Enabling Multithreaded Applications on Hybrid Shared Memory Many-core Architectures

by

Tushar Rawat

A Thesis Presented in Partial Fulfillment
of the Requirement for the Degree
Master of Science

Approved October 2014 by the
Graduate Supervisory Committee:

Aviral Shrivastava, Chair
Georgios Fainekos
Partha Dasgupta

ARIZONA STATE UNIVERSITY

December 2014

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 they 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. This work presents a source-to-source translator written using CETUS [7] 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.

To My Family,

for always expecting great things.

ACKNOWLEDGEMENTS

This work would not have been possible without the support and encouragement of numerous individuals. I am extremely grateful to my advisor, Dr. Aviral Shrivastava, for the introduction to power-efficient computing and for nurturing the best quality production from his students. His flexibility, help and guidance proved fundamental to achieving more than I thought possible.

I am especially grateful to Intel Corporation, not only for generous use of the MARC Datacenter and access to the Single-chip Cloud Computer, but also for the valuable time I spent working within the Storage Group and Platform Validation Engineering divisions which fostered immense professional and personal growth.

Thanks also go to my committee members, Dr. Partha Dasgupta for his instruction with Cryptography and Distributed Operating Systems, and Dr. Georgios Fainekos teaching Real-time Embedded Systems. Thanks to labmates within the Computer Microarchitecture Lab for their valuable discussions and assistence, especially Ke Bai, Jian Cai, Russel Dill, Shri Hari Rajendra Radhika, Bryce Holton, Reiley Jeyapaul, Di Lu, Jing Lu, Mahdi Hamzeh, Yooseong Kim, Jared Pager, Abhishek Rhisheekesan, and Dipal Saluja.

A very special thanks to the Mohanraj family, and friends. My time in Arizona simply would not have been the same without you all.

Last but not least, thank you to my friends Rudolph Troha, Albert Cho, Sean Leary, Arushi Kapoor, Kelly MacDonald (final proof lifesaver!), Vishwas and Vinay Seshachellam, Ying Wu, Victor Moussalem, Anantha Krishna Nelliparthi, Chase Parenteau, AJ Thau, James Ha, Dylan Knuth and members of the Davis/Meetup group. Your good humor and sage advice helped more than you know.

And without which none of this would have been possible – the unwavering dedication, love and support of my parents.

TABLE OF CONTENTS

APPENDIX Page

LIST OF TABLES

LIST OF FIGURES

LIST OF ALGORITHMS

Chapter 1

INTRODUCTION

Rapid improvement of microprocessor architectures has been foundational to the rise of advanced computer and electronic technology in development and use around the world today. The increase in integrated circuit density, following the trajectory of Moore‘s Law, along with improvements in wafer process technology, larger die package sizes with increased caches, and voltage scaling with leakage compensation to modulate chip frequency have all converged, ultimately resulting in the emergence of multicore and many-core computing systems. In turn, the performance envelope has been firmly pushed from hardware logic improvements into the realm of software development, specifically parallel programming [10]. And although concurrent systems are deployed in all manner of computing environments, from enterprise data centers to personal computing devices, parallel programming remains an often misunderstood, improperly implemented, and difficult paradigm to apply to the systems for which it is intended.

The plethora of parallel programming languages masks the lack of an easy-to-use framework for leveraging concurrent systems. Not only do many popular parallel programming languages already exist, but many parallel programming models exist as well. These models establish a concurrent systems paradigm by enabling a given programming language for an architecture or set of architectures. Building a model around the right language has a significant impact on its implementation and usage. For example, object-oriented languages have the advantage of being familiar to modern programmers, and as such, extensions and revisions to well-known languages such as C and Java are already utilized. The rise of General-Purpose Graphics Processing

Units (GPGPUs) saw the emergence of CUDA and OpenCL to take advantage of hundreds of stream-processing cores. With computer systems comprised of many nodes and distributed memory becoming widely available, Global Address Space (GAS) and Partitioned Global Address Space (PGAS) languages, particularly Unified Parallel C, Titanium, and High Performance Fortran have become popular in the scientific computing community [8] [17] [2]. In many ways, parallel programming has become a modern challenge similar to that faced by developers of early machine code. It's highly dependent on the underlying architecture and available runtime support. Porting by hand from one concurrent programming paradigm to another quickly becomes tedious for all but the simplest programs.

Crucially, managing shared memory is one of the most important challenges when scaling the number of cores. Shared memory is the part of the processor/application address space which is coherent and consistent as recognized among cores / tasks / threads. In other words, if one core / thread writes to a shared variable, and another core / thread reads it, then it will read the updated value. The presence of shared memory can simplify multithreaded programming, since the threads in a multithreaded program can communicate by just reading from and writing to shared variables. There is of course an implicit assumption of memory being shared among the cores. In general, the illusion of shared memory in todays multicore systems is instrumented by implementing hardware-based coherence protocols within the memory hierarchy. Coherence protocols essentially make the writes by any core visible to all the cores. Each core may then read and obtain the updated values. Since implementing coherence requires all-to-all communication between cores, the overhead of implementing coherency increases dramatically with the number of cores [12]. One way to scale the number of cores is to utilize Non-Coherent Cache (NCC) architectures that skip implementing cache coherence in hardware. While NCC architectures are power-efficient and scalable, it is difficult to program them [20]. Such architectures are excellent for programs written in a Message Passing Interface (MPI) 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. Because shared memory is not provided by the hardware in the NCC architecture, communication methods must be explicitly present in the software for the programs to work correctly. One option to make multithreaded programs execute correctly on NCC architecture is to identify the communication between the threads and convert them to explicit communication commands. Even when possible, this approach may suffer from significant performance overhead since now all communication has to be performed in the software through instructions.

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) many-core 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 as 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 preformance, 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 (8 KB per core).

This research is composed of two main parts. First, identification of a tight superset of shared data. Existing hardware techniques focus on cache improvements and detecting sharing between threads in order to reduce the overhead of implementing cache coherence protocols. Some work implements shared data detection at runtime through profiling techniques. While effective, these require multiple runs of the application which uses valuable compute time and resources. Static analysis techniques to detect and prevent race conditions resulting from improper access of shared variables, and limiting the lifetime of shared data in memory, are both used to detect and minimize the impact of improper use of shared variables. However, none of these techniques are directly applicable as we require a compile-time approach to identify shared data in a multi-threaded application. Second, we provide shared data partitioning and memory management. Many previous techniques partition data between on- and off-chip shared memory yet fail to take into account parallel systems. The work presented here estimates the number of accesses to program variables in both serial and multi-threaded applications. This work extends prior solutions by implementing a partitioning scheme which takes into account parallelism for both multicore and many-core environments.

To enable the execution of multithreaded programs on HSM many-core architectures, i) all the shared data must be mapped to the shared memory, and ii) the more frequently accessed shared data must be mapped to the smaller but faster on-chip shared memory. The approach to solve this problem consists of an analysis phase followed by a source-to-source transformation. The technique is divided into 5 major stages, as shown in Figure 1.1. In the first three stages, the multithreaded program source code is analyzed and each stage progressively identifies more information about the variables, including pointers, within the program. In Stage 1 basic details, including the name, size, type, read count, write count, and scope of the variables are determined. Stage 2 establishes the *sharing status* for each variable. For example, initially, if a variable is a global variable, it is classified as shared. An inter-thread analysis also determines whether the sharing status of a variable is shared or private. Some variables exist simply as an alias pointing to another variable, therefore a “Points-to” pointer analysis in Stage 3 identifies such variables and updates the sharing status for identified shared variables.

