Technical report OSU-CISRC-1/14-TR03

Automatic Generation of Coherence Instructions for Software-Managed Multiprocessor Caches

Sanket Tavarageri
Department of Computer Science and Engineering
The Ohio State University
tavarageri.1@osu.edu

Wooil Kim
Department of Computer Science
University of Illinois at Urbana-Champaign
kim844@illinois.edu

Josep Torrellas
Department of Computer Science
University of Illinois at Urbana-Champaign
torrella@illinois.edu

P Sadayappan
Department of Computer Science and Engineering
The Ohio State University
sadayappan.1@osu.edu

Abstract

The advent of multi-core processors with a large number of cores and heterogeneous architecture poses challenges for achieving scalable cache coherence. Several recent research efforts have focused on simplifying or abandoning hardware cache coherence protocols However, this adds a significant burden on the programmer, unless automated compiler support is developed.

In this paper, we develop compiler support for parallel systems that delegate the task of maintaining cache coherence to software. Algorithms to automatically insert software cache coherence instructions into parallel applications are presented. This frees the programmer from having to manually insert coherence primitives, which can be tedious and is error-prone. Experimental evaluation over a number of benchmarks demonstrates that effective compiler techniques can make software cache coherence competitive with hardware coherence schemes both in terms of energy and effectiveness in preserving cache data locality.

1. Introduction

Continuous transistor scaling has enabled increasing amounts of logic and memory on a chip. The resulting higher computational density has been used to design highly-capable many-core compute accelerators. These platforms are programmed with highly-parallel applications, which are partitioned into many threads running in parallel on the different cores.

Unlike general-purpose parallel processors, these compute accelerators do not necessarily have hardware-coherent caches. For example, commercial Graphics Processing Units (GPUs) do not have full cache coherence [1, 2]. There are several reasons behind this lack of hardware cache coherence.

First, hardware cache coherence comes with a non-trivial implementation cost. It is difficult to design and verify cache coherence protocols completely [3]. Furthermore, coherence structures such as the directory storage, typically consume a considerable amount of chip area. In many compute accelerators, such chip area is best applied to increase the computational capabilities.

Secondly, the programs that run on accelerators have intrinsically less need for full hardware cache coherence. This is because these programs are usually written in epoch-based form. They have little data sharing among cores within an epoch, while most communication occurs across epoch boundaries. Consequently, the coherence requirements of the workloads can be largely enforced at epoch boundaries.

There has been significant interest in trying to understand how to best support cache coherence in accelerator architectures. At one extreme, GPU architectures do not have any hardware cache coherence. The Intel Single Chip Cloud processor [17, 21] does not provide hardware cache coherence, but instead defines a new type of memory to facilitate communication between cores. Other designs provide limited cache coherence for programmability, but with lower hardware overhead. For example, Runnemede from the DARPA UHPC program [5] provides scratchpads and software-managed incoherent caches, shifting the responsibility of coherence to the software. The Rigel accelerator architecture has support for a hybrid hardware-software cache coherence [19].

At the other extreme, the Intel Xeon Phi accelerator has fully coherent caches, largely motivated from keeping the task of programming as close as possible to existing models. Thus we have a range of current trends, because of the different demands and constraints. At one end, we have fully non-coherent caches in GPUs, which makes the hardware more easily scalable but forces a restrictive programming model and places a greater burden on application developers. At the other end, the Xeon Phi has sacrificed some hardware scalability in order to make the programming model easy for users by providing fully coherent caches. Software controlled caches could provide the best of both worlds if compilers could shoulder the burden: provide users the same epoch-based parallel programming model like OpenMP that systems like the Xeon Phi provide, but allowing a simpler and more scalable hardware design like non-coherent GPUs.

In this paper, we present some compiler advances that make software coherent systems more attractive. We divide problem domains into applications with regular memory accesses and irregular ones. For regular memory-access programs, we develop algorithms to precisely mark variables for invalidation in a core's private cache — because they have become stale due to other cores' writes. We also develop algorithms to accurately determine data that have to be written back from the private caches to shared caches because other cores need to access the data in future epochs.

For iterative irregular applications we present *inspector*-based schemes that exactly demarcate data for coherence. Other irregular parallel applications are handled via conservative methods that write back and invalidate data across synchronization points. These schemes do not preserve cache locality across synchronization points, but still enable data reuse in the cache by keeping read-only data if possible.

Compared to prior work on compiler-directed cache coherence [7, 9, 10], the compiler support developed in this paper is more general. It is applicable to a larger class of programs, and the compiler analysis is more precise, as it takes into account the task-to-processor assignments.

The contributions of this paper are:

- Compiler algorithms to automatically instrument parallel applications with cache management instructions that write back and invalidate cached data.
- For affine computations, algorithms to precisely identify data for cache coherence using the Polyhedral-model [14];
- Efficient techniques using the inspector-executor paradigm to ensure coherence for recurrent iterative irregular computations.
- An experimental demonstration using several programs that the compiler-based techniques we develop are competitive with hardware coherence schemes in terms of performance and energy consumption, at a lower hardware cost.

2. Overview

A computer system with software managed cache coherence does not implement cache coherence protocols in hardware. It is necessary to explicitly insert coherence instructions in a parallel program to ensure correct execution. The software orchestrates cache coherence using the following coherence primitives.

- **Writeback:** The address of a variable is specified in the instruction and if the addressed location exists in the private cache and has been modified, then it is written to a shared later level cache or main memory.
- **Invalidate:** The instruction causes any cached copy of the variable in the private cache to be discarded (*self-invalidation*) so that the next read to the variable fetches data from shared later level cache.

The above coherence operations provide a mechanism for two processors to communicate with each other: if processor A has to send an updated value of a shared variable X to processor B, then processor A issues a *writeback* instruction on X, and processor B later invalidates X so that a subsequent read to X fetches the latest value from the shared cache.

Fig. 1 shows the API for the *invalidate* and *writeback* instructions. These API functions use arguments at the granularity of word, double-word, or quad-word. The *invalidate_range* and *writeback_range* functions have a start address and number of bytes as parameters.

In this paper, we address the question of *how to automatically generate cache coherence instructions for execution on software*

```
invalidate_word(void *addr);
invalidate_dword(void *addr);
invalidate_qword(void *addr);
invalidate_range(void *addr, int num_bytes);

writeback_word(void *addr);
writeback_dword(void *addr);
writeback_qword(void *addr);
writeback_range(void *addr, int num_bytes);
```

Figure 1: Coherence API list

```
1     for (t = 0; t <= tsteps-1; t++) {
2   #pragma omp parallel for
3     for (i = 2; i <= n-2; i++) {
4       S1: B[i]=0.33333*(A[i-1]+A[i]+A[i+1]);
5     }
6   #pragma omp parallel for
7     for (i = 2; i <= n-2; i++) {
8       S2: A[i]=B[i];
9     }
10    }
```

Figure 2: 1-d Jacobi stencil

```
    for (t = 0; t <= tsteps-1; t++) {
  #pragma omp parallel for
    for (i = 2; i <= n-2; i++) {
      invalidate_dword(&A[i-1]);
      invalidate_dword(&A[i]);
      invalidate_dword(&A[i+1]);
      S1: B[i]=0.33333*(A[i-1]+A[i]+A[i+1]);
      writeback_dword(&B[i]);
    }
  #pragma omp parallel for
    for (i = 2; i <= n-2; i++) {
      invalidate_dword(&B[i]);
      S2: A[i]=B[i];
      writeback_dword(&A[i]);
    }
  }
```

Figure 3: 1-d Jacobi stencil for SCC (unoptimized)

managed caches. The variables that potentially hold stale data - *invalidate set* - have to be identified in order that copies of those variables in the private cache are discarded; the data that are produced at a processor, but might be consumed at other processors - *writeback set* - need to be characterized so that those data are written to the shared cache and are available to other processors for future reads.

