Enabling Multi-threaded Applications on Hybrid Shared Memory Manycore Architectures

Tushar Rawat and Aviral Shrivastava
Computer Microarchitecture Lab
Arizona State University
Tempe, AZ, USA
Email: {tushar.rawat, aviral.shrivastava}@asu.edu

***Abstract*—As the number of cores per chip increases, maintaining cache coherence becomes prohibitive for both power and performance. Non Coherent Cache (NCC) architectures do away with hardware-based cache coherence, but become difficult to program. Some existing architectures provide a middle ground by providing some shared memory in the hardware. Specifically, the 48-core Intel Single-chip Cloud Computer (SCC) provides some off-chip (DRAM) shared memory *and* some on-chip (SRAM) shared memory. We call such architectures Hybrid Shared Memory, or HSM, manycore architectures. However, how to efficiently execute multi-threaded programs on HSM architectures is an open problem. To be able to execute a multi-threaded program correctly on HSM architectures, the compiler must: i) identify all the shared data and map it to the shared memory, and ii) map the frequently accessed shared data to the on-chip shared memory. In this paper, we present a source-to-source translator written using CETUS (Dave et al. [1]) that identifies a conservative superset of all the shared data in a multi-threaded application, and maps it to the off-chip shared memory such that it enables execution on HSM architectures. This improves the performance of our benchmarks by 32x. Following, we identify and map the frequently accessed shared data to the on-chip shared memory. This further improves the performance of our benchmarks by 8x on average.**

I. Introduction

As we transition from a few cores to many cores, scaling the memory architecture is one of the most difficult challenges. Current multicore architectures feature a fully coherent cache architecture. Coherence ensures that a write by any core is visible to all the cores. Each core may then read and obtain the updated values. This makes it easy to support the execution of applications written in the multi-threaded programming paradigm. However, implementing coherence often requires all-to-all communication between cores and the overhead of implementing cache coherency increases dramatically with the number of cores, [2, 3].

NCC (Non Cache Coherent) architectures attempt to circumvent this issue by skipping the implementation of cache coherence in hardware. Such architectures are power-efficient and scalable, but they are difficult to program [4]. NCC architectures are excellent for programs written in a Message Passing Interface paradigm where the communication between tasks is explicitly present in the application. However, programs written in the popular multi-threaded programming paradigm may not execute correctly on these architectures since the values written by one thread on a core may not be propagated to another thread on a different core.

A compromise between current multicore designs (in which all memory is shared memory, but suffers from poor scalability) and pure NCC architectures (in which there is no shared memory, but are scalable) is Hybrid Shared Memory (HSM) manycore architectures – in which there is some shared memory. In HSM architectures the private memory of the cores can be cached, but the shared memory cannot since the caches are non-coherent. Multi-threaded programs can be executed on HSM architectures by mapping the shared data to the shared memory. To enable higher performance HSM architectures may provide some limited on-chip shared memory to improve access to frequently accessed, or long-access latency, shared data. The 48-core SCC processor from Intel is a prime example. It features non-coherent caches. Pages in the off-chip memory can be configured as shared-among-all-cores or private-to-a-core through page tables. The data in the private pages are cache-able but the shared pages are not. To enable efficient execution the Intel SCC processor provides $384\ KB$ on-chip shared memory (only $8\ KB$ per core).

In the original form, multi-threaded applications can only be executed on a single core of the HSM processor. This will ensure correct execution, since the same core is writing to the memory – so it will be coherent by definition. However, clearly this approach is not scalable since we can only use one core on the HSM processor. The objective of this paper is to enable efficient and scalable execution of multi-threaded applications on HSM processors. To do that, we i) identify all the shared data in a multi-threaded application and map it to the off-chip shared memory. We do this through a series of analytic passes operating on the source-code, which creates an increasingly accurate picture of the shared nature of program variables. For example, initially we assume that all global variables are shared, but then in later stages through *points-to analysis*, we may be able to identify that some global variables cannot be accessed in more than one thread – and we can classify them as non-shared. Our approach works for well-constructed multi-threaded programs, free from improper thread accesses and race conditions. This approach is scalable, and will have better performance as all the threads can access their private data through caches – only shared data is non cache-able. Performance can be further improved by ii) mapping the more frequently accessed shared data to the on-chip shared memory to achieve efficient execution of multi-threaded applications on HSM architectures.