Following the analysis technique the multithreaded programs, in the final two stages, are converted to HSM applications executable on the SCC. This process is implemented in a source-to-source translator. In Stage 4, the results of the analytic technique are used to convert the implicitly shared variables to explicitly shared variables, as expected by the many-core architecture. In Stage 5, the program is converted from using threads to using multiple processes. The experimental test platform is the 48-core Intel SCC. The runtime is measured for each program configured for 32 threads (original) or 32 cores (converted), compiled to run on this target architecture. As unconverted programs cannot fully take advantage of the HSM memory hierarchy and are limited to single-core execution, the converted programs using only off-chip shared memory show significant performance improvement – up to 32x. As the HSM programs are further optimized to take advantage of the on-chip memory, they show up to 8x improvement on average.

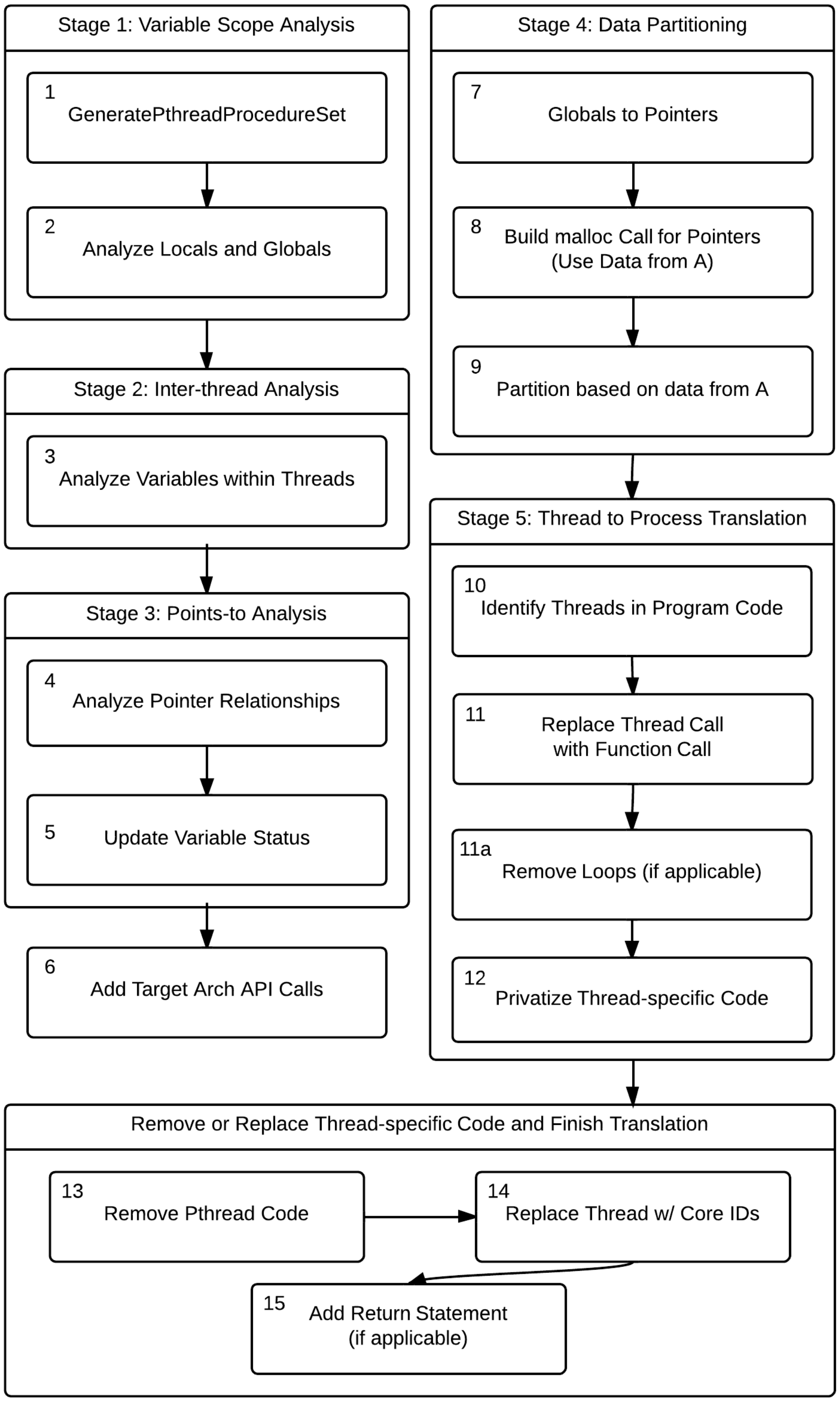
Stage 1: Variable Scope Analysis
1 GeneratePthreadProcedureSet
2 Analyze Locals and Globals
Stage 2: Inter-thread Analysis
3 Analyze Variables within Threads
Stage 3: Points-to Analysis
4 Analyze Pointer Relationships
5 Update Variable Status
6 Add Target Arch API Calls
Stage 4: Data Partitioning
7 Globals to Pointers
8 Build malloc Call for Pointers (Use Data from A)
9 Partition based on data from A
Stage 5: Thread to Process Translation
10 Identify Threads in Program Code
11 Replace Thread Call with Function Call
11a Remove Loops (if applicable)
12 Privatize Thread-specific Code
Remove or Replace Thread-specific Code and Finish Translation
13 Remove Pthread Code
14 Replace Thread w/ Core IDs
15 Add Return Statement (if applicable)

Figure 1.1: Framework overview of all major and minor stages.

Chapter 2

RELATED WORK

Multicore and many-core systems embody the many advancements developed since the birth of the planar transistor. The impetus for these developments has been a desire for improved performance and functionality as well as reduction in material cost. These include the introduction and subsequent refinement of caches, the implementation of threading (e)specially for user-level applications), optimizations for shared data in both local and distributed systems, and finally the advent of the many-core systems in response to physical design constraints.

Caches came about in an effort to bridge the performance gap between CPUs and the storage medium, namely main memory and physical media. Because CPUs operate orders of magnitude faster than the storage and memory devices, there is an element of idleness whereby the CPU is waiting for operations sent to memory or disk to complete. In such a case the CPU may be blocked from performing further operations. Since the same data is often repeatedly accessed, making that data available faster directly influences the performance of the overall system. Thus, a cache provides an intermediate storage location physically close to the CPU where data elements from memory and disk may be kept temporarily for easy reuse and fast access. Improvements to caching protocol may often improve overall performance without changes to the other components within the system. Rawat [25] predicts cache misses for scalar data towards improving performance of real-time programs while Weissman [31] uses hardware performance counters to detect cache misses and reduce conflict and capacity misses.

Pthread was originally developed to provide a standardized threading interface across POSIX-compliant systems [1]. Even though operating systems natively supported kernel-level threads for managing processes and system resources, user-level (application) threads were difficult to port between various computing systems due to their proprietary nature. For example, an application built using the Microsoft Windows threading API will run only on Windows. In contrast a program built using the standardized version of Pthreads (without the use of the non-portable functions) will run on any POSIX-compliant operating system, from basic UNIX variants, to Linux, even to Apple's Mac OS X. In basic terms, Pthreads enables a multithreaded program to be written once and simply recompiled for the target platform. This enables many systems to benefit from the same parallel program source code.

Transitioning to multicore systems also enables Pthread programs to more fully utilize the processor, as each thread may be individually assigned to different cores. A performance benefit can be realized, but only if the threads efficiently manage the shared data between the application. The shared memory is implicitly shared amongst all the threads, making the architecture easier to program. However, since scalability becomes a larger issue as core count increases, a new class of architecture called "many-core" offers improved performance yet comes at a cost whereby shared data must be managed explicitly.