In this section, we use an example to provide an overview of the analyses performed and optimizations applied by the algorithms developed in the paper. Figure 2 shows a 1d-jacobi stencil code. In an iteration of the stencil computation, elements of array B are updated using three neighboring elements of array A in parallel (line 4). Then, all values of array B are copied to array A in parallel (line 8). In the next iteration, the new values of array A are used to update elements of array B. This process is repeated tsteps times (loop at line 1).

The parallelism in the computation is realized using OpenMP work-sharing constructs: the loops with iterators i are declared parallel (line 3 and 7), and different iterations of those loops may be assigned to different processors to execute them in parallel. The value written to B[i] by first statement (S1) is read by the second statement (S2). Since a different processor may execute the statement producing B[i] than those that consume B[i], B[i] has to be written-back to shared cache after S1 and is invalidated before S2.

Similarly, array element A[i] is written at S2 and has uses in three iterations of the first loop at S1. The reference A[i] must therefore be written-back after S2, and invalidated before S1. The code obtained thus for Software Cache Coherence (SCC) is shown in Figure 3.

However, there are opportunities to improve performance of the shown code on a software cache coherence system:

1. If iterations of the parallel loops in Figure 2 are mapped to processors such that an iteration with a certain value of i is assigned to the same processor in the first and second loops (line 3 and line 7), then B[i] is produced and consumed at the same proces-

```
for (t = 0; t <= tsteps-1; t++) {
 #pragma omp parallel private(myid, i1, i2) {
   myid = omp_get_thread_num();
   for(i1=myid;i1<=floor((n-2)/16);i1+=8) {
    if (t >= 1) {
     invalidate_dword(&A[16*i1 -1]);
     invalidate_dword(&A[16*i1 +16]);
    }

    for(i2=max(i1*16,2);i2<=min(i1*16+15,n-2);i2++){
     S1: B[i2]=0.33333*(A[i2-1]+A[i2]+A[i2+1]);
    }

    if (t == tsteps-1)
     writeback_range(&B[i1*16], sizeof(double)*16);
   }
 }

 #pragma omp parallel private(myid, i1, i2) {
  myid = omp_get_thread_num();
  for(i1=myid;i1<=floor((n-2)/16);i1+=8) {
   for(i2=max(i1*16,2);i2<=min(i1*16+15,n-2);i2+=1){
     S2: A[i2]=B[i2];
   }

   if (t <= tsteps-2) {
     writeback_dword(&A[16*i1]);
     writeback_dword(&A[16*i1+15]);
   }
   else if (t == tsteps-1) {
    writeback_range(&A[i1*16], sizeof(double)*16);
   }
  }
 }
}
```

Figure 4: 1-d Jacobi stencil for execution on an 8-processor SCC system

sor. Therefore, writing back of B[i] at S1, and invalidating of it at S2 can be avoided, resulting in lower overhead and better cache locality (read misses to array B in the second loop will be avoided).

Thus, analysis to compute *invalidate* and *writeback* sets that considers the iteration-to-processor mapping can avoid many potentially conservative coherence operations.

2. Further, if iterations are mapped to processors in a block-cyclic manner, fewer invalidations and write-backs will be required. Consider for example that the iterations of the parallel loops are scheduled to processors in a block-cyclic manner with a chunk-size of 16. In that scenario, a processor writes to a consecutive set of words at S2 (line 8 in Figure 2) — from A[k+1] to A[k+16] (for some 'k') and the same processor in the next iteration reads elements A[k] to A[k+17] at S1 (due to line 4). Therefore, only A[k], and A[k+17] have to be invalidated. Equally, other processors would only reference A[k+1] and A[k+16], and hence, only those two array cells have to be written-back.

With the above considerations, one can proceed as follows.
1) By explicitly mapping iterations of the two parallel loops to processors through the use of myid and by pinning threads to cores, we can reduce the number of coherence operations;
2) The parallel iterations are distributed among processors in a block-cyclic manner.

The resulting code is shown in Figure 4. The very last write to a variable by any processor is also written-back so that results of the computation are available at the shared cache.

The algorithms that we develop in the paper automatically produce the code in Figure 4 by performing a) an exact data dependence analysis (Section 4.1), b) iteration-to-processor mapping aware code generation (Section 4.2).

3. Background

3.1 Execution Model

Release Consistency: The execution of parallel programs consists of *epochs* (intervals between global synchronization points). In an epoch, data that were written potentially by other cores in previous epochs and that a core may need to read in the epoch are *invalidated*. Before the end of the epoch, all the data that a core has written in the epoch and that may be needed by other processors in future epochs are *written-back* to the shared level cache.

Before an epoch completes, all prior memory operations, including ordinary load/store instructions and coherence instructions, are completed. Then the next epoch can start, and the following memory operations can be initiated. Further, ordering constraints between memory instructions are respected: The order of a store to address i and the following writeback for address i should be preserved in the instruction pipeline of the processor and caches. Similarly, the order of invalidation to address j and a following load from address j should be preserved in the pipeline and caches to guarantee fetching of the value from the shared cache.
Coherence Operations at Cache line granularity: Coherence operations are carried out at the granularity of cache lines — all the lines that overlap with specified addresses are invalidated or written-back. If the specified data are not present in cache, then coherence instructions have no effect.

In addition, *writeback instructions write back only dirty words of the line*. In doing so, writeback instructions avoid the incorrectness issue that may arise from false sharing: if two processors are writing to variables that get mapped to the same cache line, and whole cache lines (and not just the dirty words) are written-back, then one processor's dirty words may be overwritten with another processor's clean words. Therefore, per-word dirty bits are used to keep track of words of a cache line that are modified.

3.2 Notation

The code shown in Fig. 5 is used as a working example to illustrate the notation and the compiler algorithm in the next section.
Sets: A set s is defined as:

$$s = \{[x_1, \ldots, x_m] : c_1 \wedge \cdots \wedge c_n\}$$

where each x_i is a tuple variable and each c_j is a constraint.

The iteration spaces of statements can be represented as sets. For example, the iteration space of statement S1 in the code shown in Fig. 5 can be specified as the set I^{S_1}:
$I^{S_1} = \{S1[t_1, t_2, t_3] : (0 \le t_1 \le tsteps - 1) \wedge (0 \le t_2 \le n-1) \wedge (1 \le t_3 \le n-1)\}$
Relations: A relation r is defined as:

$$r = \{[x_1, \ldots, x_m] \mapsto [y_1, \ldots, y_n] : c_1 \wedge \cdots \wedge c_p\}$$

```
for (t1=0;t1<=tsteps-1;t1++) {
 #pragma omp parallel for private(t3)
   for (t2=0;t2<=n-1;t2++) {
     for (t3=1;t3<=n-1;t3++) {
       S1: B[t2][t3] = B[t2][t3+1] + 1;
     }
   }
}
```

Figure 5: A loop nest

where each x_i is an input tuple variable, each y_j is an output tuple variable and each c_k is a constraint.

