Caches with Compositional Performance[⋆]

Henk Muller, Dan Page, James Irwin, and David May

Department of Computer Science, University of Bristol, Bristol BS8 1UB, UK
`http://www.cs.bris.ac.uk/`

Abstract. One of the challenges in designing systems is adopting a design method with compositional properties. *Compositional functionality* guarantees that two components that each perform a task can be integrated without affecting the semantics of either task. *Compositional performance* means that two components can be integrated so that the timing of neither components changes. In this paper we describe the hardware and software needed in order to build cache memories that have those compositional properties. This *partitioned cache* allows the system designer to design individual components of an application program in the knowledge that cache performance is fully deterministic; ie. integrating these components will not affect the performance of any component.

1 Introduction

When designing systems one of the main challenges is adopting a design approach that is compositional. Compositional design means that one can design components of a system independently and compose them without affecting functionality or performance. This applies to the design of parts of an application program, an item of hardware specified in VHDL, or some software library module.

In this paper we are going to focus on one component found in many systems: the cache memory. Functionally, a cache is mostly transparent to the programmer. From a performance viewpoint, the cache may have a significant impact. On average, introducing a cache will improve the performance, but very little can be predicted about cache performance of specific applications. Even if we know the exact performance characteristics of two software modules when executing on a specific cache, it is still very difficult to predict the performance of the combined modules.

Many solutions have been proposed to improve cache performance, including set-associative caches [1] and victim caches [2], but all work on statistical properties which are of little value when designing a real-time system. We present a solution where the cache is exposed to the compiler and has some extra hardware attached that allows data streams to be segregated where necessary. The compiler analyses the application program and automatically generates code with composable performance properties.

[⋆] This work supported by EPSRC grant GR/L78970.

In Section 2 we will first elaborate on compositional properties, before quantifying conventional cache performance and its lack of composability. As a solution we suggest the use of partitioned caches as described in Sections 3 and 4. Like conventional caches they are transparent to the programmer in terms of functionality but, in addition, have compositional performance characteristics. This means that system designers can produce software components in the knowledge that the cache performance is predictable. We quantify these performance results in Sections 5 and 6.

2 Compositional Design Strategies

Composability is an issue in designing any type of system; whether it is special purpose hardware, general purpose processors, or software. This issue must be addressed by the tools and software library modules that are used for design and implementation, for example, the hardware description language used to design hardware, and the programming language used for implementing programs. Composability spans the hardware software divide in that composability of software may be restricted by the underlying hardware.

As an introduction to composability we will first discuss two programming languages in Sections 2.1 and 2.2. Compositional properties of programming languages are well studied, since a major issue in designing a programming language is to allow programmers to design, implement, and test modules independently, before integrating them in a final phase.

After that, we discuss cache memories which present some major issues in producing composable systems. A cache is functionally transparent, and improves the performance of a program on average. However, even though the performance improves on average, it is very difficult to predict what exactly happens to the performance of two components that both use the same cache. Subsequent sections present a novel caching strategy which addresses this issue.

2.1 Example 1: Haskell

Haskell [3] is a purely functional programming language, defined in the mid-1990's. Purely functional programming requires the programmer to write a function that given some input produces some output. A fundamental requirement of such a function is that it can only operate on the input passed to the function; there is no global state on which functions can operate.

As a direct consequence of this purely functional approach, the language has very strong compositional properties. Two functions that are defined can be executed in any order, including in parallel, and they will always produce the same result. Once a module has been defined to implement, for example, a hash-table, this module can be employed in any other place and will always implement a hash-table. This can be achieved in other languages, such as C, C++ and Java, but Haskell *enforces* compositional behaviour; one *cannot* implement side effects.

Haskell is strong at compositional functionality but it is not very strong at compositional performance. One example of this fact is as a result of the Haskell garbage collection system. Garbage collection is an essential part of the language but the user cannot determine at compile-time when garbage collection will take place. Even if the performance characteristics of two functions are known exactly, the performance may therefore change when they are composed, because the garbage collector may require a significant execution time in one function since the other function has produced a lot of garbage.

2.2 Example 2: Occam

Occam [4] has a different approach to guarantee composability. Side effects are an essential part of Occam, but at compile time a *disjointness* check is performed. This checks which variables are used in a module and that they are not used by any other module that executes concurrently. The disjointness check guarantees that any program that is accepted by the compiler will not contain any state modifications that are unsafe.

The Occam approach gives a degree of composability in that any concurrent activity that is accepted by the compiler will run deterministically since there are no race conditions on the global state. Of course, one can write an Occam program in which a module relies on side effects, and hence different compositions of function calls will result in different answers.

Occam has a very strong compositional performance model. The language is designed so that all memory allocation, including any stacks, is performed at compile-time. Hence, the order in which functions are executed is irrelevant to the performance. When functions are executed concurrently, the performance will rely on what the underlying hardware can achieve. Having said that, there is one element of the hardware where the performance cannot easily be predicted: the cache memory. The performance of a function will always depend on what is present in the cache, and on which other functions are being executed.

2.3 Compositional Caching

As stated earlier, composability is not strictly a property of either hardware or software. As an example, suppose that we have two modules, one producing elements and one consuming elements, which are executed a the for-loop sketched in Figure 1. Ideally the performance of this loop would be equivalent to the performance of the producer being called a 1000 times, the consumer being called a 1000 times plus the overhead of a for-loop. Unfortunately, this is not necessarily the case depending on the cache architecture of the host system.

