Critical Technology Evaluations

Technology	Description	Status
Parallel Language	Evaluation of PIL targeting Shared Memory SCALE using NAS Parallel Benchmarks	Evaluating
Parallel Language	Evaluation of PIL targeting Distributed Memory SCALE using NAS Parallel Benchmarks	Evaluating
Compiler	Design and implementation of an Exascale-friendly codelet code generation scheme	Evaluating
Compiler	Virtual DMA runtime to enhance virtual scratchpad optimization	In process
Applications	Provided main driver and input information for TCE benchmark	Done
Applications	Designed CnC flow for LULESH	Done
RTS/Compression	Comparing the effects of compression algorithms on dense data structures for a Matrix Vector multiply application using the Floating Point Compression Algorithm by Burtscher and Ratanaworabhan using the SWARM framework	Evaluating
RTS/Compression	Comparing the effects of compression formats on sparse data structures for a completed Cholesky factorization application using an extended* version of Block Sparse Row Format using the SWARM framework	Evaluating
RTS/Compression	Development of the first instance of the Architected Composite Data RTS Type Qualifier concept [reciprocal formulation to RPDT] for SWARM	Evaluating
RTS/Compression	Writing paper for ICS'14 (collaboration between PNNL, UDEL & Reservoir Labs)	In process

Summaries of Quarterly Work (Q5)

ETI Work

TCE

ETI has adapted the Tensor Contraction Engine code generator, provided by PNNL, to generate simple C code. ETI has also built a standalone C program around it, so that the generated code can be evaluated, modified and optimized directly, outside of nwchem. This test framework supports the minimal set of nwchem API functions and data structures, to allow the code generated by TCE to function. It also compares the output to the output of nwchem itself, to ensure that the operation was performed successfully.

This test framework successfully runs one iteration of cc2 on an Ozone molecule. This completes in 0.3 seconds. ETI is now working on the ccsd operations in a Benzene molecule. This requires supporting additional data structures from nwchem, but will allow much larger problem sizes to be executed, which will allow us to evaluate scaling strategies much more clearly.

LULESH

LULESH demonstrates a workload imbalance that is consistent across iterations. In LULESH, this workload imbalance is represented as a differing loop count for certain regions. Regions with high loop factors will always have more work to do than other regions. In some cases, the loop count increases the workload by more than a factor of 10.

It may be possible to re-balance the workload, by splitting up the volume of (simulated) space differently. Our next step is to produce a design and a set of code modifications to allow this. The goal will be to repartition the space such that the computational load is equal while minimizing the amount of data transmitted between compute nodes (i.e. minimizing the surface area between the partitions).

Reservoir work

Reservoir Labs has focused on two performance-related aspect of SWARM code generation. First, we have implemented an Exascale-ready codelet code generation scheme in SWARM, which is optimal in terms of sequential scheduling overhead and space overhead.
This work is presented in the “Exascale code generation” section below.
Also, we have implemented a high-performance cache-bypassing memory copy library to improve the performance of codes generated by R-Stream using our virtual scratchpad memory optimization. This library basically provides a virtual DMA abstraction, practically removing cache interference issues when executing parallel programs using virtual scratchpads. This work is presented in the “Virtual DMA” section below.

UIUC Work

UIUC has implemented six of the NAS Parallel Benchmarks using PIL HTA library for performance evaluation. The code is written in C language and includes HTA API calls. Since parallel data distribution and operations can be expressed through combinations of HTA operations, the programmers can program in a higher-level manner instead of having to directly interact with PIL.

The benchmarks can now be executed on OpenMP and Shared Memory SCALE, and the preliminary performance results were retrieved from a workstation with many cores. The work to extend PIL to support backend code generation for the distributed memory environment is still ongoing. Since the programming interface of PIL is different for the distributed memory environment, the HTA-to-PIL interface has to change as well. However, HTA APIs remain unchanged to the user level applications, thus these changes should be transparent to the user.

PNNL Work

PNNL provided the Tensor Contraction Engine (TCE) driver and input decks to the project. It also designed a LULESH CnC flow. Using SWARM as a RTS substrate, we are studying the effects of compressed formats and algorithms on standard kernels.

In particular, we are evaluating the effects of compression algorithms on dense data structures for a Matrix Vector multiply application using the Floating Point Compression Algorithm [1] by Burtscher and Ratanaworabhan within the SWARM framework. For sparse data structures, we are examining representations for a completed Cholesky factorization application using an extended* version of Block Sparse Row (BSR) Format. The extensions we developed take into account fill-ins and operator invariants. These experiments are complemented by an overarching framework which is a bottom up (runtime aware) data type qualification system for the RTS.