Array accesses appearing in the code may be modeled as relations from iteration spaces to access functions of the array references. The two accesses to array 'B' in Fig. 5, B[t2][t3] and B[t2][t3+1], are represented as the following relations:

$$r^{S_1}_{write} = \{S1[t_1,t_2,t_3] \mapsto B[t'_2,t'_3] : (t'_2 = t_2) \wedge (t'_3 = t_3)\}$$
$$r^{S_1}_{read} = \{S1[t_1,t_2,t_3] \mapsto B[t'_2,t'_3] : (t'_2 = t_2) \wedge (t'_3 = t_3 + 1)\}$$

The Apply Operation: The apply operation on a relation r and a set s produces a set s' denoted by, $s' = r(s)$ and is mathematically defined as:

$$(\vec{x} \in s') \iff (\exists \vec{y} \text{ s.t. } \vec{y} \in s \wedge (\vec{y} \mapsto \vec{x}) \in r)$$

The set of array elements accessed by an array reference in a loop (data-footprint) may be derived by *applying* access function *relations* on the iteration space *sets*. For the array accesses in the example code shown in Fig. 5, data-footprints of the two accesses are: $r^{S_1}_{write}(I^{S_1}), r^{S_1}_{read}(I^{S_1})$.

The Inverse Operation: The inverse operation $r = r_k^{-1}$ operates on a relation r_k to produce a new relation r such that r has the same constraints as r_k but with the input and output tuple variables swapped. $(\vec{x} \mapsto \vec{y} \in r) \iff (\vec{y} \mapsto \vec{x} \in r_k)$.

3.3 Polyhedral Dependences

In the Polyhedral model [14, 29], for affine computations, dependence analysis [13] can precisely compute flow (Read After Write - RAW) and output (Write After Write - WAW) dependences between dynamic instances of statements. The dependences are expressed as maps from source iterations to target iterations involved in the dependence.

The flow dependence determined by polyhedral dependence analysis (for example, using ISL [27]) for the code in Fig. 5 is:

$$\mathcal{D}_{flow} = \{S1[t_1,t_2,t_3] \mapsto S1[t_1+1,t_2,t_3-1] :$$
$$(0 \le t_1 \le tsteps-2) \wedge (0 \le t_2 \le n-1) \wedge (2 \le t_3 \le n-1)\}$$

The relation characterizes the flow dependence that exists between the write reference B[t2][t3] and the read reference B[t2][t3+1].

An analysis tool like ISL can also be used to emit information regarding *live-in* data: data that are read in the loop but are not produced by any statement instances in the scope of analysis. A list containing maps from an iteration point that reads live-in data to the live-in array elements that it reads is computed. For the running example, the live-in maps are:

$$\mathcal{D}_{live-in} =$$
$$\{S1[0,t_2,t_3] \mapsto B[t_2,t_3+1] : 0 \le t_2 \le n-1 \wedge 1 \le t_3 \le n-2;$$
$$S1[t_1,t_2,n-1] \mapsto B[t_2,n] : 0 \le t_1 \le tsteps-1 \wedge 0 \le t_2 \le n-1 \quad \}$$

The two maps capture live-in data read for read reference B[t2][t3+1]: a) at the first iteration of the outermost t1 loop, all read elements are live-in b) for later iterations, only the n^{th} element in each row of matrix B is live-in, since it is never written to by the write reference B[t2][t3].

4. Compiler Optimization for Regular Codes

The iteration space of an epoch in a parallel loop is modeled by considering iterator values of the parallel loop and its surrounding loops as parameters. In the parallel loop in Fig. 5, the t2 loop is parallel and an iteration of t2 constitutes a *parallel task* executed in an epoch. Its iteration space is modeled by considering values of iterators t1 and t2 as parameters - t_p and t_q respectively:
$I^{S_1}_{current} = \{S1[t_1,t_2,t_3] : (t_1 = t_p) \wedge (t_2 = t_q) \wedge (1 \le t_3 \le n-1)\}$.

4.1 Computation of Invalidate and Writeback Sets

Invalidate Set: The *invalidate set* for a parallel task consists of data that are consumed in the parallel task but produced elsewhere. Thus, the invalidate set is the data consumed by those iterations of the parallel task that are targets of some flow dependence (RAW) whose source iterations are *not* in the same parallel task. The live-in data (first reads to any variable in the loop) are also invalidated and hence forms a part of the invalidate set.

Algorithm 1 shows how the invalidate set for a parallel task is computed. It is computed by forming the union of invalidate data sets corresponding to all statements within the parallel loop by iterating over each statement. For each statement S_i, first the source iterations of the dependence - I_{source} - whose target iterations are in the current slice for that statement - $I^{Si}_{current}$ - are determined by applying the inverse relation of the flow dependence. From this set, any of the source iterations that lie in the current slice - $\bigcup_{Sj \in stmts} I^{Sj}_{current}$, are removed from I_{source} because the source and target iterations are run on the same processor and no coherence instruction is needed. The array elements written by iterations of I_{source} are placed in the set of data elements for which invalidation coherence instructions must be issued to guarantee coherence. To this set is added the live-in list corresponding to data elements that come in live from outside the analyzed region.

Algorithm 1 Compute Invalidate Set

Input: Flow Dependences : $\mathcal{D}_{flow}$, Live-in read maps : $\mathcal{D}_{live\_in}$, Current Iteration Slices: $I_{current}$, Write maps: r_{write}

Output: Statement and Invalidate set pairs: $D^{Si}_{invalidate}$

```
1: for all statements - Si do
2:     $D^{Si}_{invalidate} \leftarrow \phi$
3:     $I_{source} \leftarrow \mathcal{D}^{-1}_{flow}(I^{Si}_{current}) \setminus (\bigcup_{Sj \in stmts} I^{Sj}_{current})$
4:     $D_{inflow} \leftarrow \bigcup_{Sj \in stmts} r^{Sj}_{write}(I_{source})$
5:     $D_{live\_in\_data} \leftarrow \mathcal{D}_{live\_in}(I^{Si}_{current})$
6:     $D^{Si}_{invalidate} \leftarrow (D_{inflow} \cup D_{live\_in\_data})$
7: end for
```

Example: The application of the algorithm to the running example results in the following invalidate set: $D^{S_1}_{invalidate} = \{[t_q, i_1] : 2 \le i_1 \le n\}$. The array elements read in the parallel task are marked for invalidation.

Writeback Set: The *writeback set* consists of all data that are produced by a parallel task and consumed outside of the parallel task. Thus we need to identify iterations of the parallel task that are sources of a flow dependence whose targets lie elsewhere. Once we have identified such source iterations, we then determine the array elements written by them. The *last write* to a variable also belongs to the writeback set and last writes are found by using output dependence (WAW) information: iterations which are not sources of any output dependence must be last writers to the array elements that they write to.

Algoirthm 2 shows how we compute the writeback set for a parallel task that possibly has multiple statements in it. To find the writeback set corresponding to a statement S_i, first all target iterations (I_{target}) of all dependences are identified whose source iterations lie in $I^{Si}_{current}$. Those target iterations that are within the same parallel task - $\bigcup_{Sj\in stmts} I^{Sj}_{current}$ are removed from I_{target} (line 3). Then the inverse dataflow relation is applied to this set and the intersection to the current iteration slice is computed (line 4) to identify the source iterations ($I_{producer}$) in the slice that write values needed outside this slice. These values must be part of the writeback set.

