XL Compiler Support for Assist Threads for Data Prefetching

Tong Chen (chentong@us.ibm.com)
Zehra Sura (zsura@us.ibm.com)
Kevin O'Brien (caomhin@us.ibm.com)

1 Introduction

A processor thread may be forced to stall if the data needed for subsequent computation is not readily available in the processor's cache memory. Computation cycles lost while waiting for data to be loaded can adversely impact performance of the system. [cite results for SPEC misses] This impact on performance is pronounced by trends in hardware design, as processor speeds have historically improved at a faster rate than memory speeds. Data prefetching is a technique used to reduce the number of memory stall cycles, and thus improve performance. Data prefetching may be performed by hardware that is designed to detect specific memory access patterns, or by software through the use of special memory prefetch instructions (such as the dcbt and dcbtst instructions in the PowerPC architecture). Hardware data prefetching incurs minimal overhead, but is limited by the complexity of access patterns that are feasible to detect, and by the number and length of prefetch streams active at a time. Software data prefetching is flexible, but there is some execution overhead associated with the prefetch instructions inserted within the code.

In this work, we explore the use of software prefetching when the prefetching code is executed in an associated assist thread, instead of being executed in situ in the application thread. Even though this requires extra hardware resources, we choose to use a separate assist thread for several reasons.

Firstly, prefetching using a separate thread allows the prefetch code to closely mimic arbitrary access patterns, or even tailor the stream of accesses to be more inclusive (e.g. by ignoring some control flow) or more exclusive (e.g. by skipping some accesses in a prefetch sequence). Also, hardware is evolving towards systems with hundreds of hardware threads, and in many usage contexts, it is likely that there will be more hardware threads available than the number that can be exploited by application-level parallelism. Furthermore, since the assist thread executes asynchronously, it is possible to run-ahead and prefetch a large number of accesses without being bound by the speed of the application thread. However, this asynchronous execution can be challenging as well, because it makes it necessary in some cases to synchronize with the application thread to ensure timely prefetch of data. Note that prefetched data may dislodge useful data already present in cache, and if prefetching is done too early or too late, it may degrade performance instead of improving performance.

We use the compiler to automatically generate code for the assist thread, and to synchronize its execution with respect to the application thread. To generate assist thread code, the compiler first uses profiling to determine which memory accesses to prefetch into cache. The memory accesses targeted for prefetching are called delinquent loads. They are the load instructions that cause the most cache misses during execution. Currently, we consider only those delinquent loads that are contained within loops. The

compiler then uses a back-slicing algorithm to determine the code sequence that will execute in the assist thread and that computes the memory addresses corresponding to the delinquent loads that are to be prefetched. The back-slicing algorithm operates on a region of code containing the delinquent load, and this region may correspond to the containing loop nest, or some level of inner loops within a nest. The assist thread code is guaranteed not to change the visible state for the application. The code generated for the application thread is minimally changed when an assist thread is being used. These changes include creating an assist thread once at the program entry point, activating assist thread prefetching at the entry to regions containing delinquent loads, and updating synchronization variables where applicable.

