Compiling for the Impulse Memory Controller

Xianglong Huang Zhenlin Wang Kathryn S. McKinley

Department of Computer Science, University of Massachusetts, Amherst
{xlhuang, zlwang, mckinley}cs.umass.edu

ABSTRACT

The Impulse memory controller provides an interface for remapping irregular or sparse memory accesses into dense accesses in the cache memory. This capability significantly increases processor cache and system bus utilization, and previous work shows performance improvements from a factor of 1.2 to 5 with current technology models for hand-coded kernels in a cycle-level simulator. To attain widespread use of any specialized hardware feature requires automating its use in a compiler. In this paper, we present compiler cost models using dependence and locality analysis that determine when to use Impulse to improve performance based on the reduction in misses, the additional cost for misses in Impulse, and the fixed cost for setting up a remapping. We implement the cost models and generate the appropriate Impulse system calls in the Scale compiler framework. Our results demonstrate that our cost models correctly choose when and when not to use Impulse. We also combine and compare Impulse with our implementation of loop permutation for improving locality. If loop permutation can achieve the same dense access pattern as Impulse, we prefer it, since it has no overheads, but we show that the combination can yield better performance.

1. Introduction

In this paper, we develop, implement, and evaluate compiler cost models to drive compiler optimizations for the Impulse memory controller [2, 3, 15]. The Impulse memory controller provides address remapping in the memory controller for mapping sparse memory access patterns into dense ones, thus exploiting the cache more effectively and reducing the memory/bus bandwidth requirements of programs with poor memory locality.

Previous work demonstrates with simulation that using fine grain Impulse remappings can improve programs by a factor of 1.2 to 5 with current technology on scientific and image processing codes, and by a factor of more than 10 with future technology models. These results use hand coded kernels. Given the ever evolving variety and complexity of hardware, it is unreasonable to expect programmers to exploit it effectively by hand, even for features such as Impulse with large potential benefits. Instead, compilers must automate optimizations to exploit hardware effectively.

PACT. September 2001, Barcelona, Spain.

This work demonstrates that a compiler can automate the use of Impulse for kernels with a few loop nests, and leaves to future work, for the most part, automation for large applications. Our compiler uses locality analysis to detect spatial and temporal locality [10]. For array references without locality, the compiler then determines if they will benefit from an Impulse remapping. We develop cost models for three forms of fine grained Impulse remappings that consider the data set sizes, cache memory configurations, and the current cache contents. The cost model estimates the net cost of a remapping by determining the number of cache misses before and after remapping, multiplying these counts by the cost of a cache miss with and without remapping. (Impulse misses take longer to service than other misses.) The model also includes the cost to set up the remapping. We apply Impulse remapping when it is profitable, generating the Impulse system calls that set up a new array and modifying the loop to use the new array. Our results show that our cost model accurately chooses when and when not to apply Impulse remapping. We also compare Impulse remapping with loop permutation to improve locality [10], and the combination of loop permutation and Impulse remapping. Since permutation has no additional overheads, if it can attain the same access pattern as the Impulse remapping, we will always prefer it. However, we show the combination achieves better performance in some circumstances.

The remainder of this paper is organized as follows. We first give the hardware and compiler background necessary to understand our work. Next, we describe our cost models for using Impulse's fine grained data remappings and code generation. Our results show that our cost models correctly predict when and when not to use Impulse remapping for both current and future technology models for five kernels that together exercise all of the remappings. We then offer conclusions and discuss future work.

2. Background

In this section, we first review some trends in hardware techniques for hiding memory latency. Next, we give an overview of how Impulse works, and of the fine grained dynamic data remappings for which we will compile. We then briefly describe how we build on previous locality analysis.

2.1 Hardware Trends and Impulse

To address the memory gap, computer architects have recently proposed adding processing power to the memory system [8, 11, 17]. Most of these efforts aim to improve memory system performance by offloading varying amounts of work to the memory system itself. For example, the Morph architecture [17] is almost entirely configurable: programmable logic is embedded in virtually every

0-7695-1363-8/01 $10.00 © 2001 IEEE

data path in the system. The result is quite complex and likely to be slower than a conventional microprocessor. The RADram project [11] and the IRAM project [8] have the memory perform computation to eliminate memory to CPU transfers. The Impulse approach is instead in the category of "smart" memory systems that can help hide relatively high memory latencies [8, 11, 17].

The Impulse main memory controller (MMC) can be configured to respond to accesses to otherwise unused physical addresses, e.g., those at or above 0x40000000 on a system with one gigabyte of installed DRAM. We refer to such addresses as *shadow addresses*. Accesses to physical addresses that correspond to installed DRAM are treated normally. However, the processor configures the Impulse memory controller to treat accesses to shadow physical addresses as something other than a bus error, as would be generated on a system with a conventional memory controller. In particular, the MMC can *gather* cache lines in shadow memory from locations in real physical memory.

An application that is using Impulse's address remapping functionality performs a system call indicating what kind of remapping to create. For example, one remapping operation that Impulse supports is to scatter/gather array data accessed via an indirection vector, as illustrated in Figure 1. The `ams_mapshadow` system call configures the MMC to create a synthetic variable, `Pi`, that corresponds to the indirectly accessed data elements `P[ColIdx[i]]` in the original program. In response to the system call, the OS allocates a sufficiently large range of shadow physical addresses to contain `Pi`. The OS then configures the MMC by performing a series of I/O writes to indicate remapping-specific information such as which address range to configure, what kind of remapping the MMC should perform (e.g., access via an indirection vector), the size of the elements being remapped, and the location of a page table that specifies the location of the target data in real physical memory. After configuring the MMC, the OS maps the shadow physical addresses to an unused portion of the application's virtual address space and returns a pointer to where this range starts to the user process. This sequence sets up the shadow memory for Impulse remapping..