[1]. Burtscher, M.; Ratanaworabhan, P., "FPC: A High-Speed Compressor for Double-Precision Floating-Point Data," Computers, IEEE Transactions on , vol.58, no.1, pp.18,31, Jan. 2009

Topic Detail:

Reservoir: Exascale code generation

As introduced in the previous quarterly report, we have developed a new code generation scheme for codelet codes with the SWARM target that dramatically improves sequential scheduling overhead and spatial overhead of inter-codelet dependences.
Here is a summary of its governing principles:
- For every codelet instance, there exists a function that returns the number of codelet instances it depends upon (its predecessor in the codelet graph).
- A codelet can be initialized either by the main codelet (sequentially so) or by the one of its predecessor that arrives at completion first (in parallel).
- A codelet gets initialized using a counted dependence. A counted dependence associates an integer counter with a codelet and its input parameters. When the counter reaches zero, the codelet is scheduled. Every time a codelet completes, it decrements the counters associated with its successors.

Space overhead

The number of synchronization objects (and hence the space overhead associated with them) is bounded by the number of codelet instances (let us call this number n): one counted dependence per codelet. This a significant improvement over the $O(n^2)$ bound for frameworks that represent one object per dependence, which includes our previous SWARM backend version.

Two things should be noted about this new $O(n)$ bound. First, it can be brought down to less than n by using a sparse representation of the current counted dependences, such as a hash map. Also, we have assessed the practical impact of an $O(n)$ space overhead versus $O(n^2)$at *Gigascale* by forcing the sizes of the tasks formed by R-Stream to be very small (and hence numerous). In some examples, a few dozen million codelets were formed. The space required to store a representation of all the dependences (even though they were not all stored at the same time) was enough to overflow the linux memory, rendering the programs unexecutable. It is clear that systems with $O(n^2)$ space overhead are not viable in an Exascale context where memory movements dominate power consumption and the number of codelets per program will be even higher.

Sequential scheduling overhead

One of the advantages of our previous SWARM backend over other systems in the literature[1] is that instead of representing the program as an explicit graph of codelets, we produce an implicit rendering of the graph in which codelets acquire and validate dependences in the form of *tags*[2]. As a result, our preivous SWARM backend improved upon the $O(n^2)$sequential scheduling overhead required for the main codelet to set up an explicit graph representation, down to $O(n)$ since with the implicit graph representation, the main codelet only requires to

[1] M. Baskaran, B, Vydynathan, U. Bondhugula, J. Ramanujam, A. Rountev, P. Sadayappan, "Compiler-assisted dynamic scheduling for effective parallelization of loop nests on multicore processors." In Proc. of PPoPP 2009, pp. 219-228.
[2] Using SWARM's TagTables synchronization mechanism.

perform one operation (scheduling) for each codelet that needs to run (each codelet knows about its own dependences).

In our new Exascale-friendly SWARM backend, thanks to SWARM's counted dependence mechanism, a codelet is scheduled only when its predecessors have completed. Hence, the overhead on the scheduler is reduced since the scheduler is not burdened with codelets scheduled before they are ready to execute. However, without particular optimizations, and to ensure that all codelets are eventually initialized, the main codelet still runs a loop that initializes all the codelets if they haven't been so. So we still have an $O(n)$ sequential scheduling cost, even though in practice the number of non-ready codelets scheduled is reduced.

Notice that only the codelets that don't have predecessors need to be initialized by the main thread. We implemented an optimization in R-Stream that uses this very observation to remove unnecessary initializations made by the main codelet, reducing the cost of sequential scheduling overhead to $O(m)$, where m is the number of codelets that don't have predecessors. In most stencil computations, as in the two-dimensional Jacobi example code presented below, $m = 1$. Interestingly, the worst case here is embarrassingly parallel programs, since none of their codelets has predecessors. However, the initialization of tasks without predecessors can be trivially parallelized as well, either per dimension (bringing down the overhead to $O(\sqrt[d]{n})$) or by hierarchical decomposition (bringing the overhead to $O(\log(n))$).

To illustrate this new code generation scheme and its optimization, we chose a simple Jacobi-style nine-points stencil kernel, represented in Figure 1.

```
#ifndef NMAX
#define NMAX 1000
#endif
#ifndef TMAX
#define TMAX 1000
#endif

#pragma rstream map
void jac(real_t A[NMAX][NMAX], real_t B[NMAX][NMAX]) {
    int t, i, j;
    for (t = 0; t < TMAX; t++) {
      for (i = 1; i < NMAX - 1; i++) {
        for (j = 1; j < NMAX-1; j++) {
          B[i][j] = ( A[i-1][j-1] + A[i-1][j] +
                      A[i-1][j+1] +  A[i][j-1]+
                      A[i][j]     + A[i][j+1] +
                      A[i+1][j-1] + A[i+1][j] +
                      A[i+1][j+1] ) / CST(9.0); /* S1*/
        }
      }
      for (i = 1; i < NMAX - 1; i++) {
        for (j = 1; j < NMAX-1; j++) {
          A[i][j] = B[i][j]; /* S2 */
        }
      }
    }
  }
}
```

Figure 1. Original Jacobi stencil code.