We have implemented our compiler analysis in the CETUS source-to-source compiler framework [1]. We evaluate the

978-3-9815370-4-8/DATE15/©2015 EDAA

effectiveness of our techniques by measuring the runtime of several parallel benchmark kernels on the Intel SCC processor. As compared to executing multithreaded applications on a single-core, identifying the shared data and mapping it to the off-chip shared memory increases the performance by about 32x. On top of this, by identifying and mapping frequently accessed data to on-chip shared memory, we can improve performance further by about 8x on average.

II. Related Work

The work presented in this paper has two aspects. The first aspect is to identify a conservative but tight superset of shared data. Many hardware-level techniques for identifying shared data have been developed with the intent of better utilizing or improving caches. Bellosa and Steckermeier [5] utilize hardware performance counters in order to detect data sharing between threads with a goal of co-locating data on the same processor. Liu and Berger [6], Paul et al. [7] focus on cache improvement as well – detecting and preventing false sharing in cache lines or reducing the traffic overhead incurred through cache coherence protocols. In addition there is work in this domain that attempts to detect shared data at runtime. For example, shared memory spaces are explored in von Praun and Gross [8] and Poziansky and Schuster [9], where thread access is controlled in order to efficiently allocate shared data. Savage et al. [10] need to determine data sharing in order to prevent race conditions; unsafe operations in a program are prevented by employing a consistent locking discipline in order to manage resource contention. The advantage of runtime-based analyses is evident in repeated-run profiling techniques such as Xu et al. [11] and Yang et al. [12], where the former implement a detector with atomic regions that identify data sharing when multiple threads interact with the regions, and the latter in which multiple runs of the program help detect shared data. We prefer a static analysis approach to avoid the execution overhead of runtime-based techniques. Kahlon et al. [13] use a static analysis technique in order to detect and prevent race conditions which result from improper access of shared variables. Gondi et al. [14] take a different path to preventing race conditions by minimizing the time shared data is kept in memory, purging it as soon as a last-use is detected. However, none of these works is directly applicable for our approach, since we need a compile-time approach to identify shared data in a multi-threaded application.

The second component of our work deals with data partitioning and memory management. The HSM manycore architecture has both on-chip and off-chip shared memory, and both Panda et al. [15] and Kandemir et al. [16] have addressed data partitioning between on and off-chip memory. However, neither consider parallel programs in their analysis. In particular, estimating the number of accesses to program variables is different in sequential and multi-threaded applications. Our work extends theirs by implementing a data partitioning scheme which considers parallel programs and approximates data read and write counts from all the threads. Cichowski et al. [17] use a manual process to port a single multi-threaded program to the SCC. To the best of our knowledge, our technique is novel in that it combines a static shared data analysis within the context of a multi-threaded program and uses it in order to automatically enable application execution on an HSM manycore architecture.

III. Our Approach

C POSIX threads (Pthreads) [18] programs present unique challenges for HSM manycore systems due to how global variables and shared data are managed within threads versus how they are handled across processes. In multi-threaded programs, a global variable is implicitly shared between any threads since the threads share the program text, data, and heap space of the parent process. In a multiprocess application, however, each thread from the original multi-threaded program is "mapped" to a full process – one per core. Variables which are global within a process are not implicitly shared with other processes. For proper execution these must be identified and converted to explicitly shared variables accessible through the HSM manycore software API. Functions and data managed by threads must also be transformed to process-based execution. This analysis builds up an increasingly accurate picture of the state of each variable (including pointers) as it appears in the program. The sample program provided in Listing 1 should be useed as a reference for this section.