Figure 2 illustrates how the Impulse hardware then gathers data via the indirection vector. When the user process accesses an address corresponding to the remapped data structure, `Pi`, the processor converts this virtual address to the corresponding shadow physical address using its normal virtual memory mapping functionality. When the Impulse MMC sees a read (or write) request to a valid shadow address, it must determine which virtual addresses in the process address space correspond to this shadow region, convert these virtual addresses to real physical addresses, and load them from memory. To support this functionality, the MMC contains both a pipelined address calculation unit and an MTLB (the MMC translation look-aside buffer, or MTLB for short). In our example, the address calculation unit would determine from the shadow address which elements of the remapped array the program is accessing, load the corresponding elements from the indirection vector into the Indirect Vector (IV) buffer, and calculate the virtual addresses in the calling process for the corresponding elements of the original array. These virtual addresses are passed through the MTLB to determine the physical addresses of the elements. The MMC loads the elements residing at these physical addresses into an output buffer using an optimized DRAM scheduler, and sends the gathered data back to the processor to satisfy the outstanding read request.

2.2 Fine Grained Remappings

Impulse offers both fine [2, 3] and page grain remappings [15], as well as MMC prefetching. We focus only on the fine grain remappings here. The prototype Impulse hardware supports eight active remapping at once. The fine grain remappings require applications with data structures that exhibit largely static, albeit highly irregular, data access patterns (e.g., sparse matrix-vector products and ray tracing). Impulse has three remappings that we consider in the compiler: *transpose* (a.k.a. corner-turn), *base-stride*, and *indirection vector*. Section 2.1 describes the *indirection vector* remapping, and now we describe the other two with simple examples. We assume C language row-major storage order for the remainder of the paper. Consider the following code fragment.

```
for (i=0; i<N; i++)
     for (j=0; j<N; j++)
        ...A[j][i]...
```

With row-major accesses to A and an inner loop that accesses more than the number of cache lines, each access will cause a miss. Assuming of course that other array dependences preclude loop interchange, the Impulse *transpose* remapping simply swaps the dimensions to produce contiguous accesses such that the reuse of each cache line is close together in time.

```
tp_arg.newaddr=&A_New;
tp_arg.vaddr = A;
...
ams_mapshadow(TRANSPOSE, tp_arg)
for (i=0; i<N; i++)
     for (j=0; j<N; j++)
        ...Anew[i][j]...
```

The *base-stride* remapping generalizes *transpose*. Consider this simple example.

```
for (i=0; i<N; i++)
     ...  A[i*s]...
```

If the step `s`, which is a loop invariant, results in one or very few touches to each cache line of A, Impulse can remap the accesses to contiguous ones as follows.

```
bs_arg.newaddr=&A_New;
bs_arg.vaddr = A;
...
ams_mapshadow(BASESTRIDE, bs_arg)
for (i=0; i<N; i++)
     ...  Anew[i]...
```

This remapping is especially useful for C codes in which the programmer has linearized multi-dimensional array accesses.

2.3 Locality Analysis and Compiler Support

We use dependence testing and locality analysis to detect spatial and temporal reuse in a nest [10]. These are implemented in our Scale Compiler framework.[1] The compiler first constructs a dependence graph by using the Omega libraries [13]. After constructing the dependence graph, the compiler performs locality analysis and detects spatial and temporal reuse using the method described by

[1] See http://www-ali.cs.umass.edu/Scale/.

```
for (i = 0; i < n; i++) {
   sum = 0;
   for (j = Rows[i]; j < Rows[i+1]; j++)
      sum += Data[j] * P[ColIdx[j]];
   b[i] = sum;
}
```

Original code

```
arg.newaddr = &Pi;
arg.vaddr = P;
arg.ivaddr = ColIdx;
...
ams_mapshadow(VINDIRECT, &arg);
for (i = 0; i < n; i++) {
   sum = 0;
   for (j = Rows[i]; j < Rows[i+1]; j++)
      sum += Data[j] * Pi[j];
   b[i] = sum;
}
```

After remapping

Figure 1: ***Scatter/gather through an indirection vector*** **changes indirect accesses to sequential accesses.**

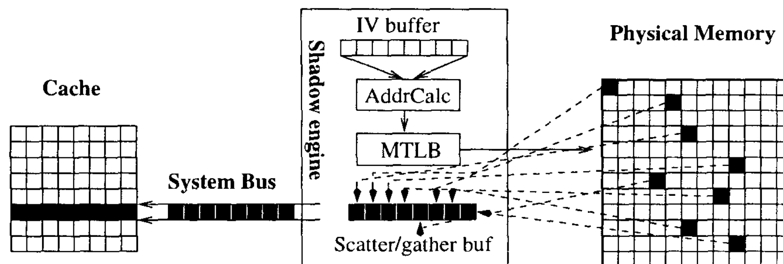

Figure 2: Gathering data through an indirection vector.

McKinley, Carr, and Tseng [10]. If necessary, our compiler applies loop permutation to improve the locality of the array references, as described in the same paper. Besides dependence testing and locality analysis, the compiler also computes a *regular section* for each array reference inside a loop. A regular section is a triple with the format: *lowerbound:upperbound:step*. If an array in the innermost loop nest still has neither spatial nor temporal locality, then we determine if its access pattern matches any of the the three Impulse remapping forms. If so, we apply our cost models, which will use the information provided by the regular section, as described in the Section 4 below, and use remapping only if the cost model indicates it will be profitable.