R-Stream decomposed this code into codelets, computed inter-codelet dependences and generated the SWARM-based parallel code. Throughout the codes below, you will notice that some function calls are actually made to a thin runtime layer, which comes with R-Stream, and that enables a more concise and readable code generation. Calls to these functions are prefixed with “rsw” (for R-Stream - SWARM).

The code generated[3] for the codelets performing the stencil operations is represented in Figure 2. It is made of three parts: the unpacking of the input arguments (from the SWARM “THIS” pointer), the computation itself, and a loop that decrements the counted dependences of the particular codelet instance’s successors. The “rswAutoDec” function name is a reminder that not only is the counted dependence decremented, but it is also automatically initialized if it hasn’t been so yet.

[3] We selected R-Stream options that give a relatively readable code. Most optimizations that reduce code readability, such as for instance global code motion, were turned off. Hence the appearance of common sub-expressions in the code.

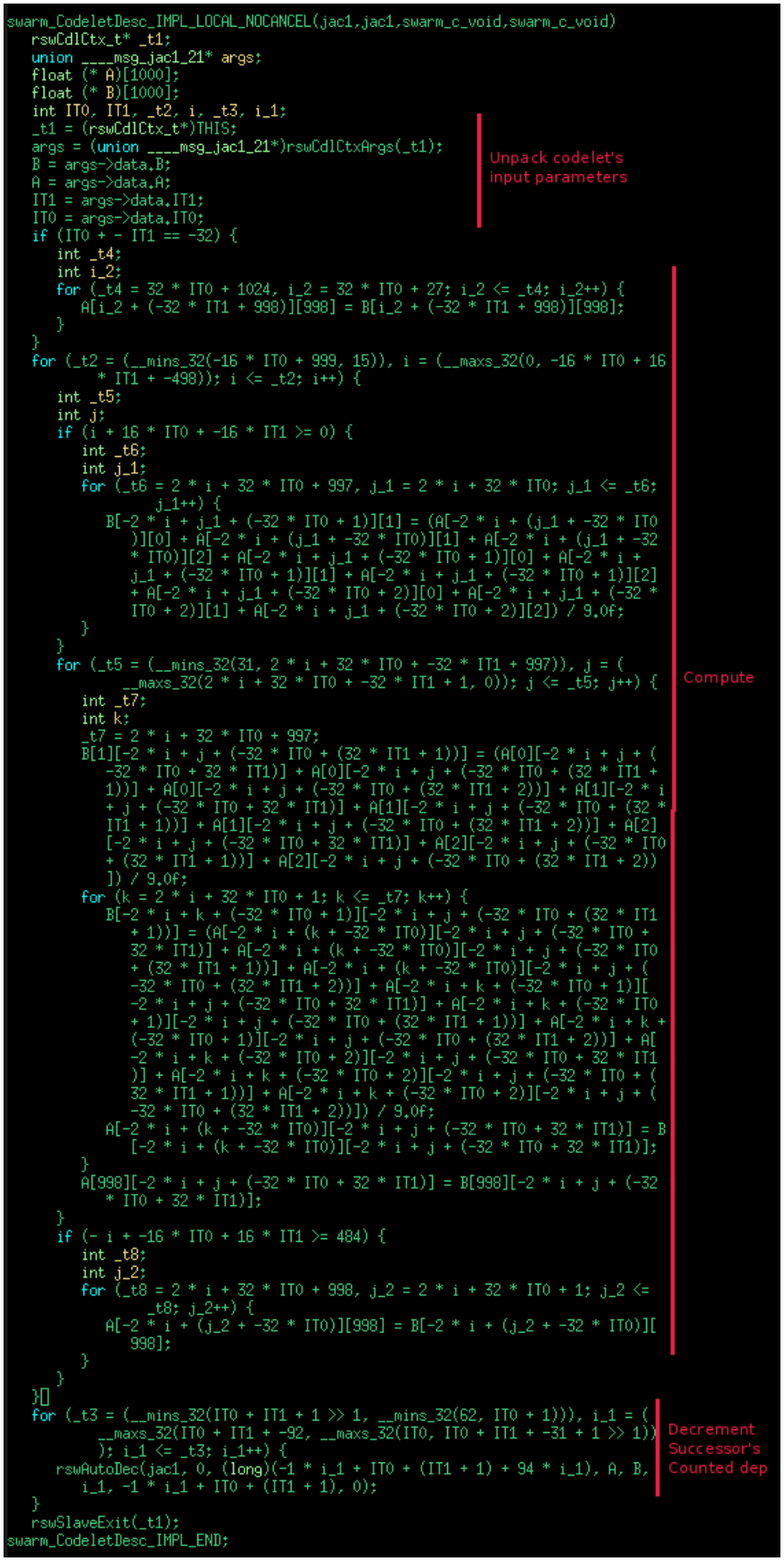


Figure 2. Generated stencil SWARM codelet

The main codelet code is represented as generated by R-Stream without the sequential scheduling overhead optimization presented here in Figure 3.

```
swarm_CodeletDesc_IMPL_LOCAL_NOCANCEL(jac_main,jac_main,swarm_c_void,
