Discovering Cache Partitioning Optimizations for the K Computer

Swann Perarnau
RIKEN AICS
perarnau@riken.jp

Mitsuhisa Sato
University of Tsukuba/RIKEN AICS
msato@cs.tsukuba.ac.jp

Abstract

The processor architecture available on the K computer (SPARC64 VIIIfx) features an hardware cache partitioning mechanism called *sector cache*. This facility enables software to split the memory cache in two independent sectors: data loads in one sector cannot trigger the eviction of data in the second one. Moreover, software is responsible for data placement in each sector by issuing special instructions tagging the various memory loads performed during execution. The implementation details of this cache partitioning mechanism also enable fast redistribution of the cache during an application's runtime, without any cost, allowing any optimization using the sector cache to be applied multiple times, with different setups, in the event of phase changes.

Unfortunately, in its current state, the compilers provided on the K Computer do not implement any automatic optimization using this cache facility. In the contrary, the only high-level interface to this mechanism is a set of directive to instruct the compiler to generate tagging instructions over a code region. Thus, only application programmers with intricate knowledge of both the memory access patterns of their code and the K Computer architecture can take advantage of this facility.

To address this issue and to study new optimization schemes using cache partitioning, we present in this paper a framework using binary instrumentation and reuse distance analysis to discover the locality of important data structures in an application and to suggest appropriate data distribution schemes for the sector cache. These optimizations are then translated into calls to the source-level API provided by the K Computer compilers. We applied our framework to analyze and optimize a set of HPC benchmarking applications and demonstrate significant performance improvements.

Categories and Subject Descriptors C.4 [*Performance of Systems*]: Measurement Techniques

Keywords Reuse distance, Binary instrumentation, Cache partitioning

1. Introduction

As the difference between memory and processor speeds continues to increase, optimizing a program locality becomes one of the most important issue in many research fields, including high performance computing. Over the years, many approaches to this problem have been evaluated, ranging from new hardware designs for the memory hierarchy to software solutions modifying either the program instruction or the data organisation to improve locality.

The improvement of the hardware cache has specifically been the focus of numerous studies. Indeed, any method reducing the average cost of a memory access will have tremendous impact on the performance of memory bound applications. Among those studies, we can cite work on scratchpad memories [1, 24] in embedded systems that allows a program to *lock* small data regions very close to the CPU or special instructions for non-cacheable memory accesses [11] to reduce cache thrashing.

In this paper we will focus on cache partitioning: a mechanism to split a cache in several *sectors*, each of them handling their data independently. In most cases, this independence guaranties that a memory load to a specific sector will not trigger the eviction of a cache line in another sector. While most research in this subject focuses on operating system schemes to forbid one process from thrashing the cache of another one, this paper discusses on the contrary the use of cache partitioning as an optimization tool for a single HPC application. Indeed, isolating a data structure in cache to protect it from streaming accesses should improve significantly the performance of a program.

Our target platform, the K computer [17] and its processor the SPARC64VIIIfx [6], features such a cache partitioning facility called the sector cache. Although multiple works [3, 5, 14, 20, 22] already studied cache behavior analysis and optimization using such a mechanism, specific architectural details of the implementation and API of the sector cache render them inefficient or impractical. Moreover, we argue that these particular traits also enables new optimization opportunities. Therefore, we discuss in the following our design for a new analysis and optimization framework for this architecture, with HPC applications as the specific target.

Our framework leverages and extends several existing methodologies. First, we use binary instrumentation of the target application along with debug information parsing to trace the various memory accesses to major data structures of a code region. This trace is then analyzed using a derivative of reuse distance to assess the locality of theses structures. Third, by modeling the impact of these localities on the performance of the application, we identify whether cache thrashing could be reduced by isolating some of these data structures to a specific sector. We envision these components as steps in an optimization loop: after identifying cache performance hotspots, a developer can analyze them, use the sector cache API to optimize them and repeat the process as much as required.

The remainder of this paper is organized as follows. Next section describes the K computer processor and in particular its sector cache. Section 3 presents existing works related to this study and discusses their applicability to our issues. Section 4 gives an overview of our memory tracing and locality analysis framework. Section 5 explains our metrics based on reuse distance and their uses to predict the impact of a sector cache configuration on the application's performance. We validate this framework in Section 6, demonstrating its potential on a custom built application and applying it on HPC benchmarks. We conclude and discuss future work in Section 7.

2. Cache Partitioning on the K Computer

The K Computer — ranked second on the Top500 issue of June 2012 — contains over 80 000 compute nodes, each composed of a single SPARC processor chip and 16 GiB of memory. The processor, a SPARC64 VIIIfx, was specifically designed for this system. This chip is produced by Fujitsu using a 45-nm process and is composed of 8 cores operating at 2 GHz for a peak performance of 128 GFLOPS [6]. It is an extended version of the SPARC-V9 architecture targeted at high performance computing, in particular it includes eight times more floating point registers (256) and SIMD instructions for improved parallelism on HPC applications.

2.1 Memory Hierarchy and Sector Cache

This processor's memory hierarchy is composed of two cache levels. Each core has two private L1 2-way associative caches of 32 KiB, for instruction and data. These caches are virtually indexed. An unified L2 cache is shared among all cores. This cache is 6 MiB wide, 12-way associative and physically indexed. All caches have a cache line size of 128 bytes and are inclusive: any data in the L1 cache is also in the L2.