3. Related Work

In previous work, Carter et al. [2, 3] present their hand-coded optimizations for the Impulse MMC. Their work shows great potential for the Impulse memory controller to improve program performance. Here we are trying to automate this process in a compiler.

Chandramouli et al. [4] have independently studied the effects of a cost model for Impulse that also considers copy-based array restructuring which ours does not. Our cost model is more detailed in estimating the cache misses and therefore the real cost of the array references before and after remapping.

Some related work has produced more accurate and expensive models of data locality, for example Ghosh et al. [5, 6], but we do not require this level of accuracy. Static data remapping for C and Fortran programs has typically been implemented with copying [16, 7, 9]. The overheads are different than in Impulse, and a full experimental comparison is beyond the scope of this paper.

4. Cost Models

This section is organized by the remappings. We begin with the simplest remapping, *transpose*, and then present *base-stride* and the *indirection vector* remappings. Each model first assumes a cold cache, which means none of the data is in the cache before the loop nest, and then we generalize for the case when data may be in the caches. Since the MMC gathers L2 cache lines in an Impulse remapping from a variety of memory locations, it lengthens the time to load a line into the L2 cache. The time to load the now dense line into the L1 cache is unchanged compared to a conventional L1 load. We thus optimize for L2 cache misses, and below we always refer to the L2 cache, unless we explicitly state otherwise. We denote the costs of the original array accesses with the subscript *org* and the Impulse accesses with the subscript *imp*. We compute the cost of the original loop nest and the Impulse nest as follows.

$$
\begin{aligned}
cost_{org} &= miss_{org} \times miss_penalty \\
cost_{imp} &= miss_{imp} \times miss_penalty_{imp} + setup_{imp}
\end{aligned}
$$

In our cost model the setup cost of Impulse remapping is calculated by the following heuristic, suppose the size of the array being remapped is M bytes and the size of shadow region being generated is Y bytes then the cycles Impulse takes to do remapping is:

$$setup_{imp} = M \times A + Y \times B + C$$

The constants A, B, and C are machine dependent. In our experiments, this estimation is within 5% compared with the real setup cost.

When $cost_{org} > 1.05 \times cost_{imp}$, we will generate the appropriate Impulse remapping for the array references (i.e., if we predict Impulse will improve performance by more than 5%, we will generate

Variables	Definitions
CS	cache size
CLS	cache line size
DV	data volume of a loop
dv	data volume of an array
E	array element size
EI	indirect vector element size
M	array size
N	indirect vector size
RS	row size of a matrix
st	stride

Table 1: Variable definitions

code that uses the Impulse system calls). We use overheads that match those gleaned from the simulator.

As we just mentioned, the Impulse L2 miss penalty is more expensive than the original; it is around 2 times higher than a conventional miss. This cost is amortized if the number of misses is reduced by more than 2 per second level cache line. Impulse can further reduce this miss penalty using prefetching, but we do not model prefetching at this point.

Our compiler framework determines E = the size of elements in the array. Let CS = the L2 cache size, and CLS = the length of L2 cache line. We set these two constants in a target architecture configuration file that Scale uses. The regular section is *lowerbound: upperbound: step*. In our cost model, we use $M = (upperbound - lowerbound) \times E$, which is the range of the target array that the loop will accessed. We thus have the data access volume, dv, of each reference:

$$dv = \begin{cases} \frac{upperbound - lowerbound}{step} \times E & : \; step \times E > CLS \\ (upperbound - lowerbound) \times E & : \; otherwise \end{cases}$$

We also compute the total data access volume in the loop nest.

$$DV_{loop} = \sum_{unique_array_references} dv$$

,

For the array references we can not get the *upperbound, lowerbound*, or *step* from the program, we will assume the array's dv is larger than the size of L2 cache size. We assume that L2 cache has set associativity S and uses a LRU replacement policy. We assume no pathological mapping problems such that if $DV_{loop} < CS$, then all the data stays in the L2 cache. This assumption is reasonable for the L2 cache. If it is not, we could use techniques such as Rivera and Tseng [14] to eliminate conflicts. We also assume that for an array which has $dv < CS$, the array data will stay in the cache as long as possible when $DV_{loop} \leq 2 \times CS$ and the data of the array will be evicted if $DV_{loop} > 2 \times CS$. Table 1 lists these definitions.

4.1 Cost Model for Transpose (Corner Turn)

Impulse uses the *transpose* remapping to map a two-dimensional matrix to its transpose without copying. We show the simplest example in Section 2.2, and below we show a linearized C array access. Let RS be the row size of the two dimension array.

```
for (i=0; i<RS; i++)
    for (j=0; j<RS; j++)
      ...A[j*RS + i]...⇒ Anew[i*RS + j]
```

Cold Cache

Consider the following two cases.

1. If $RS \leq CS/CLS$ and $DV_{loop} \leq 2 \times CS$, or if $RS \leq (CS - (DV - M))/CLS$ then after the cold miss the next access to elements in the same line will be a hit, and we clearly do not need Impulse.

2. Otherwise, after the cold misses the next access to the elements in the same line will also be a miss. $Miss_{org}$ is simply the number of elements in the original array:

$$miss_{org} = RS \times RS$$

and $miss_{imp}$ is reduced by the number of elements per line:

$$miss_{imp} = RS \times RS \times E/CLS.$$

Warm Cache

Suppose array A is initialized in `loop1` just prior to `loop2`, the one we are considering for remapping. We assume the data of A will stay in the cache as much as possible after `loop1` and $DV_{loop1} \leq CS$. The same assumptions are made in the following warm cache cost models for *base-stride* and *indirect vector.*

1. If $DV_{loop2} \leq 2 \times CS$ and $M < CS$, A will stay in the cache during `loop2`, therefore

$$miss_{org} = 0$$

.