Suppose that the system employs a direct mapped data cache. If the function produce is called 1000 times in a row, each n^{th} iteration would cause a cache miss, where n is the line size of the cache. Similarly, the consumer would have one cache miss every n iterations.

When the two functions are integrated and called subsequently from one for-loop, the arrays `y` and `z` may map on the same line in the cache (which is very

```
int y[ 16384 ];
int z[ 16384 ];

int produce( int index )
{
  return y[ index ];
}

void consume( int index, int value )
{
  z[ index ] = value;
}

void main()
{
  int i;
  int x;

  for( i = 0; i < 1000; i++ )
  {
    x = produce( i );
    consume( i, x );
  }
}
```

Fig. 1. Example program with poor composability due to cache performance.

likely given that the array y is exactly a power of 2 in size), and we will end up with a cache miss on every entrance to both the producer and the consumer. One cache miss to load the line covering the y array, which will displace the line of z, which is to be reloaded when consumer is called.

Although this may seem like a contrived example there are actually many programs where this interference occurs. Powers of two are often used as array sizes, and even within functions such as matrix multiplications, various data structures can interfere in the cache. An interesting example is a GIF decoder which uses two arrays, each sized a power of 2. Figure 2 plots the execution time of a GIF decoder. We have manually inserted some padding between those arrays; and depending on the amount of padding we can see the performance vary. On this particular machine, a Silicon Graphics infinite reality engine, the effects are relatively minor, but on other architectures we have observed performance variations of up to 30%.

Unpredictable behaviour is exacerbated when we run programs on a multi-threaded machine, modelled on the HEP [5] and *T [6] multi-threaded architectures, as is shown in Figure 3. Here we show the hit-ratios of a cache running 1, 2, 4, and 8 GIF decoders in parallel on a multi-threaded machine. One can

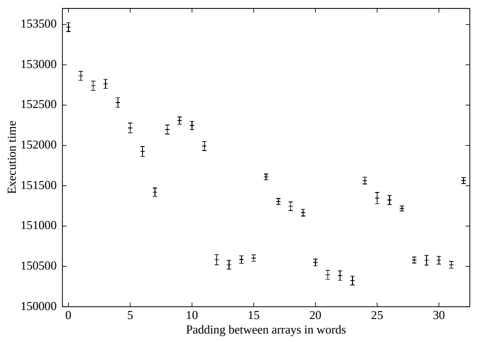
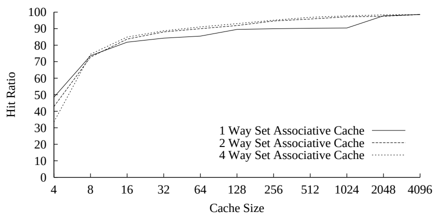
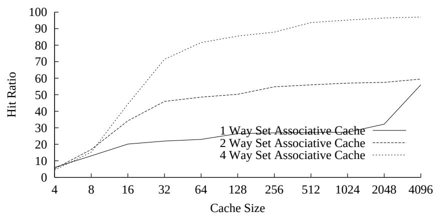


Fig. 2. Performance of a GIF decoder, along the horizontal axis we have inserted a varying amount of padding. The vertical error bars denote the standard deviation.



(a) 1 GIF Decoder thread(s).



(b) 4 GIF Decoders thread(s).

Fig. 3. Multi-threaded performance of multiple GIF decoder threads.

observe that depending on the number of threads and the associativity of the caches performance deteriorates; we will come back to this in Section 6.

3 Partitioned Cache Hardware

In order to address some of the problems with designing composable cache systems we propose the use of a partitioned cache. This novel design consists of a direct-mapped style cache that can be dynamically partitioned into protected regions by the use of specialised cache management instructions. By modifying the load/store mechanism and tagging each memory access with a partition identifier, each access is routed through a partition dedicated to dealing with it.

Unlike conventional caches, the partitioned cache is visible to software running on the host processor. This allows the compiler and operating system to allocate partitions of the cache to specific data objects and streams of instructions so as to control persistence and eliminate interference. This has the knock on effect of improving the predictability and determinism of the cache.

The idea of exposing the cache to the programmer has been proposed before [7], and has been worked on by Juan et. al. [8], Kirk [9] and Mueller [10]. The

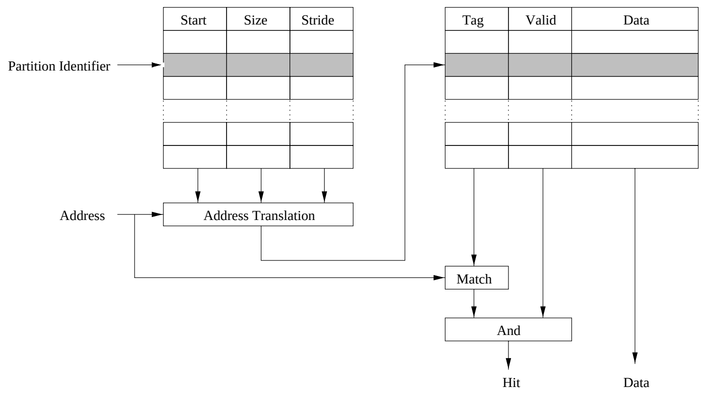


Fig. 4. A flow-diagram of a partitioned cache.

most obvious deficiency of all these systems is the inflexible manner in which cache partitions are allocated and the blinkered approach in using entirely hardware or software based designs. These techniques lack interaction from the one component, the compiler, that enables them to revolt against conventional, average case optimised cache design. By employing the compiler to analyse the source program and configure the cache so as to achieve the best performance possible, a partitioned cache benefits from a combined hardware/software approach. This sidesteps the problems of being tied to one paradigm in particular, and allows the cache to reap the benefits of being specialised to any given application program.

