Identifying Working Data Set of Particular Loop Iterations for Dynamic Performance Tuning

Yukinori Sato*§, Hiroko Midorikawa†§ and Toshio Endo‡§

*Research Center for Advanced Computing Infrastructure, Japan Advanced Institute of Science and Technology (JAIST)
†Graduate School of Science and Technology, Seikei University, Japan.
‡Global Scientific Information and Computing Center, Tokyo Institute of Technology, Japan
§JST CREST, Japan

***Abstract*—Improving data locality and cache hit rates are key to obtain performance gain in modern computer systems. While loop tiling or blocking is a technique that can enhance spatial and temporal locality of data accesses, it is hard for compilers to fully automate it. This might be because sometimes the number of iterations of a loop is decided at runtime based on the size of input data or static dependency checking is too conservative for such loop transformation. Therefore, currently most of them are done by expert HPC programmers instead of compilers or advanced tools that guide and support them. In this paper, we present a mechanism for identifying the working data set of a particular loop iteration space using a dynamic binary translation system. From preliminary evaluation results using a benchmark program, we demonstrate how we measure the locality of data accesses via our mechanism and how we guide blocking strategies of data accesses. Also, we discuss the applicability of our mechanism to dynamic performance tuning mechanism in a dynamic compiler framework performed by a binary translation system.**

I. Introduction

One of the most challenging issues for designing and realizing future exascale supercomputer systems including the development of application programs for them is to handle the performance gap between CPU and memory. This gap is widely called memory wall and caused by the fact that the growth of memory bandwidth, capacity and latency has been slower than that of CPU throughput. While emerging new memory technologies such as NVRAM and 3D integration are expected to moderate the memory wall, these will lead to multi-level caches and deeper memory hierarchies to keep relevant data for a program to the faster but smaller memories close to the processing logic [1].

Managing data locality in the light of demands from application programs and the characteristics equipped in the each memory device is an essential for making fully use of such complex memory hierarchies. Traditionally, cache memories attempt to handle data locality by hard-wired logic based on the locality of memory accesses. However, conventional cache mechanisms are not an universal mechanism good for everything. It has been found that cache-management policies implemented by hardware are not effective in some cases, especially in scientific code that has large data structures such as 3-dimensional arrays [2].

Loop tiling, or also known as blocking, has been performed to the application codes especially in HPC field in order to compensate the weakness of the data locality management mechanism of cache memories. Here, loop tiling is a transformation that tailors an application's working data set to fit it in the memory hierarchy of the target machine. By reorganizing a loop nest structure and choosing tile size to allow the maximum reuse for a specific level of memory hierarchy, it optimizes locality of code for targets. Since this can be used for different levels of memory hierarchy such as physical/virtual memory, caches , registers, this is a powerful technique that can manage and enhance spatial and temporal locality of data accesses. Also, multi-level tiling can be used to achieve locality in multiple levels of the memory hierarchy simultaneously [3], so it is inevitable for a memory subsystem realized by different types of devices such as SRAM, DRAM, NVRAM.

However, it is hard for compilers to fully automate it because it often needs runtime information that cannot be used at compile time. For example, sometimes the number of iterations of a loop is decided at runtime based on the size of input data. Or, static dependency checking performed by compilers might be too conservative for loop transformation. As some of memory blocks are allocated at runtime, changing the data layout to match the data access pattern is difficult. Additionally in current mainstream multicore CPUs, the last level cache is shared among cores in the CPU, so there are some resource conflicts among co-running processes or threads and these cannot find until these codes run together. Due to the factors discussed above, compiler cannot generate highly optimized codes desired by the production HPC field and currently most of loop transformations are done by expert HPC programmers instead of compilers or advanced tools that guide and support them.

Therefore, tools or mechanisms that enable automated mapping and tuning of programs toward a particular target machine are strongly requested for system software and compiler technologies. The situation that programmers are

explicitly restructuring codes to perform well on particular memory subsystems leads to machine-specific programs. However, creating programs that are specific to a particular machine should be avoided in the view of portability and productivity. Instead, the programmers should write machine-independent code, and then low-level mechanisms should generate a specialized code for the target machine transparently from programmers. Future exascale machine designs are certain to have increasingly complex memory hierarchies, sophisticated memory-management strategies conducted by low-level mechanisms other than traditional compilers should be developed.

