Toward Automated Cache Partitioning for the K Computer

Swann Perarnau[1] Mitsuhisa Sato[1,2]

Abstract: The processor architecture available on the K computer (SPARC64VIIIfx) features an hardware cache partitioning mechanism called *sector cache*. This facility enables software to split the memory cache in two independent sectors and to select which one will receive a line when it is retrieved from memory. Such control over the cache by an application enables significant performance optimization opportunities for memory intensive programs, that several studies on software-controlled cache partitioning environments already demonstrated.
Most of these previous studies share the same implementation idea: the use of page coloring and an overriding of the operating system virtual memory manager. However, the sector cache differs in several key points over these implementations, making the use of the existing analysis or optimization strategies impractical, while enabling new ones. For example, while most studies overlooked the issue of phase changes in an application because of the prohibitive cost of a repartitioning, the sector cache provides this feature without any costs, allowing multiple cache distribution strategies to alternate during an execution, providing the best allocation of the current locality pattern.
Unfortunately, for most application programmers, this hardware cache partitioning facility is not easy to take advantage of. It requires intricate knowledge of the memory access patterns of a code and the ability to identify data structures that would benefit from being isolated in cache. This optimization process can be tedious, with multiple code modifications and countless application runs necessary to achieve a good optimization scheme.
To address these issues and to study new optimization strategies, we discuss in this paper our effort to design a new cache analysis and optimization framework for the K computer. This framework is composed of a binary instrumentation tool to measure the locality of a program data structures over a code region, heuristics to determine the best optimization strategy for the sector cache and a code transformation tool to automatically apply this strategy to the target application. While our framework is still a work in progress, we demonstrate the usefulness of our locality analysis and optimization strategy on a handcrafted application derived from classical stencil programs.

Keywords: cache partitioning, reuse distance, binary instrumentation

1. Introduction

As the difference between memory and processor speeds continued to increase, optimizing a program locality has become 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 a program organization via static analysis.

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 [19], [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 [3] 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 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 [5], features such a cache partitioning facility called the sector cache. Although multiple works [2], [4], [14], [18], [21] 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 the specific goal to automate as much as possible the discovery and application of optimization opportunities in a target HPC application.

Our framework leverages and extends several existing methodologies. First, we use binary instrumentation of the target application to measure the locality (*i.e.* reuse distance) of major data structures in a code region. Then, 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. Finally, we modify automatically the application by source-to-source transformations to

[1] Riken Advanced Institute for Computational Science
[2] University of Tsukuba

configure and activate the sector cache according to our optimization scheme.

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 details our cache behavior analysis and optimization framework, while Section 5 validates the locality analysis on an handcrafted application. We conclude and discuss future work in Section 6.

2. Cache Partitioning on the K Computer

The K Computer — ranked second on the Top500 issue of June 2012 [16] — 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 [5]. 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 application.

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. This 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 ensure that a cache line retrieved for one sector cannot evict a cache line from 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 on the L2 cache.

The technical name for the hardware implementation of the sector cache is *instruction-based way partitioning*. To activate this partitioning, an unprivileged instruction specify 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 cache ensure that its sector is not 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 that was 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 for obvious simplicity reasons we will 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. In this case, any sector is allowed to use the remaining ways, causing the cache to always be used in full. Thus, it is not possible to limit the cache of the application by restricting it to a small sector, as the sector will grow above its configured size if no memory is loaded in the other sector.

Consequently, 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 $(1,11),(2,10),(3,9),...,(11,1)$.