3.1 Architecture

Our partitioned cache [11] is a direct-mapped like block of cache memory which uses a *partition descriptor table* to hash incoming memory address traffic into cache line addresses. Figure 4 shows a flow-diagram of the operation of a partitioned cache. Information is extracted from the partition descriptor table by using an index, the *partition identifier*, provided with each memory access. This information is used to hash the memory address so that it maps onto a subsection, or *partition*, of the cache.

In order to maximise the density of useful data stored, the partitioned cache allows each cache line to be filled with data from non-contiguous, regularly strided memory locations. The stride, or distance between each address in a cache line, is a property of the partition owning the cache line and is used in the translation of memory accesses to map memory locations into the cache. For example, as demonstrated in Figure 5, a conventional cache with 4 words in each cache line will need 4 lines to store the addresses 0, 4, 8, 12 which could be generated by an array reference such as $a[4 * i]$, without conflict. A partitioned cache can store these accesses in one line provided the partition is configured

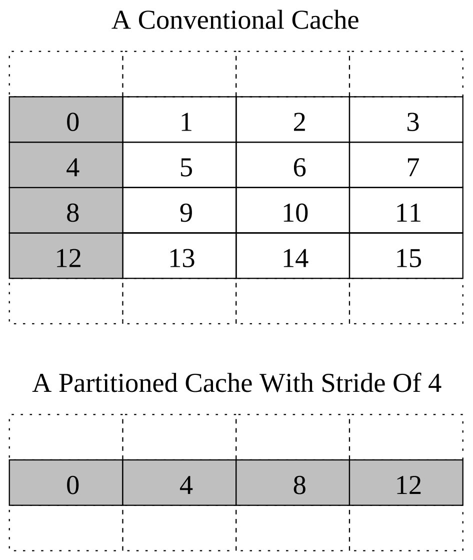


Fig. 5. Strided cache lines in a partitioned cache.

with a stride of 4. In order for this scheme to be successful, we assume that lower levels of the memory hierarchy [12–14] can efficiently service such requests.

The presence of strided cache lines requires that the address translation mechanism which converts memory addresses into cache addresses is slightly more complex than usual. The purpose of the address translation mechanism, or address hashing function, is to distribute memory addresses in the cache so as to use all of the available cache space. In the context of a partitioned cache, the hash function needs to restrict the mapping so that memory addresses can only appear in the cache space owned by the target partition, and to populate the cache partition as fully as possible with incoming references.

Considering Figure 4 where p_{start}, p_{stride} and p_{size} are the start line, stride and size of a partition p, we can express our hashing function as follows:

$$\begin{aligned} p_{offset} &= ((addr/nwords) \gg lsb(p_{stride})) \bmod p_{size} \\ p_{line} &= p_{start} + p_{offset} \end{aligned}$$

That is, the cache line p_{line} for an address $addr$ is given by the sum of the starting line of the partition, p_{start}, and an offset p_{offset}. The first stage in computing the offset is given by the incoming address, $addr$, divided by the number of words per cache line, $nwords$, which must be a power of two. This is shifted right by the position of the least significant 1 bit of the partitions stride, p_{stride}, and finally restrict the cache line to the available space by taking the remainder after division by the partition size, p_{size}.

To see how this scheme will distribute the data, consider a stride of twenty which has a binary representation of 10100. After dividing the incoming address by the number of words per line we shift it right by two, the least significant non-zero bit of the stride being the bit two. If we set size of the partition to eight lines and we assume a standard of four words per cache line, the first ten accesses, to addresses 0, 20, 40, 60, 80, 100, 120, 140, 160, 180, will be placed in lines $0, 1, 2, 3, 5, 6, 7, 0, 2, 3$ of the partition. This technique ensures that as long as the partition descriptor table is configured correctly, accesses to memory using different partition identifiers will always map into disjoint partitions and will utilise all cache lines in the partition. However, the scheme may not use the lines in an optimal manner, with some being used more often than others depending on the size, stride and line length of the partition.

3.2 Partition Management

Before we consider how memory is accessed though cache partitions, it is important to understand how the partition descriptor table is managed so that the cache can operate correctly. The management of the cache is performed dynamically, under control of the running program or operating system. This is done by using additional machine level instructions which alter the configuration of the cache. Table 1 details the extra instructions required to manage the table of partition descriptors. An alternative to this implementation may be to offer the partition descriptor table as a memory mapped block of address space so that the programmer can access it using standard load and store instructions.