Listing 1. Store thread ID sums and a locally defined shared variable

```
#include <stdio.h>
#include <pthread.h>
int global;
int *ptr;
int sum[3] = {0};
void *tf(void * tid) {
  int tLocal = (int)tid;
  sum[tLocal] += tLocal;
  sum[tLocal] += *ptr;
  pthread_exit(NULL);
}
int main() {
  int local = 0;
  int tmp = 1;
  ptr = &tmp;
  pthread_t threads[3];
  int rc;
  for(local = 0; local < 3; local++) {
    rc = pthread_create(&threads[local], NULL,
         tf, (void *)local);
  }
  for(local = 0; local < 3; local++) {
    pthread_join(threads[local], NULL);
    printf("Sum Array: %d\n",sum[local]);
  }
  return 0;
}
```

A. Variable Scope and Access Frequency Analysis (Stage 1)

This first stage takes as input the multi-threaded program source code and performs a rudimentary analysis of local and global variables. Details such as size of the variable, type, read and write counts, as shown in Table I are extracted. We implement a technique similar to that of Pabalkar et al. [19]. We assess variables based on their context – whether within a procedure, within a loop or nested loops, and whether called within a thread. Such a procedure provides approximate relative read and write counts for each variable. Each step in this, and following stages, represents an analytic *pass* through an abstract intermediate representation (IR) of the source code [1]. Passes are designed to look as narrowly or broadly within

TABLE I. INFORMATION EXTRACTED PER VARIABLE (POST STAGE 3)

Name	Type	Size	Rd	Wr	Use In	Def In
global	int	1	0	0	null	null
ptr	int*	1	1	1	tf	main
sum	int*	3	2	2	tf, main	tf
tLocal	int	1	3	1	tf	tf
tid	n/a	n/a	1	0	tf	null
local	int	1	8	4	main	main
tmp	int	1	1	1	main	main
threads	pthread_t*	3	2	0	main	main
rc	int	1	0	3	null	main

TABLE II. VARIABLES SHARING STATUS

	Shared Status After		
Variable	Stage 1	Stage 2	Stage 3
global	true	true	false
ptr	true	true	true
sum	true	true	true
tLocal	null	false	false
tid	null	false	false
local	null	false	false
tmp	null	false	true
threads	null	false	false
rc	null	false	false

the IR as necessary. For example, to constrain a pass to local variables only, it may restrict its search to only within procedures, as anything outside would constitute a global variable. When seeking only global variables, procedures within the IR are excluded. At this early stage, global variables are initially assumed to have a sharing status of *shared=true* while all other variables' sharing status is initialized to *null*. During a subsequent stage, the sharing status may be refined from *true* to *false* or *false* to *true* once, but will not revert or flip-flop. Prior to this stages exit, each variable is "seen" at least once, and identified as local or global. As only global variables have had a proper sharing status assigned, remaining variables retain the temporarily assigned sharing status of *null*, as shown the second column of Table II.

B. Inter-thread Analysis (Stage 2)

This stage identifies which variables exist within threads and also which are shared. In Algorithm 1, given a variable name and a list of procedures, the IR is traversed in a DFS manner to locate the variable and the procedure within which it appears. The IR is then searched for the thread which executes this procedure. Based on whether the thread is launched only once, or several times (for example, within a loop), a decision is made whether the variable is within a single thread or multiple threads, and this information is returned. Based on this result the sharing status of each variable (Table II) is updated. Referring back to Listing 1, even though both the variables *sum* and *tLocal* exist within the function *tf* which is launched by a thread, *tLocal* is defined in the scope of the function (not shared between threads), and has the sharing status set to *false*. Table I is updated to reflect the name of the function within which each variable was used and/or defined.