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 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 is 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++)
        a[i] = 0;
        for(j = 0; j < N; j++)
            a[i] += b[i][j]*c[j];
}
```

Fig. 1 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 any instruction that touches a data structure assigned to sector 1 to be preceded by the special `sxar` instruction, tagging the memory access as belonging to sector 1. 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 touches 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

The most straightforward way of using this kind of software-controlled cache partitioning facility is to preserve from eviction data that the programmer knows for a fact will be used in a close future. Another possibility is to avoid cache thrashing by isolating in a small partition accesses without reuse. As a motivating example of the performance improvements that can be achieved by these optimizations strategies, one can look at the BLAS library provided on the K Computer. In particular, the level 3 DGEMM routine, known for being at the heart of the Linpack benchmark (used for the Top500), uses the sector cache to improve its data reuse by more than 12%. To perform its matrix multiply operation, it recursively splits its parameters into blocks and in the process, if the sizes are small enough, reduces cache thrashing by keeping one of them in sector 1.

This example also illustrates the issues we aim to address in this paper: without intricate knowledge of both the memory hierarchy of the SPARC64 VIIIfx processor and of the target code locality, it is currently tedious to make good use of the sector cache. Consequently, our goal is to design an automated framework that can analyze the locality of an application, identify the code regions that would benefit from the sector cache and modify them to use it efficiently. This type of framework would greatly simplify the optimization of HPC applications when ported or developed on the K Computer.

Most existing works on software-controlled cache partitioning didn't have access to an hardware implementation of it, resorting to a system-software solution based on page coloring [11]. Such solution's principle is based on the fact that with a physically-indexed way-associative cache, it is possible to control its availability to a virtual memory region by mapping the latter to specific physical pages (colors). Unfortunately, such solution suffers from several drawbacks. 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 is this setup. Indeed, it requires stopping the program and moving data from every physical page that needs it to another one. Among works that use page coloring and thus suffer from these issues we can cite Soft-OLP [14], ULCC [4] and CControl [18].

In regards to our goals, the Soft-OLP paper represents the closest work available. It describes a binary instrumentation framework to analyze the locality of objects (data structures) inside a program to latter better distribute the cache among them using a cache partitioning. The cache partitioning environment the author use allows for more than two partitions, resulting in a focus of the tool on grouping objects together inside partitions. To analyze the impact of such grouping, the authors define an *inter-object interference* metric, based on a sampling the amount of data references made to a data structure between accesses to another one. 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 and dynamically allocated objects, by reading at a very simple level the program symbol table and hijacking 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. Finally, our sector cache is only capable of splitting in two and it is unlikely that its size allows optimizations with more than one structure isolated at a given time in sector 1, more so with HPC applications using a significant amount of memory. 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 [4] and CControl [18] 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. The experiment is the following. First, create two partitions in cache: one containing a single structure (P_s), the other for the rest of the application (P_r). Then, run the application with the minimal size possible for P_r and varying the size of P_s. Since the cache available to the data structures inside P_r, the amount of cache they generate is constant at each run. Thus, any change in cache misses is the result of an improvement/worsening in the cache use of the isolated structure. The authors then uses this *working set analysis* to determine which structures would benefit the most from partitioning and optimize the application accordingly. 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. As we stated before it is not possible to configure the sector cache so that both partitions occupy less than the full cache. Consequently, while we could isolate a structure inside sector 1 and increase progressively the number of ways it uses, the sector 0 will always fill the rest of cache, creating variations of the amount of cache misses triggered by its data structures and making the experiment void in the process.