In our instruction based management scheme, the *CREPAR* instruction adds an entry to the partition descriptor table, creating a new partition. The partition descriptor is configured so that the partition is tagged with an identifier of *id* and has a stride of *stride*. Furthermore, the management mechanism ensures that the partition starts on a cache line such that the partition is *size* cache lines in size. Creation of a partition which causes partitions with duplicate identifiers is considered an error.

The *DELPAR* instruction acts to delete the partition with an identifier of *id* from the partition descriptor table. In this study, the deletion of partitions under program control, thus creating truly dynamic configuration of the cache, is not discussed. Specifically, this is due to the complexity introduced by such usage which may include, for example, fragmentation of the partitionable cache space.

Partitions may be invalidated, that is their contents marked as invalid, so that the partition is effectively emptied or flushed, using an *INVPAR* instruction. Finally, the *SETIPAR* instruction is used to control the value of the partition through which instructions are loaded by the fetch/execute engine.

3.3 Partition Access

In order for accesses to memory to be routed through the correct partition in the cache, the memory access mechanism must be augmented to provide a partition

Instruction	*Meaning*
$CREPAR\ id, size, stride$	Add partition.
$DELPAR\ id$	Remove partition.
$INVPAR\ id$	Invalidate partition.
$SETIPAR\ id$	Set instruction partition.

Table 1. Partitioned cache management instructions.

Instruction	*Meaning*
$LOAD\ dst, add, id$	$R[dst] \leftarrow M[add, id]$
$LOADIDX\ dst, add, off, id$	$R[dst] \leftarrow M[add + off, id]$
$STORE\ src, add, id$	$M[add, id] \leftarrow R[src]$
$STOREIDX\ src, add, off, id$	$M[add + off, id] \leftarrow R[src]$

Table 2. Partitioned cache access instructions.

identifier. The simplest way to pass the partition identifier is by using an extra operand in each instruction. This strategy requires that the instruction set be altered and may reduce code density by increasing the number of bits required to encode each instruction. Table 2 shows the modified memory access instructions where $R[n]$ denotes an access to general purpose register n, $M[add, id]$ denotes an access to memory, through the cache, at address add using partition id.

There are other techniques for passing the partition to the cache (such as stealing memory address bits, or maintaining a "current partition register"), but because our research is mainly based in the techniques for using a partitioned cache effectively and not issues of implementation, we opted for the simplest strategy. It is important to note that the partition identifiers are effectively constants and hence it should be easy to pipeline accesses to the partition descriptor table such that the performance overhead of doing so is minimal.

Finally, where an invalid or reserved partition identifier is used, the cache foregoes the standard behaviour and passes the access straight through to the next level of the memory hierarchy. This is used to achieve *cache bypass* where the cache state is unaltered by the memory access and is useful in this context for avoiding situations where a misconfigured cache may otherwise give undefined results.

4 Using A Partitioned Cache

Automation of partition management and configuration requires the application of a number of simple algorithms in suitable compiler modules. These algorithms act to disperse a complex stream of memory accesses into a number of more simple streams. We separate memory accesses to data objects into vector, scalar and stack classes. A number of partitions are allocated to each of these classes which are configured to suit the access properties of each class.

- Vector data partitions are perhaps the most interesting data access segregation to consider. Each vector within the program normally has at least one partition dedicated to it. Deciding the parameters for a partition, and how many there should be for each vector is computed from the set of references made in the application source code.
- Scalar data partitions cache accesses to ordinary, non-vector variables. The pattern of access to scalars is determined by the register spilling and loading of scalar data generated by the compiler. Normally, the compiler can remove most scalar memory accesses using traditional optimisation techniques. The accesses that remain can be routed through a single scalar partition whose size is related to the number of scalars in each of the procedures.
- The stack partition is sized to manage stack frames effectively. In a similar way to the scalar partition, we see highly localised access patterns. The algorithm takes into consideration the stack frame size of each procedure within the program, and the existence of any recursive procedure calls. The result is a stack partition large enough to contain the most recent n-levels of recursion in protected cache space.

We first establish properties of the memory accesses, whereupon the compiler partitions the program over the cache. For a full description of the algorithms used see [15, 16]

4.1 References

The compiler takes a source program as input which declares a number of scalar and vector variables and uses a number of statements and expressions to perform operations on those variables. Figure 6 shows a kernel from the Livermore Loop Fortran Kernels [17] suite of benchmark programs, which computes an Incomplete Cholesky Conjugate Gradient or *ICCG*. This program typifies the kinds of operations performed by an application program on declared data objects.

The first section of the program declares two vector variables, named v and x, and five scalar variables, named *ipnt*, *ipntp*, *ii*, i and k. The variables are then used in a number of *references*, which mark instances of data access to the variable, in a sequence of statements or expressions. In the example program there are two references to the variable v, four to x, two to *ipnt*, six to *ipntp*, five to *ii*, four to i and nine to k.