In this paper, we present a mechanism for identifying the working data set of a particular loop iteration space using a dynamic binary translation system. We present how we measure the locality of data accesses via our mechanism. Our primary motivation of this work is to use our analysis to guide performance tuning or optimization. Our mechanism could help identify memory access locality amenable to tuning or optimization, and guide what kind of strategies to use. This would also be useful for guiding a manual tuning done by expert programmers who must re-write their codes for performance optimization. Also, implementing them on an on-the-fly binary translation mechanism has potential to be applied as dynamic optimization and parallelization techniques.

II. Monitoring working data set

In order to monitor the working data set at runtime, we extend our work on whole program data dependence profiling implemented in a dynamic binary translation system [4]. Here, we briefly explain how we profile memory dataflow on our system using the concept of paging.

We start analysis on dynamic binary translation system from the static analysis phase. Here, we formulate control flow graphs and check memory access instructions in the code. To keep track of dynamic behaviors such as memory access and control flow transitions, we prepare analysis codes. At this phase, we generate markers that point out where analysis codes are to be instrumented, and these are instrumented into the original binary codes. Since we instrument analysis code when the binary code image is loaded, our profiler can accept any executable binary code without specific support for profiling.

After the instrumentation, we run the instrumented binary code and start the dynamic analysis phase. We monitor memory access and dynamic data dependencies together with the dynamic context of loop and call activations. To realize low overhead data dependence profiling, we focus only on read after write dependencies, also called true dependencies, in whole program executions.

To monitor true dependencies efficiently, we maintain who writes the most recent value in each accessed address using a locality-aware structure referred to as a lastWrite table. In order to realize the faster access to lastWrite tables and implement them with smaller memory size, we prepare a fixed sized block for one lastWrite table. This concept is similar to pages in virtual memory system. In virtual memory system, physical memory space is relocated as a set of fixed sized block (also called a page) [5]. This concept allows on-demand allocations of tables required for actual memory access patterns. Since a new lastWrite table is allocated on-demand when an untouched memory segment is newly accessed, we can save the total amount of memory compared with the implementation that allocates tables for all memory address space in advance.

In this paper, we apply the concept of the paging of lastWrite memory access tables to monitoring working data sets of actual executions. Here, the working data set is defined as the amount of memory for data that the execution requires in a given time interval. Then, we use the number of allocated lastWrite tables as an indicator for the working data set. Since a unit of a working data set becomes a criterion for measuring locality of memory accesses, we improve the fixed paging structure for lastWrite memory access tables presented in [4] by allowing the size of each page to be parameterizable.

By specifying the appropriate page size for each memory hierarchy, we measure the locality that can fit in it. If all accesses are referenced to the same working data set page or only few pages, then we can consider that these accesses have locality within these pages. Otherwise, memory is discretely accessed from wide ranges of memory address space, and their access pattern is considered not to have much locality.

Also, we make use of trip counts and appearance counts of loop regions to monitor the working data set in a particular interval. Here, the appearance count of a loop region represents how many times the loop is activated from outside of the loop regions. The trip count of a loop region represents how many times the loop is iterated from the beginning of the loop activation, and the trip count is initialized when the loop region activated from the beginning again. By making use of these counts, we realize a mechanism for identifying the working data set of a particular loop iteration space.