Our focus in this paper is on a special feature of the data caches: software-controlled cache partitioning. Called *sector cache*, it allows software to split the cache into two independent partitions or *sectors*. Once activated, this partitioning ensures that a cache line retrieved for one sector cannot evict a cache line belonging to the other. In other words, instead of a single LRU eviction policy in the cache, the two sectors implement their own LRU. While both cache levels possess a sector cache, for the sake of simplicity, we will only discuss this feature over the L2.

The technical name for the hardware implementation of the sector cache is *instruction-based way partitioning*. To activate this partitioning, an unprivileged instruction specifies a splitting rule for the cache's associative sets: how many ways should be used by sector 0 and how many for sector 1. If the rule is valid (i.e. the two sectors contain at least one way), the information is stored in hardware and partitioning is activated. At this point, every memory load is considered to be of a specific sector (by default sector 0). Assigning a memory load to a sector uses another set of instructions: the unprivileged `sxar1` and `sxar2` instructions can specify for respectively the next and the two following memory accessing instructions the sector of each operand.

Isolation between the two sectors is ensured by the hardware. Two counters keep track inside each associative set of the number of lines belonging to each sector. When a cache line is retrieved from memory, the sector checks if it is full, in which case a line of the same sector is evicted according to a pseudo-LRU policy. When the sector is not full, due to a cache line previously invalidated for example, the size of the sector is increased and data placed in an available line. As a matter of fact, nothing in this hardware implementation restricts the two sector's sizes to sum up to 12 (the number of ways in an associative set). The behavior of the eviction/sector management mechanism becomes however much more complicated (isolation in not guarantied anymore) and we chose not discuss such setups in this paper. Another detail that will matter in the next section however is the behavior of this partitioning when ways are left unoccupied. If the sum of both sectors sizes is less than 12, any sector is allowed to use the remaining ways. Consequently, the cache is always used in full, with the sectors competing for unassigned ways in some configurations. This behavior makes it impossible for example to limit the cache of the application by restricting it to a small sector, as the latter will grow above its size limit if no memory is loaded in the other sector.

For simplicity, we will consider in the remainder of this paper that only 11 configurations are valid for the sector cache: if we note a configuration (s, t) with s the size of sector 0 and $t = 12-s$ the size of sector 1, then we will only discuss the set of configurations $(1, 11), (2, 10), \ldots, (11, 1)$.

2.2 Sector Cache Programming Interface

If one is willing to program an HPC application in SPARC assembly, the special instructions activating the sector cache and assigning to sectors some of the memory accesses are all that is required. Nevertheless, the C and Fortran compilers provided on the system also give access to an higher

```
sxar2
sllx  %xg29, 3, %xg29
fmuld  %f56, %f88, %f56
sxar2
ldd,1  [ %xg24 + %i4 ], %f324
ldd  [ %xg29 + %i5 ], %f322
sxar1
sllx  %xg27, 2, %g4
```

Figure 1. Sector cache example: the first `ldd` instruction is tagged for sector 1 by the previous `sxar2`. Disassembly indicates use of the sector 1 by the mnemonic '`,1`' as the `sxar` instructions can serve other purposes.

level interface. Such API should be easy to use to anybody familiar for the OpenMP or OpenAcc language extensions. Indeed, it is directive-based: the programmer marks by special comments (pragmaS in C) the code regions that should use the sector cache and the compiler generates the required instructions. Two directives are provided: one setting the size of each sector and one specifying the instructions to tag into the sector 1, by taking data structures (arrays) names as a parameter. These directives can either be applied to a procedure as a whole or to a smaller code regions by using begin/end delimiters.

```
double a[N],b[N][N],C[N];
void mvp(void)
{
#pragma procedure cache_subsector_size 10 2
#pragma procedure cache_subsector_assign c
    int i,j;
    for(i = 0; i < N; i++)
        for(j = 0; j < N; j++)
            a[i] += b[i][j]*c[j];
}
```

Figure 2. C matrix-vector product with a $(10, 2)$ sector cache configuration and the `c` array tagged into sector 1.

During assembly generation, the compiler will consider that any instruction touching a data structure assigned to sector 1 must be preceded by the special `sxar` instructions, tagging the memory access as belonging to the right sector. Since the sector 0 is the default sector, the user only need to specify the structures going to sector 1. Unfortunately, this interface has an obvious issue: if the compiler cannot determine that an instruction accesses a data structure, it cannot generate the right tagging instruction before it. For example, the use of *pointer aliasing*, accessing a structure by another variable pointing to the same memory, will not trigger the tagging instruction generation. Moreover, the compiler does not provide any means to automatically use the sector cache. It is up to the application programmer to know where and how this feature could improve the performance of its code.

Finally, environment variables on the computing node can exert some control on the sector cache. The runtime environment provides two of them, one to activate/deactivate the sector cache completely and one to configure an initial size for the sectors. Of course, using the latter instead of the directive inside the source code makes it impossible to dynamically change the sector sizes during runtime, to adapt to phase changes for example.

3. Related Work

This sector cache facility is closely related to two kind of works: scratchpad memories in embedded systems and recent software-controlled cache partitioning methods.

3.1 Scratchpad Memories

In embedded systems, real-time constraints often require programmers to know precisely how much processing time a given code will take. In such contexts cache memories, unpredictable as their behavior depend on the data processed, might be forbidden. Instead, scratchpad memories are used. Originally, a scratchpad memory was a memory module close to the processor where program code or data was placed statically into [1]. To use such memories efficiently, compilers analyze the locality of the program and choose statically which code or data to move to this scratchpad [24]. Recent architectures also include software-controlled scratchpad, with DMA-like special instructions used to explicitly move data in and out of this memory [12]. As a result, most works on scratchpad memories focus on static analysis to identify data with good locality across the whole program execution or on optimizing the scheduling of explicit data movement instructions.