C. Alias and Pointer Analysis (Stage 3)

Because potentially shared variables may be hidden behind pointer relationships this stage performs a "Points-to" pointer analysis leveraged from the Cetus translation framework [1]. A brief description of the basic analysis: the goal is to identify the set of memory locations that a pointer variable may be pointing to. Interprocedural pointer information is analyzed via a dataflow methodology where pointer relationships are explicitly identified from pointer assignments including function calls. At each line of the program the analyzer produces a relationship map as output. This data is merged with the pointer information collected from analyzing previous statements, building a comprehensive overview of the pointer relationships within the program. These pointer relationships are classified as *definite* or *possibly*, with the latter often occurring after analyzing pointers within an if-else statement, where the pointer relationship branches with the control flow and is no longer definite with respect to which branch might be taken at runtime. As output, this analysis produces a map of relationship-pairs that specify a pointer and pointed-at symbol. We use this map within the passes in this stage.

Algorithm 1 Variable in Thread

```
Input: P, v, F /* Program IR, variable v, set of functions called
    by $pthread\_create$ */
Output: /* How many threads v is in */

for all Variable s $\in$ Program P do
  if v matches s then
    $proc \leftarrow$ name of procedure which contains v
    if $proc \in$ F then
      $caller \leftarrow pthread\_create$ launching $proc$
      if $caller$ appears within a loop then
        return "In Multiple Threads"
      else
        $seen \leftarrow$ number of times $proc$ appears in
        $pthread\_create$ calls
        if $seen > 1$ then
          return "In Multiple Threads"
        else
          return "In Single Thread"
        end if
      end if
    end if
  end if
end for
return "Not in Thread"
```

If a particular pointer is shared then the object it points to is also accessible in the context of this sharing. Algorithm 2 describes the high-level details of this process. It is possible that the pointed-to symbol is yet another pointer, or it may be a variable. The pointed-to object is retrieved and its sharing status is updated as a shared entity, such as that of the variable *tmp* given in the last column of Table II. The Points-to analysis offers a powerful capability to extract relationships that are not evident otherwise, and additionally our analysis can be less conservative, since the set of variables which are the same as a given variable is constrained. As Stage 3 ends, refer again to Table II. Notice that global variables which were defined but entirely unused, such as *global*, may be set as private.

D. Data Partitioning (Stage 4)

This stage uses information from previous stages in order to make decisions about where to place the data in the context of

Algorithm 2 Points-to Analysis (Shared Variables)

Input: P, V, R /* Program IR, Variable Status Map, Pointer relationships map */
Output: V /* Updated Variable Status Map */

```
for all Pointer symbol $s \in$ R do
    if A relationship exists with $s$ and the relationship is "definite" then
        $ptr \leftarrow$ Pointer symbol
        $ptt \leftarrow$ Pointed-to symbol
        shared $\leftarrow$ $ptr$ status from V
        if shared is True then
            shared $\leftarrow$ $ptt$ status from V
            shared $\leftarrow$ True
            update $ptt$ status in V
        end if
    end if
end for
```

Algorithm 3 Partitioning Shared Variables

Input: P, V /* Program IR, Set of Shared Variables+properties */
Output: M /* Transformed Program IR */

```
 1: for all Shared variable $s \in$ V do
 2:     $total\_size$ $+=$ $s$.mem_size
 3: end for
 4: if $total\_size \leq$ on-chip memory then
 5:     for all Shared variable $s \in$ V do
 6:         Create on-chip malloc call, $C$
 7:         Insert $put$ and $get$ calls in P to access on-chip memory
 8:         if Previous malloc call $B$ for $s$ exists in P then
 9:             Remove $B$
10:         end if
11:         Insert $C$ in main function of P
12:     end for
13: else
14:     Sort V by size, ascending
15:     $R \leftarrow$ size of remaining on-chip memory
16:     for all Shared variable $s \in$ V do
17:         if $s$.mem_size $\leq R$ then
18:             Create on-chip malloc call, $C$
19:             Insert $put$ and $get$ calls in P to access on-chip memory
20:             $R \leftarrow R - s$.mem_size
21:         else
22:             Create off-chip malloc call, $C$
23:         end if
24:         if Previous malloc call $B$ for $s$ exists in P then
25:             Remove $B$
26:         end if
27:         Insert $C$ in main function of P
28:     end for
29: end if
```