Figure 1 shows the way how we keep track of the working data set during a given interval. Here, we monitor the working data set using a hashTable, allocated lastWrite tables, linked lists. The working data set is identified as follows: When we encounter an instrumented memory operation, a key for the hashTable is generated using its effective address (EA) of the memory operation. Accessing the hashTable using the key, we obtain a pointer to the linked list, which records a pointer to a lastWrite table from an entry of the hashTable. Also, an element of the linked list contains a upper M bit field of EA. By comparing M bit in an element and the original one, we resolve the corresponding lastWrite table for the memory operation. Finally, the N bit field

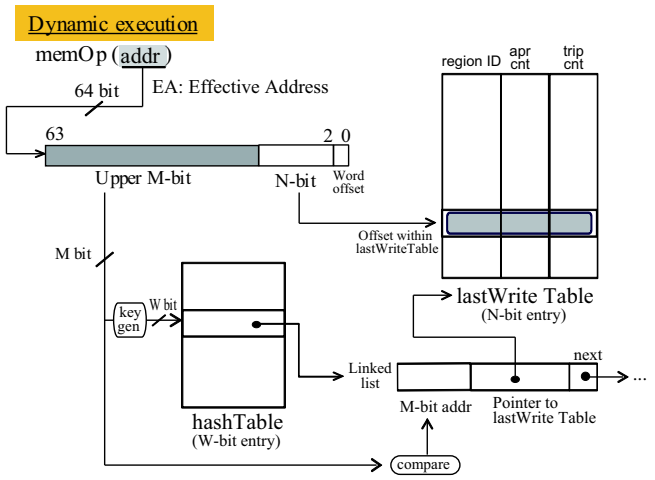

Figure 1. Our mechanism for identifying the working data set of a particular loop iteration space.

```
1  for(n=0 ; n<nn ; ++n){
2
3    for(i=1 ; i<imax-1 ; i++)
4      for(j=1 ; j<jmax-1 ; j++)
5        for(k=1 ; k<kmax-1 ; k++){
6          s0 = a[0][i][j][k] * p[i+1][j  ][k  ]
7            + a[1][i][j][k] * p[i  ][j+1][k  ]
8            + a[2][i][j][k] * p[i  ][j  ][k+1]
9            + b[0][i][j][k] * ( p[i+1][j+1][k  ] - p[i+1][j-1][k  ]
10                             - p[i-1][j+1][k  ] + p[i-1][j-1][k  ] )
11           + b[1][i][j][k] * ( p[i  ][j+1][k+1] - p[i  ][j-1][k+1]
12                             - p[i  ][j+1][k-1] + p[i  ][j-1][k-1] )
13           + b[2][i][j][k] * ( p[i+1][j  ][k+1] - p[i-1][j  ][k+1]
14                             - p[i+1][j  ][k-1] + p[i-1][j  ][k-1] )
15           + c[0][i][j][k] * p[i-1][j  ][k  ]
16           + c[1][i][j][k] * p[i  ][j-1][k  ]
17           + c[2][i][j][k] * p[i  ][j  ][k-1]
18           + wrk1[i][j][k];
19
20         ss = ( s0 * a[3][i][j][k] - p[i][j][k] ) * bnd[i][j][k];
21         wrk2[i][j][k] = p[i][j][k] + omega * ss;
22       }
23
24   for(i=1 ; i<imax-1 ; ++i)
25     for(j=1 ; j<jmax-1 ; ++j)
26       for(k=1 ; k<kmax-1 ; ++k)
27         p[i][j][k] = wrk2[i][j][k];
28
29 } /* end n loop */
```

Figure 2. The outline of the computation kernel of the Himeno Benchmark.

of an EA are used as an offset to access an entry within the lastWrite table. Here, the allocated lastWrite table itself corresponds to a page for a working data set, in other words, the number of allcated page tables or all the elements of linked lists represents the working data set.

The key for the hashTable is generated by the following equation:

$$key = M\%(1 << W) \quad (1)$$

Here, the hashTable entry size is represented as W bit, the lastWrite table entry size is represented as N, and the M bit field is calculated by $M = EA >> (N+2)$. Also, all of memory accesses are set to be monitored in the granularity of 4 bytes.

In order to change the page unit of working data set, we reconfigure the size of a hashTable and that of a lastWrite table. Here, we make the size of each page to be chosen from 256 Byte, 4 kByte and 64 kByte, and the size of pages chosen as a parameter is equal to the 2^{N+2}. We selected these sizes assuming to be a page or a cache line size for future memory hierarchies. Also, we set the W to 16 in this work.