Farther from our objectives, Mowry *et al.* discussed in 1996 a compiler framework detecting cache partitioning opportunities in a parallel program and generating hints for a page coloring solution inside the operating system. This study was focused on multi-threaded programs working on shared arrays and the use of cache partitioning to isolate each thread's accesses to the arrays from the others'. While the use of cache partitioning for such purposes is interesting in itself, the sector cache and its only two partitions is a poor fit.

Finally, Wang *et al.* discussed in a series of papers [7], [8] issues regarding the locality of specific instructions in a program. Using reuse distance analysis the authors identify critical instructions in a program, responsible for most of the cache misses. They identify that these instructions possess multiple major reuse distances: i.e., the instruction generates several classes of accesses, at different distance from each other. They also identify that these classes correspond to phase change in the program or at least execution path differences. In other words, some critical instructions appear in multiple execution path and each path has its own locality pattern. A simple example of such instructions would be the ones in an utility function called from multiple sites in the program. While we expect to discover such kind of function/instruction in HPC applications, we chose to address these issues in a future work. One possible solution for this issue in our setup is to generate several versions of the function according to its different call sites or execution paths and to optimize each version independently. For the rest of this paper we will consider that the performance critical code is contained in functions with a single incoming path.

4. A Framework for Analysis and Optimization of Partitioned Programs

As stated in the introduction, our framework is composed of three phases. We first study the locality of a code region using binary instrumentation to create a trace of all memory accesses and apply reuse distance analysis to it. Then, the framework identify which of the data structures will be benefit from the sector cache by predicting the cache misses of a partitioning configuration. Finally we modify the application's source code to activate and apply partitioning in the relevant function. Such process can then be repeated for each code region generating a significant amount of cache misses.

In its current state 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 [20], 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 quadratically with the number of structures, making such solution expensive.

4.1 Reuse Distance Analysis by Binary Instrumentation

Numerous studies have demonstrated the use of the reuse distance as a measure of a program locality and a good predictor for cache misses [1], [22]. Traditionally, the reuse distance of a memory access is defined as the number of unique memory accesses separating it from the previous one touching the same location. First accesses to a location are considered to be of infinite distance. Assuming a fully-associative cache and a perfect LRU policy, if a memory access has a reuse distance greater than the size of the cache it will trigger a cache miss: the LRU policy necessarily evicted the data from the cache. While such cache model might seem unrealistic, the reuse distance as proven to be a good predictor of cache miss rates on commodity architectures. Generally, the reuse distance of all memory accesses in a program will be merged in a reuse distance histogram, giving insight about the number of cache misses triggered for any given cache size.

For the purpose of our analysis, we will however distinguish memory accesses in two categories: those touching a given data structure and those outside its address range. In other words, our objective here is to compute for each data structure two reuse distance histograms: the first one for accesses internal to the data structure and the second for all the others. Analyzing the former will allow us to predict the cache misses triggered by the structure if it was alone in a sector of the cache and the latter to predict the cache misses triggered by the rest of the program isolated in the other sector.

To trace all the memory accesses in a target program, we use the binary instrumentation framework Pin [15]. Schematically, this framework allows us to execute custom code each time an instruction results in a memory access by modifying the application binary at runtime. The complete process of this analysis is the following. Upon startup, the user provides us with information on the data structures to analyze and a code scope over which to perform this analysis. We define structure information as the name of the variable referencing it and, in the case the name can appear multiple times in the program, enclosing scope (i.e. enclosing function or object file). The code scope is either a function or a range of source code lines. With this information, our program compile a table of the data structures locations by reading the program's DWARF debugging information. Once each data structure's address range is known, the program instruments every instruction of the traced scope. During the target application execution, each memory access will trigger our tracing code, registering locality information about this memory access for all the data structures under study. We detail these two steps in the following subsections.