2. If $DV_{loop2} > 2 \times CS$ and $M < CS$, A will always be evicted by other data in the loop, then

$$miss_{org} = RS \times RS$$

.

3. If $M \geq CS$, the elements accessed by `loop2` will be evicted out of cache before they are accessed. So this case is the same as the cold cache model above.

Because Impulse flushes remapped elements out of the caches when setting up a new remapping, $miss_{imp}$ will be the same as the cold cache model

$$miss_{imp} = RS \times RS \times E/CLS.$$

4.2 Cost Model for Base-Stride

For a strided array reference `A(i*st)`, Impulse can produce contiguous accesses `Anew(i)` with its *base-stride* remapping, as illustrated in Section 2.2. If we have a multi-dimension array where N is the size of the innermost dimension, we require an invariant stride through the array: $N \bmod st = 0$.

```
for (i=0; i<N/st ; i++)
    ...  A[i*st]...⇒ Anew[i]
```

Cold Cache

Assuming a cold cache, we have the following cases.

1. If $st \times E > CLS$, every element we read from the array in the original loop will be in a different L2 cache line. Thus $miss_{org}$ is the number of elements accessed in the loop.

$$miss_{org} = M/(st \times E)$$

Remapping yields sequential array references, which reduce the misses by the number of elements CLS/E per line.

$$\begin{aligned} miss_{imp} &= \frac{M/(E \times st)}{CLS/E} \\ &= \frac{M}{st \times CLS} \end{aligned}$$

2. If $st \times E \leq CLS$, then the loop accesses more than one element per L2 cache line. $Miss_{org}$ is thus the number of cache lines we need to load into the cache, i.e., the size of this array divided by the size of L2 cache line size.

$$miss_{org} = \frac{M}{CLS}$$

With the Impulse *base-stride* remapping, we have the same result as previous case.

$$miss_{imp} = M/(st \times CLS).$$

Warm Cache

Suppose array A is initialized in `loop1` just prior to `loop2`, the one we are considering for remapping. Here we assume `loop1` initializes all the elements in array A, not only the elements accessed by `loop2`.

1. If $DV_{loop2} \leq 2 \times CS$ and $M < CS$, all of A accessed in `loop2` is in the cache, and

$$miss_{org} = 0$$

2. In all other cases, that is, when $DV_{loop2} > 2 \times CS$ or $M \geq CS$, the total misses of accessing elements in A will be the same as the misses we calculated in the cold cache case because no data loaded in `loop1` will stay in the cache when `loop2` accesses it.

3. Because Impulse flushes remapped elements out of the caches when setting up a new remapping, we have

$$\begin{aligned} miss_{imp} &= array_size/(stride \times cache_line_size) \\ &= M/(st \times CLS) \end{aligned}$$

4.3 Cost Model for Indirect Vector

Impulse provides a remapping of a region of shadow addresses to a data structure through an *indirection vector*. In this case, a shadow address at offset *soffset* in a shadow region is mapped to a pseudo-virtual address $pvaddr + stride \times vector[soffset]$, as illustrated below.

```
for (i=0; i<X; i++)
    ...A[index[i]]...⇒ Anew[i]
```

Let N be the size of index array. Let EI be the size of elements in index array. For simplicity, we suppose each access of $A[index[i]]$ will cause a miss on array A. Because the index array is accessed sequentially, the misses for the indirect array are $miss_{ia} = N/CLS$.

Cold Cache

Considering we will access N/EI elements in array A, we have the number of misses as follows, which includes the misses to A and the index array.

1. If $DV \leq 2 \times CS$ and $M + N \leq CS$, we have only one cold miss for each cache line,

$$miss_{org} = \begin{cases} (M+N)/CLS & : N/EI > M/CLS \\ N/EI + N/CLS & : otherwise \end{cases}$$

2. Otherwise, if $DV > 2 \times CS$ or $M + N > CS$, we will get all misses for each access in array A:

$$miss_{org} = N/EI + N/CLS$$

But if $M < CS$ and $N/EI > M/E$, it is reasonable to assume the access to the same element on A will be a hit, then the misses will be:

$$miss_{org} = M/E + N/CLS$$

For Anew, the number of misses is the same as the number of misses in a sequential accesses of an array which has N/EI elements.

$$miss_{imp} = (N/EI) \times E/CLS$$

Warm Cache

Again, suppose array A is initialized in `loop1` just prior to `loop2`, the one we are considering for remapping.

1. If $DV \leq 2 \times CS$ and $M + N \leq CS$, all the data accessed in `loop2` is in the cache, and

$$miss_{org} = 0$$

2. If $DV \leq 2 \times CS$ and $M + N \geq CS$, there may be data left in the cache that will result in some hits. The probability of a cache miss is a very complicated mathematical formula which is not practical to implement in the compiler. Therefore, we use a simple estimation of the probability P.