Further, if a write by an iteration is the last write to a certain variable, it must also be written back since it represents a live-out value from the loop. The iterations that are not sources of any output dependencies produce live-out values. Such iterations are determined by forming the set difference between $I^{Si}_{current}$ and domain of output dependences - $dom\ \mathcal{D}_{output}$.

Algorithm 2 Compute Writeback Set

```
Input: Flow Dependences : $\mathcal{D}_{flow}$, Output Dependences : $\mathcal{D}_{output}$, Current Iteration Slices: $I_{current}$, Write maps: $r_{write}$
Output: Statement and Writeback set pairs: $D^{S_i}_{writeback}$
1: for all statements - Si do
2:     $D^{S_i}_{writeback} \leftarrow \phi$
3:     $I_{target} \leftarrow \mathcal{D}_{flow}(I^{Si}_{current}) \setminus (\bigcup_{Sj\in stmts} I^{Sj}_{current})$
4:     $I_{producer} \leftarrow \mathcal{D}^{-1}_{flow}(I_{target}) \cap I^{Si}_{current}$
5:     $D_{outflow} \leftarrow r^{Si}_{write}(I_{producer})$
6:     $I_{live\_out} \leftarrow I^{Si}_{current} \setminus dom\ \mathcal{D}_{output}$
7:     $D_{live\_out\_data} \leftarrow r^{Si}_{write}(I_{live\_out})$
8:     $D^{S_i}_{writeback} \leftarrow (D_{outflow} \cup D_{live\_out\_data})$
9: end for
```

Example: The algorithm produces the following writeback set for the example in Fig. 5: $D^{S_1}_{writeback} = \{[t_q, i_1] : (t_p \leq tsteps - 2 \wedge 2 \leq i_1 \leq n-1) \vee (t_p = tsteps - 1 \wedge 1 \leq i_1 \leq n-1)\}$.

For 0 to tsteps-2 iterations of the outermost t1 loop, only elements B[t2][2:n-1] need to be written back as they will be read in the next iteration of t1 loop. Array cell B[t2][1] does not need to be written back because it is overwritten in a later t1 iteration and its value is not read. But the very last write to B[t2][1], i.e., when t1 = tsteps-1 has to be written back as it is a live-out value of the loop.

Code Generation The invalidate and writeback sets are translated to corresponding cache coherence instructions by generating a loop to traverse elements of the sets using a polyhedral code generator — ISL [27]. The invalidations and writebacks are combined into coherence range functions whenever elements of a set are contiguous in memory: when the inner-most dimension of the array is the fastest varying dimension of the loop.

4.2 Optimization

Analysis Cognizant of Iteration to Processor Mapping The techniques described until now do not assume any particular mapping of iterations to processors. However, if a mapping of processors to iterations is known, many invalidations and write-backs could possibly be avoided. For example, in the code shown in Fig. 5, the flow dependence (mentioned in §3.3) is: $S1[t_1,t_2,t_3] \mapsto S1[t_1+1,t_2,t_3-1]$. If parallel iterations of the ‘t2’ loop are mapped to processors such that an iteration with a particular ‘t2’ value always gets mapped to the same processor, the source and target iterations of the flow dependence get executed on the same processor,

```
for (t1=0;t1<=tsteps-1;t1++)
#pragma omp parallel private (myid,t2,t3) {
 myid = omp_get_thread_num();
 for (t2=myid;t2<=n-1;t2+=8) {
  if (t1 == 0) {
   invalidate_range(&B[t2][2], sizeof(double)*(n-2));
  }
  invalidate_dword(&B[t2][n]);
  for (t3=1;t3<=n-1;t3++) {
   S1: B[t2][t3] = B[t2][t3+1] + 1;
  }
  if (t1 == tsteps-1) {
   writeback_range(&B[t2][1], sizeof(double)*(n-1));
  }
 }
}
```

Figure 6: Optimized loop nest for SCC

making invalidations and write-backs due to the dependence unnecessary.

In order to incorporate this optimization, Algorithm 1 and 2 are modified to take iteration to processor mapping into account. Line 3 of Algorithm 1 is now changed to:
$I_{source} \leftarrow \mathcal{D}^{-1}_{flow}(I^{Si}_{current}) \setminus (\bigcup_{Sj\in stmts} I^{Sj}_{current} \cup I_{same\_proc})$
and line 3 of Algorithm 2 is changed to:
$I_{target} \leftarrow \mathcal{D}_{flow}(I^{Si}_{current}) \setminus (\bigcup_{Sj\in stmts} I^{Sj}_{current} \cup I_{same\_proc})$,
where I_{same_proc} is the set of iterations that is executed on the same processor as the processor on which $I_{current}$ is executed.

For the working example, let us say that the OpenMP scheduling clauses specify that iterations are cyclically mapped onto processors and the number of processors used is 8. Then, we encode that information into the following iteration to processor map: $r_{i2p} = \{S1[t_1,t_2,t_3] \mapsto [t'_2] : t'_2 = t_2 \mod 8\}$. The parallel region code is all the iterations that are mapped to a parametric processor ‘myid’: $I_{my\_proc} = r^{-1}_{i2p}(myid)$. The iteration set $I^{S_1}_{current}$ is a subset of I_{my_proc} with the values of the t1 and t2 loop iterators parameterized.

Using the modified algorithms, the cache coherence code generated for the working example is presented in Fig. 6. In the optimized code, only the live-in data is invalidated: elements B[t2][2 to n] at time-step $t_1 = 0$, only a single element – B[t2][n] at later time-steps, since other elements are written to by the same processor ensuring that the updated values are present in the processor’s private cache. Only the live-out data is written back at the last time-step: $t_1 = tsteps - 1$.

Iteration to Processor Mapping A *cyclic distribution* of iterations to processors yields a better load balance especially for triangular iteration spaces; a *block distribution* of iterations to processors on the other hand, maps consecutive iterations to the same processor and hence reduces the amount of invalidations if flow dependences exist mainly between consecutive iterations. Further, if successive iterations write to consecutive locations in memory, a block distribution of iterations may enable a thread to collect writebacks together and perform coherence operations on a set of consecutive array elements.

Therefore, for triangular iteration spaces, a block-cyclic distribution of iterations among processors is employed, for others a block distribution is performed.

5. Compiler Optimization for Irregular Codes

5.1 Basic Approach

The tasks that are executed in an epoch (interval between synchronization points) by construction do not have any dependences between them (otherwise, the dependences would induce serialization of tasks and hence, the tasks would have to be executed in different epochs). Therefore, all data accessed *within* an epoch can be safely cached and cache coherence is not violated.

```
invalidate_all();
writeback_all();
```

Figure 7: Coherence API for conservative handling

For irregular applications that have non-affine references and hence, are not amenable to the analysis presented in the previous section, software cache coherence is achieved conservatively: at the beginning of an epoch, the entire private cache is invalidated and at the end of the epoch, all data that are written in the epoch (dirty words) are written to the shared cache. The coherence API functions shown in Fig. 7 are inserted in the parallel program at epoch boundaries to conservatively manage software coherence.

The basic approach outlined above preserves intra-epoch cache data locality, but cannot exploit any temporal locality that exists across epoch boundaries.