Together with the working data set size, we monitor trip counts and appearance counts of loop regions to specify a particular interval using them. An element of a lastWrite table is composed of region ID, two counters for recording the number of appearances and trip counts of the region. These counters are represented as apr cnt and trip cnt in the Figure 1. Using these runtime information, we identity a particular loop iteration space where the working data set is monitored. Also, we keep track of dependencies among iterations and appearances using LCCT+M representation as discussed in [4].

III. Experimental framework

A. Methodology

We implement our dynamic data dependence profiling using the Pin tool set [6]. Pin is a well-known dynamic binary translation system that provides the same ISA translation applicable to dynamic binary optimization and parallelization.

To verify and evaluate our method, we use the Himeno Benchmark, which is widely known as a program that requires large memory bandwidth. Later in this Section, we briefly explain the overview of this benchmark.

As compiler tool sets, we use both of the GNU Compiler Collection 4.1.2 for x86_64 Redhat linux and the Intel C++ Compiler 11.1. Here, we compile the codes with '-O3 -g' option. We run our system on a single node of Appro gB222X-SM32 cluster servers, which is composed of two Intel Xeon X5570 CPUs, 24GB memory, Red Hat Enterprise Linux 5.4. The other detail parameters are similar to the ones in [4].

B. The Himeno benchmark

The Himeno benchmark, which originally developed by Dr. Ryutaro Himeno, measures performance in solving the Poisson equation using the Jacobi iterative method appeared in incompressible fluid analysis code [7]. Since this benchmark is known to be highly memory intensive and bound by memory bandwidth [8], this has grown in popularity and has been used by the HPC community to evaluate the worst-case performance for bandwidth intensive codes [9] [10].

Figure 2 shows the main solver which applies a 19-point stencil computation to the 3D array *p*. This main solver is composed of one outermost loop indexed by *n* for the Jacobi iterations, and two triply-nested loops indexed by *i, j, k*. Here, *p* is pressure, and it becomes the output of

Table I
THE NUMBER OF WORKING DATA SET PAGES DURING GIVEN LOOP ITERATIONS.

(a) gcc.4.1.2 with '-O3 option'

loopID	Analysis window	# of working set pages 64kB	4kB	256B
-	all	3669	58411	932669
8	apr=1, itr=1	3589	57180	896233
9	apr=1, itr=1	47	522	8134
10	apr=1, itr=1	17	26	106
11	apr=1, itr=1	17	18	23

(b) icc.11.1 with '-O3 option'

loopID	Analysis window	# of working set pages 64kB	4kB	256B
-	all	3674	58420	932704
9	apr=1, itr=1	3605	57209	896279
10	apr=1, itr=1	45	521	8133
11	apr=1, itr=1	16	19	104
13	apr=1, itr=1	16	18	22

Table II
THE NUMBER OF INCREMENTED WORKING DATA SET PAGES FROM THE PREVIOUS ITERATION.

(a) gcc.4.1.2 with '-O3 option'

loopID	Analysis window	Incremented page counts 64kB	4kB	256B
8	apr=1, itr [2-1]	0	0	0
9	apr=1, itr [2-1]	28	454	7104
10	apr=1, itr [2-1]	0	1	63
11	apr=1, itr [2-1]	0	0	0

(b) icc.11.1 with '-O3 option'

loopID	Analysis window	Incremented page counts 64kB	4kB	256B
9	apr=1, itr [2-1]	0	0	0
10	apr=1, itr [2-1]	29	454	7104
11	apr=1, itr [2-1]	1	7	64
13	apr=1, itr [2-1]	0	0	7

this computation, and all of data in arrays is represented in single precision floating point format. The body of this computational kernel originally involves 34 floating point calculations. Based on this, the performance of this benchmark is measured in FLOPS (FLoating-point Operations Per Second), where the total number of floating point operations is divided by the execution time.

In this experiment, we use Himeno benchmark version 3.3 (C, static allocate) and its medium size data sets for our evaluation.