The work presented here has two aspects. The first aspect is to identify a conservative yet 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 [3] 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 [18], Roy and Jones [26] and Paul et al. [22] 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 [30] and Pozniansky and Schuster [23], where thread access is controlled in order to efficiently allocate shared data. Savage et al. [27] 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. [32] and Yang et al. [33], 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. Due to the overhead of increased time spent during analysis and execution of runtime-based schemes, a static analysis approach is preferred. Kahlon et al. [15] use a static analysis technique in order to detect and prevent race conditions which result from improper access of shared variables. Gondi et al. [11] 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 this approach, since a compile-time approach is needed to identify shared data in a multi-threaded application.

The second component of this 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. [21] 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. This work extends theirs by implementing a data partitioning scheme which considers parallel programs and approximates data read and write counts from all the threads. This 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 enable application execution on an HSM manycore architecture.

Chapter 3

MOTIVATION AND BACKGROUND

The advent of multicore systems has put renewed emphasis on the development and optimization of multithreaded programs. Prior to multicore computing, commercial CPUs had a single core which ran threads sequentially one after another. With the introduction of Simultaneous Multithreading (SMT), superscalar CPUs began to exhibit support for multiple hardware-level threads, effectively enabling the operating system to schedule multiple threads for execution at the same time. How SMT is implemented is largely architecture-dependent. In 2002, Intel Corporation introduced Hyper-threading technology (HTT) for the Xeon and Pentium line of x86-Architecture CPUs. HTT implements SMT by making each physical core appear as two logical cores. Thus, an operating system and application supporting HTT can increase the amount of work processed by distributing it to more than one core and better utilizing the existing core architectural execution units which previously waited idle for instructions and data. Although HTT was meant to increase the performance of applications taking advantage of the technology on HTT-enabled processors, Hyperthreading and other technology that increases chip utilization has some interesting consequences. On a single-core system which utilizes HTT, applications more efficiently utilizing the processor resources meant that the energy consumption of the overall package increased. Prior to the emergence of HTT many of the processor resources were waiting idly while only a single thread was executing. This scenario could be quite common: suppose that a particular workload does only integer operations and completely ignores the floating point units. Thus two threads, one focused on integer operations and the other on floating point operations, could better utilize these resources by being scheduled simultaneously. However, going from idle to active increases the current these units draw, leading to increased heat dissipation from the chip. Some Intel chips that implemented Hyperthreading ran too hot and too fast, subsequently overheating.

In addition to increasing the capabilities of the processing core through additional hardware-level threads, semiconductor chip manufacturers also moved to multicore processors where each processor had two or more cores on the same package die. While the move to multicore systems benefits multithreaded applications because each thread can be assigned to an individual core, the transition to multicore systems is promoted for several reasons.

The primary reason is that increasing the core clock frequency no longer delivered guaranteed performance gains for applications as in the past. Chip designers were forced to abandon ever-higher clock frequencies due to increased voltage required to drive the clocks faster, which in turn also increased the current leakage on-chip, and both of which contribute to higher heat dissipation [24]. From a system integration perspective this would immediately require better cooling solutions which aren't readily available. Thus if frequency is increased ad infinitum under the status quo the chip will quickly experience catastrophic failure – obviously an unacceptable outcome. This phenomenon has come to be known as the power wall and is a fascinating study of the thermal, power and performance tradeoffs in modern microprocessor development. Note that the "free performance lunch" that enabled software developers to make minute or no changes to their software and automatically enjoy the benefits from the next generation of processors has ended [28]. Since the emphasis is now on multiple cores, hardware designers and software developers must optimize applications for these new architectures.

Task 1
Task 2
Time
Task 1
Task 2
Time

Figure 3.1: Tasks in parallel may finish earlier than tasks ordered sequentially.

This is where the value of multithreaded programming via Pthreads becomes readily apparent. Although it is possible to run POSIX thread applications on a single core processor, the threads are each limited to a given time quanta cut out of the overall processor time. Thus, on a single-core processor, no two threads run simultaneously, making the use of parallelism impossible even if the application is written with a task-concurrency design. Increasing software performance through parallel programming for better utilization of multicore systems has become a primary goal of computer scientists and industry professionals. Pthreads is widely available with stable libraries on Linux, Unix, and Mac OS X based operating systems. Even Windows, which does not natively support Pthreads, has a third-party Pthread-like API called Pthreads-Win32 built on top of the Windows proprietary threads [9].

Multicore processors provide the capability for a given process or thread to be assigned to a core other than that which is running the operating system. Each core duplicates certain essential hardware components, such as having a private L1 and L2 cache. Other components are shared by the cores, such as an optional L3 cache. Pthread programs running on a multicore system distribute the threads amongst participating cores. These threads have access to and share the process address space. When a process is assigned to a particular core, it retains its own stack, stack pointer, heap and program data. Additionally, it utilizes the cores L1 and L2 caches, if available, as well as the shared L3 cache and main memory. Remarkably, it is the convenience of sharing data that also introduces inefficiencies into multicore programming.

In the single-core processor, a program may request data from the cache hierarchy. If the data is in the cache it is provided to the processor, however, if the data is not in the cache then the request is made to main memory or disk to retrieve the appropriate data. Moving to multicore processors, main memory is shared but the caches and execution hardware are private to the core they are on. Thus if two or more threads in the same program are working on the same data, initially that data is copied to each cache for each participating core. However, a situation arises whereby a thread may read and modify data in its cache while the same piece of data in the remote cache in another processing core could potentially have a more recent copy of the data. Thus characterizes the issue of cache coherency. In multicore systems it is normal that multiple copies of a shared-memory object reside in several different caches at the same time. Unless each copy is updated to reflect a change in one cache, it may be that one or more copies become outdated in their respective locations. If this happens, and if that piece of data is read or transformed by the processor core, invalid operations or results will likely occur. To prevent this and ensure all changes to the data are properly communicated, a cache coherence protocol adhering to a coherency model is required.

Cache coherent hardware, on the other hand, introduces performance penalties with increasing core count, as more updates need propagation to more cores. Also, each additional cache also draws power, quickly making large multicore systems extremely costly in both power and performance. One solution is to write a parallel program for multicore systems that is easily ported to many-core systems. There are several challenges involved. For example, in multicore systems, threads are assigned and managed by the operating system; however, in many-core systems – where a global application runs across several cores – each core may in fact be running its own OS. Fundamentally, in order to support the distributed memory environment found on many-core systems, a thread-to-process mapping must be established for a proper program conversion.

Pthread is uniquely suited for such a mapping. In many-core systems there are a large number of processing elements (cores) that accept processes as the execution vector of choice. Programs written for many-core systems may utilize a master-worker or multi-worker paradigm in which the tasks are distributed to the processes and via some communication mechanism data is shared between the processes. Threads are very similar. The heart of Pthread computation lies in the creation of the thread entity using the *pthread_create* library routine. This function takes as parameters the thread ID in the form of *pthread_t* data type, thread attributes, function pointer to the routine the thread should execute, and any function arguments to be passed to that thread.

```
pthread_create(&thread, NULL, function, (void *)argument)
```

Note that the Pthread application calls *pthread_create* from within the parent process and that the threads are assigned to cores based on core availability and preference of the operating system scheduler. Programs which expect a predefined or fixed order of execution for the threads and attempt to assign threads to cores in such a manner are not valid POSIX-compliant programs and are not considered in this work. In a Pthread program, *pthread_create* calls are encountered sequentially, one by one, but their actual execution order is dependent on the operating system scheduler. Converting the threads into processes that interact with each other requires modifications to the programming model based on the nature of the parallelism within the Pthreads application. Consider the two following parallelism scenarios: in the first case each thread handles a standalone task. The tasks may communication with each other on different threads and facilitate the exchange of data. Each task is largely independent and performs work which does not have dependencies on all the threads finishing their execution. Such a program may have two or more *pthread_create* calls which initiate different functions and sit outside of a for-, while-, or do-while-loop. In the second case, each thread operates on different parts of the same larger problem, dividing the work amongst the threads. Because each thread is given a portion of the overall work, the entirety of the program cannot finish until all the threads have finished their portion of the computation and returned to the managing thread. In this situation, a *pthread_create* call might be inside of a loop structure where the work is divided among the threads according to thread ID. Alternatively, threads may divide the work outside of a loop structure in a similar manner. This second case has each thread utilizing essentially the same code but accessing different offsets (usually thread or core ID) to index the computation. Essentially, each thread does the same type of computation as part of a divide-and-conquer strategy.