$$\begin{aligned} P &= \frac{array_size + index_array_size - cache_size}{array_size + index_array_size} \\ &= \frac{M + N - CS}{M + N} \end{aligned}$$

$$miss_{org} = P \times (N/EI + N/CLS)$$

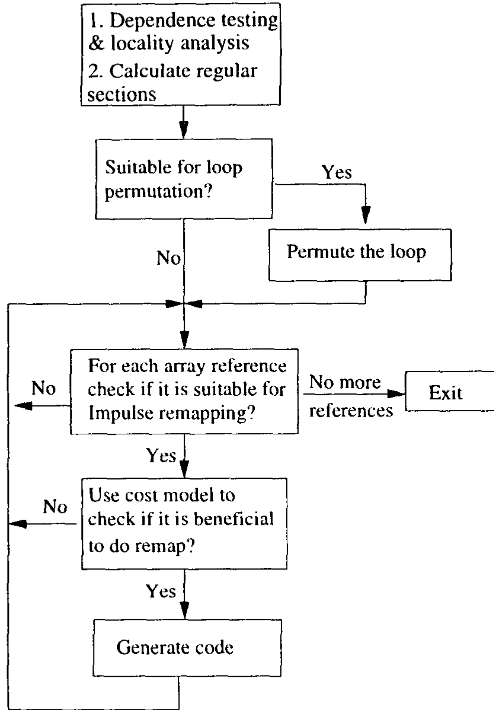

Figure 3: Process of code generation.

3. Otherwise, when $DV > 2 \times CS$, the cache misses will be the number of cold misses.

$$miss_{org} = N/EI + N/CLS$$

Again if $M < CS$ and $N/EI > M/E$, we assume the access to the same element on A will be a hit, then the misses will be:

$$miss_{org} = M/E + N/CLS$$

Since the Impulse remapping function flushes the cache before doing remapping, the misses in the loop with a warm cache will be the same as the misses with the cold cache.

$$miss_{imp} = (N/EI) \times E/CLS$$

5. Code Generation

In Figure 3 we show the process of analysis and code generation in our compiler framework. We describe each steps below.

1. First the compiler performs dependence testing and locality analysis as described in Section 2.3. Regular sections are also calculated for each array reference in the loop.

2. The compiler then performs loop permutation if doing so will improve the locality and permutation is legal.

3. Then compiler checks each array reference to see if it conforms to a pattern without locality that we can remap. For example, if we find a two dimensional array has a transpose access pattern, then it is a candidate for *transpose* remapping.

4. If we find an array reference is a candidate for remapping, the compiler uses the regular section in the cost models we introduced in the previous section to decide if remapping the array reference is beneficial. All our benchmarks have a warm cache and thus we only test the warm cache cost models.

5. If the cost model indicates remapping is beneficial, the compiler inserts the remapping system call at the proper position and replaces the original array references with remapped array references.

6. Experimental Results

In this section, we first introduce our experimental environments and measurements.

6.1 Experimental Environment

We implement our locality analysis, cost models and code generation in Scale, a compiler infrastructure implemented by our research group. Scale accepts Fortran or C input, and translates it into an intermediate representation called Scribble. Scale includes optimizations, such as partial redundancy elimination, value numbering, and sparse constant propagation. Scale produces SPARC assembly code, which we feed to the Impulse simulator, URSim. URSim emulates Impulse remappings and is built on top of RSim [12].

We configure URSim to use an aggressive 4-way issue, out of order execution, 64 instruction window, and a 32K L1 cache with 32 byte cache lines and set associativity of 2. The latency for L1 cache access is 2 cycles. We use a 128K L2 cache with 128 byte cache lines, set associativity of 2, and an 8 cycle latency. The latency for memory access is between 48 and 200 cycles, depending on the outstanding misses which URSim models. The shadow memory access latency is about 100 cycles more than a normal memory access. We also examine some of our benchmarks under a 5 year hardware projection, where the L1 cache latency is 3 cycles, L2 cache latency is 20 cycles, and memory latency is 200-500 cycles. We use a processor clock-rate of 5GHz versus bus/memory clock-rate of 300MHz which results in the ratio of 16 : 1, for the projected hardware in 5 years compared with the ratio of 3:1 in our current model. These projections of the cache and memory latencies in 5 years from now come from the work of Agarwal et al. [1].

6.2 Case Studies

We use five benchmark kernels to examine the performance of Impulse remapping. The following table describes each of our benchmark kernels. We validate our cost model on all of the benchmarks, and we compare and combine it with loop permutation on $m \times m$, and *vpenta*. The majority of our input sizes are close to the crossover point between when we want to use Impulse in order to test our cost models. Carter et al. [2, 3] show larger improvements for larger data sets.

Benchmark	Functionality	Arrays	Nests	Depth
base-stride	test	1	2	1
$m \times m$	matrix multiply	1	3	3
transitive	all pairs shortest path	4	5	2
matrix	sparse matrix multiply	1	20	2
vpenta	invert 3 pentadiagonals	41	7	2

Base-Stride

The *base-stride* kernel consists of the following, plus an initialization loop.

```
for (i=0; i<N; i++)
  s=s+A[i*STRIDE];
```

The code generated by Scale after inserting the Impulse *base-stride* remapping function is as follows.

```
bs_arg.newaddr=&A_New0;
bs_arg.vaddr = A;
bs_arg.count = ARRAYSIZE/STRIDE;
bs_arg.stride = STRIDE*sizeof(int);
bs_arg.offset = 0;
...
ams_mapshadow(BASESTRIDE, bs_arg);
for (i=0; i<N; i++)
    s=s+A_New0[i]
```

We tested a wide range of array sizes and strides, and include representative results in Table 2. In all the tables, the "Yes" in the "Cost Model" column means our cost model determine that it is beneficial to perform Impulse remapping.

The results in Table 2 indicate that remapping has the effect of increasing the total number of accesses (L1 hits + L2 hits + Misses) because of the additional accesses to shadow memory. However, the Impulse remapping improves cache locality and therefore reduces total execution time when the array size is large. Our cost model correctly determined whether Impulse remapping would be beneficial in 68 of the 72 configurations (94.4%) that we tested, which ranged from array sizes from 32K to 1024K, with strides of 32, 64, and 133.

