract

This paper describes a novel memory interface architecture, called burst buffers. Although simple, it regularly attains more than a factor of two improvement in performance for media algorithms above a normal data cache using conventional DRAM technology. This paper shows that exposing the features of main memory permits architectural and compiler innovations. Performance gains are achieved by improving the use of available bandwidth from DRAM by utilising three techniques: long bursts, burst pipelining and buffer re-use. This paper demonstrates the benefits of the new memory interface architecture when included in an embedded system alongside the data cache. This interface adds little or no extra system cost.

Note:

This document is based on the paper submitted to ISCA 98. This means that a revised version may become available should this paper be accepted but require modification.

1. Introduction

This paper describes work undertaken to improve throughput within computer architectures. The work reflects a careful balancing of system constraints, memory hierarchy architecture, compiler technology and media processing in order to maximise system efficiency, in terms of performance and cost. The result is a new, very simple, compiler-directed memory interface architecture, called *burst buffers*, which has been shown to accelerate the processing of media algorithms on microprocessors by factors greater than two, compared with a normal data cache. There are three major contributors to the performance gains:

1. using long bursts to dynamic random access memories (DRAM) to improve memory bandwidth usage;
2. pipelining the bursts to permit concurrent computation and burst transactions, in order to saturate the memory bus with useful traffic;
3. re-using data held in the burst buffers through compiler transformations and optimisations.

The work recognises the importance of the burst capability of DRAM and exposes this feature to the compiler. The compiler automatically detects candidate loop structures within a media algorithm and schedules read and write bursts to the DRAM, whilst maintaining synchronisation with the computation.

This work has been targeted towards embedded systems. These have moved away from the small processor with a dedicated application-specific integrated circuit (ASIC) performing much of the computations, to a processor-centric solution, often based around a 'workstation-class' processor. This trend has introduced new problems. Embedded systems always have aggressive cost goals, which often forces a primitive memory hierarchy comprising the lowest cost memory components available, namely DRAM. With no level-2 cache, there is often a mismatch between an on-chip cache and external DRAM memory, which is exacerbated by the need for the processor to perform media processing on large data flows, such as images.

The approach taken here was to develop generic structures and techniques to improve the situation. As it turned out, the architectures developed here have as much potential in general purpose computation environments as they have in embedded systems, though the work naturally emphasised the latter domain.

The work has resulted in a stable architecture, however the implementation of the compiler is less mature. Compiler activities have focused on the transformation techniques, rather than their implementation. The evaluations performed in section 6 reflect this as some amount of hand coding of optimisations has been performed, due to the complexities of writing a compiler to attack all potential cases. The first compiler, based on SUIF [1], has been used successfully to demonstrate many of the transformation and optimisation techniques that the new architecture requires.

This paper begins by discussing the current state of media algorithms, caches and DRAM. It then explores architectural considerations that influenced the buffer architecture. Following a description of the architecture itself, the paper presents the compiler strategy and transformations. This evaluation process is described and the results presented. The paper concludes with a discussion of the results and some concerns and proposals raised during the execution of this work.

2. Reasoning

Much has been made of the divergent trends of processor speed and memory bandwidth. Beginning with the recognition of the 'memory wall' [2], many papers have challenged the existing caching paradigms [3,4,5,6]. Although there is no doubting the benefits of existing cache architectures, there are three underlying issues which require review. These are: the importance of vector-like computations associated with media algorithms; the trends in cache architectures; and the state of DRAM architecture.

2.1 Media algorithms

The trends associated with computation on media data are widely recognised. These have driven the implementation of multimedia instruction set extensions for most processor families. Providing the support for the computation is only part of the answer - two other issues also need to be faced: compilation strategies to exploit micro-SIMD or packed arithmetic; and improvements in memory bandwidth. Diefendorff and Dubey [7] have commented on this and documented the need for more bandwidth and the inefficiencies of caches within media processing environments. The work undertaken in this study focuses in this area.

The issues with media processing relate to the characteristics of the algorithms and their required data flows. Media processing involves operating on large, regular data streams, such as image, audio and video. These streams have characteristics which differ from those normally exploited by a data cache, such as temporal locality. In fact, these data streams tend to degrade data cache performance due to conflict misses, redundant line fetches and flushes (in a write-back cache). With the processor now responsible for much of the data processing, as well as the control, such degradation is disappointing.

The need to provide capabilities for media processing is not restricted to the general purpose computing domain. Embedded systems, such as printers and scanners also require support for media processing, with the additional constraint of very low cost.

2.2 The case for and against caches

The heritage of caching structures is beyond question. Proof of their value is well documented in several classic papers, notably [9]. The original intent of the name 'cache' was to describe some local memory, largely transparent to the system. In many ways, the work proposed in this paper fulfils this definition, though the term 'cache' is now popularly narrowed. Today, it describes the familiar structure attached to most processors, comprising fast memory and tags. For the remainder of this paper, the term 'cache' will be used in this narrower sense.

Caches provide the majority of memory transaction requests. This means that their operating model largely determines the efficiency with which the memory bus is used. For programs that use scalar data sets, the cache is a proven winner. However, when vector data sets are processed, inefficiencies introduced by the interface between cache and memory arise. There are five contributions to the inefficiencies.

1. The line fetch mechanism is managed by hardware, which makes it highly restrictive and speculative. This means that it cannot be optimised for every data flow - in practice, it is matched to very few.

2. Due to the requirements to balance issues of locality and latency, short line sizes cause short bursts to DRAM. For systems with a 16 byte line size and a 32 bit memory bus, this equates to a DRAM burst of only 4 memory accesses.
3. Simple caches issue a request only when a miss occurs. This generally has two effects: the first is to stall the processor until the requested item is fetched; and the second is that, until a miss occurs, the memory bus is not being used - wasting bandwidth.
4. Low associativity and small cache size often lead to conflict misses which generate much redundant memory traffic.
5. Vector data sets, greater than the cache size, cause thrashing. Non-ideal placement of data, and its access pattern, also contribute.

Attempts to improve these cases have been made by extending the basic cache architecture. Obvious improvements, such as using higher associativity, simple prefetching, longer lines and critical-word-first mechanisms, have been followed with more complex structures, including victim caches [10], lock-up free (non-blocking) caches [11] and prefetching. Long lines mean longer DRAM bursts, but often introduce other side-effects, such as redundant fetches and high latency. Prefetching has been proposed and assessed from both hardware [13,14,16,17,19,20] and software perspectives [12,15,18]. The benefits of providing 'hints', from either the program or specialised hardware, to prefetch mechanisms, have been shown to be effective, but not necessarily a justification for their automatic inclusion in all processor designs. Such schemes suffer from five weaknesses:

1. The prefetched data is written into the cache, which potentially causes conflict misses depending on cache state and prefetch line size. This leads to redundant data traffic - which is the case it is trying to prevent.
2. There is no equivalent write-back policy which would automatically free space in the cache.
3. All program-generated hints introduce instruction overhead.
4. Hardware requirements for supporting prefetch introduce extra complexity which may negate the gains.
5. Performance gains are non-deterministic due to the interactions between program, data and cache characteristics.

To address these issues, this paper proposes that a new memory interface is introduced alongside the data cache, much like the stream caches in [19]. The removal of the data cache is not advocated since it is well proven to accelerate scalar data sets. Supplementing it with a new interface, comprising memory buffers, rather than cache lines, and directing appropriate data traffic to either cache or buffers, is the thesis under test. Prefetching is known to be particularly effective for vector streams. In these streams, the regularity of the data accesses are easily detected by the compiler. In this paper it is proposed to exploit the knowledge of these streams at the compiler level and make the compiler schedule transactions between the local buffers and main memory. However, adding a region of localised memory alongside a write-back cache exacerbates the problems of memory coherency. Prefetching mechanisms avoid this at the cost of performance gains.