2 Assist Threads for data prefetching

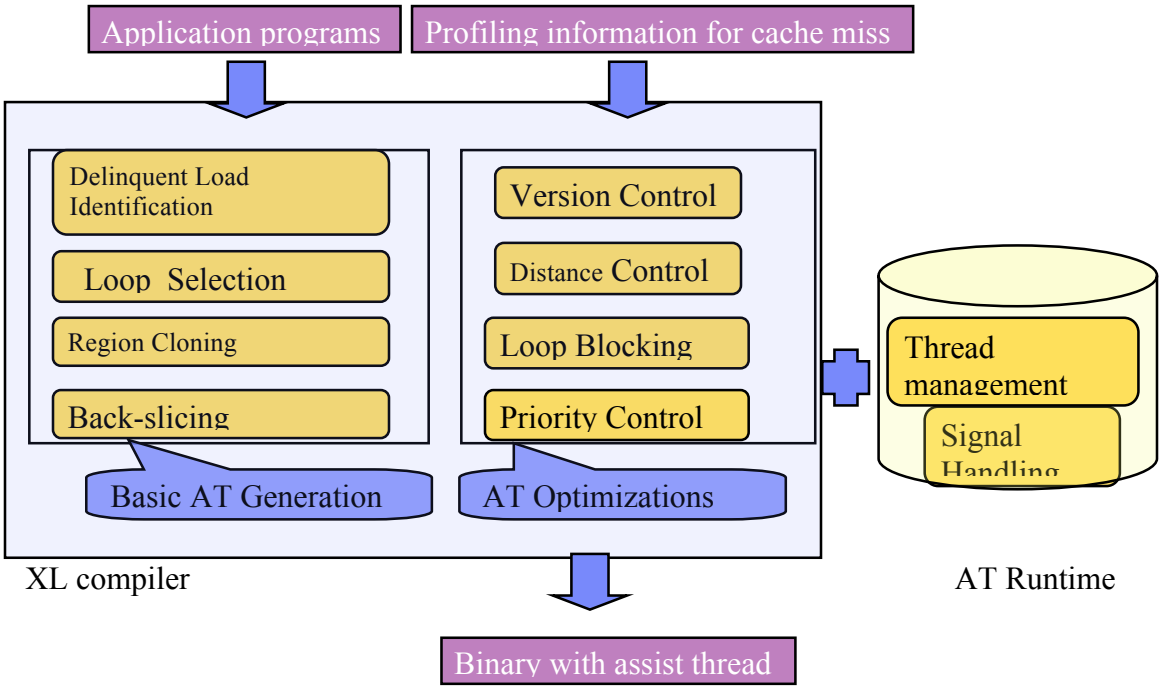


Figure 1 Overview of assist thread

2.1 Delinquent load identification

Identification of delinquent loads is the key to effective cache optimizations. Delinquent loads can be identified by runtime profiling or static analysis.

Dynamic profiling can be done by the compiler instrumentation or performance tools. XL compilers extend dynamic profiling infrastructure to support multiple pass profiling and cache miss profiling (Figure 1). The compilers insert instructions in an application to read hardware performance counter data directly for fine-grain cache miss profiling or invoke APIs provided by performance library packages for coarse-grain cache miss profiling to get delinquent load information.

Delinquent loads can be also identified statically by the compiler or user using memory access pattern and cache behavior analysis. XL compilers provide a directive void *__mem_delay(const void *address, const unsigned int delay_cycles)*, to annotate delinquent loads. Static delinquent load identification can avoid the runtime overhead imposed by profiling but may not be precise or sometime unavailable.

2.2 Loop Selection

After delinquent loads are identified, proper loops that contain the delinquent loads are selected as the work items for assist thread. In order to amortize the overhead of waking up the assist thread, the selected loop level should have enough iterations. However, compiler can not simply choose the outermost loops because they may introduce too much redundant operations to the assist thread. Moreover, the outer loops may introduce extra synchronization or speculation. Since the assist thread can not contain operations that change the application's status, assist thread has to either ignore those operations or get up-to-date value from the main thread.

The compiler searches inside out in the loop nests from innermost loop that contains delinquent loads. Compiler stops at the loop level that have enough accumulated iteration number. In implementation, the execution profiling information, if available, will be used in loop selection.

2.3 Outlining

The outlining step is to clone the selected loops for the assist thread. For each selected loop, a new procedure is generated with copied statements and symbols. The variables that need to be copied in are identified and their values passed to the cloned procedure. New procedures are generated with outlining utilities in the compiler. These procedures are the base of the code for assist thread. Further optimization will be applied to them.

2.4 Back-slicing

The back-slicing algorithm operates on a region of code containing the delinquent load, and this region may comprise of the whole procedure, or the containing loop nest, or some level of inner loops within a nest. Command-line options exist to allow the user to specify the size of the region considered for slicing. Compiler heuristics are also being developed to automatically determine the region to use such that the resulting slice is optimal. This slice should pre-fetch a large number of data items ahead of their use in the primary thread, but it should do this without negatively impacting the cache usage of the primary thread by bringing in too much data ahead-of-time. Compiler heuristics to control the back-slicing region (and hence the size of the slice generated) are based on the ratio of computation needed for delinquent address calculation and the computation in the whole region. In some cases, the delinquent load may not be contained in any loop within its own procedure (say *foo*), though it is contained in a loop within another procedure (say *bar*) that invokes procedure *foo*. In this case, the compiler attempts to inline *foo* into *bar* before applying back-slicing, so as to enable better control over the granularity of the assist thread code generated. The basic algorithm for back-slicing makes use of the procedure control flow graph as well as the SSA graph, and it proceeds as follows:

1. Initialize the set of instructions in the slice (say SliceSet) to be dcbt instructions whose operands are the address expressions in the delinquent load instructions.
2. Initialize the set of upward exposed reads (say UpwardReads) to be empty.
3. Repeat until there is no change to SliceSet:
   For each instruction in SliceSet:
   a) If the instruction is in a control-dependent basic block, then:
      For each predecessor basic block:
      i. Find the conditional branch instruction that leads into this instruction's basic block.
      ii. Add this conditional instruction to set SliceSet if it has not already been added, and if it is within the back-slicing region.
   b) For each read reference (memory or register) in the instruction:
      i. Determine the SSA definition (say *def*) for the reference.
      ii. If *def* has been processed earlier, then continue to the next reference.
      iii. If *def* occurs outside the back-slicing region, add it to the set UpwardReads if it defines a procedure-local variable, and continue to the next reference.
      iv. If *def* is not an SSA Phi node, then add it to the set SliceSet, and continue to the next reference.
      v. *def* is a Phi node; add both its condition expression, as well as the definition expressions to SliceSet.
4. Remove from SliceSet all instructions that store to a variable that is not local to the procedure.

The sets SliceSet and UpwardReads determined by the back-slicing algorithm are used to generate a separate slice function that will execute as part of assist thread code. First, the original procedure is cloned to create a copy with its own local variables. Then, only instructions in SliceSet are retained in the cloned copy, and other instructions are discarded. Next, for each instruction in UpwardReads, two new assignment statements are generated: one in the original procedure, and one in the cloned copy. The assignment in the original procedure is placed directly before the entry to the back-slicing region, and it assigns the value of the upward exposed read variable to a shared memory location. The corresponding assignment statement in the cloned procedure is placed at the beginning of that procedure, and it assigns the value from the shared memory location to the procedure-local copy of the upward exposed read variable. The original procedure is executed in the primary thread, whereas the transformed cloned procedure is executed in the assist thread.

2.5 Runtime support

The original procedure is also responsible for signaling to the assist thread that it should start pre-fetching. The code for the original procedure is transformed so that after it has copied the values of the upward exposed read variables, it copies to a shared memory location (say *func_addr*) the address of the slice function for the succeeding back-slicing region. It also copies to another shared memory location (say *data_ptr*) the address of the

memory area containing the values of the upward exposed reads for that slice function. Finally, it signals to the assist thread to start pre-fetching using the procedure given in *func_addr*, while the original procedure continues executing the original application code. The following pseudo-code illustrates the code transformation for slice function generation:

Original Code

```
// A and x are globals

y = func1();
…
i = func2();
// start of back-slice
while (i < condition) {
        …
        x = func3();
        …
        // delinquent
load
```

Primary Thread Code

```
y = func1();
…
i = func2();

// y and i are upward-exposed
reads
char *tmp =
malloc(sizeof(y)+sizeof(i));
*((int *)tmp) = y;
*((int *)(tmp+sizeof(y))) = i;

// start pre-fetching in assist
thread
data_ptr = tmp;
func_addr = &slice_func_1;
signal assist thread;

// start of back-slice
while (i < condition) {
        …
        x = func3();
        …
        func4( A[i] );
        …
        i += y;
}
// end of back-slice
```

Assist Thread Slice Function

```
void slice_func_1(char
*dptr) {

   int y = *((int *)dptr);
   int i = *((int
*)(dptr+sizeof(y)));
```

The compiler inserts code in the program entry point procedure (i.e. in the main procedure) to create and run the assist thread. This thread is bound to a specific processor, which is either the same processor as the one executing the primary thread in case of SMT, or a processor on the same chip in case of CMP. When it starts up, the assist thread executes a continuous loop, in each iteration waiting for the primary thread to signal start of pre-fetching, and then invoking the corresponding slice function.

```
while (1) {
```

```
wait for primary thread to signal start of pre-fetching;
// Global shared variable func_addr gives assist thread slice function to invoke.
// Global shared variable data_ptr gives address where values of upward exposed
//         read variables have been stored by the primary thread.
(*func_addr)(data_ptr);
}
```