The following sections describe the novel analytics and translation framework developed to assess, extract, convert and transform variables, memory and program data from one parallel processing paradigm to another.

Chapter 4

TRANSLATION FRAMEWORK

The objective of this work is to enable the efficient execution of multithreaded programs on a Hybrid Shared Memory (HSM) many-core architecture – one that expects a multiprocess/many-core program. This methodology starts with considering multithreaded applications [1] using the C POSIX threads (Pthreads) library. In such a program, a global variable is implicitly shared between any threads within that process since they share the program text, data, and heap space of the parent process. However, in a multiprocess application – where each thread from the multithreaded process can be mapped to a full process – the global variables cannot be implicitly shared; instead, they must be identified and converted to explicitly shared variables accessible through the HSM many-core software API. Determining which variables are shared and which are private must be done extremely carefully. Marking a variable as private when it should have been shared can and will result in erroneous behavior ranging from incorrect results to program crash, or even the programming falling into an infinite loop. Variables shared across threads must be properly identified, and variables that are referenced in a shared context must also be processed. Superficially, both threads and processes may run on the same multicore architecture, as processes are composed of one or more threads. However, the architecture may lend itself to particular softwar that better utilizes threads over processes, or vice-versa, and is highly dependent on how memory is shared (see Figure 4.1 for similarities and differences). Threads active within one process do not have access to global variables within adjacent processes unless the data sharing is explicitly managed. Due to this difference, memory management and conversion of global variables in Pthreads

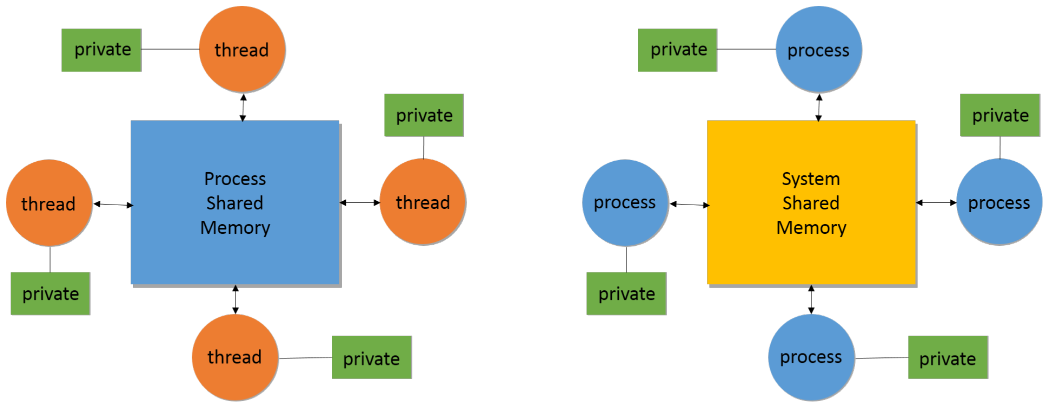

Figure 4.1: A shared memory space within a process (left) provides easy and immediate access to global variables. In contrast, processes with access to shared memory must explicitly define global variables in such a space for use as a shared resource.

programs is a key aspect of the transformation process. This process is composed of several stages as shown in Figure 1.1. In each stage the multithreaded program source code, which is parsed into a Cetus intermediate representation (IR) tree [7], is analyzed. The analysis builds up an increasingly accurate picture of the state of each variable as it appears in the program – including pointers. The sample program given in Example Code 4.1 will serve as a guide for this process. The program source code is passed into a lexical analyzer which assists in building a tree-based structure for identifying components of the C language. Once the syntax-tree exists, the parser does pass-based analysis whereby each pass establishes a search pattern based on transformation rules intended to convert Pthread program features first into an IR and subsequently into the source code deployable to the target architecture (Example Code 4.2).

4.1 Stage 1: Variable Scope Analysis

The first stage performs a rudimentary analysis of local and global variables, extracting basic properties such as size, type, and read/write counts, as shown in Table 4.1. Each step in this stage represents an analytic pass used to extract information from the source code. If the pass looks only within procedures in the program IR then the analysis is constrained to local variables only. The opposite is done when traversing the program for all globals – procedures within the IR are excluded.

Table 4.1: 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

Example Code 4.1: Store thread ID sums and a locally defined shared variable

```c
#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;
}
```

Threads implemented in the program are not analyzed at this stage and only variables identified as global are assumed to have a sharing status of *shared* = *true*. During a subsequent stage the sharing status may be refined from *true* to *false* or *false* to *true* once, but it will not revert. Changes from *null* are always accepted. At the end of this stage each variable has been seen at least once; therefore, sets which comprise *all variables*, *local variables*, and *global variables* are created and populated. Sets which contain variables that are defined or used within the procedure and thread which executes once it is launched are built during the next stage. Because only the global variables have had a valid sharing status assigned, the remaining variables are temporarily assigned a sharing status of *null* as shown in the second column of Table 4.2.

Table 4.2: 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

4.2 Stage 2: Inter-thread Analysis

This stage identifies the variables existing within threads and determines which ones are shared. Recall that in Pthread applications the threads are launched from the parent process. In the target architecture applications are instead run via a launcher which sends the executable to each core involved in the computation. To clarify, in Pthreads programs, the threads are launched from a single point of control: the parent process. These threads may all access the shared memory of that parent process and execute the function that is passed to them through the *pthread_create* library routine. Now consider the translated application. There is no *pthread_create* call in the target code. Instead, each core is running one process which replaces one thread from the Pthreads application. Given a variable name and a list of procedures which are executed by threads, the IR is traversed in a depth-first-search 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 qualification is made whether the variable is seen within a single thread or multiple threads, and this information is returned as a single result. With these results the variable might be placed into either the single thread execution set, the multiple thread execution set, or neither if it is not found within any thread execution environment at all. Algorithm 1 details this operation. In the third column of Table 4.2 the sharing status of each variable is updated with a boolean value. Note that 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 4.1 is also updated to reflect the function within which each variable was used and defined.

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 */

```
1: for all Variable s $\in$ Program P do
2:   if v matches s then
3:     $proc \leftarrow$ name of procedure which contains v
4:     if $proc \in$ F then
5:       $caller \leftarrow$ pthread_create launching $proc$
6:       if $caller$ appears within a loop then
7:         return "In Multiple Threads"
8:       else
9:         $seen \leftarrow$ number of times $proc$ appears in pthread_create calls
10:        if $seen > 1$ then
11:          return "In Multiple Threads"
12:        else
13:          return "In Single Thread"
14:        end if
15:      end if
16:    end if
17:  end if
18: end for
19: return "Not in Thread"
```

4.3 Stage 3: Alias and Pointer Analysis

Potentially shared variables may be hidden behind pointer relationships. This stage leverages the built-in Points-to analysis that the Cetus translation framework provides [7]. Here's a description about how this analysis works – the goal is to identify the set of memory locations that a pointer variable may point to: A dataflow methodology is used to analyze interprocedural pointer information. Pointer relationships are explicitly identified from pointer assignments, including through function calls. Once a fixed point is reached, the analyzer produces a relationship map as output. This data is updated at each statement in the program and is merged with the existing pointer information that was collected before it. Pointer relationships are classified as *definite* or *possibly*, the latter often occurring after analyzing pointers within an if-else statement. Such control-flow information is contained within the traversal of Cetus-generated control-flow graphs.