We can compute a set of references for all variables in the program, even if the set turns out to be empty. From this information we can determine further properties, called the *stride*, the *group* and the *window*, which are associated with variable references and will guide the partitioning process. Given that the $\mapsto$ operation is used to denote a mapping between a symbol and a set of references to that symbol, the reference sets for our *ICCG* example in Figure 6 look like:

```
void iccg()
{
  double v[ 1024 ];
  double x[ 1024 ];
  double ipnt;
  double ipntp;
  double ii;
  int i;
  int k;

  ii_0    = 1024;
  ipntp_0 = 0;
  do
  {
    ipnt_0  = ipntp_1;
    ipntp_2 = ipntp_3 + ii_1;
    ii_2    = ii_3 / 2;
    i_0     = ipntp_4 - 1;
    for( k_0 = ipnt_1 + 1; k_1 < ipntp_5; k_2 = k_3 + 2 )
    {
      i_1         = i_2 + 1;
      x_0[ i_3 ] = x_1[ k_4 ] - v_0[ k_5 ]     * x_2[ k_6 - 1 ]
                              - v_1[ k_7 + 1 ] * x_3[ k_8 + 1 ];
    }
  }
  while( ii_4 > 0 );
}
```

Fig. 6. An example implementation of an *ICCG* kernel annotated with reference numbers.

$$\begin{array}{ll}
v & \mapsto \{\, v_0, v_1 \,\} \\
x & \mapsto \{\, x_0, x_1, x_2, x_3 \,\} \\
ipnt & \mapsto \{\, ipnt_0, ipnt_1 \,\} \\
ipntp & \mapsto \{\, ipntp_0, ipntp_1, ipntp_2, ipntp_3, ipntp_4, ipntp_5 \,\} \\
ii & \mapsto \{\, ii_0, ii_1, ii_2, ii_3, ii_4 \,\} \\
i & \mapsto \{\, i_0, i_1, i_2, i_3 \,\} \\
k & \mapsto \{\, k_0, k_1, k_2, k_3, k_4, k_5, k_6, k_7, k_8 \,\}
\end{array}$$

4.2 Strides

The *stride* of a reference is the distance between successive accesses to the associated variable. This comes about through the use of subscripted references to vector variables in the program. As the subscript expression changes value, usually due to iteration of a loop, successive accesses to the variable are generated that are spaced apart by a potentially constant distance. Because our partitioned cache has a facility for strided cache lines which improve the density of useful data, the detection and use of this property is desirable.

The *ICCG* kernel uses only simple, one-dimensional array subscripts of a standard form. The strides for references to v and x are calculated and marked for use in configuring the partitions allocated to them:

$$\begin{aligned}
&v && \mapsto \{\ v_0[\text{stride=2}],\ v_1[\text{stride=2}]\ \}\\
&x && \mapsto \{\ x_0[\text{stride=1}],\ x_1[\text{stride=2}],\ x_2[\text{stride=2}],\ x_3[\text{stride=2}]\ \}\\
&ipnt && \mapsto \{\ ipnt_0,\ ipnt_1\ \}\\
&ipntp && \mapsto \{\ ipntp_0,\ ipntp_1,\ ipntp_2,\ ipntp_3,\ ipntp_4,\ ipntp_5\ \}\\
&ii && \mapsto \{\ ii_0,\ ii_1,\ ii_2,\ ii_3,\ ii_4\ \}\\
&i && \mapsto \{\ i_0,\ i_1,\ i_2,\ i_3\ \}\\
&k && \mapsto \{\ k_0,\ k_1,\ k_2,\ k_3,\ k_4,\ k_5,\ k_6,\ k_7,\ k_8\ \}
\end{aligned}$$

4.3 Groups

The purpose of a *group* is to collect together all references to a variable that have a similar access properties and should therefore be placed in the same cache partition. One method for creating groups of references is to collect them using the stride of the reference to determine which group it should go into. Using this strategy, each item of vector data may produce a number of reference groups, each with different strides and hence different access patterns. References to scalar variables will only ever produce one group because of the lack of associated stride based access.

For example, in the *ICCG* kernel in Figure 6, both the references to v have the same stride and so will be grouped together while the references to x will produce two groups because there are two different strides:

$$\begin{aligned}
&v && \mapsto \{\ v_0[\text{stride=2}],\ v_1[\text{stride=2}]\ \}\\
&x' && \mapsto \{\ x_0[\text{stride=1}]\ \}\\
&x'' && \mapsto \{\ x_1[\text{stride=2}],\ x_2[\text{stride=2}],\ x_3[\text{stride=2}]\ \}\\
&ipnt && \mapsto \{\ ipnt_0,\ ipnt_1\ \}\\
&ipntp && \mapsto \{\ ipntp_0,\ ipntp_1,\ ipntp_2,\ ipntp_3,\ ipntp_4,\ ipntp_5\ \}\\
&ii && \mapsto \{\ ii_0,\ ii_1,\ ii_2,\ ii_3,\ ii_4\ \}\\
&i && \mapsto \{\ i_0,\ i_1,\ i_2,\ i_3\ \}\\
&k && \mapsto \{\ k_0,\ k_1,\ k_2,\ k_3,\ k_4,\ k_5,\ k_6,\ k_7,\ k_8\ \}
\end{aligned}$$

4.4 Windows

The final useful property of a group of references is something termed the *window*. This property is a measure of the range of different data items which are accessed with the same stride and, to some extent, relates to the persistence of data within the cache. We use the window of a group of references as a heuristic to guide the sizing of partitions created for that group. The window is only useful in groups for vector variables because scalar partitions are sized to accommodate all variables rather than a subset of some larger data object.