   swarm_c_void)
   rswCdlCtx_t* _t1;
   union ____msg_jac_main_25* args;
   float (* B)[1000];
   float (* A)[1000];
   int i;
   rswSinkInit(NEXT, NEXT_THIS);
   for (_t1 = (rswCdlCtx_t*)THIS,
        args = (union ____msg_jac_main_25*)rswCdlCtxArgs(_t1),
        B = args->data.B,
        A = args->data.A,
        i = 0;
        i <= 62; i++) {
      int _t2;
      int j;
      for (_t2 = (__mins_32(93, i + 32)), j = i; j <= _t2; j++) {
         rswPreSchedule(jac1, 0, (long)(j + 94 * i), A, B, i, j, 0);
      }
   }
   rswSlaveExit(_t1);
swarm_CodeletDesc_IMPL_END;
```

Figure 3. Unoptimized main codelet

A two-dimensional (i,j) loop nest is making calls to “rswPreSchedule”, which initializes a counted dependence for each codelet instance if it wasn’t initialized yet by concurrent calls to rswAutoDec. In this example, it happens that only one codelet does not have a predecessor. We can confirm this by looking at the optimized version on Figure 4, in which only one call is made to rswPreSchedule by the main codelet. Figure 4 also represents the new code for the original “jac” function, as well as a helper function, generated by R-Stream, which packs a codelet’s input data into a data structure (which becomes the “THIS” pointer in the destination codelet).

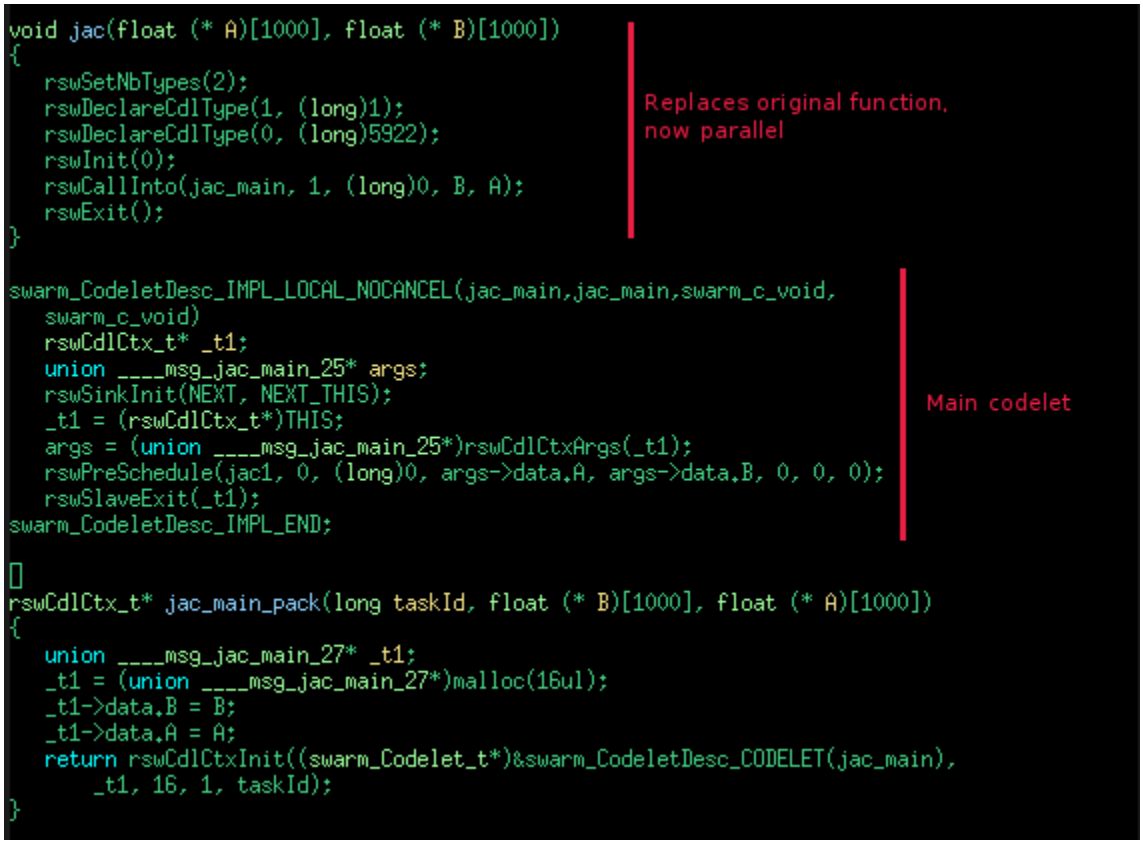


Figure 4. Optimized main codelet, packing function, new “jac” function.