Through this analysis a map of relationships from pointer to pointed-at symbol is constructed. Based on this information, 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 this is another pointer or it may be a variable. In any case, the pointed-to object is retrieved and its sharing status is updated as a shared entity – such as that of the variable *tmp* in the last column of Table 4.2. The Points-to analysis offers a powerful capability to extract relationships that may not be evident otherwise. Additionally the analysis can be less conservative since the set of variables which are the same as a given variable may be constrained.

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 */

```
1:  for all Pointer symbol $s \in$ R do
2:    if A relationship exists with $s$ and the relationship is “definite” then
3:      $ptr \leftarrow$ Pointer symbol
4:      $ptt \leftarrow$ Pointed-to symbol
5:      shared $\leftarrow$ $ptr$ status from V
6:      if shared is True then
7:        shared $\leftarrow$ $ptt$ status from V
8:        shared $\leftarrow$ True
9:        update $ptt$ status in V
10:     end if
11:   end if
12: end for
```

As Stage 3 ends, refer again to Table 4.2. Notice that global variables which were defined but entirely unused (such as *global*) may be set as private and potentially removed from the source altogether. After this post-processing, the analytics are used to influence the data partitioning and translation framework that follows.

4.4 Stage 4: Data Partitioning

Without the analytics from the previous stages this stage would lack crucial information. For example, the size and type of variables are identified in order to gain insight into how much space they may take up in memory. Additionally, with the read/write counts for each piece of data, variables which are accessed frequently may be mapped to the on-chip memory whereas data that is not in high demand might go to the off-chip memory – if space is a constraint. The partitioning scheme starts off simple but is also very flexible. In the best-case scenario all the shared data identified during stages 1–3 will fit into the on-die shared memory and provide the best possible performance. However, if the total shared data exceeds the capacity of the shared SRAM available, a few opportunities for improved performance exist. First, small shared scalars may be mapped to on-chip memory readily, with further granularity provided by frequency of access to those variables. Second, larger arrays may be allocated entirely in DRAM or split between DRAM and SRAM. High level details are given in Algorithm 3 which allocates memory by first considering if it can fit all of it onto the on-chip shared memory, and if not, partitioning based on size of variables and the space remaining on the on-chip memory. Note that *mem_size* is a combination of the *Size* and *Type* properties as shown in Table 4.1. These are architecture-dependent. The shared memory declaration is identical to a dynamically allocated variable in C, with the difference being the name of actual function call. This function call is dependent on how the software API enables access to various parts of the memory hierarchy on the HSM architecture. 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.

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
```

4.5 Stage 5: Translation Framework

In this final stage a source-to-source translator is implemented that takes as input a well-defined Pthread program and generates a transformed intermediate representation which is output as C source code. Several passes are executed during the transformation. Each pass consists of an algorithm that analyzes and shapes the IR into the final representation. The main passes in the framework focus on handling the conversion from threads to processes as well processing the conversion of implicit shared variables to explicit shared variables based on the information collected in stages 1–4.

The thread-to-process pass (Algorithm 4) consists of the traversal of the program IR which looks primarily at function calls, attempting to find those that match the *pthread_create* call and extracting the relevant information from them. The *pthread_create* routine accepts four parameters: the thread ID, a thread attribute (which also allows NULL values), the function executed by the thread, and an argument (or NULL) which is passed to the executing function (See Example Code 4.1). Once a function is matched as *pthread_create* 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 following two conditions apply. These conditions are that the argument passed to the function would be a thread ID and the target architecture supports such an identifier in the translated program. If these are satisfied, a core ID may be inserted in place of the argument. 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 are noted for subsequent use

Algorithm 4 Threads to Processes

Input: P, T /* Multithreaded Program IR and set of thread IDs (user supplied) */
Output: M /* Transformed Multiprocess Program IR */

```
 1: ProcList ← List of Procedures in P
 2: for all functions ∈ P do
 3:    UseCoreID ← False
 4:    if function name is pthread_create then
 5:       ProcName ← argument 3 from pthread_create call
 6:       ProcArg ← argument 4 from pthread_create call
 7:       if ProcArg ∈ T then
 8:          UseCoreID ← True
 9:       end if
10:    end if
11:    if ProcName ∈ ProcList then
12:       NewFunction ← ProcName /* Create new function from ProcName */
13:       if UseCoreID is True then
14:          Set NewFunction argument to ‘CoreID‘
15:       else
16:          Set NewFunction argument to value in ProcArg
17:       end if
18:    end if
19:    if pthread_create ∈ Loop then
20:       Insert NewFunction outside Loop
21:    else
22:       Insert NewFunction before pthread_create call
23:    end if
24:    Remove pthread_create call
25:    if Loop contains no pthread_create then
26:       Remove Loop
27:    end if
28: end for
29: return P as M
```

and stored within a hash table. Consider the scenario for the usage of this information. After the thread to process conversion, an application may run the same executable on multiple cores. If this is the case (and 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). This isolation is straightforward. Each object, if assessed and found to be a function call, is checked for existence within the hash table populated previously. This function is wrapped in an if-condition where the conditional statement checks if the program is running on the proper core with the associated core ID. The core ID is the value associated with the function name in the hash table. This process adheres to the POSIX specification whereby threads are not specifically tied to any particular core. It is irrelevant whether a given task executes on Core 2 vs Core 3. All that matters is that the thread ID given for the function originally launched through *pthread_create* corresponds one-to-one to the same core ID in the target program that is used for wrapping the function found within the hash table.

Working with multithreaded programs necessitates handing synchronization issues, and although synchronization is not a main facet of the parser, some basic conversion of the mutual exclusion (mutex) variables and functions is implemented. A mutex variable is essentially a lock enabling or preventing access to some shared resource. On its own a mutex does nothing, but if threads requiring access to a common resource all participate in utilizing a lock, it ensures that only one thread can access the resource at a given time. This helps prevent race conditions. Recall that in traditional Pthread programming all threads and variables are contained within a process. Therefore it is simple for any thread to access the mutex variable since it is a global variable. In the target applications two concerns arise. The first is that any sort of mutex or synchronization primitive must be accessible from any process in order to be effective as a critical section control agent. Because of this, a typical Pthread mutex cannot be used. The second challenge is that the target architecture may offer a locking capability much different than that of the source architecture. Specifically, what if each core has exactly one test-and-set register? A test-and-set register is used for checking and setting (or clearing) a condition all at the same time [14]. It is a low-level hardware register that enables the construction of higher level atomic operations. To provide synchronization and control routines across processes, the mutex lock and unlock routines are converted to acquire and release methods built around the target API. However, because a Pthread mutex and hardware test-and-set register are not exactly the same, performance varies when converting a synchronization-dependent application from Pthreads on a multicore system to the target application on the many-core architecture. These changes facilitate the transition from a multithreaded application to the program consisting of multiple processes.

After taking care of the larger changes, simple cleanup of the converted application is necessary. First, all *pthread_** functions are removed from the code. A pass to remove these functions is created which contains a hash table containing one entry for each function name. As the AST is traversed, each function name is compared using an O(1) lookup in the hash table. If the entry is found in the table the function is removed from the AST with all other code being preserved. Following this, all Pthread-specific data types are removed as well, though in a separate pass with its own hash table but utilizing the same methodology. Because core IDs are used instead of thread IDs in the target application, any occurrence of thread ID must be replaced with core ID. In particular, core ID is assigned by calling a function which returns the rank of the core that the process is running on. Within the main procedure a call to initialize the target API library must be inserted. This initialization function is placed right after the entry point to the main procedure. As the *pthread_join* and *pthread_exit* calls are removed, a finalize call is placed at the end of the main procedure to clean up execution at the end of the target program – inserted just before the return statement. Described in Appendix A, Algorithms 5, 6, 7 and 8 define code removal procedures. Appendix B contains Algorithms 9 and 10 defining code addition methods. The converted program for Example Code 4.1 is given in the transformed Example Code 4.2.

Example Code 4.2: Translated RCCE source code for Example Code 4.1