While our sector cache facility might share similarities with such memories, the programmer has little control in our context over the order in which data is fetched from memory into the cache, nor over the eviction of this same data: a LRU policy is still active inside the sector cache. Moreover, we argue that different code sections of a program might exhibit different localities. In other words, the sector cache might need to be reconfigured several times over the course of a program execution.

3.2 Software-Controlled Cache Partitioning

In a different context, recent works studied the implementation and use of a software-controlled cache partitioning facility on commodity systems, mostly using page coloring [13].

Cache partitioning works use page coloring in a unintended way: two virtual memory regions not using the same colors cannot conflict each other in cache. Thus, partitioning the cache is simply the process of assigning colors to the virtual address space of a process in a specific way, so that structures are isolated from each other.

Several drawbacks to this method can be immediately identified. First, it requires changing the virtual memory manager of the underlying operating system or, at least, to extend and bypass it in significant ways. Second, it is limited by the amount of physical memory available on the system. While a complete and integrated solution to this particular issue could be implemented by rewriting the swapping system, to the best of our knowledge no existing work did it. Finally, changing a partitioning during an application runtime is very expensive in this setup. Indeed, it requires stopping the program and moving data from every discarded physical page to a new one. Among works that use page coloring and thus suffer from these issues we can cite Soft-OLP [14], ULCC [5] and CControl [20].

The Soft-OLP paper represents the closest work to our goals. It describes a binary instrumentation framework to analyze the locality of objects (data structures) inside a program to better distribute the cache among them using cache partitioning. The cache partitioning environment the author use allows for more than two partitions and the authors use this feature to create multiple groups of objects depending on their locality and relative influence. To analyze the latter, the authors define an *inter-object interference* metric. This metric tries to assess how the reuse distance of a data structure will evolve if another structure is inserted in the same partition. The authors chose to sample the number of references made to the new structure between accesses to the first one. To be meaningful, this measurement is made for every pair of data structure. Unfortunately, Soft-OLP doesn't match our goals on several issues. First, it uses page coloring for the partitioning, limiting the tool to whole program analysis since dynamic repartitioning is too costly. Second, the tool only detects global objects reading at a very simple level the program symbol table and structures created by a single call to standard allocation functions (the C `malloc` family). The authors acknowledge that this issue triggered them to modify the source code of several of the benchmarks used in SPEC CPU2000 for example. We consider that the application's code should not be modified, specially in programs where a significant effort has already been made to optimize them. Moreover, we should be able to measure the locality of parameters to a function, including subranges of an array. Thus, at this stage of our study, we do not consider intra-object interference to be required, which simplifies the locality analysis.

On the same topic, ULCC [5] and CControl [20] are two software environments allowing a user to partition the cache for its application. While ULCC relies on user knowledge of each application locality for their optimization — something we want to eliminate — CControl discusses the use of modified runs of the application to discover automatically the locality of its data structures. Unfortunately, while likely to be faster than binary instrumentation to analyze the locality of an application, this experimental scheme is not possible on the K Computer.

4. A Framework for Analysis and Optimization of Partitioned Programs

As stated in the introduction, we envision our framework as steps in an optimization process. An application developer identifies its program hotspots in terms of cache performance. Then he uses our framework to measure the locality of key data structures in an critical code region. Our framework identifies a correct optimization strategy and indicate to the developer which modifications to make. Finally, the developer can repeat this process across its program code as long as he wishes.

As we just stated, our framework does not identify by itself neither the application's hotspots nor which data structures are of most interest. Detecting which functions generate the most cache misses can easily be achieved by using one of the numerous existing profiling tools like Intel VTune [21], Likwid [23] or the one Fujitsu provide on the K Computer [10]. As for identifying the relevant structures, we rely on the user to provide us their names. While our tool is able to list every variable in the program and analyze all of them at once, the complexity of the reuse distance measurement grows with the number of structures. Assuming that the code region under study uses few of the numerous variables existing in the program, considering all of them will slow down the analysis without any purpose.

Schematically, our framework analyzes the target code in three phases. First, it extracts from the DWARF debugging information of the binary under study the location of all the data structures specified by the user. Since this location information can require runtime values, it is saved in an object table along with a description of the runtime information it needs. Second, the application binary is instrumented by Pin [15]. This instrumentation traces every memory access triggered by the target code and identifies which data structures they corresponds to. This information is outputted along with the instruction address and the memory location touched. Finally, our framework reuses this memory trace to compute locality metrics related to each data structure. These locality metrics are then used to predict the amount of cache misses all the possible sector cache configurations will trigger. Thus, the best sector cache configuration is found. Figure 3 summaries theses different phases.

4.1 Extracting Data Structures Information from DWARF

DWARF is the standard debugging information format used under Linux (which the K Computer uses both on the frontend and the computing nodes). It describes all the functions, variables and constants in the program, providing enough information for a debugger to be implemented. The format organizes its information into a tree of DIEs (Debugging Information Entries), with a top DIE representing the compilation unit and having as children DIEs representing enclosed types, functions and variables.