A note on hierarchical mapping

As part of our previous X-Stack research, we explored the use of hierachical mapping to reduce both space and sequential scheduling overheads. The sequential scheduling overhead is brought down to O(log(n)) , and the space overhead associated with dependences is down to O(nt), where t is the number of codelets in the penultimate level of recursion. Interestingly enough, while these overheads are acceptable for Exascale, they are not as low as with our new SWARM backend presented here. Additionally, our new SWARM backend doesn’t suffer from the extra synchronizations that are idiosyncratic to hierarchical task decomposition.

Reservoir: Virtual DMA runtime for x86

Virtual scratchpad optimization

Cache memories provide additional performance potential by keeping data used by the processor in faster, closer-to-the processor memories automatically. As opposed to explicitly-managed memories such as scratchpads, the additional performance potential comes for free in terms of programming burden, since cache operations are transparent to the application programmer.

Unfortunately, this transparency comes with some amount of inflexibility, which makes optimal use of these faster memories tricky. In particular, knowing in advance whether loading data X into the cache will trigger the eviction of data Y from the cache is difficult. It is however

important to know this, since the next use of Y will require a new time- and energy-costly load of Y into cache (a.k.a. a cache miss).

In the case of loop codes, knowing how many cache misses will occur given a particular schedule of the loop iterations requires counting the number of solutions to so-called cache miss equations[4]. The solution to cache miss equations is usually a complex expression, and the computations involved in obtaining the number of such solutions are not tractable enough. Hence compilers such as R-Stream rely on heuristics and approximate computations to optimize to cache-based architectures. Global data layout optimizations are also used to pack data used at once into a fewer number of cache lines. Data layout optimizations are costly as they introduce extra data movement (and hence extra cache misses) and additional synchronizations in the program.
By opposition, since the programmer has full control over data movement to and from scratchpad memories, these are easier to optimize for. Additionally, moving data to a scratchpad offers an opportunity for local data layout optimization.

Since the days of the STI Cell processor, R-Stream has been capable of generating optimized code for scratchpad memories. The gist of R-Stream's virtual scratchpad optimization is to get closer to the type of control of data movement that we have on scratchpad memories, but on cache architectures. To do so, roughly, we treat the cache as a scratchpad. This process is illustrated in Figure 5. Before a codelet executes, it copies its input data into a buffer that is as small as (or smaller than) the cache. The codelet executes on the copy of its input data that lies in the "local" buffer (which is presumably cached), and writes back its output data from the buffer when done.

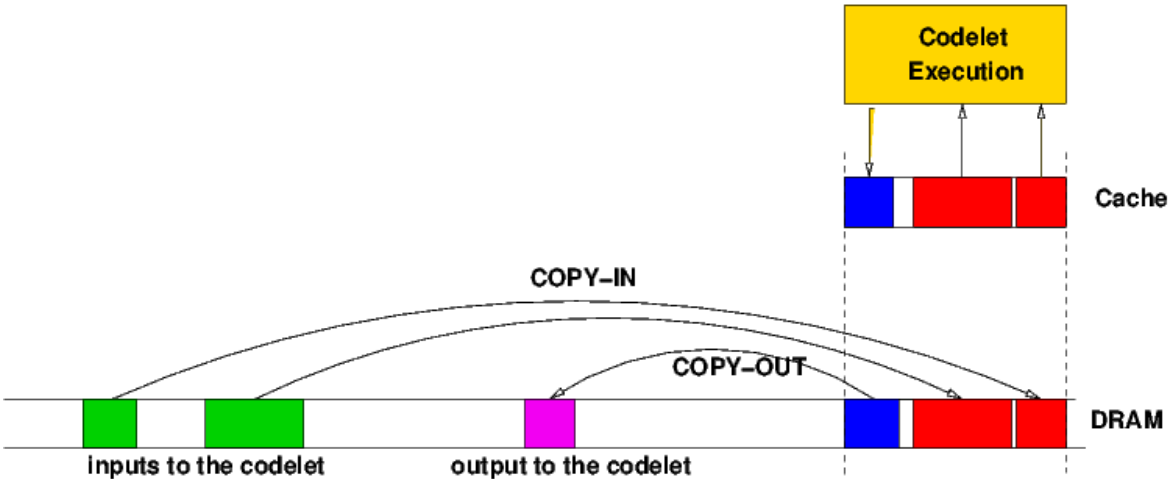


Figure 5. Virtual Scratchpad optimization

While we have experienced significant gains using this technique with codes that reuse data enough, we believe that the current implementation suffers from unnecessary cache misses during the copy-in and copy-out phases. Indeed, the data copied in is only accessed once and won't be accessed during the computation (which uses its "local" copy). Idem for the data written to during the copy-out. Hence the cache misses entailed by these copy-in and copy-

[4]S. Gosh, M. Martonosi and S. Malik, "Cache Miss Equations: A Framework for Analyzing and Tuning Memory Behavior." ACM Transactions on Programming Languages and Systems, Vol. 21, No. 4, July 1999.

out operations introduce unnecessary delays, and additionally evict local buffer data from the cache, making for more cache misses during codelet execution.