the HSM memory hierarchy. Just as with traditional caches, the size and frequency of access influences where data is stored and for how long. If all of the shared data fits within the on-chip shared memory, then it is collectively allocated into the faster SRAM even if some data is accessed much more frequently than others. A tradeoff is made if not all the data fits on the on-chip shared memory. In line 14, Algorithm 3, the variables are being sorted by size, as in Panda et al. [15]. A slightly modified algorithm also accommodates for sorting by frequency of use, as that metric is retained within the properties collected during the analysis of each variable. Shared scalars may be mapped to on-chip memory readily, with further granularity provided by frequency of access to those variables. Larger arrays may be allocated entirely in DRAM or split between DRAM and SRAM. The shared memory declaration is identical to a dynamically allocated variable in C, the difference is in the name of actual function call. The newly constructed declaration is inserted into the 'main' procedure in the target program, to effectively make the variable or pointer explicitly shared across the entire multiprocess application.

E. Translation Framework (Stage 5)

The final stage implements a source-to-source translator which uses the analysis from stages 1–4 to transform the IR and output C source code. The thread-to-process pass (Algorithm 4) attempts to find functions launched via the pthread_create call. This routine accepts four parameters: the thread ID, a thread attribute (or NULL), the function executed by the thread, and an argument (or NULL) being passed to the executing function (see Listing 1). Once a pthread_create function is found, the third and fourth arguments to pthread_create are extracted and saved. A new function call is generated using the function name derived from the third argument, and is given either the original argument specified as the fourth parameter in the pthread_create call, or, a core identifier if the argument passed to the function would be a thread ID and if the target architecture supports a core ID. After inserting the new function call above the pthread_create call in the IR, the pthread_create call is removed from the IR. Last, the function name and the order of appearance of the pthread_create call is noted for subsequent use and stored within a hash table. Consider: after the thread to process conversion, an application runs the same executable on multiple cores. In this case, if a particular thread runs on all cores then the information in the hash may be discarded. However, if a task is thread-specific and not delegated across all the other threads, it must be isolated such that it executes only on the given core(s). To isolate such a function within the hash, it is wrapped in an if-condition where the conditional checks if the program is running on the core with the proper core ID. The core ID is the value associated with the function name in the hash table. We ensure that thread IDs correspond 1:1 with core IDs.

IV. Experimental Results

A. Experimental Setup

We perform our experiments on the Intel Single-chip Cloud Computer (SCC) [20]. The 48-core non-coherent cache architecture features a unique on-die shared SRAM (384 KB) called the Message Passing Buffer (MPB). Through the message passing buffer, the cores communicate a limited amount of data directly and bypass both the L2 cache and DRAM (up to 64 GB). Each benchmark test is run on the SCC, each core running Linux, at 800 MHz core frequency, 1600

Algorithm 4 Threads to Processes

Input: P, T /* Multi-threaded Program IR and set of thread IDs (user supplied) */

Output: M /* Transformed Multi-process Program IR */

```
ProcList ← List of Procedures in P
for all functions ∈ P do
  UseCoreID ← False
  if function name is $pthread\_create$ then
    ProcName ← argument 3 from $pthread\_create$ call
    ProcArg ← argument 4 from $pthread\_create$ call
    if ProcArg ∈ T then
      UseCoreID ← True
    end if
  end if
  if ProcName ∈ ProcList then
    NewFunction ← ProcName /* Create new function
    from ProcName */
    if UseCoreID is True then
      Set NewFunction argument to ‘CoreID‘
    else
      Set NewFunction argument to value in ProcArg
    end if
  end if
  if $pthread\_create$ ∈ Loop then
    Insert NewFunction outside Loop
  else
    Insert NewFunction before $pthread\_create$ call
  end if
  Remove $pthread\_create$ call
  if Loop contains no $pthread\_create$ then
    Remove Loop
  end if
end for
return P as M
```

MHz network mesh, and 1066 MHz off-chip DDR3 memory frequency. RCCE library on the Intel SCC provides support to execute shared memory programs [21].