2.3 Revisiting DRAM structure

Dynamic random access memory (DRAM) technology is well described in several places, such as [8]. This paper proposes that the DRAM structure is reviewed in order that salient features, often ignored, are given exposure at the highest architectural levels, to avoid inefficient use. In particular, bursting to DRAM is emphasised.

2.3.1 DRAM bursts

DRAMs comprise a matrix of capacitors, where each capacitor stores a binary digit. Each capacitor is connected to a transistor, which can be turned on via a word line such that the capacitor charge can be sensed on a bit line. This structure is shown in Figure 1.

The matrix requires row and column addresses to reference any random location. The row address is decoded to select a particular word line, whilst the column address enables the output of a sense amplifier, driven by the selected bit line. DRAM addressing is accomplished by sequentially presenting a row address and column address. Since the row address is latched, if a succeeding access uses the same page (i.e. uses the same word line), then there is no need to re-issue the row address, only the new column address is required. This shortens the total transaction time.

A ‘burst’ is used to describe the transfer of multiple locations to or from a single DRAM page in a single transaction. A burst always begins with the presentation of a row address, followed by any number of random column addresses that lie within the opened page. A burst may be characterised by two parameters: the time taken to perform the first access (which is the aggregate of row and column selection); and the time taken to perform a column selection, often called a ‘page mode access’. The time for the first access is always longer than for a subsequent page mode access, so, the average access time for elements in a burst may be reduced by amortising the first access with many page accesses, i.e. by using long bursts.

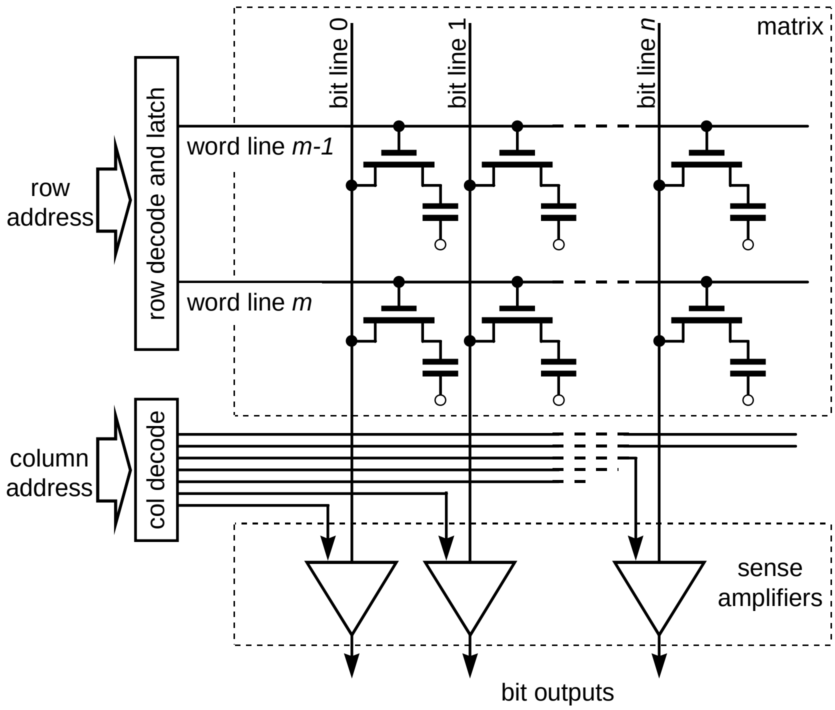

Figure 1: DRAM structure.

Newer DRAM architectures [8], such as synchronous DRAMs (SDRAM), those from Rambus and the SyncLink consortium all use DRAM arrays of this form. The major differences lie in the interface structures and the number of individual DRAM banks within each device. These features, if exploited successfully given the restrictions they impose (such as non-determinism due to cache effects), can help improve system performance. This paper shows that using bursts effectively can achieve equivalent or better performance using *any* variety of DRAM.

2.3.2 Burst efficiency model

Burst length and efficiency may be related using the following simple model[1].

1. A burst comprises an initial 'long' access, which opens a DRAM page, followed by any number of accesses within that page, ending with a precharge phase which closes the page.
2. The ratio of the first access time to the time of a subsequent access to the same page (i.e. for one burst) is r. A burst profile may then be defined as r:1:1:1:…
3. The minimum average access time to memory may be defined as the average time taken when bursting to a complete page. If the page size is $n_{\max}$, and the page mode access time is t_s, then this minimum time is:

$$t_{\min} = \frac{r + (n_{\max} - 1)}{n_{\max}} \times t_s$$

4. Burst efficiency, η_b, may be expressed as the ratio of the bandwidth for a burst of length n to the maximum bandwidth ($1/t_{\min}$). This may be expressed as:

$$\eta_b = \frac{(r + (n_{\max} - 1))}{n_{\max}} \times \frac{n}{(r + (n - 1))}$$

The burst efficiency may now be plotted against burst length for different burst profiles, as shown in Figure 2.

The region marked A is when the burst length is 4 and corresponds to a normal cache as discussed in section 2.2. The graph clearly shows that at most 45% of the DRAM bandwidth is *instantaneously* used. This use of 'instantaneous' means that this efficiency is only sustained during the interval of the burst. New architectures and techniques, described later, increase this to longer durations when running media applications.

As the ratio value r increases, so the efficiency decreases, for a given burst length. This reflects the fact that efforts to improve DRAM architecture have focused on improving the performance of the interface, while density concerns have left the core performance relatively static. This means that the first access into the core has changed little, whereas page mode accesses have become faster. This suggests that DRAM architecture trends are forcing r to increase and available bandwidth is being wasted because of the use of short bursts whenever a cache miss occurs. If this trend continues, and less attention is paid to the DRAM core, then techniques such as the ones proposed in this paper will become even more important in sustaining throughput.

For SDRAM, r is often 6 or greater. For newer DRAM technologies, such as Rambus DRAMs (RDRAM), or the proposed SyncLink DRAMs (SLDRAM), ratio values for r can approach 10. In this case, less than one third of the available bandwidth is being used. It is true that SDRAM, RDRAM and SLDRAM have multiple banks internally which may be

[1] A more sophisticated model would include other overheads associated with the manner in which the system interacts with the memory, physical issues related to board layout and clocks not running at optimum frequencies

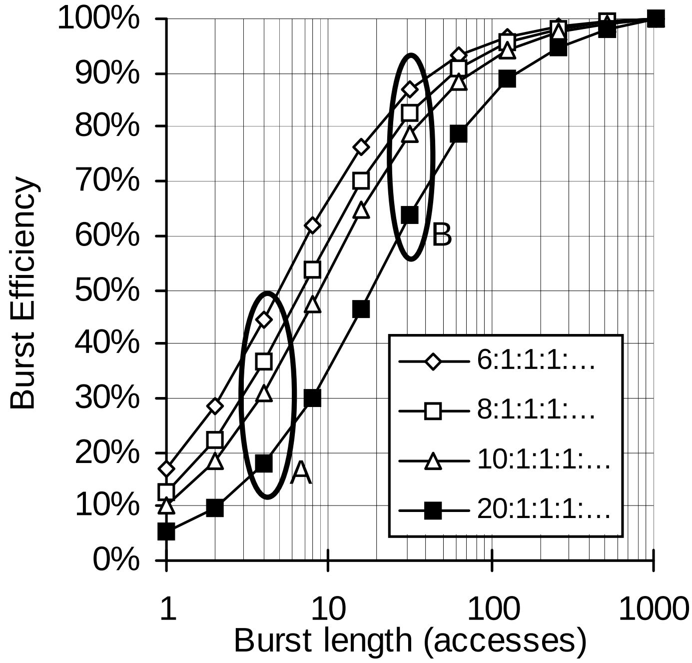

Figure 2: Burst efficiency for different DRAM burst profiles.

accessed almost concurrently, and so may hide the first access latency. However, to make use of this, data access patterns must be well known so that data may be laid out over multiple pages, and early anticipation of required bursts must be provided. More difficult is that the non-deterministic effects of the cache must be modelled to provide some insight into expected behaviour. Keeping DRAM pages open is a common technique of minimising latency, but in embedded systems, this often interferes with DMA scheduling. These factors increase the complexity of the memory controller, the compiler, and the linker. Using long bursts is a much simpler technique to achieve better bandwidth utilisation.