Our objective is to remedy this problem by performing the copies using efficient cache-bypass operations. Such operations, non-temporal vector loads and stores, were introduced in x86 architectures since SSE 4.1. More efficient (but less portable) versions of them have appeared with the AVX/AVX2 instruction sets. In order to seamlessly enable the local layout transformation opportunities that come with scratchpad memories, the API for these copy operations is that of a standard two-dimensional DMA (Direct Memory Access) engine. We have assembled a minimal necessary set of operations into a runtime component that is part of the R-Stream runtime layer.

Virtual DMA runtime

The runtime API is made of three functions: asynchronous two-dimensional get and put and a wait function. Tags represent groups of asynchronous transfers that can be waited upon using the wait function. The functions' signatures are as follows:

```
void  x86_dma_get(void * src, void * dst, uint64_t bytes, uint64_t src_stride, uint64_t dst_stride, uint64_t count, int tag);

void x86_dma_put(void * src, void * dst, uint64_t bytes, uint64_t src_stride, uint64_t dst_stride, uint64_t count, int tag);

void x86_dma_wait(int tag);
```

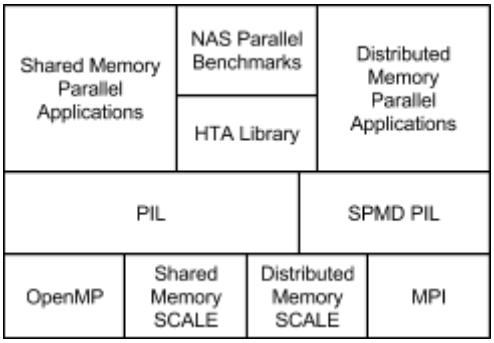
A 'get' is used to move data from DRAM to the virtual scratchpad and a 'put' is used to move data from the virtual scratchpad to DRAM.

The implementation of these functions is optimized to the SIMD nature of the underlying data transfer operations, maximizing the amount of aligned, vector transfers per function call. Because of the SIMD nature of the data transfer operations, calls representing aligned, block operations on the DRAM end (as opposed to the virtual scratchpad end) are more efficient than unaligned and strided ones.

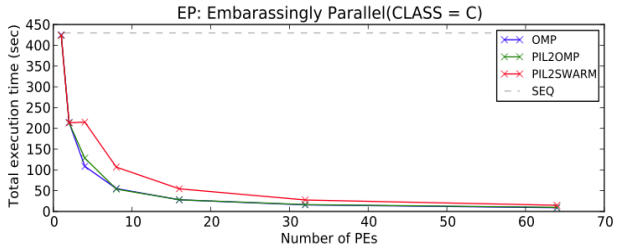
We expect that several extra optimizations will improve the virtual scratchpad technique further. We will present these in future quarterly reports.

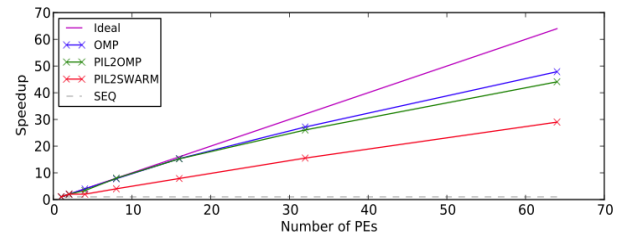
UIUC: NAS Parallel Benchmark Implementation

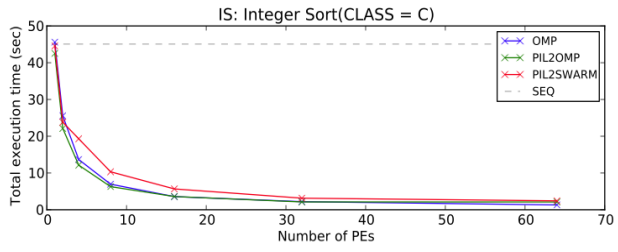
To evaluate the performance more realistic and complicated applications, six of the NAS Parallel Benchmarks (NPB) have been implemented in PIL using HTA library. We adapted from the serial C and OpenMP C implementations of SNU-NPB-1.0.3 developed at the Center for Manycore Programming of Seoul National University. The supported benchmarks are EP, IS, FT, LU, CG and MG. Their sizes vary from hundreds to a few thousands of lines. The applications are implemented in C language and they manage data and perform parallel operation by calling HTA library APIs.