4.1.1 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. In particular, 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.

Using the structure information provided by the user, our tool recursively scans the whole tree, filtering nodes until the DIE of

each structure of interest is found. The DIE of a data structure contains information on its type and location. This information is then used by our tool to compute the address ranges corresponding to the structure. For example, a contiguous array will be described by the DWARF type `DW_tag_array_type` and will include the size of each of its dimensions, its basic type and a location expression. If the size of such array is then obvious, translating the location expression into the starting address of the structure can require work at runtime.

Indeed, the location expression is defined in DWARF as a list of operations to apply to an integer stack. After execution of all operations, the stack's top value is the virtual address researched. Runtime information might be required to execute these operations, as it can include pushing current machine register values on the stack. This is specially needed to represent parameters passed as pointer to a function, their location thus expressed as an offset to the current stack pointer. In the event the location expression cannot be resolved statically, our tool saves a representation of the expression to use during the target application run.

4.1.2 Reuse Distances Tracing

Once the DWARF information has been parsed, Pin is used to instrument each instruction of the tracing scope. By default, our code only instruments the instructions of the top-level scope, not the code that could be called from it. This enables us to reduce the number of instrumented instructions and greatly improves the slowdown of the application. It is still possible to instrument all called functions if the user requires it.

For each instrumented instruction triggering a memory access, our code collects the required register values (for location expressions) along with the address and the size of the memory access. It then iterates over the structure table, registering this event for all structures. If the memory address falls into a structure, the internal reuse distance of the access is computed, otherwise the reuse distance for the other accesses is found. In both cases the according histogram is updated.

To compute a reuse distance, our tool implement one of fastest algorithm known [6]. This algorithm relies on two data structures: an hash table saving for each memory location the last time it was accessed and a splay tree sorted by this time. The splay tree keeps track of all memory locations touched as well as for each node the number of left and right children. A recursive lookup through the tree can then be used to compute the number of locations touched since the last access to the current location.

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

4.2 Identifying Cache Optimization Opportunities

Once the tracing is completed, the binary instrumentation tool outputs each structure's histograms. From these histograms 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 tuples (pos, a) with $pos \in \mathbb{N}$ and $a \in M$. Let $A(s)$ the memory addresses of a structure s analysed by our experiments. We can now express the reuse distance as $h(m, d)$ as 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 it 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. We then determine the best configuration for each data structure and thus the best configuration overall.

4.3 Code Transformations for the Sector Cache

The next step of the process is the optimization of the application itself. With the knowledge of which sector cache configuration to apply, this requires inserting at the beginning of the tracing scope we just analyzed. To do so, we intend to use the XcodeML framework [25]. This framework provides a source-to-source transformation facility based on an intermediate code representation in XML. It supports transformation from and to C and Fortran code with an helper library to inspect, modify, remove and add code to the abstract tree of a target program. This framework was for example used to develop the Omni OpemMP compiler [12] and the XcalableMP parallel language compiler [13].