```
#include <stdio.h>
#include "RCCE.h"

int * ptr;
int * sum;

void * tf(void * tid)
{
  int tLocal = ((int)tid);
  sum[tLocal]+=tLocal;
  sum[tLocal]+=( * ptr);
}

int RCCE_APP(int * argc, char * argv[])
{
  RCCE_init(&argc, &argv);
  sum=(int * )RCCE_shmalloc((sizeof(int)*3));
  ptr=(int * )RCCE_shmalloc((sizeof(int)*1));
  int myID;
  myID=RCCE_ue();
  int tmp = 1;
  ptr=( & tmp);
  tf(((void * )myID));
  RCCE_barrier(&RCCE_COMM_WORLD);
  printf("Sum Array: %d\n",sum[myID]);
  RCCE_finalize();
  return(0);
}
```

Chapter 5

EXPERIMENTAL SETUP

Each experiment consists of either a multithreaded Pthread application or the same program converted via the translation framework to run on the multiprocess SCC architecture, through use of the RCCE library. RCCE is the C-based, low-level communication library purpose-built for the SCC architecture which supports explicit communication between cores as well as memory and power management routines [29]. The foundation of RCCE lies in one-sided *put* and *get* primitives similar to those of MPI. These functions work by moving data from the private memory and L1 cache of the sending core into the MPB whereby it is retrieved into the L1 cache and memory of the receiving core. In such a manner, the data moves from one core to another without either core accessing the off-chip shared memory. The compiled program, run the same way for each core, is written in such a way that each core does different work given different conditional inputs. One way different inputs are given to the program is through varying core IDs. In RCCE programs, each executable is "owned" by a Unit of Execution (UE) tied 1-to-1 to a core. At runtime, the UE, physical core and rank (a sequence number given to each participating core from 0 to $N-1$ where N processors are involved) are all linked, making it simple for a program to obtain core-specific information (such as a core ID) from the core it is executing on. This section details the test environment, programs, and infrastructure utilized during the analysis, translation, and experiments.

5.1 Architecture

For the experimental HSM platform the Intel Single-chip Cloud Computer (SCC) is utilized [13]. The 48-core non-coherent cache architecture features a unique on-die shared SRAM called the Message Passing Buffer (MPB). Through the MPB the cores may communicate a limited amount of data directly and bypass both the L2 cache and DRAM. The overall size of the MPB is quite limited at 384 KB, or 8 KB per core. For small messages this allows for very fast, very efficient inter-core communication, presenting a significant improvement over message passing in traditional cluster computing environments. The cores are connected to one another and to the off-chip memory through a mesh-grid of routers as shown in Figure 5.1. Each tile has a mesh interface unit (MIU) that connects to the router and facilitates data transfers. The mesh as a whole can operate up to a maximum frequency of 2 GHz which is significantly faster than the 1 GHz maximum processing frequency of the cores. The frequency of the mesh and the cores is variable and can be set in a variety of ways. First, the frequency for each core can be set all at the same time by setting the frequency of the entire chip. Second, groups of cores may have their frequency changed by changing the frequency of the power domain they fall under. Third, both of these steps can be carried out dynamically within a program by making procedure calls to the power management API.

With operating ranges of 0.7 V and 125 MHz (25 W at 50°C) up to 1.14 V and 1 GHz (125 W at 50°C) the SCC platform provides extensive control over power management at varying granularity making it a very flexible power-sensitive architecture. Each core is a full P54C Pentium-class processor. Up to 64 GB of memory is available for use as either private or shared space using the off-chip DRAM [19]. RCCE accommodates both the shared memory and message passing paradigms of

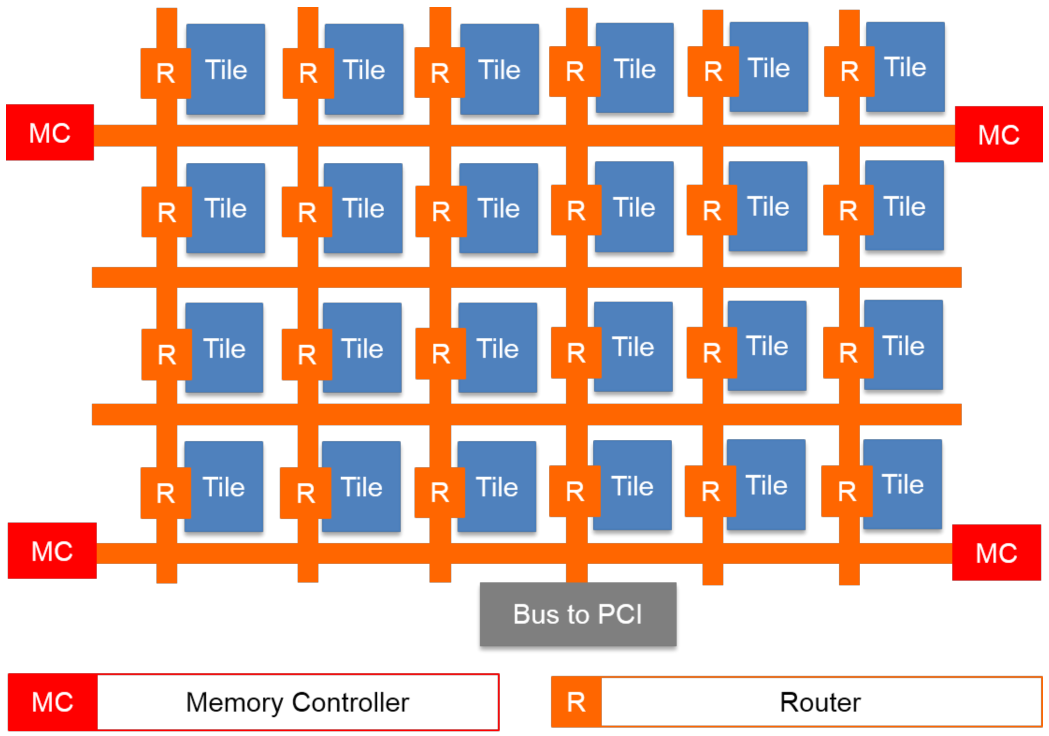

Figure 5.1: SCC core layout. Each tile is composed of two IA-32 P54C Pentium-class processors. Tile locality impacts memory access time relative to each memory controller.

sharing data. Each benchmark was run on the SCC, each core running Linux, at 800 MHz core frequency, 1600 MHz network mesh, and 1066 MHz off-chip DDR3 memory frequency – see Table 6.1. The Pthread benchmarks are built for 32 threads whereas the RCCE applications utilize 32 cores.

Being comprised of Pentium-class cores, the SCC utilizes an x86-based instruction set which helps facilitate a familiar programming environment. The Linux operating system can be run on *each* core for a total of 48 concurrent operating system environments running at once. Generic x86-compatible programs written in C and Fortran can be recompiled with supported Intel and gcc compilers and run easily on any single core of the SCC – in a manner similar to any general x86 chip.

All applications were compiled for the SCC using the Intel C++ compiler (icc) version 8.1 (gcc 3.4.5), and RCCE version 2.0.

5.2 Benchmarks

The transformation is optimized for enabling multithreaded programs to run on HSM architecture. Therefore, the primary focus is regarding applications that distribute the work into a set of tasks which can be executed in parallel over multiple compute units. The programs divide up their respective computation into a number of threads, each of which performs the same task on a subset of the data. These applications include a program to *Count Primes*, to do a *Pi Approxmiation*, 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*. The latter three programs exhibit a high degree of memory operations: allocating, copying and loop traversals. The former two applications perform more division and modulo compute operations rather than specifying high memory activity. Understanding this becomes crucial in properly handling data in the HSM environment.

