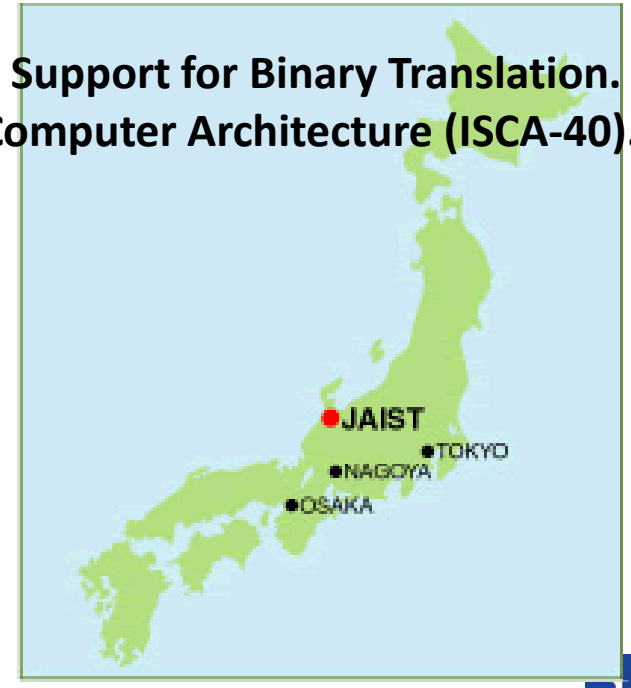
Identifying Working Data Set of Particular Loop Iterations for Dynamic Performance Tuning

Yukinori Sato (JAIST / JST CREST)
Hiroko Midorikawa (Seikei Univ. / JST CREST)
Toshio Endo (TITECH / JST CREST)

**6th Workshop on Architectural and Microarchitectural Support for Binary Translation.
Held in conjunction with the 40th Int'l Symposium on Computer Architecture (ISCA-40).
June 24, 2013**



Introduction

"**Memory Wall**" is one of the most challenging issues

- For developing future exascale supercomputer systems
  - To develop application programs is also important (= co-design)
- Emerging new memory technologies are expected to moderate the memory wall
  - Such as NVRAM and 3D integration
  - But, these will lead to multi-level caches and deeper memory hierarchies

Anyway, we have to keep relevant data for a program to the faster but smaller memories close to the processing logic

Managing **data locality** based on the characteristics of each memory device and the behavior of application programs is essential

Managing data locality

- Cache memory is not an universal mechanism
  - cache-management policies are not effective especially in scientific code that has large data structures such as 3D arrays
- Loop tiling (or also known as blocking) has been performed to the application codes especially in HPC field
  - Loop tiling is a transformation that tailors an application's working data set to fit it in the memory hierarchy of the target machine
  - Loop tiling
    - can manage spatial and temporal locality by reorganizing a loop nest structure or choosing an appropriate tile size
    - can be used for different levels of memory hierarchy such as physical/virtual memory, caches, registers, and different types of devices such as SRAM, DRAM, NVRAM

      ⇒ Maximizing reuse for a specific level of memory hierarchy
    - can compensate the weakness of cache memories.

However, it is hard for compilers to fully automate **loop tiling** because it often needs runtime information that cannot be used at compile time

Mapping and tuning of programs toward target machine

Compiler cannot generate highly optimized codes desired by HPC field

Due to the lack of runtime info.

- the # of iterations of a loop depends on the size of input data
- static dependency checking might be too conservative
- Effects of runtime memory allocation
- Effects by co-running processes or threads

Most of code transformations are done by expert HPC programmers

- Explicitly restructuring leads to machine-specific code
- These should be avoided in the view of portability and productivity
- Instead, using machine-independent code as an input, low-level mechanisms should generate specialized code transparently

Current obstacles: These are not "**automated**"

We need **system software and compiler technologies** that enable **automated** mapping and tuning of programs toward a particular target machine