IV. PERFORMANCE EVALUATION

First, we profile the whole program execution of Himeno benchmark executables. Table I shows the number of working data set pages obtained in this evaluation. The row indexed by 'all' in the column of Analysis window represents the working data set size of the whole program execution. Here, we measure working data sets for different page sizes from 256 Byte to 64 kByte.

From the results, we find that the working data set size of the whole program execution is about 229 MB when assuming 64 kB pages, which can be calculated by multiplying the number of pages with the page size. This is almost equal to the the maximum allocated memory size provided by operating system, which is the VmHWM size at `/proc/<pid>/status`.

Next, we monitor working data sets of particular loop iterations based on the visualized LCCT+M graphs as shown in Figure 3 and Figure 4. Here, the circles represent loops, the boxes represent procedure calls. Dynamic control flows of L-CCT are represented in solid (black) lines. So, an outer loop become a parent node and an inner loop become a child node of this graph. Here, we represent loop regions with their own ID number called loopID. In addition to call and loop context flows, dynamic data dependencies between nodes are represented in arrows with dashed lines. In each node, the percentage of the total cycles accumulated over all of its successors, and inside the parenthesis the percentage of the cycles executed in the node itself are represented, respectively. Also, we can find the # of appearances of each node, and the average # of loop iterations is outputted if the node is a loop.

Table I also shows the number of working data set pages for particular loop iterations. Using the loopID of the loop region and the number of appearances (apr), the number of iterations (itr) within a loop region, we represent regions of interests for monitoring working data set analysis. These are represented at the columns of loopID and Analysis window.

Here, we set the intervals of measurement to the first one iteration of the outermost loop and each of triply-nested loops in the Himeno benchmark code. Here, the outermost loop *n* is represented loopID=8 and the triply-nested loops *i*, *j*, *k* are represented loopID=9, 10 and 11 in the gcc.4.1.2. In the icc.11.1 case, these loops are represented loopID=9, 10, 11 and 13, respectively. Also, we only focus on monitoring working data set at the first iteration of the first appearance of each loop.

It is observed that there are not so much difference of memory access locality between gcc 4.1.2 and icc.11.1 while the FLOPS score of the icc code is 2.9 times higher due to its aggressive optimization. We observe that the loop structure generated by icc 11.1 is converted to achieve SIMDization and the innermost loop is divided into a few loops. Here, we monitor the largest one (loopID=13) in the innermost loops.

It is also observed that there are not so much differences of necessary pages among all page sizes when we execute one iteration of the innermost loop (loopID=11 of gcc 4.1.2 or loopID=13 of icc 11.1). These imply that there are little spatial locality inside an innermost loop iteration. In the loops that are next level of the innermost toward outer, the number of pages is increased only in 256 B case, so these