Matrix Multiplication

For classic matrix multiplication, the original code is as follows.

```
for (i=0; i<Vertices; i++)
     for (j=0; j<Vertices; j++)
        for (k=0; k<Vertices; k++)
            c[i][j] += a[i][k]*b[k][j];
```

In this version, the inner two loops are out of order. Scale generates the following code for Impulse (without performing loop permutation) using the transpose remapping.

```
tp_arg.newaddr=&bNew;
tp_arg.vaddr = b;
tp_arg.elemsize = sizeof(int);
tp_arg.rownum = Vertices;
tp_arg.rowsize = Vertices*sizeof(int);
...
ams_mapshadow(TRANSPOSE,&tp_arg);
for (i=0; i<Vertices; i++)
     for (j=0; j<Vertices; j++)
        for (k=0; k<Vertices; k++)
            c[i][j]+=a[i][k]*bNew[j][k];
```

The results in Table 3 demonstrate that the Impulse remapping will reduce the total number of misses, and yield a speedup when the matrix is bigger than 130×130. Loop permutation gives an even better performance improvement because it does not have the overheads of Impulse. Note that our cost model has correctly predicted that no speedup will be gained for the smaller matrix sizes, e.g., 32 and 64. The cost model fails to choose Impulse for the matrix size of 130 where it has an 11% speedup. However, it resumes accurate predictions from matrix sizes of 150 upwards.

Transitive

Transitive is a Darpa Data Intensive System (DIS) Stress benchmark. The following array reference pattern, to which we can apply the Impulse *transpose* remapping, appears in three loops within the benchmark.

```
for (i=0; i<VERTICES; i++)
     for (j=0; j<VERTICES; j++)
        dout[j*VERTICES + i] = new1;
```

After remapping we have:

```
tp_arg.newaddr=&doutNew;
tp_arg.vaddr = dout;
tp_arg.elemsize = sizeof(int);
tp_arg.rownum = VERTICES;
tp_arg.rowsize = VERTICES*sizeof(int);
...
ams_mapshadow(TRANSPOSE,&tp_arg);
for (i=0; i<VERTICES; i++)
     for (j=0; j<VERTICES; j++)
        doutNew[i*VERTICES + j] = new1;
```

Tables 4 and 6.2 show simulation results for the Transitive benchmark with current, and 5 year projected, hardware parameters respectively. The results indicate a large performance improvement when the array is sufficiently large. Note that our cost model has correctly predicted that no speedup will be gained for the smaller array sizes. The speedup due to Impulse is reduced when the number of *Vertices* is increased from 411 to 697, which is counter intuitive,. This reduction is because there are significantly more TLB misses (71 compared to 366,748) with the larger data set size. We could use Impulse's superpages to eliminate these misses, but that improvement is beyond the scope of this paper.

Matrix

Matrix is another DIS stress benchmark. The following array pattern, to which we can apply Impulse remapping, appears in three loops within the benchmark:

```
static double vectorX[DIM];
static int col_ind[DIM+NUMBERNONZERO];
for (ll=tmp_rs; ll<tmp_re; ll++)
     ...  = ...vectorX[col_ind[ll]]...;
```

The code after remappings is:

```
iv_arg.newaddr=&vectorX_New1;
iv_arg.vaddr=vectorX ;
iv_arg.count=DIM;
iv_arg.objsize=sizeof(double);
iv_arg.iv_vaddr=col_ind;
iv_arg.iv_objcount=DIM+NUMBERNONZERO;
iv_arg.iv_objsize=sizeof(int);
...
ams_mapshadow(VINDIRECT,&iv_arg);
for (ll=tmp_rs; ll<tmp_re; ll++)
     ...  = ...vectorX[vectorX_New1]...;
```

Array Size(K)	Stride	Original L1 hits	Original L2 hits	Original Misses	Original Cycles	Impulse L1 hits	Impulse L2 hits	Impulse Misses	Impulse Cycles	Speedup	Cost Model
32	32	38233	4225	1247	179293	42025	3357	1343	201012	0.89	No
32	64	38227	3715	1244	176404	41055	3309	1328	197427	0.89	No
32	133	38266	3414	1240	175264	40816	3284	1318	235122	0.75	No
256	32	239576	25270	16012	1551960	248555	25597	8590	1248312	1.24	Yes
256	64	241458	24761	12604	1483098	250587	25212	8715	1254803	1.18	Yes
256	133	241664	24764	10453	1387045	249342	25009	8618	1309383	1.06	Yes
256	256	241664	24763	9504	1282063	245453	24905	8587	1236120	1.03	No
1024	32	950997	99388	65400	6178389	982946	102054	34146	4824273	1.28	Yes
1024	64	951004	99131	49254	5318694	968155	100520	33629	4731753	1.12	Yes
1024	133	939457	98712	41274	5312265	964224	99511	33654	4972022	1.07	Yes
1024	256	939463	98705	37457	5038962	948741	99091	33525	4829841	1.04	No

Table 2: Simulation results: Base-stride with strides and array sizes

Vertices	Original Misses	Original Cycles	Permutation Misses	Permutation Cycles	Permutation Speedup	Impulse Misses	Impulse Cycles	Impulse Speedup	Cost model
32	143	462821	142	464984	1.00	188	504780	0.92	No
64	431	3592985	430	3599255	1.00	573	3832450	0.94	No
130	122667	33870493	7963	29283061	1.16	13354	30612999	1.11	No
256	13807774	889336644	1529629	349294302	2.55	748852	498383702	1.78	Yes

Table 3: Simulation results: matrix multiplication with poor inner loop locality, loop permutation, and Impulse remapping.