While adding the required directive at the top of a function might seem too simple to require the use of a full source-to-source transformation framework, our intend is to also support the recursive modification of successive function calls, each function requiring its own sector cache configuration setup. Such goal might for example require to detect that a data structure was passed as parameter to another function and that it changed its name in the callee. Since the directives provided by the compiler do not propagate to the called functions, this can be an important issue when optimizing an application.

5. Experimental Results

Our framework is still a work in progress. In particular, the code generation phase is not complete yet. Nevertheless, we designed a simple application to validate our binary instrumentation tool and the resulting locality analysis and optimization.

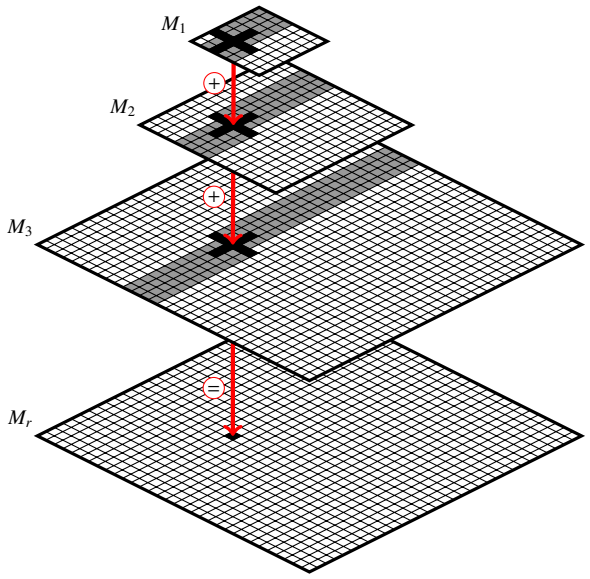

Fig. 2 Multigrid Stencil: 9 cells of matrices M_1,M_2 and M_3 are summed into a single cell of M_r. Grey areas represent cache requirements of each matrix.

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 (*working sets*) cannot fit all in cache, accesses to each matrix will thrash accesses to the others.

Figure 2 illustrate this stencil, with a 9 points cross being read in each 3 matrices to compute a single cell of the resulting matrix. Cache requirements of each matrix are also displayed. 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.

To validate our binary instrumentation tool we analyze the locality of the 3 input matrices of this program. We chose the matrices sizes so that M_3 requires 7MiB in cache so that a default cache configuration triggers numerous cache misses. The full program executes the stencil 10 times, after having initialized each matrix with random values. We applied our binary instrumentation, giving it the 3 M matrices' names and limiting the trace to one application of the stencil. Since Pin does not support the SPARC instruction set, this analysis was realized on another system, a Linux system with an Intel i7 2760 QM processor and 8 GiB of RAM. On average, the analysis induced an execution time of the application 200 slower. This value is on par with existing instrumentation methods [14].

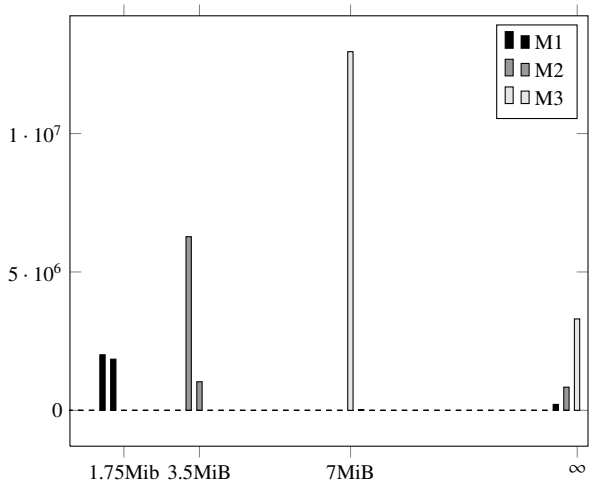

Fig. 3 Reuse distance histograms for the 3 input matrices of the multigrid stencil. For simplicity, we display distance bigger than a 12^{th} of the cache. Bins are 256KiB wide. The last bin is for infinite distances.

After analysis of these reuse distance histograms (**Fig. 3**), 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.

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

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 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. We also confirmed these results by testing other sizes for the matrices, and achieve a cache misses reduction up to 40%.

6. Conclusion and Future Works

We presented our design for an automatic analysis and optimization of HPC application regarding the use of a specific cache

partitioning facility available on the K Computer. 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 analysing independently multiple regions of code and determining the best configuration for each of them is a better strategy than whole program analysis. This approach is also expected to induce lower costs on the instrumentation of the target application, a process known to be particularly heavy.

While still a work in progress, we expect our framework to enable fast and easy optimization of HPC applications for the K Computer. Preliminary results indicate a possible improvement of the locality of application ranging from 12 to 40%. In the future, we intend to apply our framework to the full suite of the NAS Parallel Benchmarks, demonstrating its relevance to the optimization of complex applications.

Among the long term possibilities for this work, we believe 3 studies to be of particular importance. First, in its current state our framework is unable to resolve the address ranges of some variable if they are too complex, like for example a dynamically allocated multidimensional array or any function parameter whose size is not known statically. Both limitation arise from our use of the exclusive use of DWARF information for this purpose. We believe that this information could be combined to a syntactic analysis of the application's code. Indeed, it would be possible to prepare the binary instrumentation by identifying which instructions were generated from a source line accessing a data structure of interest. Second, the issue of code regions requiring multiple sector cache configurations according to the code path that precedes them could be solved either by a code transformation duplicating the code region to create independent instruction streams for each path or, in the simple case that only the size of each sector needs to be changed, by reconfiguring the cache at runtime depending on the current path. Third, we believe that in some case, an application might benefit from the sector cache being reconfigured between function calls. In other words, using the sector cache to preserve data during the call to a thrashing function. Detecting such cases would require to analyze very precisely the target code before, during and after the function call, while preserving information on the cache content between these scopes. In its current state, our framework might require additional automation in the analysis phase to allow this kind of studies.