2.6 Speculation handling

The main thread and the assist thread are running fully asynchronously after the assist thread is created. There are several issues with speculative pre-computation for data prefetch by assist threads. (1) the global variables accessed by assist threads may be modified by the main thread, which may result in invalid memory accesses; (2) assist threads may get scheduled to execute after the main thread is finished; (3) assist threads may run too faster than the main thread, which causes cache pollution, or assist threads may run too slower than the main thread, which cannot help the main thread

To solve the problems (1) and (2), XL compilers use the three approaches:

The one is to exploit compiler static analysis to avoid invalid memory accesses at runtime;

The second is to introduce a coarse-grain control to reduce the probability of invalid memory accesses. We create a global unique version number for each instance of the code region where data prefetching with assist thread is applied. For each call to wake up a thread, the version number is passed to this wake up function. When the assist thread is executed, it will first check if the global version matches with the version value that is passed. For example, when the current global version number 10 is created by the main thread, the value is passed to the assist thread. When the assist thread get executed, it first checks if the global version number is still matched to its local version number 10. If not, the assist thread exists. When the main thread finishes executing a code region, it will increase the global version number.

In the case where invalid memory accesses happen at runtime, the assist thread runtime catches the signal and the assist thread exits

3 Optimizing Assist Thread Code

3.1 Version Control

The main thread and the assist thread are running fully asynchronously after the assist thread is created. Assist threads may get scheduled to execute after the main thread is finished. We introduce a lightweight coarse-grain version control. We create a global unique version number for each instance of the code region where data prefetching with assist thread is applied. For each call to wake up an assist thread, the version number is passed to the assist thread. When the assist thread is executed, it will first check if the global version matches with its local version number. For example, when the current global version number 10 is created by the main thread, the value is passed to the assist thread. When the assist thread gets executed, it first checks if the global version number still matches its local version number 10. If not, the assist thread exits. When the main thread finishes executing a code region, it will increase the global version number. Figure 2 shows an example of the coarse-grain version control.

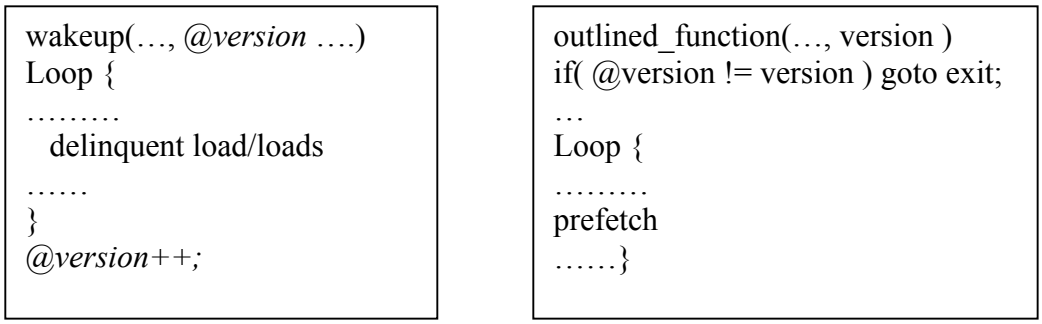


Figure 2 Coarse-grain version control

3.2 Distance control

For performance, the compiler can transform code to insert synchronization between the main thread and assist thread to keep execution of both threads within a certain number of loop iterations of each other. The compiler can choose to have the assist thread periodically check where it is relative to the main thread and wait some time if it is too far ahead, or it can choose to have the assist thread periodically skip some number of iterations if it tends to lag behind, or it can choose to avoid synchronization altogether because the overhead is high and it is not profitable. We use a compiler algorithm to automatically decide what synchronization transformations to apply in the case of each delinquent load. This algorithm relies on a heuristic to estimate the execution times for a loop iteration in the assist thread prefetch code, and for a loop iteration in the main application code assuming successful data prefetching. If the assist thread time is epsilon (30 cycles) less than the main thread time, then the compiler transforms the code so that the assist thread periodically waits for the main thread, else if the assist thread time is greater than or equal to the main thread time, then the compiler transforms the code so that the assist thread periodically skips some iterations. The number of iterations to skip or synchronize is approximated by dividing the amount of L2 cache assumed to be available for prefetching (15% of P5 L2 cache, i.e. 300K) by the amount of data fetched in each assist thread iteration (estimated in terms of number of cache lines used for all loads in the assist thread loop).