The ratio *r* is also increased in systems by the cache miss overhead of the processor to which the DRAM is connected. Although a cache hit may be resolved in a single processor clock cycle, the detection of a miss, the synchronisation of the processor clock with the bus clock and the 'flight time' between processor and memory, all add to the first access latency when computation pipeline stall cycles are counted. These delays can push the value of *r* up near 20. In this region, long bursts are the only effective mechanism to amortise the initial latency and improve bandwidth use.

2.3.3 Make bursts an architectural issue

In order to improve the use of DRAM bandwidth, bursts should be exposed at the system architecture level. This means that code must accommodate the functions required to exploit bursts, rather than hiding a restricted, transparent mechanism in hardware (such as the line fill mechanism for caches).

Using long burst lengths has other system effects which must be addressed. For example, long burst lengths require larger buffers, latencies due to long bursts are greater, and long bursts need to be easily extracted from system code. These concerns are addressed elsewhere in this paper.

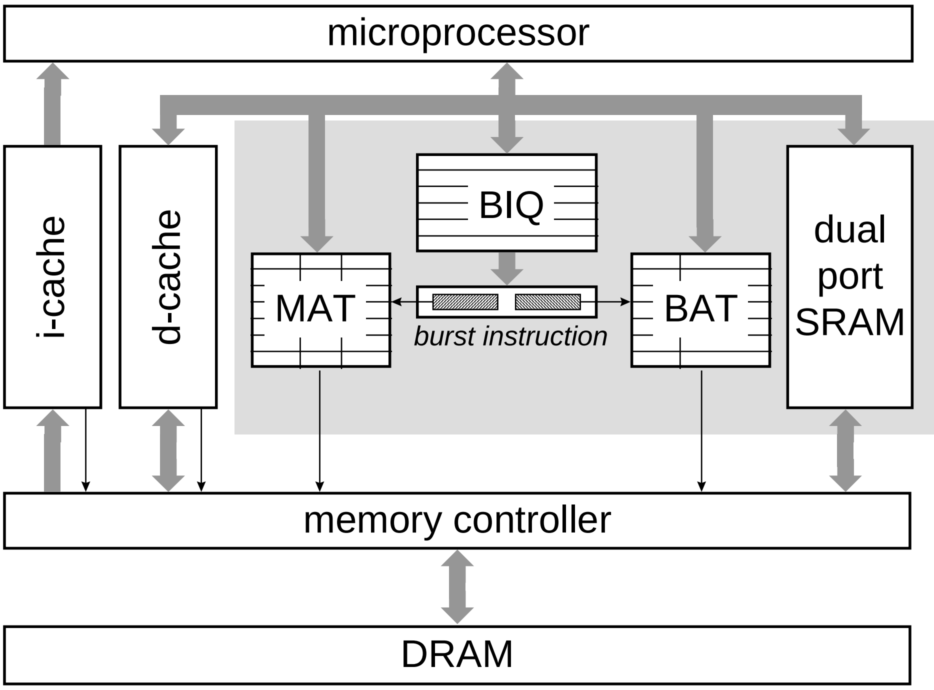

Figure 3: Overview of burst buffer architecture.

Region B in Figure 2 shows that using bursts of 32 is more appropriate than bursts of 4. This region is on the 'knee' of the curve for SDRAM and provides a factor of two improvement in average access time that ultimately equates to system throughput. Proper system design ought to be able to exploit this.

3. Architectural considerations

Section 2 reviewed three sources from which significant performance improvements could be obtained. These all indicate the need to extract maximum bandwidth from memory and the memory bus. This section considers the architectural features that are required to do so.

3.1 Exposing the memory interface

This work is based on the principle that *exposing the memory interface to the programming model provides the best way to provide the flexibility needed to ensure reliable extraction of available performance*. This implies that a compiler must be crafted to meet the development needs of users, and that the compiler must provide sufficient benefit to offset the costs of adding the extra memory interface hardware.

In order to meet these requirements, it is necessary to explicitly expose control of memory traffic to the programming model, whilst conforming to several restrictions. These include: minimising instruction overhead in the scheduling and dispatch of memory interface transaction descriptors; ensuring reliable management of data coherence, especially within a direct memory access (DMA) environment; make better use of regularity and locality; fetch accurately, rather than speculatively; and to respect the constraints imposed by the operating system, such as thread switching and interrupts.

The underlying architecture must provide mechanisms to help address these. For example, restricting the architecture to burst linear address streams enables a mechanism to be implemented to do automatic address generation, relieving the processor of this burden, whilst minimising complexity in the compiler.

Embedded systems require predictability to meet timing constraints imposed by the system. The regular nature of burst pipelining achieves this.

3.2 Burst pipelining and its implications

Figure 4 shows a timing diagram that illustrates burst pipelining. The principle is similar to double buffering bursts from each stream in a computation. As an example, consider a 192-element vector addition. The ‘C’ source code for this would be:

```
for(i=0;i<192;i++){
  a[i] = b[i] + c[i];
}
```

Figure 4 shows how this code may be decomposed into 8 stages, each operating on 32 elements from each stream.

The task is split up into blocks which are the size of the burst length. Computation is performed on these blocks in turn. Establishing a pipeline requires some architectural features which enable bursts to be issued concurrently with computation, and to have some local storage for the bursts being handled. The lower half of Figure 4 shows two traces. The top represents burst activity on the memory bus. The lower trace represents computation within the processor pipeline. A horizontal bar indicates no activity, which, for the processor, implies a stall, and, for the memory, implies wastage of available bandwidth. A vertical bar is a synchronisation point.

Figure 4 shows how the memory bus is saturated with long memory bursts when the pipeline is in its steady state. This diagram assumes that the time taken to compute the vector sum for 32 elements is less than the time to perform three 32 access bursts. In this case, the system is *bandwidth-bound* - an increase in compute performance will not improve the

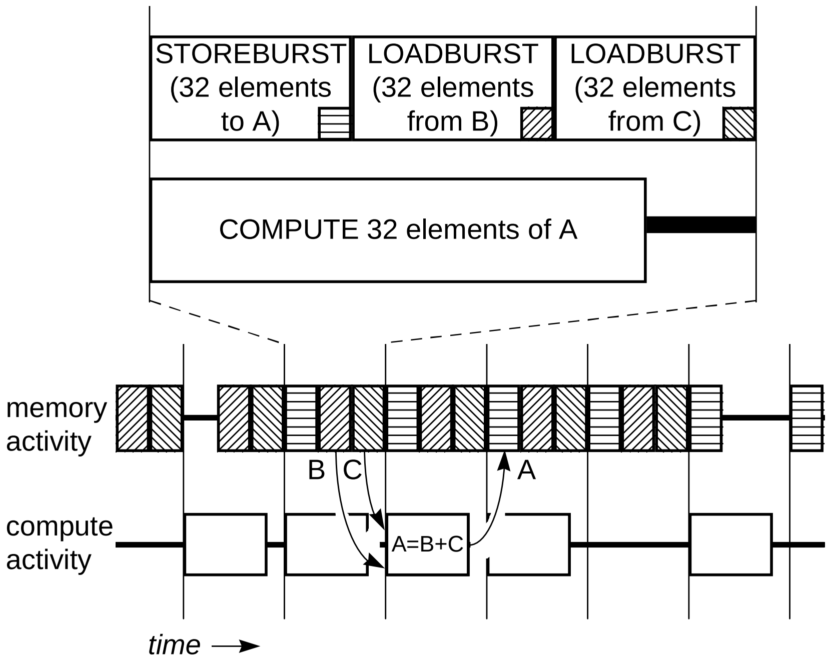

Figure 4: Burst pipelining a 192-element vector addition.

system performance. The corollary is when the system is *compute-bound*. Improvements in processor performance only help until the *bandwidth-bound* limit is met.