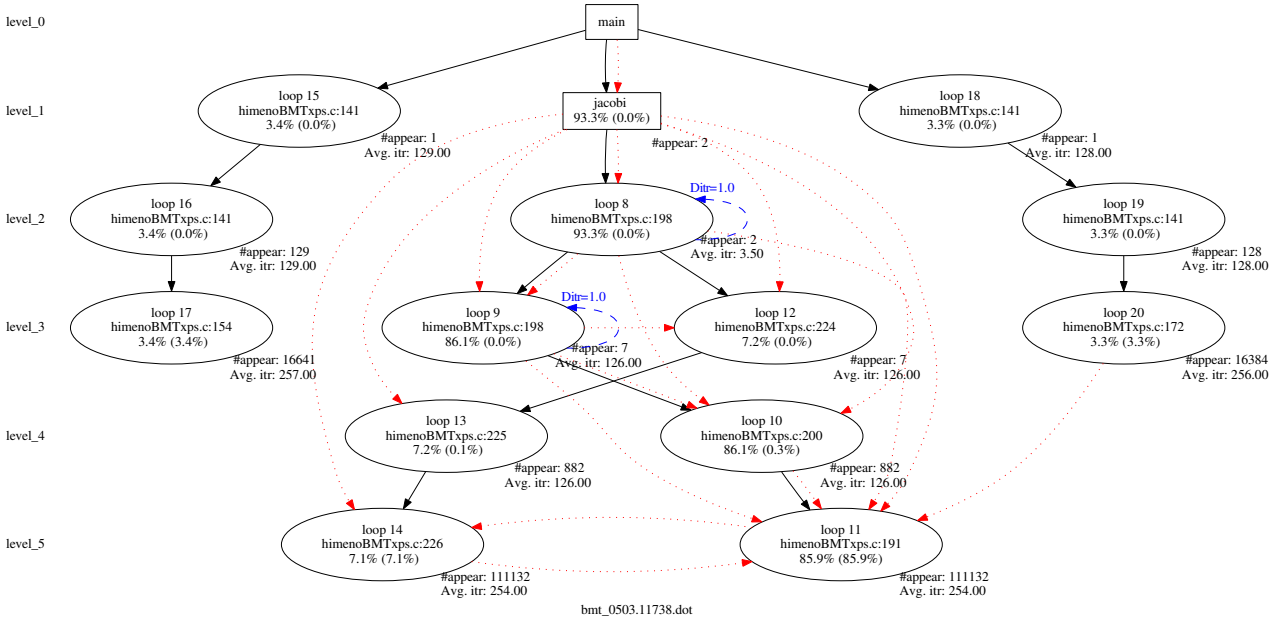

Figure 3. The LCCT+M generated by gcc4.1.2.

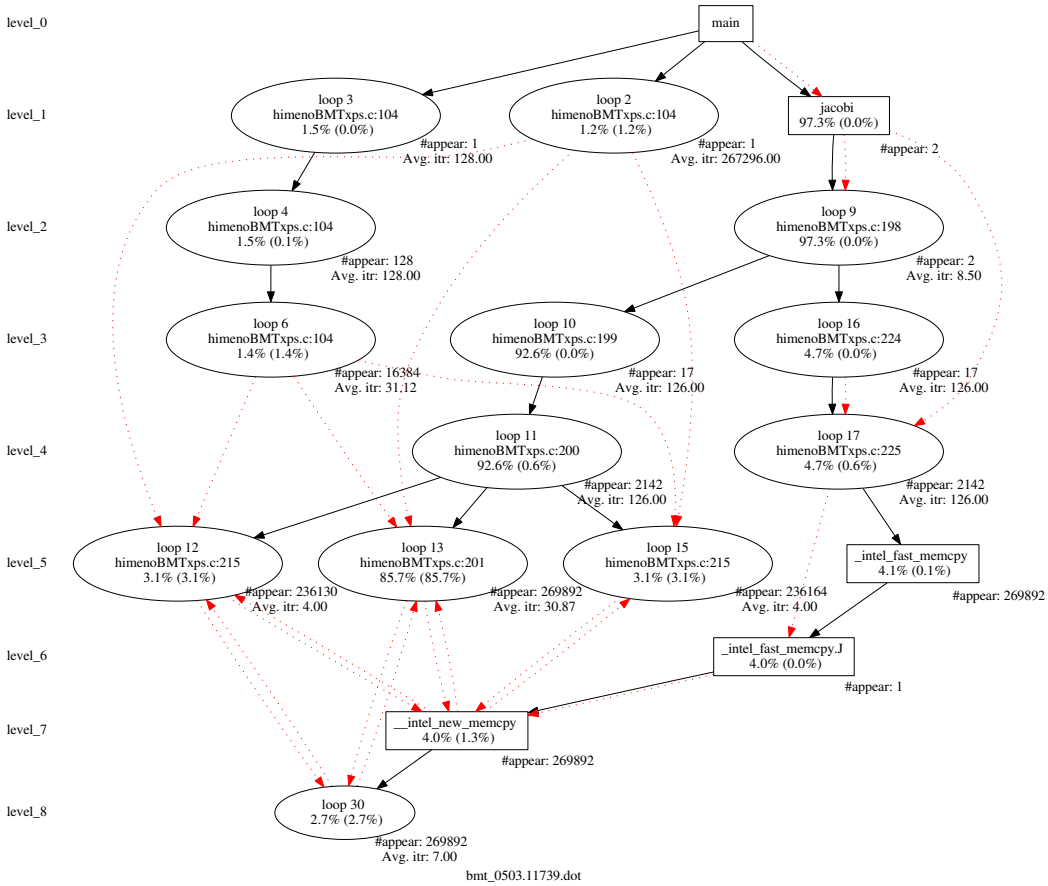

Figure 4. The LCCT+M generated by icc11.1.