The heuristic to estimate execution time uses the flow graph and profile directed feedback (PDF) data available in the XL compiler (TPO pass 2). The profiling data used includes miss rates for individual memory instructions, percent execution frequencies for basic blocks, and loop iteration counts. Initially, the number of cycles for each basic block is computed as the sum of cycles for each instruction in the block. We assign 1 cycle for almost all data manipulation instructions (2 cycles for multiplication and 15 cycles for division). For memory instructions, we use the formula ((miss latency*miss rate) + 2*(1-miss rate)), with the exception that if the memory operation is in both threads, then its miss rate in the main thread is assumed to be zero. The cycle count for each block is weighted by the execution frequency for that block. Then, for any loop in the flow graph, the execution time is the loop count multiplied by the sum of weighted cycle counts for every block within the loop region.

The heuristic algorithm was evaluated with several loops from the SPEC benchmarks. The algorithm correctly predicts whether the assist thread goes faster or slower than the main thread. In about half of all cases, it estimates the number of iterations at which to synchronize to be close to optimal. In the remaining cases, it overestimates by about 10-40%, but the impact of this on overall application performance is negligible.

3.3 Loop Blocking

To further reduce the overhead associated with distance control, we apply well-known loop blocking to control distance for each block instead of each loop iteration. Both the main thread and assist thread use the same blocking factor (BF), and distance control code is inserted out of the blocked loop (Figure 3).

```
// MT Original Loop
for (i ; i <UB; i ++) {
…
  a[i];
…
mtc++;
}
```

```
// MT Blocked Loop
for (j; j < BF; j++){
   for (i ; i < min (BF*j + BF, UB); i
++) {
    …
      a[i];
    …
   }
mtc+=BF;
}
```

Figure 3 Loop blocking to reduce the overhead of distance control

3.4 Priority control

The software-controlled priority for SMT determines how decode cycles are assigned among the SMT threads. The priority is set by special instructions and is implemented by hardware in the decode stage. In general, the thread with higher priority will be assigned more decode cycles. This is a particular feature implemented in PowerPC to provide software with more control over SMT resource usage.

There are two interesting features of instructions for software-controlled priority: they are extremely light-weight and they are stateless with respect to the OS. To make use of these instructions without any OS changes, our basic approach uses the compiler to insert the priority setting instructions into the code in such a way that they are periodically executed at runtime. As a result, the thread will be executed with the desired priority most of the time, even though we may lose control of priority for a short period at each context switch.

In our assist thread prefetch system, setting priority can be integrated with the progress control already in the system. The progress control code helps to determine:

- Insertion points. The instruction to set the priority can be inserted with the progress control check. Progress control checks need to execute at appropriate intervals: not too frequent (to limit overhead costs), and not too sparse (to keep control of the progress of AT). When setting priority is done with the progress control code, minimum extra overhead is added.
- Hints for priority. The progress of the MT and the AT is known within the progress control code. The runtime check for the progress control can tell when the AT is running too fast and when the AT is running too slow. This information can be used to guide the compiler to select the right priority for the MT and the AT.

4 Experiments

Experiments are conducted on POWER7 machine with Linux OS. Both synthetic kernels and real benchmarks are used in the experiments.