The *ICCG* example highlights this property fairly well. Within the inner loop of the kernel, the main computational expression makes a number of references to the variable x which have produced two reference groups. The second of these groups contains the references $x[k-1]$, $x[k]$ and $x[k+1]$ because they all have the same stride, as determined by k. By examining these references, we can see that as the loop is executed a number of times the value of $x[k+1]$, for example, may be reused as the value of $x[k-1]$ as the value of k changes. To accommodate this need for data persistence, the window of the group is computed as the difference between the minimum and maximum offsets from each group of references.

This is used in the *ICCG* example to guide the sizing of cache partitions allocated to each group so that this potential persistence requirement is exploited to gain higher performance:

$v[\text{window}=2]$	$\mapsto$	$\{\ v_0[\text{stride}=2],\ v_1[\text{stride}=2]\ \}$
$x'[\text{window}=1]$	$\mapsto$	$\{\ x_0[\text{stride}=1]\ \}$
$x''[\text{window}=3]$	$\mapsto$	$\{\ x_1[\text{stride}=2],\ x_2[\text{stride}=2],\ x_3[\text{stride}=2]\ \}$
$ipnt$	$\mapsto$	$\{\ ipnt_0,\ ipnt_1\ \}$
$ipntp$	$\mapsto$	$\{\ ipntp_0,\ ipntp_1,\ ipntp_2,\ ipntp_3,\ ipntp_4,\ ipntp_5\ \}$
ii	$\mapsto$	$\{\ ii_0,\ ii_1,\ ii_2,\ ii_3,\ ii_4\ \}$
i	$\mapsto$	$\{\ i_0,\ i_1,\ i_2,\ i_3\ \}$
k	$\mapsto$	$\{\ k_0,\ k_1,\ k_2,\ k_3,\ k_4,\ k_5,\ k_6,\ k_7,\ k_8\ \}$

4.5 Partitioning

Once the reference analysis process is finalised, we can create a cache partition for each group of references to vector variables and mark them so that each reference accesses the variable through the correct partition. The partitions are created so that the partition stride and size are guided by the stride and window of the group from which guided their creation. In addition to partitions for vector variables, we create single partitions for all scalar variables and another for stack accesses. The scalar partition is sized so that it can house all variables that do not fit into registers.

At this stage, the source program is annotated in such a way that the back end of the compiler can correctly generate machine level instructions to implement the partitioning scheme required. However, before this is done, the total amount of cache resource used can then be reduced by reusing partitions. A simple technique, similar to register allocation, reuses partitions allocated to one data object when its use doesn't interfere with another and the configuration of the partition matches the requirements of the two variables.

5 Compositional Performance Model

In the previous section we have described how we can segregate memory accesses. Accesses to different objects are routed through different cache partitions in order to avoid interference. Various parallel activities that operate on one or more objects will also run in separate partitions, which allows us to define an extremely simple performance model. Below, we are going to define a performance metric P for a code segment s, $P(s)$. The unit of this metric can be execution time or number of cache misses depending on the prediction model used.

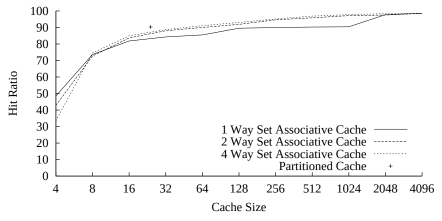
This section describes how the performance of an application may be calculated. We assume that we we have measurements of its constituent parts. That is, without necessarily computing the performance of the entire application by methods described in previous sections, we may accurately compose the applications performance from smaller, computationally cheaper and more accurate studies of the application.

5.1 Basic composition

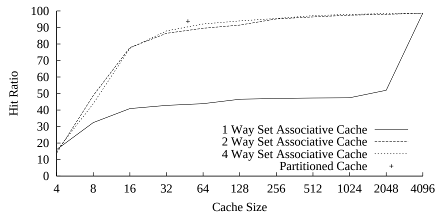
Assuming, that we can compute the performance $P(s_1)$ and $P(s_2)$ of two code fragments s_1 and s_2, then we can now trivially compute $P(s_1; s_2)$, the performance of the sequential composition of s_1 and s_2. If s_1 and s_2 share no partitions, then $P(s_1, s_2) = P(s_1) + P(s_2)$. This is valid for both time and miss performance metrics and means we can decompose large applications into more efficient components for performance prediction. If s_1 and s_2 do share partitions, then this method of calculating $P(s_1, s_2)$ can be pessimistic as the execution of s_1 may benefit the execution of s_2. In the cases where s_1 and s_2 share partitions, an accurate performance metric is made from the whole program $s_1; s_2$ and not from their constituent performance metrics. Iterated code segments can be treated similarly.

5.2 Extension by composition

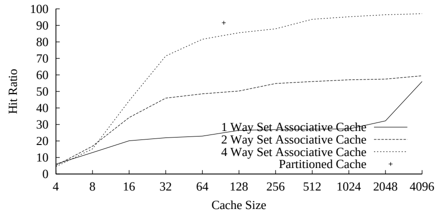
If we extend some fragment s, by the addition of a reference $v[i]$ to form s', where $v[i]$ is executed through a unique partition, we can count the extra misses incurred for $v[i]$ independently of the other references. The instruction references require a recalculation, to consider the changed program executed through the relevant instruction partition. This situation is similar to the case where $v[i]$ shares a partition with other references in the program. Prediction of the execution time for s' will require full static simulation and the classification [18] of the data reference $v[i]$ may be independent, or require the reclassification of a group of references, as before, depending on whether $v[i]$ shares a partition. The reclassification and simulation of the instruction code is also required as is a computation for the miss counting method if that is the model used.