Vertices	Original L1 hits	Original L2 hits	Original Misses	Original Cycles	Impulse L1 hits	Impulse L2 hits	Impulse Misses	Impulse Cycles	Speedup	Cost Model
113	852245	60048	1317	4851675	1414605	31459	8499	6840206	0.71	No
277	3911864	658159	403768	55205286	8110085	201972	52511	31648797	1.74	Yes
411	7147463	830385	2566442	237111690	17862096	469483	130404	71746071	3.30	Yes
697	20117449	835468	11694075	914051076	43530495	1112433	3824138	396397335	2.31	Yes

Table 4: Simulation results: Transitive

Vertices	Original L1 hits	Original L2 hits	Original Misses	Original Cycles	Impulse L1 hits	Impulse L2 hits	Impulse Misses	Impulse Cycles	Speedup	Cost Model
113	845648	59350	1412	4967264	1407949	30669	8810	17070311	0.29	No
277	3912415	656855	404744	117997408	8111553	200937	52509	101460332	1.16	Yes
411	7162959	828615	2567978	611860336	17878216	468469	130918	243588112	2.51	Yes
697	20132067	833080	11746193	2404205145	43575551	1109738	3844610	1454859792	1.65	Yes

Table 5: Simulation results: Transitive with projected hardware for 5 years out

Dim	#Non0	Original	Impulse	Speedup	Cost Model
3912	8660	7195584	7357778	0.98	No
9400	50000	26140055	25612026	1.02	No
30730	1400000	733583510	388726047	1.88	Yes
75000	3000000	2044178149	800149886	2.56	Yes

Table 6: Simulation results: Matrix

Dim	#Non0	Original	Impulse	Speedup	Cost Model
3912	8660	11217762	16398099	0.68	No
9400	50000	46613965	64162977	0.72	No
30730	1400000	1956530541	1152998801	1.70	Yes
75000	3000000	5356346911	2488114202	2.15	Yes

Table 7: Simulation results: Matrix with a hardware projection for 5 years out

Tables 6 and 6.2 show simulation results for the Matrix benchmark with current and projected hardware parameters for 5 years out respectively. The results indicate excellent performance improvement when the matrix is sufficiently large.

Vpenta

The *vpenta* benchmark includes two categories of array references that are amenable to Impulse remapping: *base-stride* and *transpose*. There are 90 candidate sites for Impulse remapping. The Impulse architecture limits us to at most 8 remapping shadow descriptors. We use our cost model to select the most productive candidates for remapping. We also combine Impulse remapping with loop permutation to get further performance improvements.

The *base-stride* pattern that appears in *vpenta* is as follows.

```
double x[SIZE][SIZE];
for (k= 0; k< SIZE; k++) {
    rld = x[k][1];
    ....
}
```

Here the stride is the size of the second dimension of the array x. After inserting the Impulse remapping function, we have the loop:

```
double x[SIZE][SIZE];
bs_arg.newaddr=&x_New0;
bs_arg.vaddr = x;
bs_arg.count = SIZE;
bs_arg.objsize = sizeof(double);
bs_arg.stride = SIZE*sizeof(double);
bs_arg.offset = 1*sizeof(double);
......
ams_mapshadow(BASESTRIDE,&bs_arg);
for (k= 0; k< SIZE; k++) {
    rld = x_New0[k];
    ....
}
```

The *transpose* pattern that appears in *vpenta* is as follows.

```
double a[SIZE][SIZE];
for (j=0; j<SIZE; j++)
     for (k=0; k<SIZE; k++)
            rld2 = a[k][j];
```

After transformation, we have the code below.

```
tp_arg.newaddr=&a_New;
tp_arg.vaddr = a;
tp_arg.elemsize = sizeof(double);
tp_arg.rownum = SIZE;
tp_arg.rowsize = SIZE*sizeof(double);
....
ams_mapshadow(TRANSPOSE,&tp_arg);
for (j=0; j<SIZE; j++)
     for (k=0; k<SIZE; k++)
            rld2 = a_New[j][k];
```

For the above loop, we can also do loop permutation to improve the locality in addition to Impulse remapping.

In Tables 8 and 9, we show the effect of Impulse remapping for current and projected hardware for 5 years out respectively. These tables also show the effect of loop permutation with and without Impulse remapping. *BS* and *TP* refer to *base-stride* and *transpose* cost models respectively. If it is not applicable to apply a certain type of remapping, there is "N/A" in the column. In the following table, the column "Row Size" means the row size of the input matrix.

Because loop permutation changes the form of array references in the permuted loops, it eliminates candidates for the *transpose* remapping in this case. However, we can still apply the *base-stride* remapping for some array references in the program. By combining loop permutation and Impulse remapping, we achieve better performance than when only one of them is used. The column "Impulse+LP" in Table 8 gives the result of performing both Impulse remapping and loop permutation.

We see from the above tables that Impulse can not only improve the performance of the original programs, but can cooperate with loop permutation to further increase the speedup. For almost all benchmarks, configurations, and data set sizes, our cost model correctly predicts whether or not it is beneficial to do Impulse remapping.

7. Conclusions and Future Work

This paper develops compiler cost models to drive the Impulse memory controller. We implement and measure the benefit of Impulse remapping using the Scale Compiler framework and URSim. We demonstrate that we are able to use Impulse effectively to attain excellent performance improvements without user intervention on five benchmark kernels. We plan to extend this work to complete applications and integrate it more fully with loop transformations to improve locality, including data copying and other data reorganization transformations.