Achieving saturation of the memory bus requires memory transactions to proceed concurrently with computation. Architectures which can issue a memory transaction without blocking the computation pipeline are needed. Decoupled architectures [25] do this, although an alternative technique might use extensions to the cache-line fetch mechanism, as discussed in section 2.2. Extending the notion of decoupled accesses to *decoupled bursts*, and by providing a queue into which burst transaction descriptors may be placed, is essential for this architecture.

Decoupling the burst issue mechanism from the actual burst transaction implies the need for a synchronisation mechanism. The pipelined form of the computation model implies that there is a natural synchronisation point at the boundaries between computation blocks. These need to be used by the compiler.

4. Description of architecture

Figure 3 shows a block diagram of the compiler-directed memory interface used in this study.

The burst buffer architecture sits between the processor pipeline and the memory controller, parallel to the data cache. It comprises three functional areas: (a) an area of dual port static random access memory (DP-SRAM); (b) a burst instruction queue (BIQ); and (c) two tables, the memory access table (MAT) and the buffer access table (BAT).

The microprocessor is responsible for generating burst instructions. These instructions ultimately reach the memory controller which copies a burst of data to or from the DP-SRAM. Issuing a burst instruction does not stall the processor. It is passed to the BIQ, which acts as a first-in first-out (FIFO) queue. The BIQ provides the *decoupling* mechanism. Burst instructions emerging from the BIQ reference slots in the MAT and BAT tables. Information contained in these two tables is *bound* together to describe a complete memory-to-DP-SRAM transaction. This binding at run-time is key to the flexibility and low processor instruction overhead incurred when using this architecture.

The basic architecture requires three burst instructions. There are the two obvious ones (*burst-load* and *burst-store*) and a third, *sync*. This latter instruction is used to allow the burst accesses to be synchronised with computation by the processor. When a *sync* instruction has been placed in the BIQ, any further burst instructions issued by the processor will cause the processor to stall until all burst instructions ahead of the *sync* have completed. Alternatively, a special register can be polled to test the synchronisation state.

Many implementations of this architecture are possible. For this study, burst instructions were issued to the BIQ by aliasing burst instructions to memory stores at a memory address which referenced the top of the BIQ. A coprocessor interface (if available on the host processor) could also be used.

The structure for the burst instructions is given in Table 1. The *burst-load* and *burst-store* instructions comprise four fields: (a) instruction identifier (b) auto-increment bit (c) MAT slot index (d) BAT slot index. (The auto-increment field is used to optionally enable a mechanism that leaves the address field in the MAT slot pointing to the start of the next burst.)

The DP-SRAM needs to be large enough for bursts from several streams in a computation. It also needed to match the demands of burst pipelining and long bursts. Examination of

	burst instruction fields			
burst-store	00	A	MAT index	BAT index
burst-load	01	A	MAT index	BAT index
sync	11	x	x	x

Table 1: Burst instruction examples.

several applications showed that providing resources to accommodate eight streams was sufficient per loop. Burst pipelining implies that there must be two sets of buffers in existence: one set for incoming and outgoing streams; and a second set for the streams undergoing computation. This means the DP-SRAM ought to have room for 16 bursts. Section 2.3.3 suggests that bursts of 32 will achieve about 80% of the available bandwidth using SDRAMs. If there is a 32 bit wide path to memory, a burst of 32 will bring in 128 bytes. To contain 16 of these bursts, the DP-SRAM area needed to be 2Kbytes.

The memory area for containing bursts is dual-ported so that bursts to and from main memory may run concurrently with accesses from the processor. Unfortunately, using dual-port SRAM effectively doubles its area, which means it is equivalent to about 4Kbytes of single-port SRAM. This is still a modest amount of SRAM when compared to a data cache. No interlocks or priority mechanisms are provided - it is the responsibility of the compiler (or system designer) to avoid write conflicts to the same DP-SRAM location. Note that the processor side of the DP-SRAM is memory -mapped.

The memory access table (MAT) is a set of indexed slots each of which describes a transaction to main memory. Each entry in the descriptor table may be represented as the 3-tuple:

{*StartAddress, Length, Stride*}

The length field specifies the burst length for the transaction and *not* the total length of the stream. The stride field defines the element separation within the burst. For example, a slot might contain the values:

```
{0x1fee1bad, 128, 16}
```

which results in a 32 word (32 4-byte words) burst with each word separated by 4 words (4 4-byte words). For a system implementation with *n* streams, it is reasonable to suggest that there are *n* slots within the MAT.

The buffer access table (BAT) has a simpler structure comprising two fields as shown in the 2-tuple below:

{*DPSRAMAddress, Length*}

The first field points to the start address of the ‘burst buffer’ in the DP-SRAM. The second field is a copy of the length of the burst most recently transferred using that buffer slot. This permits the implementation of some DP-SRAM buffer resource management, although software could perform this task itself. Due to the requirements of double buffering, a system implementation with *n* streams would have *2n* slots within the BAT.

4.1 Extended architecture features

Extending the architecture permitted more system concerns to be addressed. These are summarised here because they did not contribute to the evaluations that were performed. Figure 5 shows some of the extensions.

1. The context table (CAT) enabled burst buffer resources to be shared across contexts using offsets and masks. These allowed the operating system to add a further level of indirection into the process of indexing MAT and BAT slots.

2. The swap feature was added to permit redefinition of the field ordering in the burst instruction. Experiments showed that instruction overhead could be minimised by judicious choice of the instruction format.

3. A buffer renaming mechanism was added to minimise instruction overhead in the construction of software pipelines. The facility was extended to provide limited support for virtual memory.