5.2 Inspector-Executors

Many scientific applications use sparse and irregular computations and are often iterative in nature and furthermore, the data access pattern remains the same across iterations. (Examples include programs for solving partial differential equations, irregular stencils, the conjugate gradient method for solving systems of linear equations which uses sparse matrix-vector multiplications, atmospheric simulations that use semi-regular grids).

```
while (converged == false)
{
 #pragma omp parallel for
 for(i=0;i<n;i++) {
   read A[B[i]]; /*data-dependent access*/
 }

 #pragma omp parallel for
 for(i=0;i<n;i++) {
   write A[C[i]]; /*data-dependent access*/
 }
 /*Setting of converged variable not shown*/
}
```

Figure 8: An iterative loop with irregular data references

For such codes, we propose the use of *inspectors* to gather information on irregular data accesses so that coherence operations are applied only where necessary. The inspectors that are inserted in the parallel codes are themselves parallel and are lock-free. The cost of inspectors is amortized by the ensuing selective invalidations of data and thus fewer unnecessary L1 cache misses over many iterations of the iterative computation.

Fig. 8 shows an iterative code that has data-dependent references to a one-dimensional array, viz., A[B[i]] and A[C[i]]. We first illustrate the inspector approach for the simple example. The ideas are more generally applicable in the presence of multiple arrays and multi-dimensional arrays.

The inspector-code determines if a) the write performed at a thread has readers at other threads: if that is the case, the variable has to be written-back to shared cache so that other threads will be able to obtain the updated value of the variable. b) the variable being read at a thread was written by another thread: if yes, the variable has to be invalidated at the private cache so that the fresh value is got from shared cache.

Fig. 9 presents the inspector-inserted parallel code corresponding to the iterative loop shown in Fig. 8 for execution on software

```
1  /*Inspector code begins*/
2  #pragma omp parallel for
3  for(i=0;i<n;i++) {
4  A_thread[i] = -1;
5  A_conflict[i] = 0;
6  writeback_word(&A_thread[i]);
7  writeback_word(&A_conflict[i]);
8  }
9
10 //Phase 1: Record writer thread ids
11 #pragma omp parallel for
12 for(i=0;i<n;i++) {
13   A_thread[C[i]] = myid;
14   writeback_word(&A_thread[C[i]]);
15 }
16
17 //Phase 2: Mark conflicted if
18 //writer and reader threads are not the same
19 #pragma omp parallel for
20 for(i=0;i<n;i++) {
21  invalidate_word(&A_thread[B[i]]);
22  if(A_thread[B[i]] != -1 && A_thread[B[i]] != myid) {
23    A_conflict[B[i]] = 1;
24    writeback_word(&A_conflict[B[i]]);
25  }
26 }
27 /*Inspector code ends*/
28
29  #pragma omp parallel
30  { invalidate_all(); }
31
32 while (converged == false)
33 {
34  #pragma omp parallel for
35  for(i=0;i<n;i++) {
36   if(A_thread[B[i]] != -1 && A_thread[B[i]] != myid)
37    invalidate_word(&A[B[i]]);
38
39    read A[B[i]];
40  }
41
42  #pragma omp parallel for
43  for(i=0;i<n;i++) {
44    write A[C[i]];
45
46    if(A_conflict[C[i]] == 1)
47     writeback_word(&A[C[i]]);
48  }
49  /*Setting of converged variable not shown*/
50 }
51
52  #pragma omp parallel
53  { writeback_all(); }
```

Figure 9: An iterative loop with irregular data references for SCC system

managed caches. Two shadow arrays — A_thread and A_conflict for array A that has data-dependent accesses are initialized (lines 4, 5). In the *first phase*, A_thread records the ids of the threads that write to array cells (line 13). In the *second phase*, if an array cell is read by a thread different from the writer thread, the corresponding cell in A_conflict array is set to 1 (line 23). Since the computation loops are parallel, the inspection is also carried out in parallel. Consequently, accesses to arrays A_thread and A_conflict are guarded with coherence instructions. If there are multiple readers for an array cell then more than one thread may set the respective cell of A_conflict to 1 in phase two and multiple threads will write-back the same value, namely 1 to shared cache (in line 24). Since the same value is being written, any ordering of writes by different threads works.

Later in the computation loops, a thread invalidates a variable (line 37) before reading it if the variable has a writer (as opposed to read-only data) and that writer is a different thread. A thread after writing to a variable, writes it back (line 47) if the variable is marked conflicted.

5.3 Exclusion of Read-Only Data from Coherence

```
/* Prologue begins */
writeback_all();
 #pragma omp parallel
 { invalidate_all(); }
/* Prologue ends */
while (condition)
{
 #pragma omp parallel
   {
      /* regular/irregular code */
   }
}

/* Epilogue begins */
 #pragma omp parallel
 { writeback_all(); }
invalidate_all();
/* Epilogue ends */
```

Figure 10: An iterative loop

For irregular codes whose data access patterns potentially change with each iteration, we adopt a conservative approach that yet excludes read-only data from coherence enforcement and thus, is more accurate than a full invalidation and writeback approach outlined earlier.

We consider *parallel regions* — parallel loops along with surrounding looping structures and perform analysis of the parallel region as a stand-alone unit. The *read-only data of the parallel region* need not be invalidated/written-back. Only those variables that are both written and read in the parallel region are invalidated and written-back at epoch boundaries.

For this scheme to work however, the following conditions have to be met:

1. None of the processors should have cached stale values of read-only data of the parallel region. (This could happen for example when, a program has a parallel region $\mathcal{P}$ followed by a sequential segment $\mathcal{Q}$ and later a parallel region $\mathcal{R}$. And, variable x is read-only in $\mathcal{P}$ and $\mathcal{R}$, but is modified in $\mathcal{Q}$).
2. Since, in the parallel region coherence is enforced only on data that are both read and written, for *written-but-not-read* data coherence operations should be introduced following the parallel region to ensure that future accesses to them get updated values.

To meet condition 1), a *prologue* is introduced that writes back all dirty words from the master thread and then does a full invalidation of caches at all threads. Condition 2) is fulfilled by writing-back all dirty words from all threads and doing a full-invalidation by the master thread in an *epilogue*. The code shown in Fig. 10 uses the outlined approach.

Algorithm 3 presents the overall parallel-region analysis technique.

6. Experimental Evaluation

We evaluate the performance of compiler-generated coherence instructions for execution of parallel programs on software managed caches. The main goal of the compiler support developed in the paper is to insert coherence instructions — *invalidate* and *writeback* functions only where necessary. The conservative invalidations (of non-stale data) result in read misses which lead to degraded performance relative to a hardware coherence scheme. Therefore, to assess efficacy of the compiler techniques, we compare *read misses in L1 caches*, and *execution time* on software and hardware managed caches. (The number of misses at the shared cache is unaffected and will be the same for software and hardware cache coherence.)

Conservative coherence operations in software scheme increase accesses to the shared cache and also, cause increased traffic on the system bus. The hardware cache coherence protocol uses control messages to maintain coherence, which a software scheme does not. Therefore, if the software coherence mechanism results in comparable cache misses as a hardware protocol then, the software coherence also reduces network traffic and cache energy. We therefore measure the *number of words transferred on the system bus* and *cache energy* by software and hardware coherence systems.