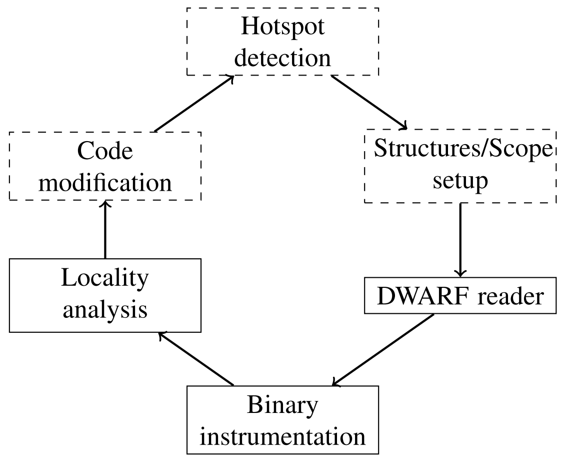

Figure 3. Our framework's analysis and optimization pipeline. Dashed boxes represent user actions.

Upon startup, the user provides to our framework a description for each data structure he wants to analyze. Such description consists of the structure's name and optionally its enclosing scope (compilation unit and/or enclosing function). Our framework then uses these description to create an object table. This table associates each structure to the virtual memory regions it occupies, so that the tracing phase can identify if a memory access touches a structure.

To find each data structure location, our tool scans recursively the whole DIE tree. As each DIE is identified by a tag, telling if it is a function, a variable or anything else, finding the structure DIE only requires to filter the evaluated nodes using the information the user gave. In its present state, our tool only supports identifying arrays, as the sector cache API can only work on such structures. DWARF identifies arrays using the `DW_tag_array_type`. This type of DIE contains the basic type (type of one element), a location expression and a list of DIEs for each of its dimensions, giving each dimension size if it can be computed statically. While computing the size of such array is easy given this information, the location expression giving the starting address of the array can require work at during the application execution.

DWARF defines a location expression as a list of simple operations to apply to an integer stack. After execution of all operations, the stack's top value is the virtual address researched. Unfortunately, some of the operations that our tool might encounter require pushing to the stack a machine register's value at the entry point of the enclosing function. DWARF uses this kind of location expressions to represent for example a parameter whose starting address was pushed into the stack when calling a function. In this case, resolving the location requires knowledge of the stack pointer value and an access to the memory of the process at this address.

When our tool cannot resolve the location expression of a structure statically, the location expression is saved along with a list of the machine registers required for its evaluation.

4.2 Tracing structure accesses

Along with the data structures description, the user provides our tool with a tracing scope. This scope defines the instructions to take into account when measuring the structures' locality. Limiting the tracing to a small part of the application allows the user to optimize each application hotspot independently, as the sector cache allows it without cost. The tracing scope is defined as either a function or a range of source lines. By default, our tool only traces the instructions of the top-level scope, not the code that could be called from it. This greatly reduce the number of instructions traced and thus limit the cost of the trace. If needed, the user can still activate a full instrumentation.

Our tool uses the binary instrumentation framework Pin [15] to modify the target application's executable. For each instrumented instruction triggering a memory access, our code collects the necessary register values (for location expressions) along with the address and the size of the memory access.

Two kinds of data structure require special handling: dynamically allocated arrays and pointer function arguments. To identify which memory addresses belong to a dynamically allocated structure, our tool uses the following steps. First, the memory address of the stack or global pointer identifying the structure is saved, and tagged as belonging to a given variable. Second, every function of the `malloc` family is traced. On each call, our framework remembers the newly allocated area and the memory address it is written to. It then compares this address to already tagged locations. If a positive match is found, the newly allocated memory is tagged accordingly. In other words, our tool iteratively associates allocated memory to already known variable locations by monitoring the values written inside. Of course, this identification assumes that the only values written to inside a pointer are memory addresses. In practice, this limitation is rarely an issue.

Unfortunately, pointers as function parameters are not as easy to handle. For such variables, the DWARF information might not contain the size of memory region pointed to, even if it is known during compilation by the compiler. In such event, our tool informs the user of missing information and recommends him to use a scope higher in the execution path for tracing.

5. Locality Analysis

In this phase, our framework uses the information of the previous step to compute different locality metrics. In its current state, all of these metrics are derived from the notion of reuse distance.

In its original definition, this distance measure the reuse potential of a memory access by relating it to the previous reference to the same location. More precisely, a memory access to location l has a reuse distance d if the number of distinct references since the last access to l is d. Assuming a fully-associative cache with a perfect LRU eviction policy, this metric captures precisely which memory references will trigger a cache miss. Indeed, with a cache of size C, any memory access with a distance $d > C$ will be misses, as the memory location was evicted from cache since its previous reference.

While originally developed for the study of virtual memory systems by Mattson et al. [16] in 1970, reuse distance has since be applied successfully in the performance analysis and optimization of numerous domains. Many works used it to estimate cache misses [2, 9] or to direct data reorganisation by the compiler [3].

5.1 Conditional Reuse Distance

Our objective in this work is to be able to identify if, when a data structure is pushed into sector 1, the performance of the program improves. This involves determining two kind of localities: the locality of a structure by itself (*i.e.* the amount of cache misses it will trigger if alone in sector 1) and the locality of all the other memory accesses. To do so, we start by defining the *conditional reuse distance* (CRD): a memory access to location l has a reuse distance d if the number of distinct references *satisfying a condition* c since the last access to l is d.