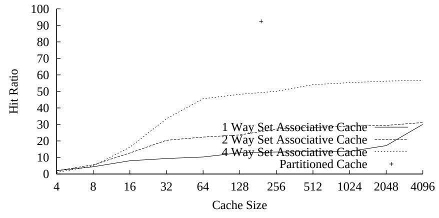
(a) 1 GIF Decoder thread(s).



(b) 2 GIF Decoders thread(s).



(c) 4 GIF Decoder thread(s).



(d) 8 GIF Decoders thread(s).

Fig. 7. Data cache performance of multiple GIF Decoder threads. The cache size is measured in lines.

5.3 Multi-threading

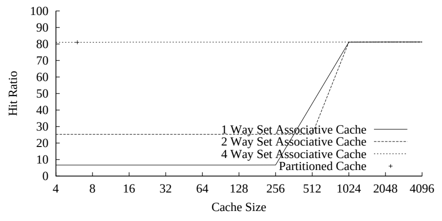
Considering two code segments s_1 and s_2 executed in parallel, denoted by $s_1|s_2$ and pre-computed performance metrics $P(s_1)$ and $P(s_2)$, a simple definition of $P(s_1|s_2)$ is $max(P(s_1), P(s_2))$. This definition has certain constraints. As in previous examples, we assume s_1 and s_2 are disjoint in their use of partitions. If this is not the case then we may either mark the shared partition references as misses or, if the combined reference sequence can be determined, accurately predict the performance. We will also ignore inter-thread communication facilities, suggesting that they do not dominate the performance of the tasks.

The parallel computation assumes that both tasks s_1 and s_2 have the same performance metrics as when they are run independently. This is true for the miss counting models but the runtime model makes further assumptions that may be included in a throughput aware model [15].

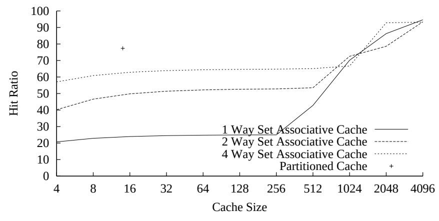
Even without extension of the utilisation metric, the model is valid and useful. It can answer, when only the cache memory hierarchy is the component under examination, the important questions about meeting computational deadlines. This model can also be used to build an optimisation scheme.

6 Results

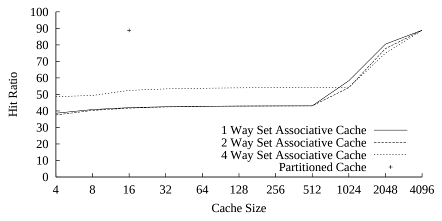
The GIF image decoder is a small and manageable example of a simple multimedia application. It is generally run in some sort of multi-threaded environment such as the HTML rendering engine of a web browser. To represent a realistic



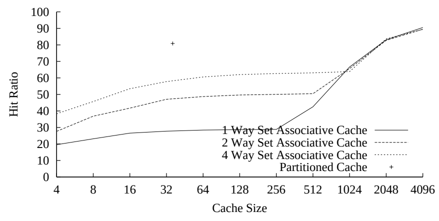
(a) Tri Diagonal Gaussian Elimination.



(b) 1D Particle In Cell (PIC).



(c) Discrete Ordinates Transport.



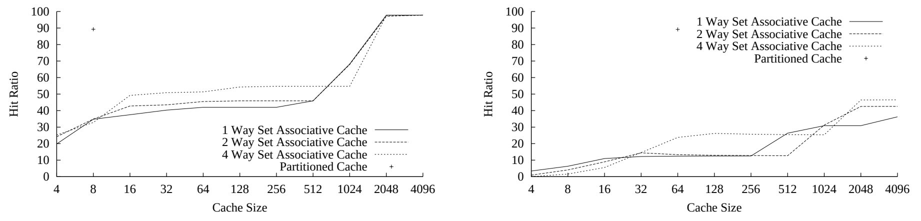
(d) All three kernels.

Fig. 8. Data cache performance of selected LLFK kernels both sequentially and in parallel. The cache size is measured in lines.

test, we ran between one and eight decoders operating concurrently, introducing the concurrency through the use of an explicit language level construct in the benchmark program. The execution trace of a GIF decoder is data-dependent and although each GIF decoder operates on a different image, all threads have a comparable run-time. The results of the experiment are shown in Figure 7.

The results for the partitioned cache are the single point in space with the overall performance of the cache is simply composed from the performance of the partitions. When a single GIF decoder is executed, conventional cache architectures show comparable performance profiles but, because of their lack of flexibility, scale badly and are not effective as the number of threads is increased. Data dependent application programs such as the GIF decoder would normally result in complex interference patterns between other threads in the system but by eliminating inter-thread interference with a partitioned cache, thread performance is guaranteed. In the case where there is only one thread running, the partitioned cache performs better than same sized set-associative caches due to the elimination of intra-thread or self interference. Although this benefit is largely uninteresting in multi-threaded situations, it is a potentially valuable property in high performance computer systems.