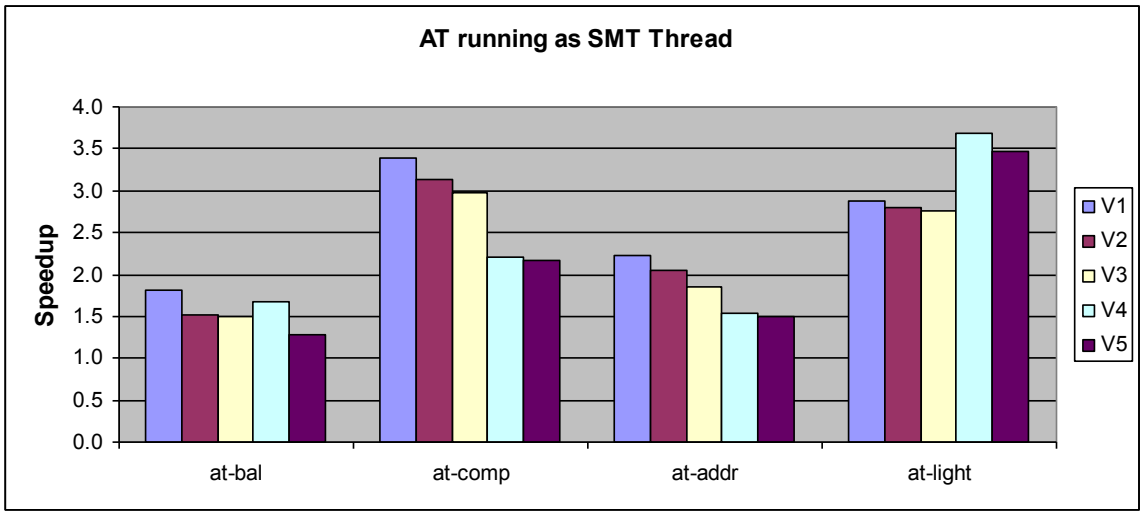
Synthetic kernels are designed to cover a wide range of test cases. They vary in the operations and cache miss rates. All the operations in the main thread can be partitioned into two groups

- Address calculation for delinquent load. These operations will be part of the assist thread code.
- Computation work in the loop. These operations will NOT be part of the assist thread code and will be executed by the main thread only.

For each of the two groups, we can assign either many operations (about 10 cycles) or only a few operations (about 3 cycles). So there are four combinations:

- *both-heavy*: both address and computation group have many operations
- *compute*: the computation work has many operations while the address calculation has only a few operations
- *address*: the address calculation has many operations while the computation work has only a few operations
- *both-light*: both address and computation group have a few operations. Compared with both-heavy, both-light is more memory bound.

Another varying factor is the cache miss rate for the delinquent load. We mixed random accesses and continuous accesses to get the desired L2 miss rate from the delinquent load. We code kernels to exhibit miss rates from high to low (from v1 to v5): 95%, 70%, 45%, 25% and 15%. These miss rates are chosen to represent the miss rates of delinquent loads found in real benchmarks. The different miss rates are combined with the different functional unit usages, resulting in a total of 20 test cases. These test cases cover a wide spectrum of scenarios in real applications.