We will use this conditional reuse distance to measure these two classes of locality. First, for a given data structure s occupying the set of addresses M and a memory access $a \in M$, the isolated distance of a is its CRD with the condition: $\in M$. In other words, only memory accesses touching the same structure are taken into account in the reuse distance computation. Such reuse distance gives us the amount of cache misses a structure will trigger by itself if it was isolated in sector 1. Second, for all accesses not touching s, we compute their CRD with condition: $\notin M$ (all accesses that not touching s). This gives us the cache misses triggered by the rest of the application if we only isolate s in sector 1, all other accesses going in sector 0. Both measures are required to compute the amount of cache misses triggered by a sector cache configuration with only one data structure isolated.

As previous works already discovered [7, 8], a small set of instructions can sometimes be responsible for numerous cache misses, due to the poor locality of their accesses. To take this into account, we measure an *infinity ratio* for each instruction. This ration measure the amount of accesses done by an instruction that have an infinite reuse distance: the access is the first to a location, or its reuse distance is greater than the cache capacity. Such accesses always trigger cache misses. Thus, instruction with a high infinity ratio have poor locality and could be improved by being replaced with their non-temporal version.

Our framework stores the two kinds of CRD as histograms associated with a given structure, while the infinity ratio is stored on a per instruction basis. Overall, for S structures analyzed, $2 * S + 1$ CRD are computed, and only $S + 1$ for each memory access traced by our framework.

5.2 Implementation

This subsection describes the algorithm computing for each memory access traced by our instrumentation phase, the associated reuse distances.

To compute a reuse distance, our tool implement one of fastest algorithm known [19]. This algorithm relies on two data structures: an hash table and a splay tree (balanced binary tree). The hash table maps memory references to the time of their last access. The splay tree is used to register one node per unique location, sorted by their timestamps. To help compute the reuse distance of a memory access, each node of the tree keeps track of the size of its subtrees. Thus, counting the number of distinct locations between two access to a memory reference is the process of traversing the tree from the root to the node of that reference, accumulating the sizes of the subtrees containing bigger timestamps than the research access. Algorithm 1 give a pseudo-code for this distance computation.

Algorithm 1 Reuse distance of a memory access by tree traversal.

```
Input: $T, c$: a binary tree and a timestamp.
Output: $d$: number of nodes with timestamps $> c$.
  $cur \leftarrow T.root$
  $d \leftarrow 0$
  loop
      if $c > cur.timestamp$ then
          $cur \leftarrow cur.right$
      else if $c < cur.timstamp$ then
          $d \leftarrow d + 1$
          if $cur.right \neq null$ then
              $d \leftarrow d + cur.right.size$
          end if
          $cur \leftarrow cur.left$
      else
          if $cur.right \neq null$ then
              $d \leftarrow d + cur.right.size$
          end if
          return $d$
      end if
  end loop
```

With such algorithm, computing a conditional reuse distance is just a matter of choosing which memory locations to store in the splay tree. Consequently, the algorithm to analyze the locality of our application simply scans through all structures, checking if the tracing phase identified it as touching each of them. If the access is inside the structure, it then computes its isolated reuse distance and saves it in the corresponding histogram. The memory location is added to the splay tree, and the last timestamp to this location is updated in the corresponding hash map. Otherwise, the reuse distance for all other accesses is computed in the same fashion. Finally, the traditional definition of the reuse distance is computed (taking into account all accesses) and used to compute the infinity ratio of the current instruction. Algorithm 2 gives a pseudo code for this instrumentation.

Algorithm 2 Tracing algorithm executed for every memory access made by an instrumented instruction.

```
Input: $A$: this access address, $pc$ the program counter, $T$ the
    structure table, $c$ this access timestamp.
  for all $S$ in $T$ do
    if $A \in S$ then
      $d \leftarrow$ UPDATE($c$,$S.ihash$,$S.itree$,$A$)
      $S.ihistogram[d]$++
    else
      $d \leftarrow$ UPDATE($c$,$S.ohash$,$S.otree$,$A$)
      $S.ohistogram[d]$++
    end if
  end for
  $d \leftarrow$ UPDATE($c$,$All.hash$,$All.tree$,$A$)
  $All.histogram[d]$++
  $All.infinityR[pc].count$++
  if $d = \infty$ then
    $All.infinityR[pc].inf$++
  end if
  function UPDATE($c$,$H$,$T$,$A$)
    $d \leftarrow \infty$
    if $A \in H$ then
      $d \leftarrow$ distance($T$,$H(A)$)
      delete($T$,$H(A)$)
    end if
    insert($T$,$c$,$A$)
    $H(A) \leftarrow c$
    return $d$
  end function
```

The complexity of this reuse distance measurement can be analyzed as the following. For each reference of a trace of size N, we scan S data structures, and compute one reuse distance. One additional reuse distance is computed for the infinity ratio of the current instruction. The reuse distance algorithm lookups the reference in the hash table in $O(1)$ and performs at most three traversals: distance computation, deletion of the reference's node, insertion of a new one. Each of these operations has a complexity of $O(\log M)$, M being the number of unique references in the trace. Finally, insertion inside the histogram costs $O(1)$. Thus, the whole complexity of the measurement is $O(N.S. \log M)$.

Given that we target a single architecture, each reuse distance analysis is optimized to only remember the amount of locations that can fit in our target cache, considering that any access with a distance greater than that will always trigger a cache miss.

5.3 Identifying Cache Optimization Opportunities

Once the locality analysis is completed, our framework outputs each structure's histograms and the conflict table. From these we predict the amount of cache misses that a sector cache configuration will trigger. Let us start by formalizing our reuse distance and cache misses model.