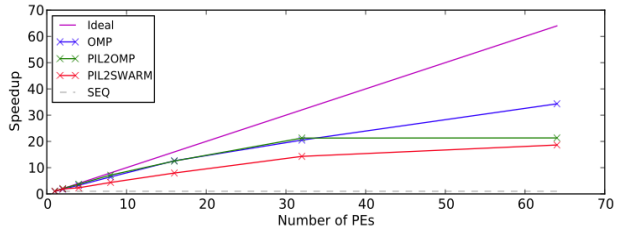


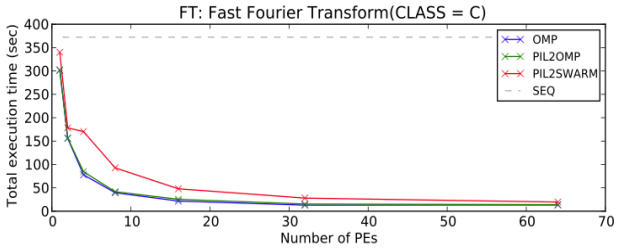
By using PIL compiler, it is possible to generate code for different backend target parallel programming languages without the need to change the application level code. Two different PIL backends for OpenMP and ETI SCALE (shared memory) are used in the performance comparison with a hand-coded OpenMP NPB implementation. Performance experiments are conducted on a multi-core shared memory machine with Intel Xeon E7-4860 CPU using up to 64 cores. The speedup is calculated using the serial C implementation execution time as the baseline.