Future exascale machine: increasingly complex memory hierarchies

Need automated and sophisticated memory-management strategies

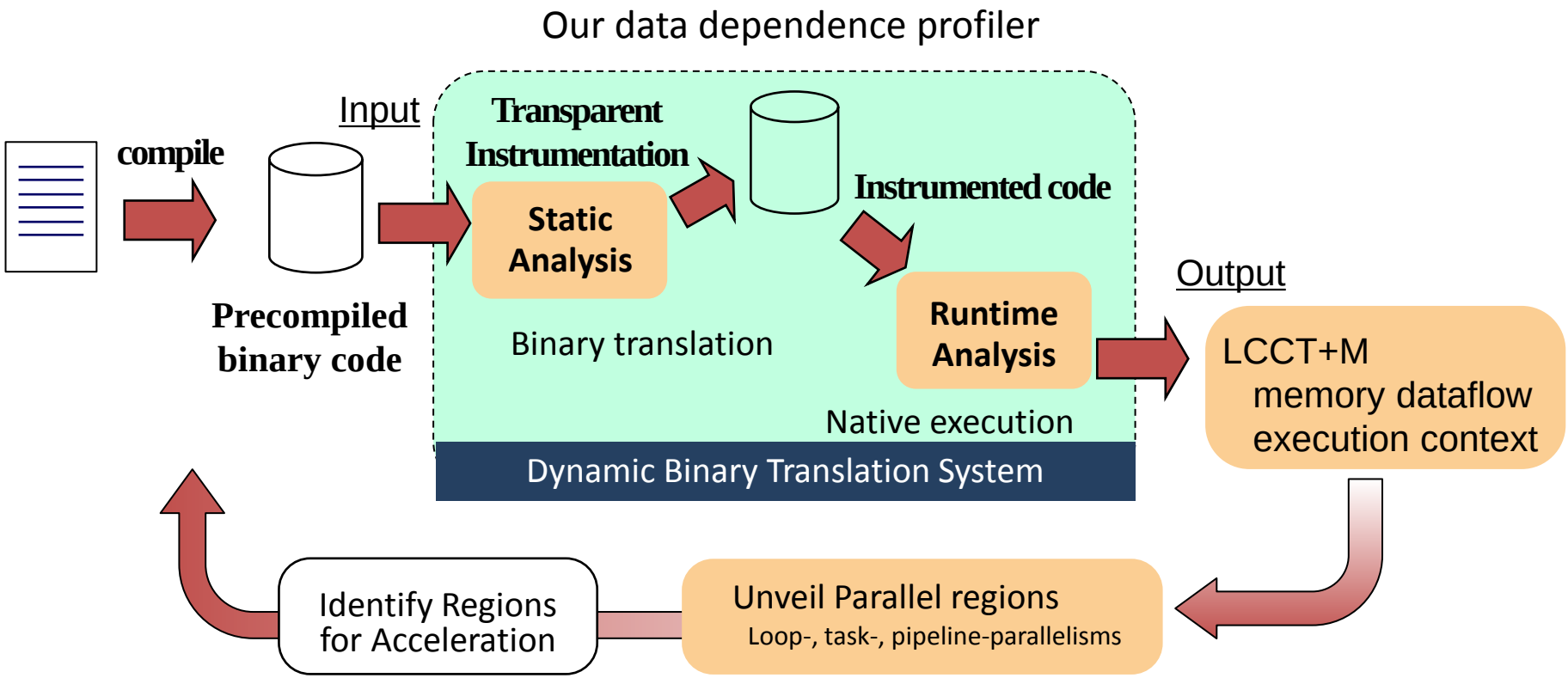
In this paper

We present a mechanism for identifying the ***working data set*** of a particular loop iteration space using a DBT system

- Our mechanism could help identify **memory access locality**
  - Amenable to tuning or optimization
  - Also, 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.
- Implemented on an binary translation mechanism
  - Based on **data dependence profiling** using the LCCT+M [Sato, IISWC2012]
  - Our mechanism has potential to be applied to dynamic optimization and parallelization

[Sato, IISWC'12] Yukinori Sato, et al. Whole Program Data Dependence Profiling to Unveil Parallel Regions in the Dynamic Execution. 2012 IEEE International Symposium on Workload Characterization

An overview of our data dependence profiling



- Using executable binary code as input, we start analysis on DBT system
- At static analysis phase, we insert analysis codes for dynamic analysis
- At runtime analysis phase, the instrumented binary code is executed
  - Keeping track of the dynamic context of loop and call activations
  - Monitoring data dependencies using memory access table with paging