Table 1: Benchmarks

Benchmark	Description
gemm	Matrix-multiply : $C = \alpha.A.B + \beta.C$
gemver	Vector Multiplication and Matrix Addition
jacobi-1d	1-D Jacobi stencil computation
jacobi-2d	2-D Jacobi stencil computation
LU	LU decomposition
trisolv	Triangular solver
CG	Conjugate Gradient method
backprop	Pattern recognition using unstructured grid
hotspot	Thermal simulation using structured grid
kmeans	Clustering algorithm used in data-mining
pathfinder	Dynamic Programming for grid traversal
srad	Image Processing using structured grid

Table 2: Simulator parameters

Processor chip	8-core multicore chip
Issue width; ROB size	4-issue; 176 entries
Private L1 cache	32KB Write-back, 4-way, 2 cycle hit latency
Shared L2 cache	1MB Write-back, 8-way, multi-banked 11 cycle round-trip time
Cache line size	32 bytes
Cache coherence protocol	Snooping-based MESI protocol
Main Memory	300 cycle round-trip time

Algorithm 3 Generate Coherence Instructions using Parallel Region Analysis

```
Input: AST of Parallel region: $\mathcal{P}$
Output: AST of Parallel region for SCC: $\mathcal{P}_{SCC}$
 1: $Prologue \leftarrow$ API to write-back all dirty words from master thread; API to invalidate entire cache of all threads
 2: $Read\_Set \leftarrow$ Arrays and scalars that are read in $\mathcal{P}$
 3: $Write\_Set \leftarrow$ Arrays and scalars that are written in $\mathcal{P}$
 4: $Coherence\_Set \leftarrow Read\_Set \cap Write\_Set$
 5: for all epoch code $e \in \mathcal{P}$ do
 6:     $Invalidate\_Set_e \leftarrow Read\_Set_e \cap Coherence\_Set$
 7:     $Writeback\_Set_e \leftarrow Write\_Set_e \cap Coherence\_Set$
 8:     Insert API for $Invalidate\_Set_e$ and $Writeback\_Set_e$
 9: end for
10: $Epilogue \leftarrow$ API to write-back all dirty words from all threads; API to invalidate entire cache of master thread
11: $\mathcal{P}_{SCC} \leftarrow$ Append $\{Prologue, \mathcal{P}, Epilogue\}$
```

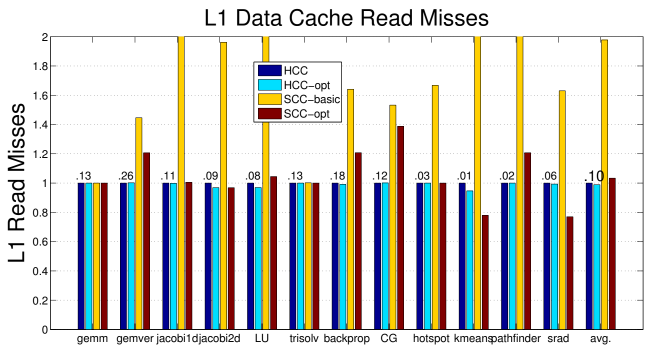

Figure 11: L1 data cache read misses (lower, the better). The L1 read miss ratios for HCC are also shown.

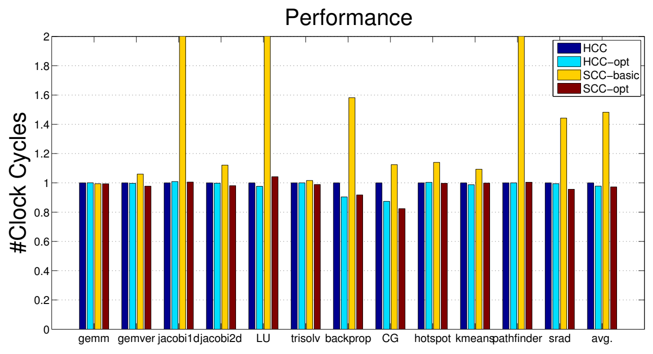

Figure 12: Execution time (lower, the better)

6.1 Benchmarks

The benchmark programs listed in Table 1 are used for the experiments. The benchmark codes are from Rodinia [6] and PolyBech [25] benchmark suites. The Rodinia suite provides parallel programs from various application domains. The backprop, hotspot, kmeans, pathfinder, srad applications are taken from Rodinia suite, and they contain affine as well as irregular data references. The PolyBench benchmark suite is a collection of widely used linear algebra, and stencil codes. The benchmark programs — gemm, gemver, jacobi-1d, jacobi-2d, LU, trisolv are taken from PolyBench suite. The codes are parallelized using a polyhedral compiler – PoCC [24]. All array references in the PolyBench programs are affine.

6.2 Set-up

The snooping-bus MESI protocol hardware coherence (referred to as *HCC* in the following text), and software cache coherence (referred to as *SCC*) have been implemented in an architectural multi-processor simulator — SESC [23]. Details of the simulator setup are described in Table 2.

We compare performance and energy of the following four coherence schemes:

1. **HCC:** Parallel programs are executed using MESI hardware coherence.
2. **SCC-basic:** The coherence instructions are inserted without iteration-to-processor aware analysis for affine references and without the use of inspector-executor or read-only data exclusion scheme for irregular accesses. That is, coherence instructions are generated with methods described in Sections 4.1 and 5.1 only without further optimizations. The resulting codes are run on software managed caches.
3. **SCC-opt:** The coherence management is optimized using compiler optimizations presented, and the resulting programs are executed on software managed caches.
4. **HCC-opt:** To study if any optimizations applied to SCC codes (such as explicit mapping of iterations to processors) can also benefit the benchmarks for hardware coherence, SCC-opt programs are adapted to run on HCC systems: coherence operations and any *inspectors* inserted are removed from SCC-opt codes and these variants are run on the HCC system.

The performances of only parallel parts of benchmarks are measured — sequential initialization and finalization codes are excluded from measurements because the performance of sequential code is expected to be the same on SCC and HCC systems. Threads are pinned to cores for both schemes.

6.3 Performance Results

Fig. 11 plots the read misses in L1 cache; Fig. 12 shows the execution time. The number of L1 read misses and execution cycles are normalized with respect to HCC statistics (the number of misses and execution cycles of HCC is considered 1). The L1 read miss ratios (fraction of L1 reads that are misses) for HCC are also indicated in the graph.

On average (geometric mean) across benchmarks, HCC-opt has the same number of cache misses as HCC; SCC-basic suffers 98% more misses and SCC-opt experiences only a 3% increase (avg. column in the graph). The geometric mean of normalized execution time for the three variants — HCC-opt, SCC-basic, and SCC-opt are, 0.97, 1.48, and 0.97 respectively. We observe that SCC-opt greatly improves performance over SCC-basic and brings down cache misses comparable to those of HCC. Further, performance of HCC-opt is very similar to that of HCC.

The gemm and trisolv benchmarks exhibit the so-called *communication free* parallelism: the outer loop in these codes is parallel. Therefore, there is no communication between processors induced by data dependences. All code variants of gemm and trisolv have virtually the same number of cache misses and execution cycles.

In applications that have irregular references, namely backprop, CG, hotspot, kmeans, pathfinder, srad, the parallel region boundaries are guarded with full-invalidation and full-writeback instructions (described in 5.3) The affine accesses in the parallel regions are optimized; irregular accesses are handled using *inspectors* or invalidation and write-back of entire arrays that are both written and read in the parallel region (read-only arrays and scalars are excluded).

For backprop and pathfinder, full invalidation of cache at parallel region boundaries results in some loss of data locality which results in increased L1 cache read misses.