A note about these particular benchmarks. In order to have a meaningful comparison between the performance of a given application – across the Pthreads software environment and the RCCE environment – it is beneficial to have a variety of benchmarks that operate in the same manner in these two multiprocessing paradigms. The intention is to have a program which takes the same input and produces the same output – with both Pthreads and RCCE source codes. Pursuant to the goals of this report, the intent was to utilize the Pthread-to-RCCE parser to convert existing Pthread benchmarks to RCCE source code. These are subsequently compiled to binary programs executable on the SCC. A wide variety of parallel applications and scientific programs exist. Despite this, obtaining benchmarks developed using Pthreads proved especially challenging. One reason is that standardized benchmarks are strictly controlled (regarding how they are modified and run), in order to preserve accurate reporting of results across many different platform configurations. If official results are desired then modification of a benchmark to utilize Pthreads instead of another parallel implementation, such as CUDA or OMP, must be done with the benchmark maintainers permission and some form of supervision. This brings up a second reason: numerous many-core applications are tailored towards scientific computing. These types of programs emphasize optimization, efficiency, and advanced operations using model-specific programming languages – including but not limited to, CUDA, OpenMP, OpenCL, C++ with Intel KML, and MPI [4] [5]. Finally, some benchmarks utilize well–aged programming languages such as Fortran and as such require translation to C, conversion to Pthreads, and then subsequent conversion to RCCE. The general-purpose nature of Pthreads lends itself well to platform-independent multithreaded programming but not very well to the architecture-specific performance benefits of more specialized programming languages and parallelism models. For this reason a set of common, albeit comparatively simple, parallel programs have been written in Pthreads and converted to RCCE using the analytic parser and translator utility. These were subsequently run on the SCC. The various workloads are separated into three categories: linear algebra programs, approximation and number theory applications, and memory operations benchmarks. The performance of these microbenchmarks is determined by using a timestamping function which returns the systems wall-clock time. Each application generates a timestamp just prior to launching threads as well as just after all the threads complete execution.

Algorithms 12, 13, 14, 15, and 16 describe these benchmarks in Appendix C.

5.3 Compilation Framework

Without this analysis and transformation, programs written in Pthreads (multithreaded programs) can only execute on a single core. To enable them to run on multiple cores requires convertion to RCCE programs – to use RCCE primitives for communication. The analysis and transformation is implemented in the source-to-source CETUS compiler framework [7]. Each component, or 'pass', of the 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 intermediate representation (IR) of the program remains in a self-consistent state. The *Driver* class brings together all passes and executes them in series, providing fine-grain analysis and making iterative changes to the IR. The software packages that Cetus requires are Java 1.6 from the OpenJDK runtime environment, and ANTLR 2.7.5. Cetus 1.3 runs on Linux Mint 12.

Chapter 6

RESULTS

Multithreaded applications do run on the SCC, however they can only take advantage of a single core. Each Pthread application is run on one core of the SCC and the time measured to obtain a baseline. In each program 32 threads compete for processor time which greatly reduces the efficiency of each given thread. The translation framework enables conversion of the Pthread applications to RCCE programs which can take advantage of 32 cores of the SCC.

Mapping shared data to off-chip shared memory improves performance by 32x

As an evaluation baseline each Pthread application is run on a single core of the SCC. Then a RCCE variant is generated for each program which takes advantage of 32 cores of the SCC and utilizes off-chip shared memory. Every program has its runtime measured. 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. 6.1 shows

Table 6.1: SCC Configuration

	RCCE	Pthreads
Core Frequency	800 MHz	800 MHz
Communication Network	1600 MHz	1600 MHz
Off-chip Memory	1066 MHz	1066 MHz
Execution Units	32 cores	32 threads

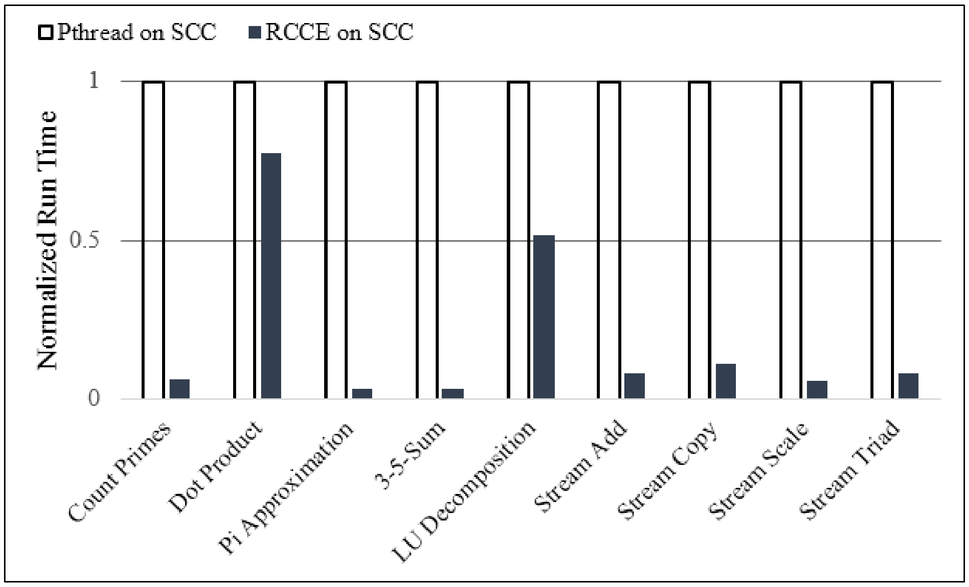

Figure 6.1: Performance of RCCE applications utilizing off-chip shared memory and 32 cores normalized to the performance of the 32-thread Pthread programs running on a single core.

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, this work of converting multi-threaded programs to run as HSM applications makes this comparison possible.

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. 6.2. As the MPB is a small, on-chip memory, programs which either exhibit a high degree of memory usage or those that balance memory use and core computation see the most performance improvement. For example, Stream already benefits from the parallelism via 32 cores, versus being run on a single core where each thread competed for processor time. In addition, when

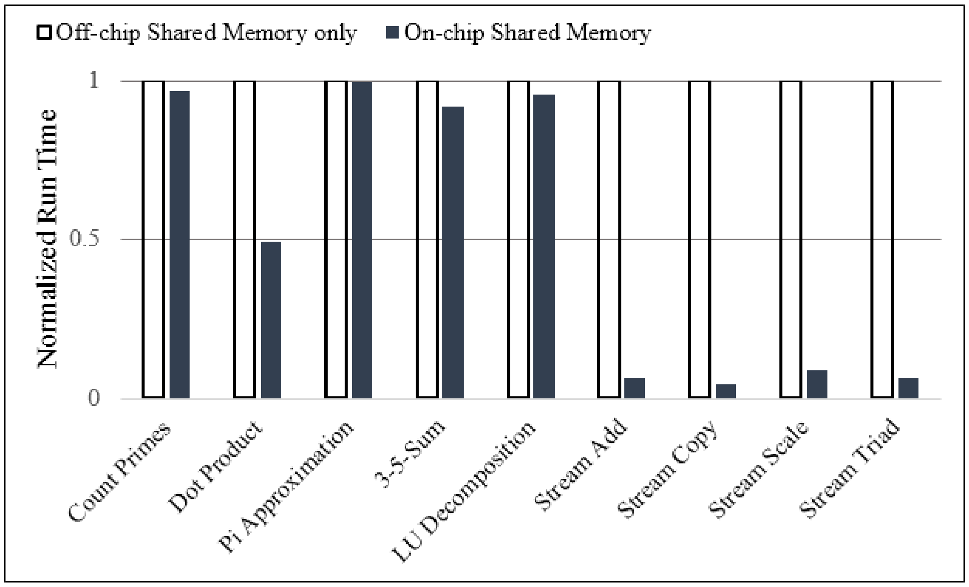

Figure 6.2: Run time performance comparison of RCCE programs utilizing shared off-chip memory against the on-chip shared memory provided by the MPB.

converted to utilize the MPB, not only are the memory accesses distributed across the cores but the locality for core-to-MPB is much closer than than of core-to-DRAM. Finally, transfers to and from the MPB 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, for example a few rows, may be allocated separately on the MPB.