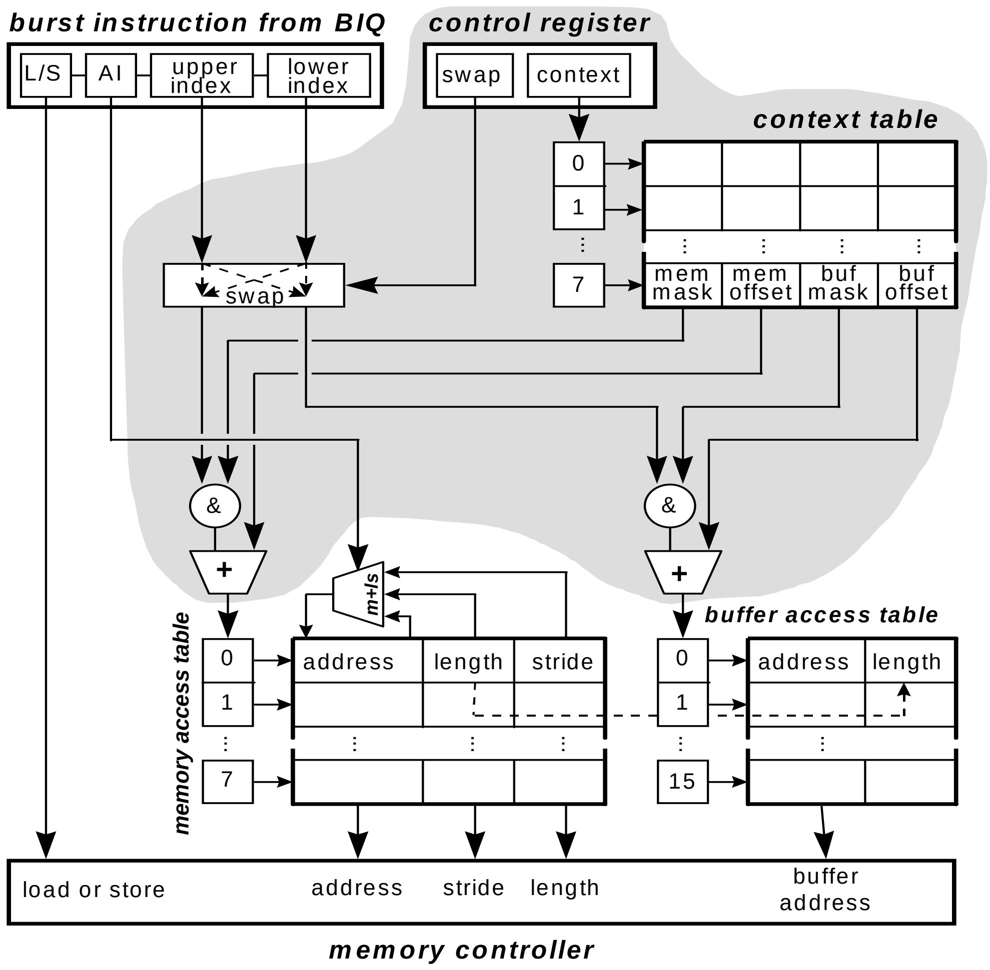

Figure 5: Block diagram of extended architecture.

5. Compiler transformations

The burst buffer compiler is a source to source code transformer which uses the SUIF [1] compiler tool kit to provide mechanisms for manipulating the ‘C’ source code. This transforms the code before it is passed to the original compiler chain. The principle task of the burst buffer pre-processor is to identify statements within loops which would benefit from block data transfers to and from the DP-SRAM. There are many similarities between the burst buffer pre-processor and the functionality of a vectorizing compiler [24]. However, there is less need here to consider the functional aspects of the computation.

Loops which contain operations using arrays are considered as candidates for the transforms. For this first implementation of the compiler there are four prerequisites to transforming a loop:

- The array accesses contained within the loop are assumed not to alias to one or more of the same locations in the address space.
- The array indices must evaluate to a loop-constant stride and offset. e.g.
```
for(i=lb;i<ub;i+=step)
  A[i*k+6].elem = B[i+2];
```
- There must be no predicates within the loop which alter the pattern of vector data flow.
- Inter-loop dependencies [23] are not handled, although transforms have been developed which enable small constant re-use distances using buffer overlapping and element transfer.

It is also necessary for the pre-processor to determine if the overheads in setting up the burst buffer architecture negate any potential performance gain. Typically, if the loop iteration count is small, then the set up costs exceed the performance gain. Where the trip count cannot be determined at compile time, the user can indicate whether to transform a particular loop with a pragma.

5.1 Managing burst buffer resources

Before one or more loops can be transformed within a function, it is necessary to ensure that there are enough burst buffer resources available. The resources which must be managed at runtime include: MAT; BAT; and areas in the DP-SRAM. The resource management is either performed statically or dynamically. The pre-processor can generate code for both policies through a command line switch.

Static management requires that the complete program is made visible to the burst buffer pre-processor at compile time. In this case, particular slots in the tables and areas in the DP-SRAM are bound to specific loops within the program.

For dynamic management, the pre-processor in-lines simple routines which manage the burst buffer resources. So as to keep the overhead for this policy low, the allocation and deallocation routines are based on a simple FILO (stack-based) mechanism.

In both cases, the resources required to transform a particular loop may exceed the available burst buffer resources. In these cases, calls are made to a run-time support library which overflows and underflows the resources using main memory. Clearly this is very expensive, and careful consideration must be given to the sizes of the tables and DP-SRAM for a particular application.

```
for(i=0;i<200;i++)
  A[i] = B[i];
```

becomes:

```
Set_matslot(0,&B[0],128,4); /*Slot 0 {&B[0],32,1}*/
Set_matslot(1,&A[0],128,4); /*Slot 1 {&A[0],32,1}*/
Set_batslot(0,0); /*1 buffer - Slot 0 offset 0 */

for(i=0;i<200;i+=32){ /* stripped to 32 words */

  if(200-32<i){ /* handle last iteration */
    *BIQ = SYNC; /* await burst completion */

    /* set burst length (bytes) to remainder */
    Set_length(0,200-i<<2);
    Set_length(1,200-i<<2);
  }
  *BIQ = BL(0,0); /* burst-load MAT 0 BAT 0 */
  *BIQ = BS(1,0); /* burst-store MAT 1 BAT 0 */
  *BIQ = SYNC; /* issue SYNC, blocks next inst. */
}
*BIQ = SYNC;
```

Figure 6: Example non-pipelined code.

5.2 Non-pipelined transform

If the loop consists exclusively of data movement using array copies and constant initialisations, then the pre-processor does not attempt to pipeline the code. The loop is stripped using the burst length parameter as a strip length and burst instructions are inserted. An example of this is shown in Figure 6.

Note that the original (`A[]=B[]`) assignment has been replaced with a sequence of load and store burst instructions. `BL()`, `BS()`, `SYNC`, `Set_matslot()`, `Set_batslot()` and `Set_length()` are macros, which typically scale to one or two processor instructions.

5.3 Pipelined transform

When the loop contains computation, the compiler attempts to pipeline the code. Essentially, the original code is stripped and unrolled twice before having the burst buffer instructions inserted. Figure 7 shows the output of the pre-processor when the code in Figure 3 was used as input.

5.4 Types and burst lengths

The buffers which are created for the computation are always cast to be the type of the original element accessed. When a loop contains different array types, the programmed stride is affected by the coefficient of the access vector in conjunction with the size of the type. Where two or more different types are combined in the same loop, the burst lengths are adjusted to maintain the semantics of the buffer index variable.

Accessing an array of structures results in a single member being burst through the modification of the *stride* parameter. However, unions which may pack different types into a single word are aliased to the same buffer.

```
Set_matslot(0, &B[0], 128, 4);
Set_matslot(1, &C[0], 128, 4);
Set_matslot(2, &A[0], 128, 4);

/* Allocate stage 1 buffers */
buf0 = (int *)&bb_buffer[0];        /* B */
Set_batslot(0, 0);
buf1 = (int *)&bb_buffer[128];      /* C */
Set_batslot(1, 128);
buf2 = (int *)&bb_buffer[256];      /* A */
Set_batslot(2, 256);

/* Allocate stage 2 buffers */
buf3 = (int *)&bb_buffer[384];      /* B */
Set_batslot(3, 384);
buf4 = (int *)&bb_buffer[512];      /* C */
Set_batslot(4, 512);
buf5 = (int *)&bb_buffer[640];      /* A */
Set_batslot(5, 640);

/* start pipeline */
*BIQ = BL(0,0); /* burst B buf 0 stage 1 */
*BIQ = BL(1,1); /* burst C buf 1 stage 1 */

for(i=0;i<192;i+=64){

  *BIQ = BL(0,3); /* burst B buf 3 stage 2 */
  *BIQ = BL(1,4); /* burst C buf 4 stage 2 */

  for(tmp=0;tmp<32;tmp++) /* 1st stage (A=B+C) */
    buf2[tmp] = buf1[tmp] + buf0[tmp];

  *BIQ = SYNC; /* wait for loadbursts */
  *BIQ = BS(2,2); /* burst buf 2 into A */

  if(i<192-64){ /* if not last iteration */
    *BIQ = BL (0,0); /* burst B buf 0 stage 1 */
    *BIQ = BL (1,1); /* burst C buf 1 stage 1 */
  }

  for(tmp=0;tmp<32;tmp++) /* 2nd stage (A=B+C) */
    buf5[tmp] = buf4[tmp] + buf3[tmp];

  *BIQ = SYNC; /* wait for loadbursts */
  *BIQ = BS(2,5); /* burst buf 5 into A */
}
*BIQ = SYNC; /* cause next inst. to stall */
*BIQ = SYNC; /* stall until bursts complete */
```

Figure 7: Piplined vector addition.