The CG and srad benchmarks have iterative loops and irregular accesses whose indexing structures do not change across iterations. Therefore, for those two benchmarks, *inspector* codes are inserted for deriving coherence operations. The inspectors contribute to a certain number of L1 read misses. The reduced cache misses in SCC-opt of srad compared to its HCC counterpart is an artifact of the interaction that exists between cache coherence and cache replacement policy (LRU): false-sharing in HCC can cause soon-to-be-reused data to be evicted, which favors SCC. The migratory writes may sometimes cause invalidations of not-to-be-reused data and thus, making way for other to-be-reused data and this benefits

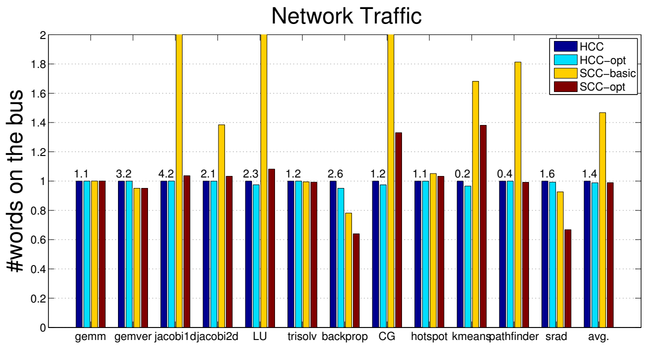

Figure 13: Traffic on the system bus (lower, the better). Average number of words per cycle for HCC is also shown.

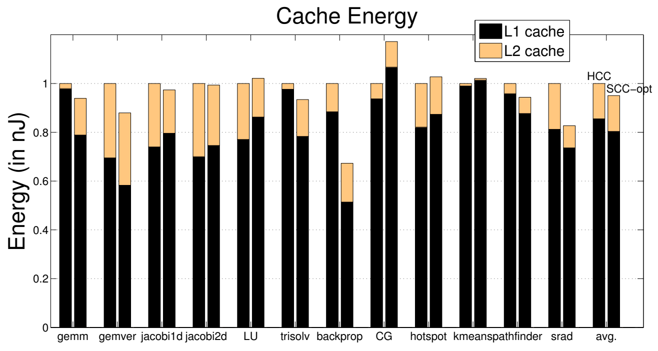

Figure 14: L1 and L2 Cache Energy (lower, the better). The first bar shows HCC energy and second bar SCC-opt energy

HCC. Conversely, the gemver is an example of hardware cache coherence working to HCC advantage, where the number of misses for HCC is lower compared to SCC-opt.

The running time (depicted in Fig. 12) shows a strong correlation between L1 cache read misses and performance. In HCC, the snooping overhead plays a significant role in determining execution time: In our implementation, we assign 1 cycle to a read-/write snooping request. In SCC, each coherence instruction incurs a two-cycle overhead. In addition to these overheads, there may be additional overheads depending upon the response to a snooping request in HCC (e.g., a read request may return an updated value from another processor) and the number of cache lines specified in the coherence instruction in SCC — each cache line incurs a 2-cycle delay. Because of removal of hardware cache coherence, we observe a 3% performance gain for SCC-opt over HCC on average.

Discussion: The performance results obtained for HCC and SCC schemes are sensitive to architectural choices made in the simulator implementation. And, we have opted for architectural choices that favor HCC even though on a real system they may be impractical or too costly. E.g., we have allotted 1-cycle delay for a snooping request and on a real system it might take multiple cycles. The implemented HCC protocol in the simulator concurrently sends a snoop request to other cores, and also a memory request to L2 cache. Alternately, the L1 cache can also be designed to send a memory request to L2 cache after a snoop miss, but this will increase the delay when there is a snoop miss.

6.4 Energy Results

Bus data transfers: Fig. 13 shows the traffic (number of words transferred) on the system bus for different schemes. All values are normalized with respect to HCC. The average number of words transferred per cycle (obtained by dividing total number of words with number of execution cycles) for HCC is also shown. For hardware coherence scheme, the traffic on the bus includes snoopy-bus coherence related exchange of messages, transfers between private L1 caches and shared L2 cache triggered by cache misses at L1 and replacement of cache lines at L1. For SCC, this includes data transfers between L1 caches and L2 cache prompted either by L1 misses and evictions, or invalidation and writeback coherence instructions. The HCC normalized data transfers on the bus for HCC-opt, SCC-basic, and SCC-opt are 0.99, 1.46, and 0.99 respectively, on average (geo-mean).

In backprop and srad, SCC-opt does a fewer write-backs to L2 cache compared to HCC; the L1 cache misses are lower for SCC-opt in the case of srad. Consequently, SCC-opt incurs a fewer data transfers in backprop and srad. Conservative writebacks in kmeans increases the traffic on the bus for SCC-opt compared to HCC.

L1 and L2 cache energy: The cache SRAM is a major consumer of energy in a processor. We compare cache energy consumption for HCC and SCC-opt schemes based on the number of accesses to tag SRAMs and data SRAMs. Using the SESC simulator, event counts for all relevant activities in the L1 and L2 caches are collected to account for all tag and data accesses to SRAMS. CACTI [28] is used to obtain the energy per access for tag and data for each cache level. The L1 cache employs dual ported SRAM to service snoop requests quickly. For SCC also we used the same dual ported SRAM for a fair comparison (per-access cost is a function of, inter alia, number of ports). The L1 cache accesses tag and data together for local processor requests while for snooping requests it accesses data SRAM only after tag hit. The L2 cache is configured to be in sequential access mode — it starts to access data SRAM after tag matching. We did not consider main memory energy because main memory accesses would be the same for both HCC and SCC schemes.

Fig. 14 plots relative energy consumption in caches for hardware and software cache coherence approaches: energy expenditure by HCC is considered 1 and energy dissipation by SCC-opt is scaled with respect to HCC. The break-down of energy expended in L1 and L2 caches is indicated. On average (arithmetic mean) SCC-opt energy consumption in caches is 5% less than that of HCC.

Most of the savings in SCC-opt come from two sources: elimination of snooping requests in L1 cache, and reduction in the number of *writeback words* by partial line transfers (only dirty words are written back to shared L2 cache in a software managed cache as opposed to entire cache lines which are the granularity of communication for HCC). We also observe that energy spent in all L1 caches together is around 86%, while the rest — 14% is expended in L2 cache.

7. Related Work

Some prior studies [7, 9, 10] have developed compiler analysis techniques to generate cache coherence instructions for software managed caches. The work in this paper distinguishes itself from prior efforts both by being more general as well as more precise, as we elaborate below.

Cheong et al. [7] use data flow analysis to classify every reference to shared memory either as a *memory-read* or a *cache-read*. A read reference is marked as a memory-read if the compiler determines that the cache resident copy of the data might have become stale, otherwise the reference is marked as cache-read. A limitation of that work is that the data flow analysis is carried out at the granularity of arrays, which will result in invalidations for an entire array even if two processors are accessing distinct parts of it.

Choi et al. [9] propose to improve inter-task locality in software managed caches by using additional hardware support: the current epoch number is maintained at runtime using an *epoch counter* and each cache word is associated with a *time-tag* which records the epoch number in which the cache copy is created. Then they develop the so-called epoch flow graph technique to establish conditions under which it can be guaranteed that the cached copy of a variable is not stale. The analysis here too treats an entire array as a single variable.

Darnell and his colleagues [10] perform array subscript analysis to gather more accurate data dependence information and then aggregate cache coherence operations on a number of array elements to form *vector* operations. The method however can handle only simple array subscripts: only loop iterators are allowed as subscripts.