A second set of multi-threaded experiments are constructed using kernels taken from the LLFK benchmark suite. We performed two experiments, running three independent kernels in three threads, and running a parallelised kernel against a non-parallelised version. These experiments demonstrate inter and intra-thread interference between shared and non-shared data objects in a number of different threads of comparable run-time. The results are shown in Fig-



(a) 2D Implicit Hydrodynamics Fragment. (b) Parallelised 2D Implicit Hydrodynamics.

Fig. 9. Data cache performance of parallelised 2D Implicit Hydrodynamics Fragment kernel.

ures 8 and 9 respectively and graph the hit-ratio of a partitioned cache compared with a number of set-associative caches of a similar size.

In both experiments, as interference is introduced by running more threads either as disjoint processes of part of a parallelised algorithm, the partitioned cache can sustain high performance while the conventional caches suffer a significant drop. Although the parallelised kernel was constructed by hand, the drop in performance demonstrates the problems that can be introduced by parallelising compilers. If these compilers are not considerate of the issues involved as a result of their transformations, the good work done by finding and exploiting parallelism within a program will be undone by poor memory performance. With careful use of a partitioned cache and associated compiler technology, the gains from parallelisation of programs are protected against this kind of problem.

7 Conclusions

We have presented a cache architecture for which it is easy to predict the performance of programs running on it, because of its compositional properties. The ability to predict performance is particularly useful in real time environments, such as a set-top box. The software modules, such as the video decoder and audio decoder, can be developed and tuned independently, and the cache architecture guarantees that when integrated the performance of the two modules is a simple composition of the performance of all of the components.

Our solution partitions a direct mapped cache, each partition is used for one or more reference streams. The use of the cache is completely under control of the compiler. Because we expose the cache to the compiler, we need to modify the instruction set so that we can pass the partitioning information to the cache.

The performance of each partition can either be predicted analytically, or be measured using simulation tools. Analytical prediction of the performance of partitioned caches is far simpler than performance prediction of traditional caches because interference is controlled [15]. Whatever prediction method is chosen, compositional properties allow us to reason about the performance of the partitioned cache when the system is integrated.

References

1. A. Smith. A Comparative Study of Set Associative Memory Mapping Algorithms and Their Use for Cache and Main Memory. *IEEE Transactions on Software Engineering*, March 1978.
2. N. Jouppi. Improving Direct-Mapped Cache Performance by the Addition of a Small Fully-Associative Cache and Prefetch Buffer. In *17th International Symposium on Computer Architecture*, pp 364–373, June 1990.
3. J. Peterson and K. Hammond, editors. *Report on the Porgramming Language Haskell.* Yale University, 1996.
4. Inmos Ltd. *Occam-2 Reference Manual.* Prentice Hall, 1988.
5. J. Kowalik. *Parallel MIMD Computation: The HEP Supercomputer and its Applications.* MIT Press, 1985.
6. R. Nikhil, G. Papadopoulos, and Arvind. *T: A Multithreaded Massively Parallel Architecture. In *19th International Symposium on Computer Architecture*, pp 156–167, May 1992.
7. R. Wagner. Compiler-Controlled Cache Mapping Rules. Technical Report CS-1995-31, Duke University, December 1995.
8. T. Juan, D. Royo, and J. Navarro. Dynamic Cache Splitting. *15th International Conference of the Chilean Computational Society*, 1995.
9. D. Kirk. SMART (Strategic Memory Allocation for Real-Time) Cache Design. In *IEEE Symposium on Real-Time Systems*, pp 229–237, December 1989.
10. F. Mueller. Compiler Support for Software-Based Cache Partitioning. In *ACM SIGPLAN Workshop on Language, Compiler, and Tool Support for Real-Time Systems*, pp 137–145, June 1995.
11. D. May and H. Muller. Cache Memory. Patent Number WO045269, August 2000.
12. J. Carter, W. Hseih, L. Stoller, M. Swanson, L. Zhang, E. Brunvard, A. Davis, C. Kuo, R. Kuramkote, M. Parker, L. Schaelicke, and T. Tateyama. Impulse: Building a Smarter Memory Controller. *5th Conference on High Performance Computer Architecture*, January 1999.
13. A. Ki and A. Knowles. Secondary Cache Data Prefetching for Multiprocessors. Technical Report UMCS-97-3-1, Department of Computer Science, University of Manchester, 1997.
14. J. Fu, J. Patel, and B. Janssens. Stride Directed Prefetching in Scalar Processors. In *25th International Symposium on Microarchitecture*, pp 102–110, 1992.
15. J. Irwin. *Systems With Predictable Caching.* PhD thesis, Department of Computer Science, University of Bristol, 2001.
16. D. Page. *Effective Use of Partitioned Cache Memories.* PhD thesis, Department of Computer Science, University of Bristol, 2001.
17. F. McMahon. *The Livermore Fortran Kernels: A Computer Test Of The Numerical Performance Range.* Lawrence Livermore National Laboratory, Livermore, California, December 1986.
18. R. Arnold. Bounding Instruction Cache Performance. Master's thesis, Department of Computer Science, Florida State University, 1996.