Let M be the set of memory addresses touched by our program and T the trace of these memory accesses. We can express it as a set of pairs (pos, a) with $pos \in \mathbb{N}$ and $a \in M$. Let $A(s)$ be the memory addresses of a structure s analyzed by our experiments. We can now define the reuse distance as $h(m, d)$: the number of accesses of distance d in the trace $T - (pos, a), \forall a \in m$. That is, we remove from the trace of memory accesses a number of addresses, while preserving order, before computing the reuse distance of it. For future convenience, we will note $h_0(s, d) = h(A(s), d)$ and $h_1(s, d) = h(M - A(s), d)$. The former represents the reuse distance without accesses to a specific data structure and the latter to the reuse distance of these accesses by themselves.

Let C_0, C_1 be respectively the sizes in bytes of Sector 0 and Sector 1 and let s be the data structure in Sector 1. Then we can express the cache misses we observe in our experiment as:

$$Q_s(C_0, C_1) = \sum_{d=C_0+1}^{\infty} h_0(s, d) + \sum_{d=C_1+1}^{\infty} h_1(s, d)$$

This equation formalizes the cache miss model we presented earlier: for a given cache size, any access with a reuse distance greater than this size will trigger a cache miss. Computing the amount of cache misses triggered by a sector cache configuration thus just requires iterating through both histograms built during the reuse distance analysis and summing their bins for values higher than the corresponding cache size.

6. Experimental Results

We now validate our framework by analysing and optimizing several memory intensive applications. After describing our general methodology, we will validate our framework on two aspects. First, we use a custom-built application inspired from stencil codes to ensure that the cache behavior we measure correspond to knowledge we have of the application. Since this application is built on purpose, it also gives us a estimate of the performance improvement we can expect from using the sector cache. We then analyze and optimize code regions from the NAS Parallel Bencharmks. This demonstrate that our framework can be applied to standard HPC codes.

6.1 Methodology

All our experiments follow the optimization process we described earlier. First, a performance analysis tool is used on the K Computer to identify which functions in the application are performance hotspots. We specifically measure running time and L2 cache miss rate of each function. We then execute our binary instrumentation on the same application, but compiled on a Linux system with an Intel i7 2760 QM processor and 8 GiB of RAM. This system is required as no binary instrumentation framework is able to modify binaries compiled for the SPARC64 VIIIfx architecture of the K Computer. While the use of a different architecture for the locality analysis might seem an issue, we argue that using another architecture should not change in any significant way the memory accesses that the program code dictates. Moreover, the reuse distance metric is by design architecture independent. A slight change in ordering of instructions should not change the global cache requirements of a data structure neither.

Each function that we identified as an hotspot is thus analyzed by our tool, tracking all arrays touched by the code (deduced by reading the source). The instrumentation is directed to stop the application after one execution of the function. After analysis, if our framework identifies that a function can be optimized, the necessary source code modifications are made. As we only analyze complete functions here, these modifications consist of setting the sector cache size and directing the compiler to push one structure in sector one.

This code is then recompiled for the K Computer and its running time and cache miss rate measured again.

6.2 Multigrid Stencil

Our test application makes a simultaneous use of three different matrices that reside in memory to compute the elements of a result matrix. The input matrices form a multigrid structure, it is made of a large matrix ($Y \times X$ double-precision floating point values), a medium-sized matrix (one fourth of the large matrix size) and a small matrix (one sixteenth of the large matrix size). The output matrix has the same size as the large matrix. Each of its elements is a linear combination of nine points stencils taken from each input matrix at the same coordinates (interpolated for smaller matrices). This application is interesting for two reasons: it is extremely memory intensive and each of its matrices has a different cache size requirement. Our nine points stencil forms a cross (a center element, the two elements above it, the two elements on the right, and so on) and it is included in five lines of a matrix. Thus, in the ideal case, if five lines of each input matrix can remain in the cache during the computation, the stencil will be computed with a maximal reuse. This translates into a cache space of $X \times 8 \times 5$ bytes for the large matrix, half of this size for the medium one and one fourth of this size for the small one. Of course if these requirements cannot all fit in cache, accesses to each matrix will thrash accesses to the others. Matrices are named from the smallest one M_1 ($X/4$ by $Y/4$) to the biggest M_3 (X by Y), the result matrix is named M_r.

We analyze here the locality of the 3 input matrices. We chose the matrices sizes for M_3 to requires 7MiB in cache, such that a default cache configuration triggers numerous cache misses.

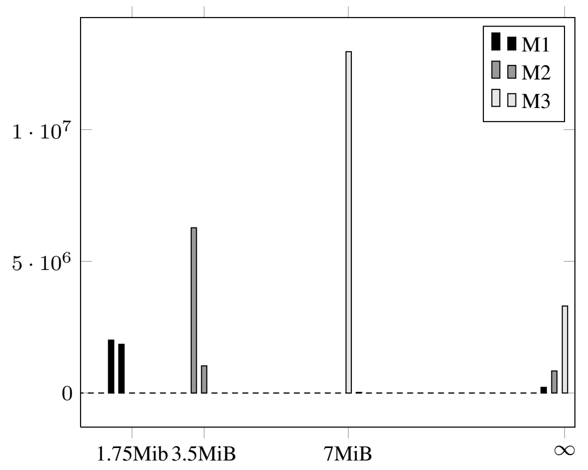

Figure 4. Reuse distance histograms for the 3 input matrices of the multigrid stencil. For simplicity, we display distances bigger than a 12^{th} of the cache. Bins are 256KiB wide. The last bin is for infinite distances.