5.5 Optimisations

The burst buffer architecture permits a code optimisation that relieves the processor of some complex address generation. Other optimisations which are applicable in vectorizing compilers are also applicable with the burst buffer architecture. Techniques such as code avoidance through copy optimisation and redundant code elimination are implemented in the first generation compiler.

6. Results

6.1 Evaluation architecture

Tests were performed within a simulation environment combining the SimpleScalar simulator [21] and the Seec memory hierarchy simulator [22]. This tool permitted cycle-accurate timing of memory transactions within configurable memory hierarchies. The evaluation architecture is shown in Figure 8.

The architectural parameters for the evaluation are given in Table 2. These represent a typical embedded system utilising a workstation-class processor with only a level-1 data cache. Both i- and d-caches share a bus to external memory, which comprises burst read-only memory (ROM) and SDRAM. Burst profiles for random bursts within these memories are given. Note that 66MHz SDRAM is modelled.

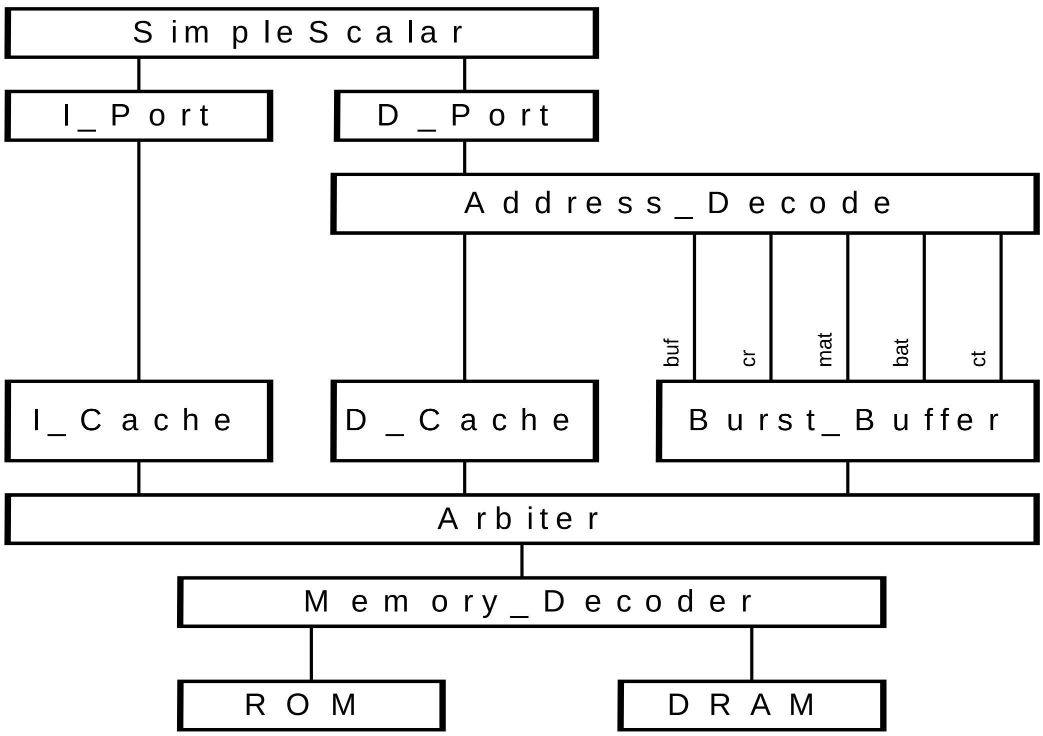

Figure 8: Evaluation architecture.

Description	Value	Units
Instruction issue width	1	
Processor clock frequency	133	MHz
Processor bus frequency	66	MHz
Processor clock cycle (pcyc)	7.5	ns
Processor bus cycle (bcyc)	15	ns
DRAM burst access profile	7:1:1:1:…	bcyc
ROM burst access profile	8:2:2:2:…	bcyc
I-cache size	16	Kbytes
I-cache associativity	2	way
I-cache line size	32	bytes
D-cache size	8	Kbytes
D-cache associativity	{1,2}	way
D-cache write policy	write-back	
D-cache line size	16	bytes
D-cache replacement policy	L.R.U.	
D-cache miss penalty	5	pcyc
Burst buffer size	2	Kbytes
Burst buffer access time	1	pcyc
Burst buffer register access time	1	pcyc
Burst buffer MAT entries	8	
Burst buffer BAT entries	16	
DRAM bus width	32	bits
Optimal burst length	128	bytes

Table 2: Evaluation architecture parameters.

Name	Description	Compiled?
memcpy	word-aligned memory copy	yes
vecadd	two integer vector addition	yes
onedconv	one-dimensional convolution with 5-tap kernel, integer samples	hand-optimised
matrix	integer matrix multiply	hand-optimised
haar	wavelet transform using integer samples	hand-optimised
sharpen	image sharpener - comprises chain of: vector subtraction; memcpy; vector addition; and saturated arithmetic.	yes

Table 3: Benchmark suite.

6.2 Evaluation algorithm suite

A non-standard benchmark suite was adopted to evaluate the architecture. The primary motivation for this was that tests were needed to evaluate the memory hierarchy performance with and without the burst buffer memory interface, and not to concentrate specifically on the computation pipeline. Additionally, the media processing domain places different constraints on the benchmark suite, since deadlines often need to be met, and it is these which drive performance. Since few performance gains were expected with scalar code suites, and algorithms representative of real applications within products were available, a suite comprising several ‘data-intensive’ algorithms was constructed. These are shown in Table 3.

The table also indicates which algorithms were processed by the compiler. In fact, all algorithms were processed by the compiler, but the transformations implemented did not optimally exploit buffer re-use. This meant that better performance improvements were obtained by hand-optimising three of the algorithms.

The suite was constructed to permit investigation into several architectural issues which follow. Each issue is appended with the benchmarks with which it was investigated.

1. *performance constraints* - develop understanding of behaviour of bandwidth- and compute-bound algorithms. (memcpy, vecadd, sharpen)
2. *word-alignment* - understand whether bursts need to respect type sizes or may be confined within fixed sized words during transactions with memory. There are two possible techniques - dynamic bus sizing in hardware or emulation in software. (memcpy)
3. *re-use* - study the techniques required to maximise the use of available burst buffer resources. (onedconv, matrix, haar, stereo)
4. *applications* - evaluate the effects of Amdahl’s Law over larger programs aggregating many algorithms. (sharpen)
5. *locality* - investigate the restrictions of loop-constant stride and offset of the compiler on algorithms transformation.

6.3 Test procedure

The code used in each benchmark underwent two separate transformations. The first optimised it for use with a data cache; the second optimised it for use with burst buffers. Ultimately, compilers will be able to perform most of these transformations, but for this experiment, some hand optimisations for the two architectures were performed. These optimisations were intended to extract as much of the performance gains as possible to enable justifiable comparison.