8. Acknowledgment

We wish to thank Lixin Zhang and John Carter from the University of Utah for providing technical support on Impulse and URSim. We especially thank Eliot Moss and Steve Blackburn for their help and suggestion.

This work is supported by Darpa grant 5-21425, NSF ITR grant CCR-0085792, and NSF grant ACI-9982028. Any opinions, findings and conclusions or recommendations expressed in this material are the authors and do not necessarily reflect those of the sponsors.

9. REFERENCES

[1] V. Agarwal, M. Hrishikesh, S. W. Keckler, and D. Burger. Clock rate versus ipc: The end of the road for conventional microarchitectures. In *Proceedings of the 27th International Symposium on Computer Architecture*, pages 248–259, June 2000.

[2] J. Carter, W. Hsieh, L. Stoller, M. Swanson, L. Zhang, E. Brunvand, A. Davis, C.-C. Kuo, R. Kuramkote, M. Parker, L. Schaelicke, and T. Tateyama. Impulse: Building a smarter memory controller. In *Fifth International Symposium on High Performance Computer Architecture*, Orlando, FL, Jan. 1999.

[3] J. Carter, W. Hsieh, L. Stoller, M. Swanson, L. Zhang, and S. McKee. Impulse: Memory system support for scientific applications. *The Journal of Scientific Programming*, 7(3-4):195–209, 1999.

[4] B. Chandramouli, J. B. Carter, W. Hsieh, and S. McKee. Cost-model driven integration of restructuring optimizations. In *The 2001 International Conference on Parallel Architectures and Compilation Techniques*, Barcelona, Spain, Sept. 2001.

[5] S. Ghosh, M. Martonosi, and S. Malik. Cache miss equations: An analytical representation of cache misses2. In *Proceedings of the*

Row Size	Original	Impulse	Speedup	Cost Model BS	Cost Model TP	Loop Permutation	Speedup	Impulse+LP	Speedup	Cost Model BS	Cost Model TP
129	25235679	20655571	1.22	No	Yes	5116014	4.93	5105157	4.94	No	N/A
340	178724430	132720091	1.35	Yes	Yes	31671588	5.64	31238970	5.74	Yes	N/A
649	668200554	645233502	1.04	Yes	Yes	109186137	6.12	108363505	6.17	Yes	N/A

Table 8: Simulation results: Vpenta with input of Row Size*Row Size matrix

Row Size	Original	Impulse	Speedup	Cost Model BS	Cost Model TP	Loop Permutation	Speedup	Impulse+LP	Speedup	Cost Model BS	Cost Model TP
129	64624209	56424652	1.15	No	Yes	8792529	7.35	8794017	7.35	No	N/A
340	436522193	358632460	1.22	Yes	Yes	49275729	8.86	47660950	9.16	Yes	N/A
649	1632989889	1567619918	1.04	Yes	Yes	167842129	9.73	162752716	10.03	Yes	N/A

Table 9: Simulation results: Vpenta with input of Row Size*Row Size matrix and a hardware projection of 5 years out

1997 ACM International Conference on Supercomputing, pages 317–324, Vienna, Austria, July 1997.

[6] S. Ghosh, M. Martonosi, and S. Malik. Precise miss analysis for program transformations with caches of arbitrary associativity. In *Proceedings of the Eighth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 228–239, San Jose, CA, Oct. 1998.

[7] M. Kandemir, A. Choudhary, J. Ramanujam, and P. Banerjee. A matix-based approach to the global locality optimization problem. In *The 1998 International Conference on Parallel Architectures and Compilation Techniques*, Paris, France, Oct. 1998.

[8] C. E. Kozyrakis et al. Scalable processors in the billion-transistor era: IRAM. *IEEE Computer*, pages 75–78, Sept. 1997.

[9] J. R. M. Kandemir and A. Choudhary. Improving cache locality by a combination of loop and data transformations. *IEEE Transactions on Computers*, 48(2):609–623, Feb. 1999.

[10] K. S. McKinley, S. Carr, and C. Tseng. Improving data locality with loop transformations. *ACM Transactions on Programming Languages and Systems*, 18(4):424–453, July 1996.

[11] M. Oskin, F. T. Chong, and T. Sherwood. Active pages: A model of computation for intelligent memory. In *Proceedings of the 25th International Symposium on Computer Architecture*, pages 192–203, Barcelona, Spain, June 27–July 1, 1998.

[12] V. S. Pai, P. Ranganathan, and S. V. Adve. RSIM reference manual (version 1.0). Technical Report Technical Report 9705, Rice University, Dept. of Electrical and Computer Engineering, Aug. 1997.

[13] W. Pugh. A practical algorithm for exact array dependence analysis. *Communications of the ACM*, 35(8):102–114, Aug. 1992.

[14] G. Rivera and C. Tseng. Data transformations for eliminating conflict misses. In *Proceedings of the SIGPLAN '98 Conference on Programming Language Design and Implementation*, pages 38–49, Montreal, June 1998.

[15] M. Swanson, L. Stoller, and J. Carter. Increasing TLB reach using superpages backed by shadow memory. In *Proceedings of the 25th International Symposium on Computer Architecture*, pages 204–213, Barcelona, Spain, June 1998.

[16] O. Temam, E. Granston, and W. Jalby. To copy or not to copy: A compile-time technique for assessing when data copying should be used to eliminate cache conflicts. In *Proceedings of Supercomputing '93*, pages 410–419, Portland, OR, Nov. 1993.

[17] X. Zhang, A. Dasdan, M. Schulz, R. K. Gupta, and A. A. Chien. Architectural adaptation for application-specific locality optimizations. In *Proceedings of the 1997 IEEE International Conference on Computer Design*, 1997.