After analysis of these reuse distance histograms (Figure 4), our cache model predicts that isolating the M_2 matrix with a sector cache configuration giving more than 7 ways to sector 1 would reduce by 23% the amount of cache misses triggered by one stencil. We applied this optimization to our application code and compared the resulting program to the unoptimized one on the K Computer. We compiled a different version of the program for each sector cache configuration tested. To also validate that our optimization was among the best available, we tested every configuration of the sector cache for every data structure.

Version	Stencil Miss Rate (%)	Reduction (%)
Unoptimized	2.10	-
$M_2(5,7)$	1.68	20
$M_2(1,11)$ *best*	1.62	22
$M_1(7,5)$ *best*	1.84	12
$M_3(11,1)$ *best*	2.08	0.1

Table 1. Cache misses reduction: comparison between the chosen optimization against best configuration for each isolation.

Table 1 gives the resulting cache misses for the different versions. Notice that the very best configuration available is for M_2 to be isolated in a sector bigger than 7 ways. However, given the reuse distance histograms we measured, our cache model does not predict any performance difference between the two configurations. The fact that we do not take into account the influence of associativity on cache misses could explain this small performance difference. Our tool still achieves a very good optimization of the application.

6.3 NAS Parallel Benchmarks

We analyzed two benchmarks from the Omni OpemMP C version 2.3 of the NAS Parallel Benchmarks: CG, LU. These benchmarks were only using one thread. In both cases, significant optimizations were found.

Most of the computation time of the CG benchmark is spent inside the `conj_grad` function. This function does not call any other, and is repeated multiple times during the benchmark's lifetime. The core of this function is a sparse matrix-vector product, with most of the memory accesses touching 3 data structures: the sparse matrix a, the column index $colidx$ and a dense vector p. We analyzed the locality of these structure and, unsurprisingly, our framework indicated that the p vector could benefit for isolation using the sector cache. Indeed, both other structures exhibit streaming access patterns due to indirect accesses that could impact negatively the caching of p. Our optimization thus isolates p in sector 1, with enough space to allow good caching. Such optimization, adding only two lines to the source code of the benchmark reduces the execution time of this function by 10%.

Our process to analyze and optimize the LU benchmark was as follows. First, LU spends almost all of its runtime in the `ssor` function. This function contains a loop, calling successively several subroutines over shared data structures. Iteratively, these calls solves a system of Navier-Stokes equations by successive over relaxation, decomposing it into lower and upper triangular matrices. Overall, eight structures are of interest here: $flux$, u, rsd and $frct$, which are global arrays used as input and results storage, and a,b,c and d which are working arrays used across subroutines to hold partial results (triangular matrices). To analyze this benchmark, we configured our framework to trace recursively all instructions of the `ssor` function or of any other function called from it. The resulting analysis identified each of a, b, c and d to benefit from the sector cache in the same way. The cache requirements of the other arrays could not fit in any sector configuration.

While isolating only one of the 4 arrays identified by our framework only improved by 2% the benchmark's execution time, another optimization gave more interesting results. Indeed, protecting those 4 arrays for streaming accesses to the other variables of the program by pushing them all together in sector one proved to be a better optimization. It resulted in a 8% reduction of execution time of this function. We should note that, as the `ssor` function passes these arrays to some subroutines as parameters, we had to change the sector cache directives in them to match the actual parameter names. Overall, code modification added 10 lines of directives: for each of the 5 functions, one line for sector size and one for variable isolation. We excluded two functions (`rhs` and `l2norm`) from these modifications, as they do not use these arrays.

Table 2 describes the exact optimization on each benchmark's functions, and the resulting improvements. Note that the cache misses reduction reported are `direct` cache misses: cache misses triggered by the speculative hardware prefetcher are ignored.

7. Conclusion and Future Works

We presented a cache analysis and optimization framework specifically targeted at optimization of HPC application running on the K Computer and using its cache partitioning facility. While previous work presented interesting analysis and optimizations techniques using cache partitioning, we demonstrated that specific implementation details of the sector cache rendered such solutions impractical, requiring us to develop our own environment.

Using state of the art tools in binary instrumentation techniques, we discussed the analysis of the cache requirements of each data structure of interest inside an application. Because the sector cache provides dynamic reconfiguration of the partitioning during execution, we argue that a solution analyzing independently multiple regions of code and determining the best configuration for each of them is a better strategy than whole program analysis. This approach can still be fast compared to the execution time of the original benchmark, as the instrumentation is limited to few instructions of the program (i.e. the rest of the binary executes at full speed).

By analyzing and optimizing a custom-built application, we demonstrated that our framework can predict with good accuracy the cache requirements of each data structure and suggest valid optimizations. The additional analysis and optimization of NAS parallel benchmarks also confirmed the applicability of our work. Still, our optimization process for the NAS LU benchmark also illustrates the kind of additional work that could improve our framework.

First, pushing several data structures inside sector 1 could be made easier by including in our analysis phase the study of inter-structures conflicts in cache either by a costly metric similar to the one in Soft-OLP, or simply by detecting if several objects are loaded in cache during the same timeframe. Second, while our framework is already capable of locality analysis over several functions called successively, it will be interesting to gather reuse histograms per function while the reuse distance is computed globally. Such analysis could help detect better sector cache configurations from one function to the next. Third, analysis time could be improved further, either by trading accuracy for speed [25] or by parallelization on the reuse distance algorithm [4, 18].