In order to limit the results presented here, the tests evaluated the speed-ups of the burst buffer implementation over the best available code implementation optimised for variants of data cache. The initial conditions assumed that all data was resident in memory (i.e. none in the data cache) and that the cache was either clean or dirty. One and two-way set associative data caches were assessed, as these are the most common data cache configurations. Data sets were ordered in memory to *prevent* excessive aliasing of addresses that would degrade cache performance due to conflict misses. The end conditions in each case were different: the burst buffer version left the data in main memory; the cache-optimised version left some data in cache. In all cases, the simulations included instruction cache performance and start-up costs. All bursts were capped to a maximum length of 32 memory accesses.

6.4 The numbers

Figure 9 shows the results from simulations of the benchmark suite. All the graphs examine speed-up over 1- and 2-way set associative data caches over varying data set sizes. Many observations may be made.

1. For the simple algorithms (memcpy, vecadd), the graphs illustrate the benefits of the burst buffer architecture, rather than cache degradation effects, which appear in the more complex algorithms.
2. The major benefit is due to the exact data placement policy imposed by the compiler, which ensures efficient use of any data copied to local memory and prevents any conflicts.
3. As the data set size approaches the cache boundary, the cache exhibits instability and becomes unpredictable. The burst buffers do not imply any boundary discontinuity, so their behaviour is deterministic and efficient.
4. In most cases, the burst buffer architecture begins to win with data set sizes greater than 100. This is a measure of the overheads associated with burst buffers.
5. Factors of two or more improvement occur for data set sizes of around 1000 integers. This corresponds to the point at which the cache is filled by the number of data streams.
6. All curves become asymptotic to some value, which is when the overheads of the burst buffers have been amortised and cache behaviour has become poor.
7. The cache results were obtained under best data placement conditions. As such, they will be close to the lower bound of burst buffer performance improvement expectations. Further tuning of the benchmarks ought to achieve more performance, e.g. haar was not burst pipelined for the burst buffer version.
8. The effect of starting with a clean cache compared to a dirty cache is to add a time overhead corresponding to the time taken to flush some or all dirty data. This means that both cases ultimately converge as this overhead is amortised. In fact, sharpen illustrates that for large applications, initial cache state is irrelevant.
9. As expected, direct-mapped (1-way) caches seriously degrade as data set sizes increase. This is exemplified by onedconv, matrix and sharpen. This explains the dramatic speed-ups.
10. The speed-ups of matrix are largely due to thrashing effects of data with the cache. *Note that this version used simple matrix code for cache comparisons.*

It is also necessary to pass some comments on the issues that were being addressed.

1. *performance constraints* - the experiments proved that saturating the memory bus may be achieved using a simple memory hierarchy. The burst buffer architecture is the mechanism which achieves this.
2. *word-alignment* - the cost of emulating bus sizing in software can be minimised by burst pipelining the required shift operations. This makes the hardware simple and avoids performance penalties on the data path. It is not yet clear whether a hardware version would give better performance for the extra complexity.
3. *re-use* - the optimisations performed by hand stressed the importance of this issue. Matrix and onedconv gained some of their improvements through optimal re-use of burst buffers.

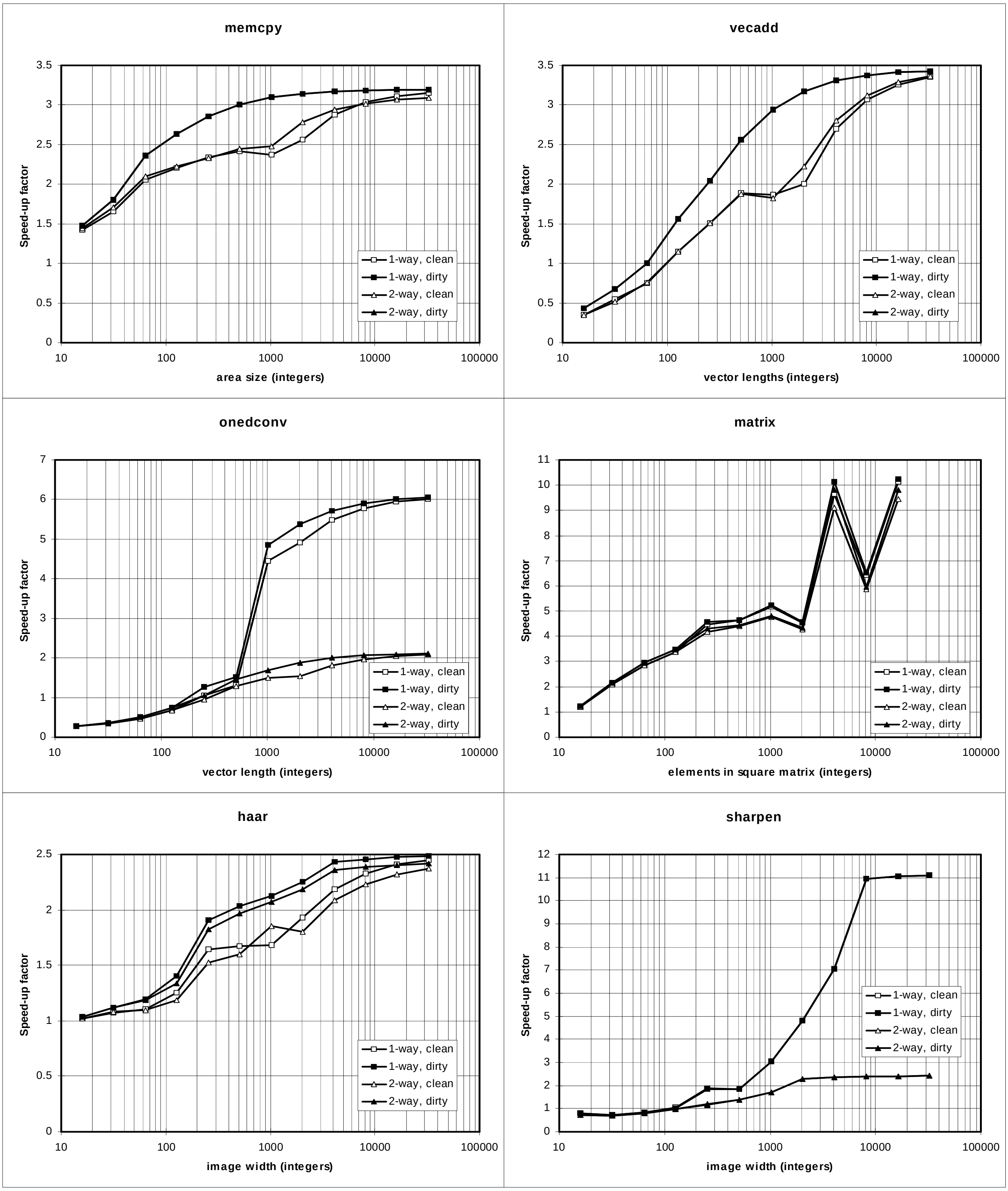

Figure 9: Burst buffer speed-ups over data cache variants.

4. *applications* - sharpen was the only benchmark comprising multiple media algorithms. Because the compiler could target each of these, local speed-ups still managed to aggregate into a significant overall speed-up.
5. *locality* - the improvements in performance were unhindered by the linear addressing scheme imposed by the architecture and compiler. In fact, stride capability proved to be one of the significant contributions to performance improvement, especially for matrix, haar and sharpen.