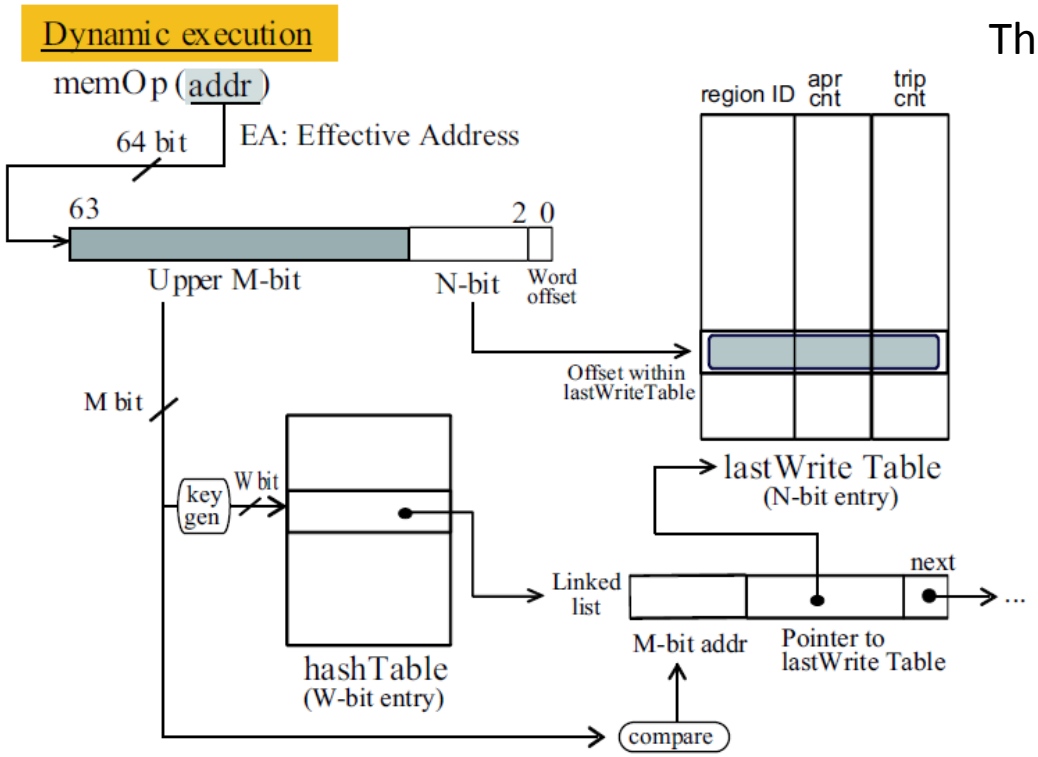
Monitoring working data sets of actual executions

We apply the concept of the **paging** of memory access tables

- ***The working data*** set is defined as the amount of memory for data that the execution requires in a given time interval.
- We use the number of allocated **lastWrite tables** as an indicator for the working data set.
  - **The lastWrite tables** are a mechanism for effectively monitoring memory accesses presented in [Sato, IISWC2012]
  - We make the size of each page of the tables to be parameterizable
    - This is because a unit of a working data set becomes a criterion for measuring locality of memory accesses
- We make use of **trip counts** and **appearance counts** of loop regions to monitor the working data set in a particular interval.

By specifying the appropriate page size for each memory hierarchy and intervals, we measure the locality that can fit in it.

How we keep track of the working data set in a given interval



The working data set is identified as follows:

- Generate a key for the hashTable using its effective address (*EA*) of the memory operation

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

- Access hashTable and obtain a pointer to the linked list
- Compare *M* bit in an element and the original one, and obtain the corresponding lastWrite table for the memory operation.
- Finally, the *N* bit field of an EA is used as an offset to access an entry within the lastWrite table.

The allocated lastWrite table itself is regarded as **a page** for a working data set

- We make the size of each page to be chosen from 256 Byte, 4 kB, 64 kB
  - These sizes are selected assuming future memory hierarchies (the size of a page/ cache line)
- To specify an interval , the **apr cnt** and **trip cnt** of loop regions are monitored

** Here, all of memory accesses are set to be monitored in the granularity of 4 bytes.
** Also, we set the W to 16 in this work.