We run several multithreaded applications on the Intel SCC with and without our analysis and transformation. These applications include a program to *Count Primes*, to do a *Pi Approximation*, sum increasingly large multiples of 3 and 5 in *3-5-Sum*, *LU Decomposition*, *Dot Product*, and also a synthetic benchmark for memory operations, *Stream* from McCalpin [22]. All applications were compiled for SCC using the Intel C++ compiler (icc) version 8.1 (gcc 3.4.5), and RCCE 2.0.

Without our analysis and transformation multi-thread Pthread programs can only execute on a single core. To enable them to run on multiple cores we need to convert them to RCCE programs. We have implemented our analysis and transformation in the source-to-source CETUS compiler framework [1]. Each component or ‘pass’ of our framework is a subclass of either the *AnalysisPass* or *TransformPass* classes. These classes provide boilerplate code as well as perform some consistency checking to ensure that the program IR remains in a self-consistent state. The *Driver* class brings together all passes and executes them in series to analyze and make iterative changes to the IR. We use Java 1.6, ANTLR 2.7.5 and Cetus 1.3 running on Linux Mint 12.

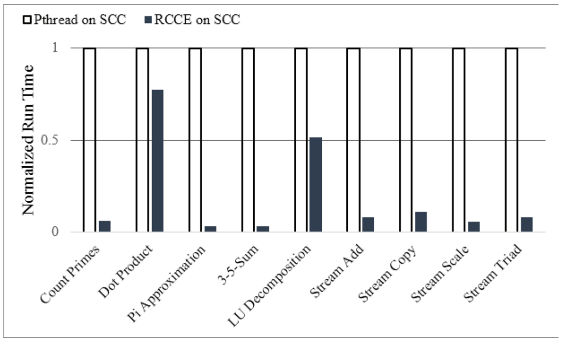

Fig. 1. Performance of RCCE applications utilizing off-chip shared memory and 32 cores normalized to the performance of the 32-thread Pthread programs running on a single core.

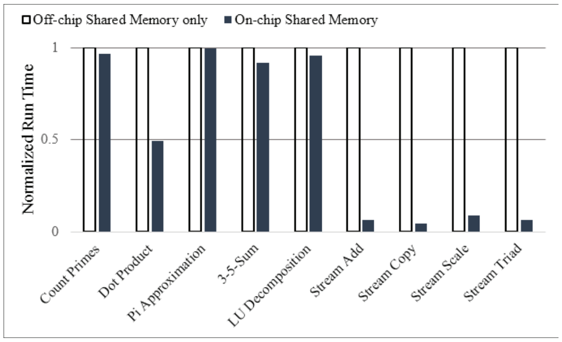

Fig. 2. Run time performance comparison of RCCE programs utilizing shared off-chip memory against the on-chip shared memory provided by the MPB.

B. Mapping shared data to off-chip shared memory improves performance by 32X

As an evaluation baseline we run each Pthread application on a single core of the SCC. We then generate a RCCE variant for each program which takes advantage of 32 cores of the SCC and utilizes off-chip shared memory, and measure the runtime. The Pthread benchmarks were built for 32 threads and the RCCE applications utilize 32 cores. Pi Approximation, 3-5-Sum, Count Primes and Stream achieve improvements of 32x, 29x, 16x and 17x, respectively. Fig. 1 shows the relative performance increase for each application (using only off-chip shared memory). The RCCE applications for Dot Product and LU Decomposition have large arrays in off-chip memory and have at least 8 cores in contention per memory controller. Although the performance benefits of 32 vs 1 core are hardly surprising, our work of converting multi-threaded programs to run as HSM applications makes this comparison possible.