Acknowledgments Part of the results were obtained by early access to the K computer at the RIKEN AICS.

References

[1] Beyls, K. and DHollander, E.: Reuse distance as a metric for cache behavior, *Proceedings of the IASTED Conference on Parallel and Distributed Computing and systems*, pp. 350–360 (2001).

[2] Bugnion, E., Anderson, J., Mowry, T., Rosenblum, M. and Lam, M.: Compiler-directed page coloring for multiprocessors, *ACM SIGOPS Operating Systems Review*, Vol. 30, No. 5, p. 255 (1996).

[3] Corporation, I.: Intel Architectures Optimization Reference Manual (March 2010).

[4] Ding, X., Wang, K. and Zhang, X.: ULCC: a user-level facility for optimizing shared cache performance on multicores, *Proceedings of the 16th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, (PPOPP)*, pp. 103–112 (2011).

[5] et al., T. M.: SPARC64 VIIIfx: A New-Generation Octocore Processor for Petascale Computing, *IEEE Micro*, Vol. 30 (2010).

[6] F., O.: Efficient methods for calculating the success function of fixed-space replacement policies, Technical report, Lawrence Berkeley Laboratory (2009).

[7] Fang, C., Carr, S., Önder, S. and Wang, Z.: Reuse-distance-based miss-rate prediction on a per instruction basis, *Proceedings of the 2004 workshop on Memory system performance*, MSP '04, New York, NY, USA, ACM, pp. 60–68 (2004).

[8] Fang, C., Carr, S., nder, S. and Wang, Z.: Path-Based Reuse Distance Analysis, *Compiler Construction* (Mycroft, A. and Zeller, A., eds.), Lecture Notes in Computer Science, Vol. 3923, Springer Berlin / Heidelberg, pp. 32–46 (2006).

[9] Hennessy, J. L. and Patterson, D. A.: *Computer Architecture: a quantitative approach*, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2nd edition (1996).

[10] Ida, K., Ohno, Y., Inoue, S. and Minami, K.: Performance Profiling and Debugging on the K computer, *Fujitsu Scientific and Technical Journal*, Vol. 48 (2012).

[11] Kessler, R. E. and Hill, M. D.: Page Placement Algorithms for Large Real-Indexed Caches, *ACM Transactions on Computer Systems*, Vol. 10, pp. 338–359 (1992).

[12] Kusano, K., Satoh, S. and Sato, M.: Performance Evaluation of the Omni OpenMP Compiler, *High Performance Computing* (Valero, M., Joe, K., Kitsuregawa, M. and Tanaka, H., eds.), Lecture Notes in Computer Science, Vol. 1940, Springer Berlin / Heidelberg, pp. 403–414 (2000).

[13] Lee, J. and Sato, M.: Implementation and Performance Evaluation of XcalableMP: A Parallel Programming Language for Distributed Memory Systems, *ICPP Workshops*, pp. 413–420 (2010).

[14] Lu, Q., Lin, J., Ding, X., Zhang, Z., Zhang, X. and Sadayappan, P.: Soft-OLP: Improving Hardware Cache Performance through Software-Controlled Object-Level Partitioning, *Proceedings of the 18th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pp. 246–257 (2009).

[15] Luk, C., Cohn, R., Muth, R., Patil, H., Klauser, A., Lowney, G., Wallace, S., Reddi, V. and Hazelwood, K.: Pin: building customized program analysis tools with dynamic instrumentation, *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, pp. 190–200 (2005).

[16] Meuer, H., Strohmaier, E., Simon, H. and Dongarra, J.: 39th Release of the TOP500 List of Fastest Supercomputers (2012).

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

[18] Perarnau, S., Tchiboukdjian, M. and Huard, G.: Controlling Cache Utilization of HPC Applications, *International Conference on Supercomputing (ICS)* (2011).

[19] Rajeshwari, B., Stefan, S., Bo-Sik, L., M., B. and Peter, M.: Scratchpad memory: design alternative for cache on-chip memory in embedded systems, *Proceedings of the tenth international symposium on Hardware/software codesign*, CODES '02, New York, NY, USA, ACM, pp. 73–78 (2002).

[20] Reinders, J.: *VTune Performance Analyzer Essentials*, Intel Press (2005).

[21] Sherwood, T., Calder, B. and Emer, J. S.: Reducing cache misses using hardware and software page placement, *Proceedings of the 13th International Conference on Supercomputing*, pp. 155–164 (1999).

[22] Snir, M. and Yu, J.: On the theory of spatial and temporal locality, Technical report, University of Illinois at Urbana-Champaign (2005).

[23] Treibig, J., Hager, G. and Wellein, G.: LIKWID: A lightweight performance-oriented tool suite for x86 multicore environments, *Proceedings of PSTI2010, the First International Workshop on Parallel Software Tools and Tool Infrastructures*, San Diego CA (2010).

[24] Verma, M., Steinke, S. and Marwedel, P.: Data partitioning for maximal scratchpad usage, *Proceedings of the 2003 Asia and South Pacific Design Automation Conference*, ASP-DAC '03, New York, NY, USA, ACM, pp. 77–83 (2003).

[25] XcalableMP/Omni Compiler Project: XcodeML: Compiler intermediate code in XML, `http://www.hpcs.cs.tsukuba.ac.jp/omni-openmp/xcodeml/`.