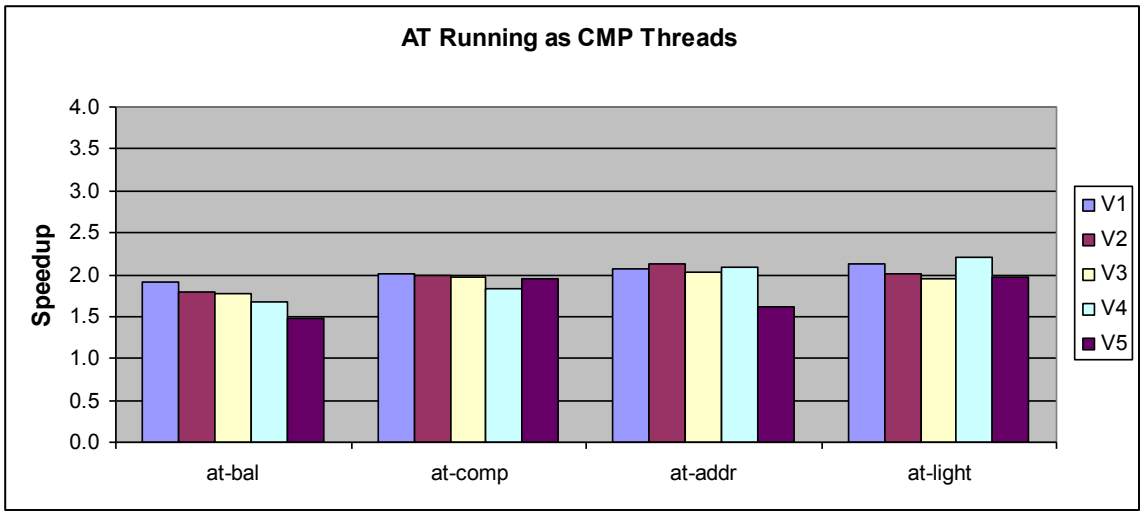


Figure 4 Speedup of Kernels

The speedups from AT, when AT is running as SMT thread or CMP thread, are reported in Figure 4. First, significant speedup, up to 3, can be gained with AT for prefetching for different kernels with different cache miss rates. The performance varies with different kernels. AT performs well on at-comp because its assist thread consumes not much resource. The kernel, at-light, can be improved dramatically because it is simply load bounded. The AT performance with SMT is better than with CMP because the data is brought to closer cache by SMT thread. Since the SMT may have resource conflicts, the speedup of at-bal and at-addr, which have more operations in AT than the other two, is similar in SMT and CMP.

Assist thread is also tested with selected benchmarks, which are listed below:

- art openMP: image recognition in SPEC OMP
- lbm: fluid dynamics in Spec2006
- IS: Integer sorting in NAS benchmark
- CG: conjugate gradient in NAS benchmark
- CC: Calculate Connected Components in a graph
- MST: Calculate minimum Spanning Tree in a graph
- Bigbyte: user application

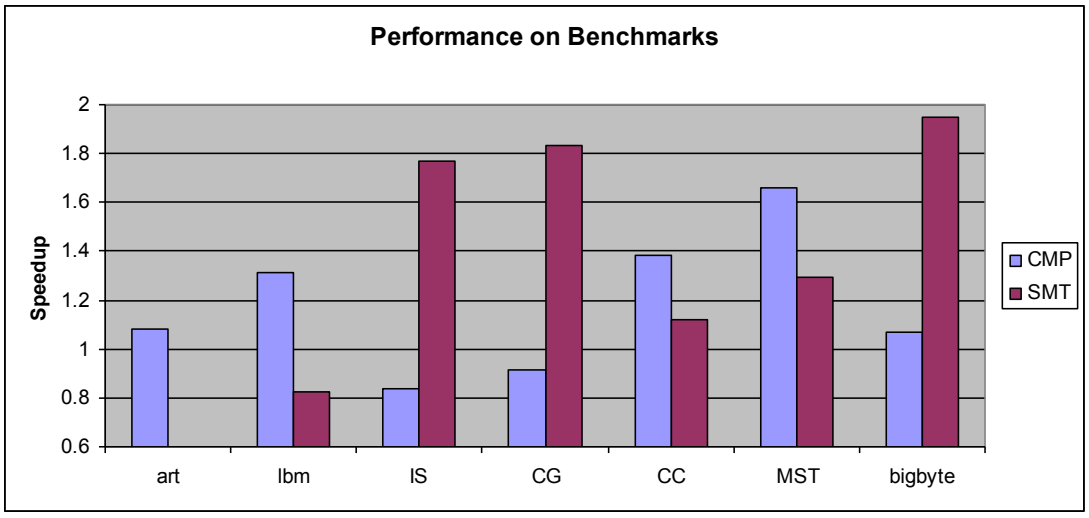


Figure 5 Speedup of benchmarks

The speedup with CMP and SMT for the benchmarks can be found in Figure 5. Many benchmarks can be speed up from 10% to almost 100%. Whether CMP or SMT performs better depends on the characteristic of the benchmarks. IS and CG are slowed down in CMP because of the cache line competition between the MT and AT.

5 Conclusions

A compiler infrastructure of using assist thread for prefetching data is developed in this work. This infrastructure contains a profiling tool to identify delinquent loads in applications, compiler transformation to generate the code, and a runtime library that provide utilities for runtime thread management.

In compiler transformation, first the proper loops in which delinquent loads are embedded are selected and cloned. Then back slicing algorithm is applied to remove all the operations that are unnecessary to prefetching. The basic assist thread code is optimized with progress control, loop transformation and priority control for better performance.

The assist thread approach is experimented with both synthetic kernels and real benchmarks. Significant performance gain, speedup up to 3, is observed on POWER7 systems. The assist thread features have been put into IBM XL compiler product.