O'Boyle et al. [22] develop techniques to identify data for Distributed Invalidation (DI) for *affine loops* that are run on distributed shared memory architectures. The DI scheme uses a directory to maintain coherence, and where possible it seeks to eliminate invalidation messages from the directory to remote copies and associated acknowledgments. Their analysis to minimize invalidation messages has similarities to our analysis for minimize invalidations. But the coherence equations in DI place some restrictions on the kinds of affine loops that can be analyzed: for example, conditional execution within the loop is disallowed, and increments of loop iterators must be unity. The approach presented in this paper efficiently handles arbitrary affine loops including those whose iterator values are lexicographically decreasing, that have a non-unit trip-count, or have a modulus operator etc., and conditionals are permitted. The DI work does not involve writebacks, which however are a part of our software cache coherence model, and we develop techniques to optimize writebacks as well. We also optimize irregular codes using an inspector-executor approach, while such codes are not optimized in the DI scheme.

Inspector-executor approaches have been used in the context of parallelization [11, 16, 26, 30] run-time reorderings [12, 15], but to our knowledge have not previously been developed for optimizing for cache coherence.

Kaxiras et al. [18] seek to improve scalability of directory coherence by creating *tear-off copies*: the read-only data are cached in the private cache of a core, but are not registered in the directory. And, at the first synchronization event, non-registered copies are self-invalidated without generating invalidation traffic.

Kontothanassis et al. [20] present a software cache coherence protocol with page granularity in large scale machines. Our approach differs from their work in that fine-grained sharing in on-chip multi-core processors is accurately handled with compile-time analyses. Ashby et al. [4] propose a software cache coherence scheme that uses a bloom filter to avoid unnecessary invalidations, but their work does not develop any compiler support. DeNovo [8] simplifies complicated hardware cache coherence protocols by enforcing a disciplined parallel programming model. The compiler support proposed in this work can complement the DeNovo project in automatically identifying self-invalidation regions.

8. Conclusion

The complexity of developing efficient hardware coherence protocols for emerging manycore heterogeneous systems makes software controlled coherence schemes attractive. However, a significant challenge for software controlled cache coherence is that of generation of efficient coherence instructions.

The automatic coherence management and optimization approaches developed in the paper advance compiler technology towards making software cache coherence a viable solution on shared-memory multiprocessor systems. Simulation results demonstrate the effectiveness of the compiler algorithms in achieving performance and cache-energy comparable to that of a hardware cache coherence scheme.

References

[1] AMD Accelerated Parallel Processing OpenCL Programming Guide, 2012.

[2] NVIDIA's Next Generation CUDA Compute Architecture: Kepler GK110, 2012.

[3] D. Abts, S. Scott, and D. J. Lilja. So many states, so little time: Verifying memory coherence in the cray x1. In *IPDPS*, 2003.

[4] T. J. Ashby, P. Díaz, and M. Cintra. Software-based cache coherence with hardware-assisted selective self-invalidations using bloom filters. *IEEE Transactions on Computers*, 60(4), 2011.

[5] N. Carter, A. Agrawal, S. Borkar, R. Cledat, H. David, D. Dunning, J. Fryman, I. Ganev, R. Golliver, R. Knauerhase, R. Lethin, B. Meister, A. Mishra, W. Pinfold, J. Teller, J. Torrellas, N. Vasilache, G. Venkatesh, and J. Xu. Runnemede: An architecture for ubiquitous high-performance computing. In *HPCA*, 2013.

[6] S. Che, M. Boyer, J. Meng, D. Tarjan, J. Sheaffer, S.-H. Lee, and K. Skadron. Rodinia: A benchmark suite for heterogeneous computing. In *IEEE International Symposium on Workload Characterization*, 2009.

[7] H. Cheong and A. V. Vaidenbaum. A cache coherence scheme with fast selective invalidation. ISCA, 1988.

[8] B. Choi, R. Komuravelli, H. Sung, R. Smolinski, N. Honarmand, S. V. Adve, V. S. Adve, N. P. Carter, and C.-T. Chou. Denovo: Rethinking the memory hierarchy for disciplined parallelism. PACT, 2011.

[9] L. Choi and P.-C. Yew. A compiler-directed cache coherence scheme with improved intertask locality. In *Supercomputing*, 1994.

[10] E. Darnell, J. M. Mellor-Crummey, and K. Kennedy. Automatic software cache coherence through vectorization. In *International Conference on Supercomputing*, 1992.

[11] R. Das, P. Havlak, J. Saltz, and K. Kennedy. Index array flattening through program transformation. In *SC*, 1995.

[12] C. Ding and K. Kennedy. Improving cache performance in dynamic applications through data and computation reorganization at run time. PLDI, 1999.

[13] P. Feautrier. Dataflow analysis of array and scalar references. *IJPP*, 20(1), 1991.

[14] P. Feautrier. Some efficient solutions to the affine scheduling problem: I. one-dimensional time. *IJPP*, 21(5):313–348, 1992.

[15] H. Han and C.-W. Tseng. Exploiting locality for irregular scientific codes. *IEEE Transactions on Parallel and Distributed Systems*, 2006.

[16] A. Hartono, M. M. Baskaran, J. Ramanujam, and P. Sadayappan. Dyntile: Parametric tiled loop generation for parallel execution on multicore processors. In *IPDPS*, 2010.

[17] J. Howard and et.al. A 48-core ia-32 message-passing processor with dvfs in 45nm cmos. In *Solid-State Circuits Conference*, 2010.

[18] S. Kaxiras and G. Keramidas. Sarc coherence: Scaling directory cache coherence in performance and power. *Micro, IEEE*, 30(5), 2010.

[19] J. H. Kelm, D. R. Johnson, W. Tuohy, S. S. Lumetta, and S. J. Patel. Cohesion: A hybrid memory model for accelerators. ISCA, 2010.

[20] L. Kontothanassis and M. Scott. Software cache coherence for large scale multiprocessors. In *HPCA*, 1995.

[21] T. G. Mattson and et.al. The 48-core scc processor: The programmer's view. SC, 2010.

[22] M. O'Boyle, R. Ford, and E. Stohr. Towards general and exact distributed invalidation. *Journal of Parallel and Distributed Computing*, 63(11), 2003.

[23] P. M. Ortego and P. Sack. Sesc: Superescalar simulator. In *Euro micro conference on real time systems*, 2004.

[24] PoCC: the Polyhedral Compiler Collection. `http://sourceforge.net/projects/pocc/`.

[25] PolyBench: The Polyhedral Benchmark suite. `http://sourceforge.net/projects/polybench/`.

[26] M. Ravishankar, J. Eisenlohr, L.-N. Pouchet, J. Ramanujam, A. Rountev, and P. Sadayappan. Code generation for parallel execution of a class of irregular loops on distributed memory systems. In *SC*, 2012.

[27] S. Verdoolaege. isl: An integer set library for the polyhedral model. *Mathematical Software–ICMS 2010*, pages 299–302, 2010.

[28] S. J. E. Wilton and N. P. Jouppi. Cacti: An enhanced cache access and cycle time model. *IEEE Journal of Solid-State Circuits*, 31, 1996.

[29] M. E. Wolf and M. S. Lam. A data locality optimizing algorithm. In *PLDI 1991*.

[30] X. Zhuang, A. E. Eichenberger, Y. Luo, and K. O'Brien. Exploiting parallelism with dependence-aware scheduling. In *PACT*, 2009.