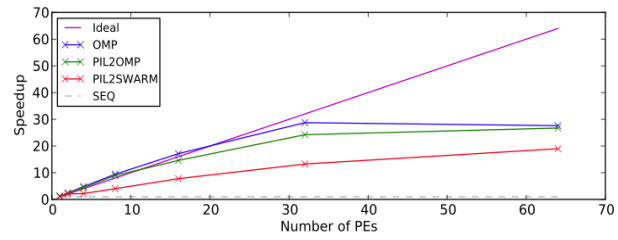


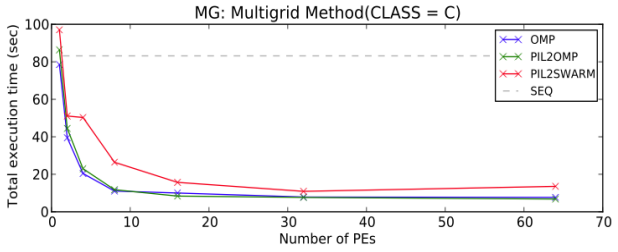


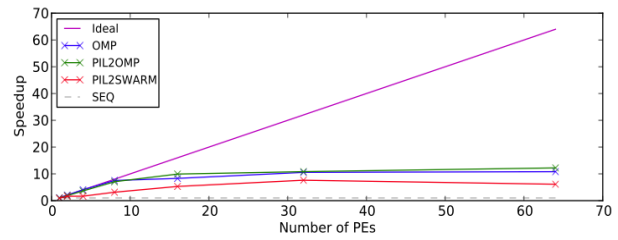


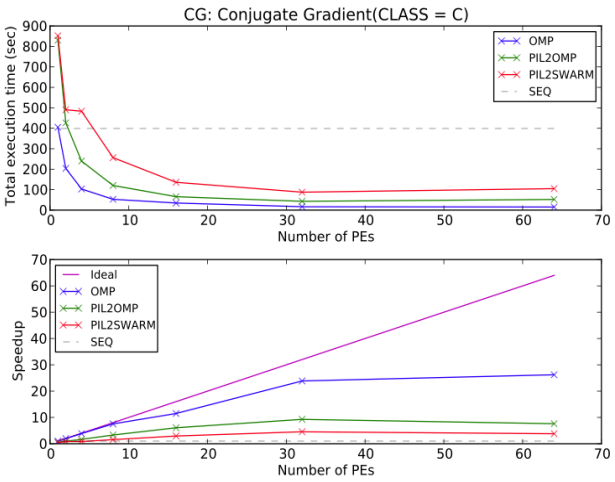


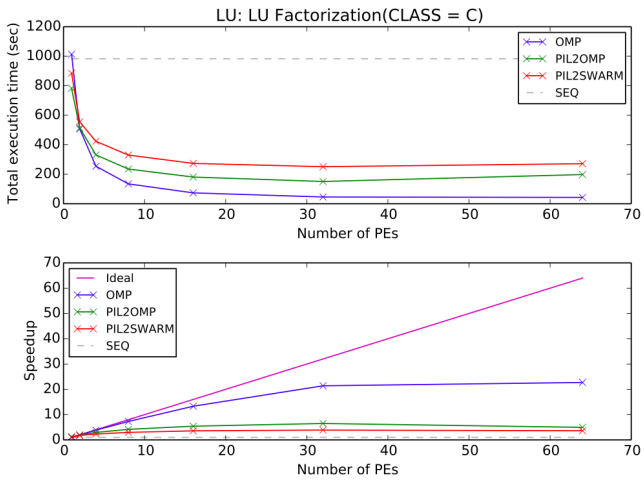






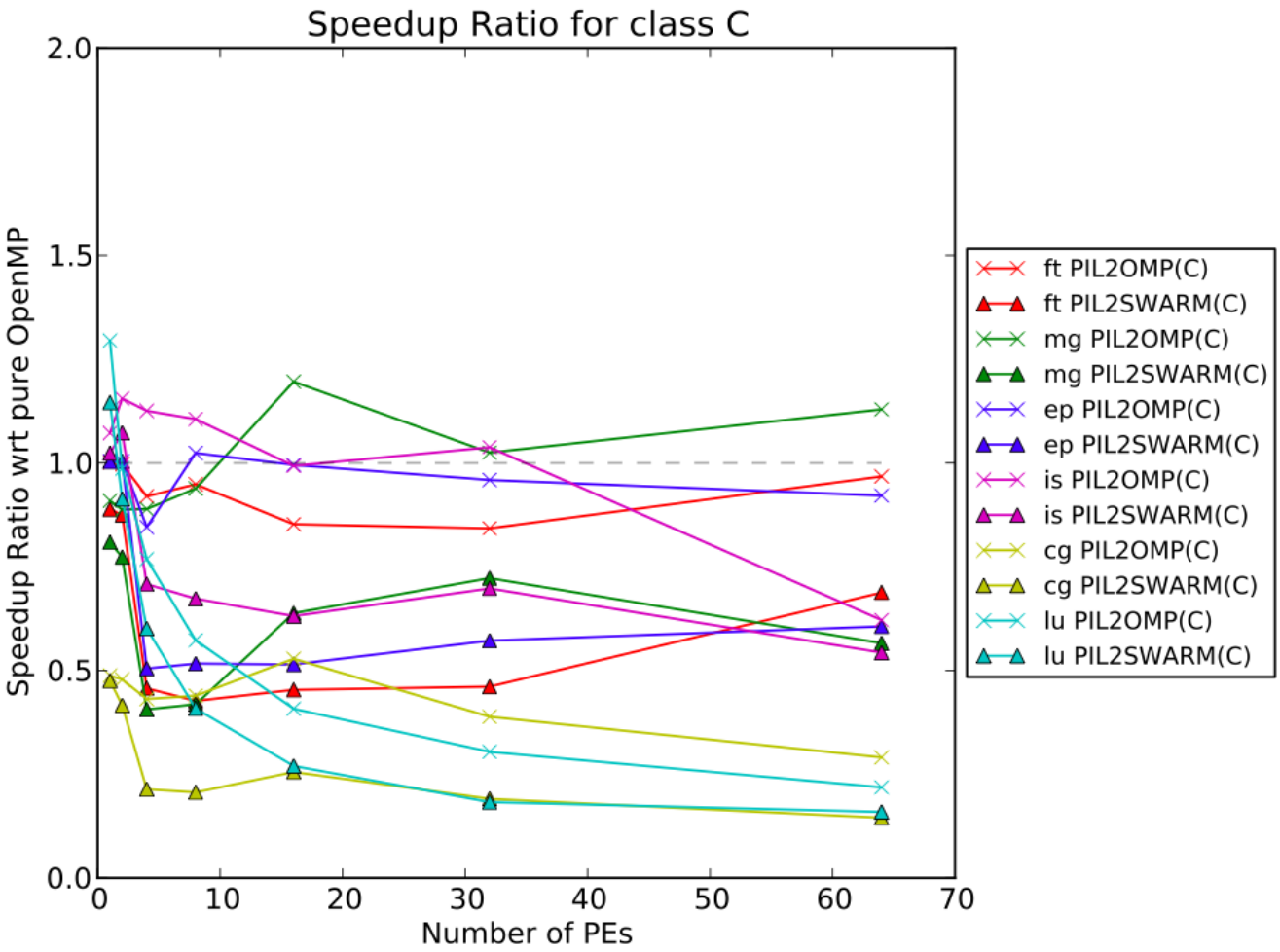






In general, the PIL-to-OpenMP version can achieve similar performance results with hand-coded OpenMP version (except for CG and LU, which use a different parallel algorithms), if the HTA operations are implemented to exploit the advantage of having global memory address space. For example, when a transposition or a circular shift is performed, swap the pointers instead of actually copying data. However, this approach is not general enough for the application to work on the distributed memory environments. We plan to make necessary modifications at the HTA library level and save the users from having to program for different memory address space.

The PIL-to-SCALE (shared memory) version shows worse performance results than the PIL-to-OpenMP version. From the following figure, we can see the speedup ratios of both versions compared with hand-coded OpenMP.



The lines with triangular markers are the PIL-to-SCALE results showing around 0.4~0.7 of the speedup of the hand-coded OpenMP version in most cases except for CG and LU. And we also found the performance scales badly when we increase the number of threads from 2 to 4. Further analysis is required to determine where the performance overhead comes from. It is possible that the code generation scheme in PIL-to-SCALE backend still requires optimization.

In the next quarter, we plan to cooperate with ETI team closely to figure out how to fine-tune and optimize the PIL-to-SCALE compiler backend, while also working on targeting distributed memory SCALE. Since the programming model for the distributed memory environment is significantly different from the shared memory environment, efforts are expected in: (1) changing the PIL compilation, and (2) HTA library implementation to hide the details of message passing from the application level.