C. Using on-chip shared memory further improves performance by 8X on average

Comparison of RCCE programs which only use off-chip memory vs those that utilize on-chip memory is given in Fig. 2. Programs which either exhibit a high degree of memory usage or those that balance memory use and core computation see the most performance improvement using the MPB. For example, Stream already benefits from the parallelism via 32

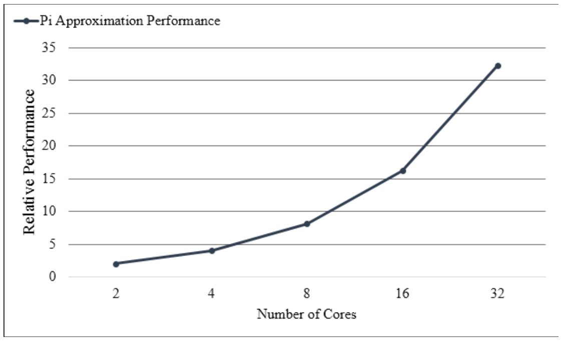

Fig. 3. Relative performance improvement over single-core Pthread application of multiprocessor RCCE program with varying core count on SCC.

cores, versus a single core with competing threads. In addition, when converted to utilize the MPB, not only are the memory accesses distributed across the cores, the locality for core-to-MPB is much closer than than of core-to-DRAM. Finally, MPB transfers may be done in bulk copy of memory (often contiguous addresses) further improving performance for an all-memory synthetic benchmark. LU Decomposition is an interesting case, as the matrix within that program does not fit into the on-chip shared memory. For a very slight performance improvement a small portion of the matrix (few rows) may be allocated separately on the MPB.

D. Enabling Scalable Applications on HSM Architecture

Converting multi-threaded programs to take advantage of multiple cores of the HSM architecture enables scalability. While this is application-dependent, programs with sufficiently large computations and which transfer data between cores using the on-die MPB can achieve significant performance increases with increasing core count. See Fig. 3.

V. Conclusion

We present a novel analysis and translation framework used to convert and enable applications to run on the Intel Single-chip Cloud Computer. Our approach automatically analyzes the multi-threaded source program and extracts the properties of all variables (shared and private) and efficiently maps the shared data to available on-chip and off-chip shared memory. Our technique is used to convert incompatible or inefficient programs by leveraging architecture-specific transformations, with experimental results demonstrating the suitability and performance benefits of enabling multi-threaded applications for efficient execution on HSM manycore architectures. The limitations of our work are: we limit source programs to those which do not use the non-portable (_np) Pthread extensions. Our analysis is also limited to the maximum number of cores as are on our experimental platform (48). However, the framework is not dependent on, or limited by, a given number of cores and is scalable to platforms with different core counts.

References

[1] Chirag Dave, Hansang Bae, Seung-Jai Min, et al. Cetus: A Source-to-source Compiler Infrastructure for Multicores. *Computer*, 42(12):36–42, 2009.

[2] Vishal Gupta, Hyesoon Kim, and Karsten Schwan. Evaluating Scalability of Multi-threaded Applications on a Many-core Platform. *Technical report GIT-CERS-12-03*, 2012.

[3] Qiang Yang, Jian Fu, Raphael Poss, and Chris Jesshope. On-chip Traffic Regulation to Reduce Coherence Protocol Cost on a Microthreaded Many-core Architecture with Distributed Caches. *ACM TECS*, 13(3s):103, 2014.

[4] Timothy G. Mattson, Rob Van der Wijngaart, and Michael Frumkin. Programming the Intel 80-core Network-on-chip Terascale Processor. In *Proc. SC08*. IEEE Press, 2008.