Finally, we are also interested in the long term on automating as much as possible our framework. By automating we mean integrating a performance measurement tool to detect performance hotspots, source analysis to extract

Benchmark	Function	Isolated Variables	Sector Size	Miss Reduction (%)	Runtime Reduction (%)
CG	conj_grad	p	(1,11)	19	10
LU	ssor	a,b,c,d	(2,10)	48	8
	blts	ldz,ldy,ldx,d		75	10
	buts	d,udx,udy,udz		18	3
	jacld	a,b,c,d		64	14
	jacu	a,b,c,d		57	6

Table 2. Optimization of NAS Benchmarks.

from the hotspot identification the data structures' names and other necessary information and code transformation at the end of the pipeline to modify the application without user intervention. This last step requires in particular a source-to-source transformation tool and, while an optimization can be as simple as the insertion of two pragma at the top of a function, the LU example showed that different function can have different names for the same memory locations, which could be more difficult to take into account.

Acknowledgments

Part of the results were obtained by early access to the K computer at the RIKEN AICS. This work was supported by the JSPS Grant-in-Aid for JSPS Fellows Number 24.02711.

References

[1] R. Banakar, S. Steinke, B.-S. Lee, M. Balakrishnan, and P. Marwedel. Scratchpad memory: design alternative for cache on-chip memory in embedded systems. In *Proceedings of CODES 2002*.

[2] K. Beyls and E. D'Hollander. Reuse distance as a metric for cache behavior. In *Proceedings of PDCS 2001*.

[3] E. Bugnion, J. Anderson, T. Mowry, M. Rosenblum, and M. Lam. Compiler-directed page coloring for multiprocessors. *ACM SIGOPS Operating Systems Review*, 30(5), 1996.

[4] H. Cui, Q. Yi, J. Xue, L. Wang, Y. Yang, and X. Feng. A highly parallel reuse distance analysis algorithm on gpus. In *Proceedings of IPDPS 2012*.

[5] X. Ding, K. Wang, and X. Zhang. ULCC: a user-level facility for optimizing shared cache performance on multicores. In *Proceedings of PPoPP 2011*.

[6] T. M. et al. SPARC64 VIIIfx: A new-generation octocore processor for petascale computing. In *IEEE Micro*, volume 30, 2010.

[7] C. Fang, S. Carr, S. Önder, and Z. Wang. Reuse-distance-based miss-rate prediction on a per instruction basis. In *Proceedings of MSP 2004*, .

[8] C. Fang, S. Carr, S. Önder, and Z. Wang. Path-based reuse distance analysis. In *Proceedings of CC 2006*, .

[9] S. Gupta, P. Xiang, Y. Yang, and H. Zhou. Locality principle revisited: A probability-based quantitative approach. In *Proceedings of IPDPS 2012*.

[10] K. Ida, Y. Ohno, S. Inoue, and K. Minami. Performance profiling and debugging on the K computer. *Fujitsu Scientific and Technical Journal*, 48, 2012.

[11] Intel Corporation. Intel architectures optimization reference manual, 2010.

[12] M. Kandemir, J. Ramanujam, M. Irwin, N. Vijaykrishnan, I. Kadayif, and A. Parikh. Dynamic management of scratch-pad memory space. In *Proceedings of the Design Automation Conference 2001*.

[13] R. E. Kessler and M. D. Hill. Page placement algorithms for large real-indexed caches. *ACM Transactions on Computer Systems*, 10, 1992.

[14] Q. Lu, J. Lin, X. Ding, Z. Zhang, X. Zhang, and P. Sadayappan. Soft-OLP: Improving hardware cache performance through software-controlled object-level partitioning. In *Proceedings of PACT 2009*.

[15] C. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. Reddi, and K. Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *Proceedings of PLDI 2005*.

[16] R. L. Mattson, J. Gecsei, D. R. Slutz, and I. L. Traiger. Evaluation techniques for storage hierarchies. *IBM Systems Journal*, 9(2), 1970.

[17] H. Miyazaki, Y. Kusano, N. Shinjou, F. Shoji, M. Yokokawa, and T. Watanabe. Overview of the K Computer system. *Fujitsu Scientific and Technical Journal*, 48, 2012.

[18] Q. Niu, J. Dinan, Q. Lu, and P. Sadayappan. PARDA: A fast parallel reuse distance analysis algorithm. In *Proceedings of IPDPS 2012*.

[19] F. Olken. Efficient methods for calculating the success function of fixed-space replacement policies. Technical report, Lawrence Berkeley Laboratory, 2009.

[20] S. Perarnau, M. Tchiboukdjian, and G. Huard. Controlling cache utilization of HPC applications. In *Proceedings of ICS 2011*.

[21] J. Reinders. *VTune Performance Analyzer Essentials*. Intel Press, 2005.

[22] T. Sherwood, B. Calder, and J. S. Emer. Reducing cache misses using hardware and software page placement. In *Proceedings of ISC 1999*.

[23] J. Treibig, G. Hager, and G. Wellein. LIKWID: A lightweight performance-oriented tool suite for x86 multicore environments. In *Proceedings of PSTI 2010*.

[24] M. Verma, S. Steinke, and P. Marwedel. Data partitioning for maximal scratchpad usage. In *Proceedings of ASP-DAC 2003*.

[25] Y. Zhong, X. Shen, and C. Ding. Program locality analysis using reuse distance. *ACM Trans. Program. Lang. Syst.*, 31 (6), 2009.