4. Experiment

We implement our mechanism on Pin tool set, and we use the Himeno Benchmark to verify and evaluate our mechanism

System configuration

We run our system on a single node of a cluster server system

A node is composed of two Intel Xeon X5570 CPUs, 24GB memory

Red Hat Enterprise Linux 5.4

Compiler tool sets

GNU Compiler Collection 4.1.2 for x86 64 Redhat linux

Intel C++ Compiler 11.1

Here, we compile the codes with '-O3 -g' option.

The Himeno Benchmark

- Developed by Dr. Ryutaro Himeno, RIKEN, Japan
- A highly memory intensive application kernel
  - Becoming popular in the HPC community to evaluate the worst-case performance for memory bandwidth intensive codes
  - The kernel is a linear solver for 3D pressure Poisson equation which appears in an incompressible Navier-Stokes solver:

$$\frac{\partial^2 p}{\partial x^2} + \frac{\partial^2 p}{\partial y^2} + \frac{\partial^2 p}{\partial z^2} + \alpha \frac{\partial^2 p}{\partial xy} + \beta \frac{\partial^2 p}{\partial xz} + \gamma \frac{\partial^2 p}{\partial yz} = \rho$$

  - The Poisson equation is solved using the Jacobi iterative method
  - The performance of this is measured in FLOPS (FLoating-point Operations Per Second)

Source code of the Himeno benchmark

```
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 */
```

(Only the primary loop region)

19-point stencil computation to the 3D array *p*

p is pressure,
a, b, c are coefficient matrices,
wrk1 is a source term of Poisson equation,
omega is a relaxation parameter,
bnd is a control variable for boundaries and objects,
wrk2 is a temporary working area for computation
Also, data are represented in single precision floating point format.

Loop nest structure

Outermost loop by *n*

Main triply nested loop by *i,j,k*

Triply nested loop by *i,j,k*

We note tat the body of this computational kernel involves 34 FP ops, so the FLOPS of this benchmark can be measured by dividing the total number of FP ops by the exe time.

The results obtained in this evaluation

The # of working data set pages during the whole or particular intervals of the execution

(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'

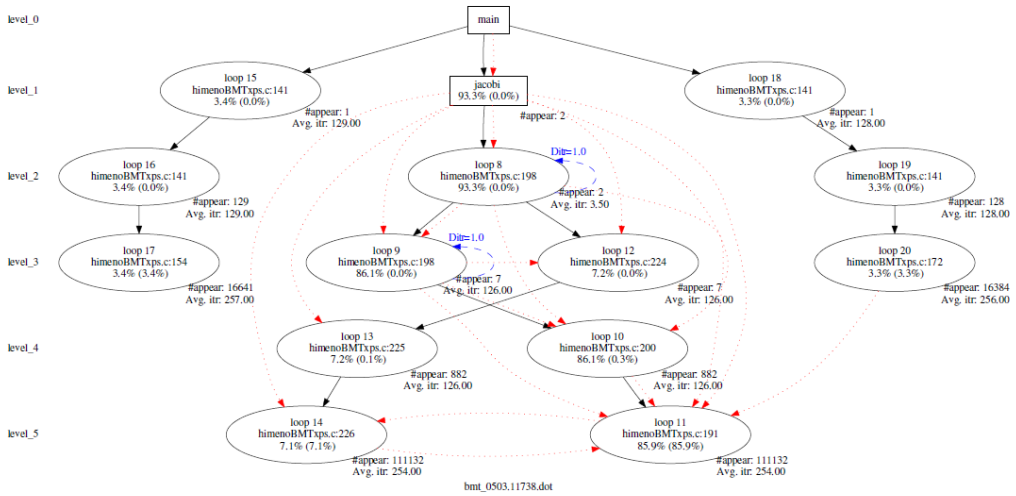
loopID	Analysis window	# of working set pages 64kB	4kB	256B
-	all	3674	58420	932704
9	apr=1, itr=1	3605	57209	896279
10	apr=1, itr=1	45	521	8133
11	apr=1, itr=1	16	19	104
13	apr=1, itr=1	16	18	22

- 'all': the working data set size of the whole program execution
  - It is observed that the working data set size 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 maximum allocated memory size provided by operating system, which is the VmHWM size at /proc/<pid>/status.