[5] Frank Bellosa and Martin Steckermeier. The Performance Implications of Locality Information Usage in Shared-memory Multiprocessors. *J. Parallel Distr. Com.*, 37(1):113–121, 1996.

[6] Tongping Liu and Emery D. Berger. SHERIFF: Precise Detection and Automatic Mitigation of False Sharing. *ACM SIGPLAN Notices*, 46(10):3–18, 2011.

[7] Johny Paul, Walter Stechele, Manfred Kroehnert, and Tamim Asfour. Improving Efficiency of Embedded Multi-core Platforms with Scratchpad Memories. In *Proc. ARCS*, pages 1–8, VDE, 2014.

[8] Christoph von Praun and Thomas R. Gross. Static Conflict Analysis for Multi-threaded Object-oriented Programs. *ACM SIGPLAN Notices*, 38(5):115–128, 2003.

[9] Eli Poziansky and Assaf Schuster. Efficient On-the-fly Data Race Detection in Multithreaded C++ Programs. *ACM SIGPLAN Notices*, 38(10), 2003.

[10] Stefan Savage, Michael Burrows, Greg Nelson, et al. Eraser: A Dynamic Data Race Detector for Multithreaded Programs. *ACM TOCS*, 15(4):391–411, 1997.

[11] Min Xu, Rastislav Bodík, and Mark D. Hill. A Serializability Violation Detector for Shared-memory Server Programs. *ACM SIGPLAN Notices*, 40(6), 2005.

[12] Yu Yang, Xiaofang Chen, and Ganesh Gopalakrishnan. Inspect: A Runtime Model Checker for Multithreaded C Programs. 2008.

[13] Vineet Kahlon, Yu Yang, Sriram Sankaranarayanan, and Aarti Gupta. Fast and Accurate Static Data-race Detection for Concurrent Programs. *Computer Aided Verification*, pages 226–239, 2007.

[14] Kalpana Gondi, A. P. Sistla, and V. N. Venkatakrishnan. Minimizing Lifetime of Sensitive Data in Concurrent Programs. In *Proc. CODASPY*. ACM, 2014.

[15] Preeti Ranjan Panda, Nikhil D. Dutt, and Alexandru Nicolau. On-chip vs. Off-chip Memory: The Data Partitioning Problem in Embedded Processor-based Systems. *ACM TODAES*, 5(3): 682–704, 2000.

[16] M. Kandemir, J. Ramanujam, Mary J. Irwin, et al. Dynamic Management of Scratch-pad Memory Space. In *Proc. DAC*, pages 690–695. ACM, 2001.

[17] Patrick Cichowski, Gabriele Iannetti, and Joerg Keller. Towards Converting POSIX Threads Programs for Intel SCC. *MARC Symposium at RWTH Aachen University*, 2012.

[18] Blaise Barney. POSIX Thread Programming, 2014. URL https://computing.llnl.gov/tutorials/pthreads/.

[19] Amit Pabalkar, Aviral Shrivastava, Arun Kannan, and Jongeun Lee. SDRM: Simultaneous Determination of Regions and Function-to-Region Mapping for Scratchpad Memories. In *Proc. HIPC*, pages 569–582. Springer Berlin Heidelberg, 2008.

[20] Jason Howard, Saurabh Dighe, SR Vangal, et al. A 48-core IA-32 Processor in 45 nm CMOS Using on-die Message-passing and DVFS for Performance and Power Scaling. *IEEE Solid State Circuits*, 46(1):173–183, January 2011. ISSN 0018-9200.

[21] Timothy G. Mattson, Rob F. Van der Wijngaart, et al. The 48-core SCC Processor: The Programmer's View. In *Proc. SC10*, pages 1–11. IEEE Computer Society, November 2010.

[22] John D. McCalpin. Memory bandwidth and machine balance in current high performance computers. *IEEE TCCA Newsletter*, pages 19–25, 1995.