imply there are spatial locality that fits in larger pages.

In order to investigate temporal locality, we measure the increments of pages from the previous iterations. Table II shows the number of incremented working data set pages. Here, the itr[2-1] in Analysis window column represents that we measure the increments from the first iteration to the second iteration. From the results, we find that the all increments are smaller than the case that the number of pages required for executing an iteration is doubled. Therefore, we can find there are reuse of data within the already assigned pages.

Based on the obtained spatial and temporal locality information, we can make strategies of loop tiling or blocking. For example, if the number of pages or the size of working data set is greater than these that can be available in the memory hierarchy, we should perform loop tiling to fit these data into the space. In the case of cache, we should focus on the 256 B page size and care about the interval when the number of pages dramatically increased. Based on these information and the size of each cache from L1 to L3, we can make use of locality of references by deciding the blocking factors or each tile size fitting to them.

We believe these information can be applied to performance tuning of real application codes. Combined with a mechanism of a dynamic compilation framework performed by a binary translation system, we would like to enhance the potential of automated performance tuning and optimization of code.

V. Conclusions

In this paper, we have discussed that loop tiling or blocking is a technique that can enhance spatial and temporal locality of data accesses and the working data set of the actual execution could help these transformation. Then, we have presented a mechanism for identifying the working data set size of a particular loop iteration space using a dynamic binary translation system. Using the Himeno benchmark, we have demonstrate how we measure the temporal and spatial locality of data accesses via our mechanism.

Current and future work includes applying the working data set analysis toward loop tiling on the actual code, exploring methods for automating the application of such transformation on a binary translator, and studying how our approach can be improved to apply real problems in the HPC field.

References

[1] P. Kogge *et al.*, "Exascale computing study: Technology challenges in achieving exascale systems," DARPA, Tech. Rep., 2008.

[2] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Addison Wesley, 2006.

[3] M. S. Lam and M. E. Wolf, "A data locality optimizing algorithm," *SIGPLAN Not.*, vol. 39, no. 4, pp. 442–459, Apr. 2004.

[4] Y. Sato, Y. Inoguchi, and T. Nakamura, "Whole program data dependence profiling to unveil parallel regions in the dynamic execution," in *2012 IEEE International Symposium on Workload Characterization (IISWC2012)*, Nov. 2012, pp. 69–80.

[5] D. A. Patterson and J. L. Hennessy, *Computer Oganization and Design , 2nd ed.* Sun Francisco: Morgan Kaufmann Publishers, 1997.

[6] C.-K. Luk *et al.*, "Pin: building customized program analysis tools with dynamic instrumentation," in *PLDI2005*, pp. 190–200.

[7] http://accc.riken.jp/2444.htm.

[8] E. Phillips and M. Fatica, "Implementing the himeno benchmark with CUDA on GPU clusters," in *IEEE Int'l Symp. on Parallel Distributed Processing*, 2010, pp. 1–10.

[9] S. Matsuoka, T. Aoki, T. Endo, A. Nukada, T. Kato, and A. Hasegawa, "GPU accelerated computingfrom hype to mainstream, the rebirth of vector computing," in *SciDAC 2009, Journal of Physics: Conference Series 180*, 2009.

[10] Y. Sato, Y. Inoguchi, W. Luk, and T. Nakamura, "Evaluating reconfigurable dataflow computing using the himeno benchmark," in *2012 International Conference on Reconfigurable Computing and FPGAs (ReConFig)*, Dec. 2012, pp. 1 –7.