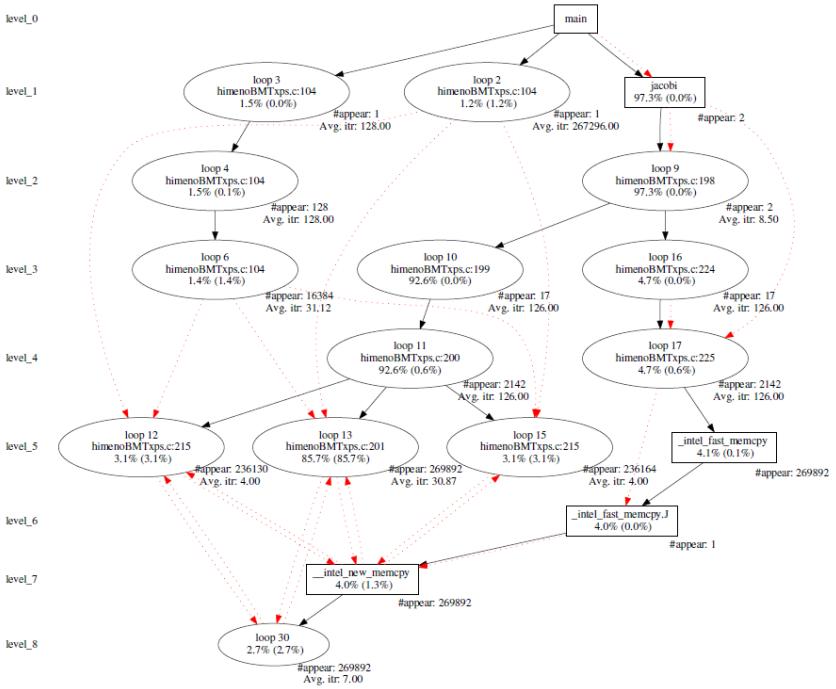
In order to monitor pages for a particular interval as a region of interest, we need to specify the information for the loop region.

Setting up particular loop iterations using LCCT+M

Legend:
- Circle node: Loop
- Box node: Procedure call
- Solid line Edge: Control flow
- Dashed Red line Edge: Data dependency between nodes
- Dashed Blue line Edge: Dependency btw loop iterations



The LCCT+M representation of gcc.4.1.2



The LCCT+M representation of icc.11.1

loops in the source code	gcc.4.1.2	icc.11.1
Outermost loop *n*	loopID= 8	loopID= 9
Main triply-nested loop *i, j, k*	loopID= 9, 10, 11	loopID= 10, 11, 13

- The loop structure of icc is converted to achieve SIMDization. Here, we monitor the largest one (loopID=13) in the innermost loops.
- The FLOPS score of the icc code is 2.9 times higher due to its aggressive optimization

The number of pages in a particular interval

loops in the source code	gcc.4.1.2	icc.11.1
Outermost loop *n*	loopID= 8	loopID= 9
Main triply-nested loop *i, j, k*	loopID= 9, 10, 11	loopID= 10, 11, 13

(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

- We set the intervals to the first one iteration of the outermost loop and each of triply-nested loops in the Himeno benchmark code
  - Also, we only focus on the first appearance of each loop.
  - It is observed that there are not so much differences 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
  - There are not so much differences of necessary pages among all page sizes when we execute one iteration of the innermost loop (little spatial locality inside an innermost)
  - In the loops that are next level of the innermost, the # of pages is increased only in 256 B case (so these imply there are spatial locality that fits in larger pages

To investigate temporal locality

We measure the increments of pages from the previous iterations

Here, the itr[2-1] represents that we measure the increments from the first iteration to the second iteration

The # 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

- From the results
  - We find that the all increments are smaller than the case that the required number of pages for an iteration is doubled
  - Therefore, we can find there are reuse of data within the already assigned pages

Strategies for loop tiling

- We can make strategies for loop tiling
  - Based on the obtained spatial and temporal locality info.
  - 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, loop tiling should be performed
    - In the case of cache, we should focus on the same page size as the cache line and care about points where 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 tile size (blocking factor)

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

Conclusions

- We have presented **a mechanism for identifying the working data set size of a particular loop iteration space**
  - Implemented using a dynamic binary translation system.
  - The working data set of the actual execution could help loop tiling or blocking that enhances spatial and temporal locality of data accesses
- 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:
  - 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
  - Studying how our approach can be improved to apply real problems in the HPC field.