7. Conclusions

The experiments described in this paper have demonstrated how the addition of a new, compiler-directed memory interface can dramatically improve system performance by factors greater than two when processing media algorithms. These gains are achieved by exploiting long bursts to any variety of DRAM, concurrently scheduling the bursts with computation, and efficient re-use of local memory. The memory interface generated to do this was extremely simple; it comprised 2Kbyte of dual port SRAM, some tables and a FIFO queue. The cost, though small, may be further reduced by trading off data cache size, since the addition of the burst buffers reduces conflict misses by removing the media data streams, and so improves cache behaviour. Note that for all the burst buffer experiments, at most only 1 Kbyte of DP-SRAM was used, compared with the full 8 Kbytes for the data cache.Amdahl's Law will dilute the gains for large applications, but this does not reduce their value. As media data streams become more dominant, the processing of these flows will be required to meet specific deadlines within real-time contexts (such as video), and it is these contexts, not the overall application, which will define performance requirements.

The work in this area has also addressed many of the issues related to the integration of such a memory interface into a complete system. Although not described here, coherency strategies have been investigated, other architectures exploiting the same DRAM characteristics have been assessed, and virtual memory extensions have been devised. These use combinations of architecture, compiler and operating system support. Many issues in these areas still need resolution and will be reported at a later date.

A preliminary investigation into the sensitivity of performance to latency between processor and burst buffers has shown a gradual roll-off in performance. This means that greater latencies may be tolerated for little loss of performance.

The benefits of the architecture are gained using a compiler, which is best viewed from two perspectives: the transformations; and the implementation. Work in the former helped define the final architecture, along with other constraints, such as minimal instruction overhead and decoupling bursts. The latter resulted in a 'first generation' compiler which implemented only a subset of these transformations. Future generations of the compiler will benefit from more complex transformations that will move the compiled code further towards the performance achieved by hand coding. Many of the techniques scale from complementary activities in vector and parallel processing - in fact, the burst buffers may be viewed as a set of vector registers. One issue to be resolved is: *given the successful investments in compiler transformations, how much further investment is required to get more sophisticated optimisations and are they on the curve of diminishing returns?* This will be continually reviewed by this research.

The memory area in the burst buffers need not be restricted to the use proposed in this paper. As a piece of generic memory, it may also be used for local variable storage or look-up tables, for example.

The final question to address is: *how relevant is this technology to general purpose computing?* The answer to this is complex, but depends largely on the expected merger of general purpose computing with media processing. The burst buffer architecture may be extended to meet the requirements of this domain, which means that this is a viable, low cost method for moving the data in and out of the processor, using most of the available DRAM bandwidth.

Acknowledgements

The authors wish to thank all members of the Appliance Computing Department at HP Laboratories, Bristol, for their contributions to, and reviewing of, this paper. Thanks are also expressed to Jack Meador at HP Boise for his endeavours.

References

[1] *The SUIF Library*. Stanford University Compiler Group. Version 1.0 1994.

[2] McKee, S.A.; Wulf, W.A: *Hitting the Memory Wall: Implications of the Obvious*. Technical Report No. CS-94-48, University of Virginia, Dept of Computer Science, December 1994.

[3] Burger, D; Goodman, J.R; Kagi, A: *Quantifying Memory Bandwidth Limitations of Current and Future Microprocessors.* May-96 In Proceedings of the 23rd Annual International Symposium on Computer Architecture.

[4] McKee,S.A; Moyer, S.A; Wulf, Wm.A; Hitchcock, C.Y: *Increasing Memory Bandwidth for Vector Computations.* Dept of Computer Science, University of Virginia Aug-93 Tech Report No. CS-93-34

[5] Huang, A.S; Shen, J.P: *The Intrinsic Bandwidth Requirements of Ordinary Programs.* Oct-96 In Proceedings of ASPLOS-VII.

[6] McKimley, K.S; Temam, O: *A Quantitative Analysis of Loop Nest Locality.* Oct-96 In Proceedings of ASPLOS-VII.

[7] Diefendorff, K; Bubey, P.K: *How Multimedia Workloads Will Change Processor Design.* IEEE Computer, Vol. 30 No. 9, September 1997.

[8] Przybylski, S.A: *New DRAM Technologies: A Comprehensive Analysis of the New Architectures.* Second Edition, MicroDesign Resources, 1996.

[9] Smith, A.J: *Cache Memories.* University of California, Berkeley Jun 82 ACM 0010-4892/82/0900-0473.

[10]Jouppi, N.P: *Improving Direct-Mapped Cache Performance by the Addition of a Small Fully-Associative Cache and Prefetch Buffers.* Jun-90 In Proceedings of the 17th Annual International Symposium on Computer Architecture, P364-373.

[11]Kroft, D: *Lockup-free instruction fetch/prefetch cache organisation.* In Proceedings of the 8th Annual International Symposium on Computer Architecture, P81-87.

[12]Chen, W.Y; Mahlke, S.A; Chang, P.P; Hwu, W-M.Wa: *Data Access microarchitectures for Superscalar Processors with Compiler-Assisted Data Prefetching.* Center for Reliable and High Performance Computing, University of Illinois Jan-91 ACM 0-89791-460-0/91/0011/0069.

[13]Fu, J.W.C; Patel, J.H: Data *Prefetching in Multiprocessor Vector Cache Machines.* Center for Reliable and High Performance Computing, University of Illinois Jun-91 ACM 0-89791-394-9/91/005/0054.
[14]Chen, T-F; Baer, J.L: *Reducing Memory Latency via Non-blocking and Prefetching Caches.* Jun-92 In Proceedings of ASPLOS-V ACM 0-89791-535-6/92/0010/0051.
[15]Mowry, T.C; Lam, M.S; Gupta, A: *Design and Evaluation of a Compiler Algorithm for Prefetching.* Jun-92 In Proceedings of ASPLOS-V ACM 0-89791-535-6/92/0010/0062.
[16]Chen, T-F.: *Data Prefetching for High Performance Processors.* Dept of Computer Science and Engineering Jul-93 Tech Report 93-07-01.
[17]Chen, T-F, Baer, J.L: *Effective Hardware-Based Data Prefetching for High-Performance Computers.* May-95 IEEE Transactions on Computing vol 44 no 5 0018-9340/95.
[18]Lee, J-H; Lee M-Y; Choi, S-U; Park, M-S: *Reducing Cache Conflicts in Data Cache Prefetching.* Computer Systems laboratory, Dept of Computer Science, Korea University Jun-95.
[19]Zucker, D.F; Flynn, M.J; Lee, R.B: *A Comparison of Hardware Prefetching Techniques for Multimedia Benchmarks.* Stanford University Dec-95 Tech Report No. CSL-TR-95-683.
[20]Shlomit S. Pinter, Adi Yoaz: *Tango: a Hardware-based Data Prefetching Technique for Superscalar Processors.* Dec-96 In Proceedings of Micro-29 ISBN 0-8186-7641-8.
[21]Burger, D; Austin, T.M; Bennett, S: *Evaluating Future Microprocessors: The SimpleScalar Tool Set.* July 1996 Technical Report No 1308 Computer Sciences Department University of Wisconsin, Madison WI.
[22]Quick, S.V: *The Computational demands of the modified Newton-Raphson algorithm in electrical impedance tomography.* April 1993 MSc Thesis Dept. of Electrical Engineering and Electronics, UMIST, Manchester, UK.
[23]Wolfe, M: High performance Compilers for Parallel Computing. Addison-Wesley ISBN 0-8053-2730-4.
[24]Hwang, K; Briggs, F.A: *Computer Architecture and Parallel Processing.* McGraw-Hill International Editions. ISBN 0-07-Y66354-8.
[25]Hennesssy, J.L; Patterson, D.A: Computer Architecture: A Quantitative Approach. 2nd Edition. Morgan Kaufmann ISBN: 1-55860-329-8.