Enabling Scalable Applications on HSM Architecture

Converting multi-threaded programs to take advantage of multiple cores of the HSM architecture enables scalability. In general this is application-dependent. However, programs which have a sufficiently large amount of computation and which transfer data between cores using the on-die MPB can gain significant performance benefits with increasing core count. Performance relative to scaling core count for Pi

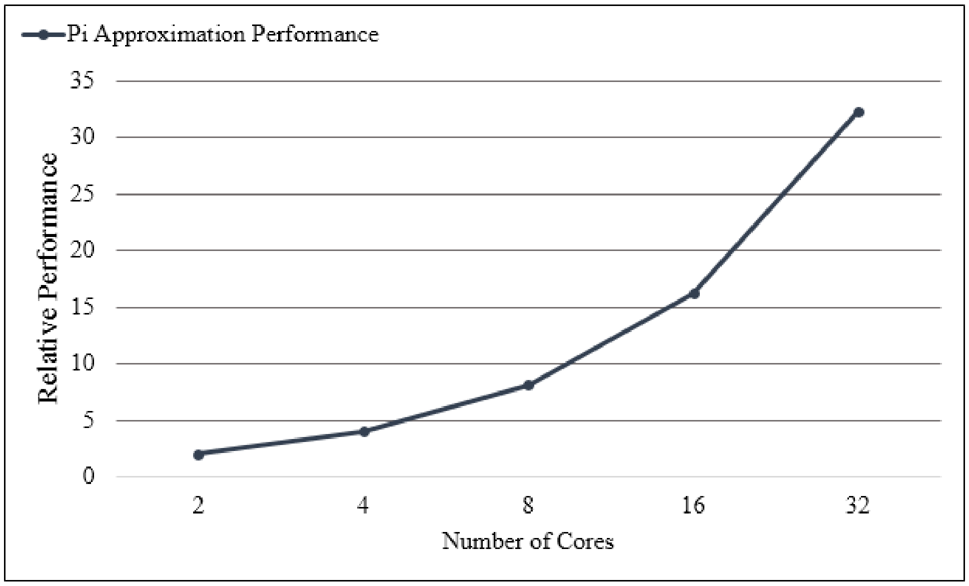

Figure 6.3: Relative performance improvement over single-core Pthread application of multiprocessor RCCE program with varying core count on SCC.

Approximation is shown in Fig. 6.3.

Chapter 7

FUTURE WORK

The compute potential of many-core processors is that of a datacenter-on-chip and merits further study as its use matures. The work presented here is a proof-of-concept parser-analyzer and translation utility which identifies shared data and efficiently maps it into memory regions on the many-core architecture. Both an abstract representation of the program, as well as a final product ready for deployment to the experimental SCC system, are generated. This work represents a preliminary foundation framework with potentially rewarding enhancements.

7.1 Translation Candidates Expansion

The analyzer accepts nearly all standard Pthread applications. However, parsing and/or translation may suffer in certain cases. For example, Pthread code wrapped within macros is inaccessible to the parser and cannot be sufficiently translated. One possibility is mapping the macro abstractions such as *CreateThread* and *Barrier* to their appropriate counterparts for the many-core architecture. This presents at least one more layer of abstraction and serves to make the parser too specialized for the general Pthread program. To provide the foundation most compatible with the Pthread specifications, this enhancement is left to subsequent improvements.

7.2 Core Count

Programs with large numbers of threads cannot be converted 1:1 with this experimental architecture. This is an artificial limitation as the parsing technique is actually scalable with the availability of additional cores. Along the same lines, programs with greater than 48 threads are currently not handled by this technique. However, [6] have implemented a many-to-one methodology to run multiple threads upon one core, and this work may provide a springboard to build upon. Since thousand-plus core count processors are possible with NCC architectures, such an avenue for further development is promising.

7.3 Code Optimizations

This work, although a proof-of-concept, does present optimizations based on memory space and locality. Many further optimizations, such as improving for data content and repeated code execution, are open to study.

Chapter 8

CONCLUSION

Developing and utilizing tools to better exploit the capabilities of many-core systems is crucial to enabling and unlocking the potential for improving performance and parallelization. The presented technique is used to convert otherwise incompatible, or inefficiently executing, programs by leveraging the Intel SCC through architecture-specific transformations. The described approach automatically analyzes the multithreaded source program and, extracting the properties of all variables, efficiently maps the shared data to available on-chip and off-chip shared memory. The procedure is well-defined, automated and repeatable which enables porting of Pthread applications for execution onto a many-core architecture. Experimental results from benchmarks indicate 32x performance improvement over the baseline when using off-chip memory alone, and 8x improvement when utilizing the on-chip memory over the off-chip memory, on average. The results demonstrate the suitability and performance benefits of enabling multi-threaded applications for efficient execution on HSM manycore architectures.

Already many-core systems have trickled down to the high-performance computing space from the supercomputing realm and are making their way further into specialized consumer applications. Speed and reusability are primary factors in adopting new architectures. Without new tools existing multithreaded applications will not be able to take full advantage of the many-core revolution, and opportunities for improving scalability might be overlooked. This technique seeks to address this shortcoming and demonstrates the suitability and performance benefits of enabling multithreaded applications for efficient execution on HSM many-core architectures.

BIBLIOGRAPHY

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

[2] Christian Bell, WY Chen, Dan Bonachea, and Katherine Yelick. Evaluating support for global address space languages on the Cray X1. In *Proceedings of the 18th annual international conference on Supercomputing*, pages 184–195, 2004. ISBN 1581138393. URL http://dl.acm.org/citation.cfm?id=1006209.1006236.

[3] Frank Bellosa and Martin Steckermeier. The performance implications of locality information usage in shared-memory multiprocessors. *J. Parallel and Distributed Comput.*, 37(1):113–121, 1996.

[4] Christian Bienia, Sanjeev Kumar, JP Singh, and Kai Li. The PARSEC benchmark suite: characterization and architectural implications. pages 72–81, 2008. URL http://dl.acm.org/citation.cfm?id=1454128.

[5] S Che, M Boyer, J Meng, D Tarjan, and J Sheaffer. Rodinia: Accelerating Compute-Intensive Applications with Accelerators, 2008. URL http://www.cs.virginia.edu/~skadron/wiki/rodinia/index.php/Rodinia:Accelerating_Compute-Intensive_Applications_with_Accelerators.

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

[7] Chirag Dave, Hansang Bae, Seung-Jai Min, Seyong Lee, Rudolf Eigenmann, and Samuel Midkiff. Cetus: A source-to-source compiler infrastructure for multicores. *Computer*, 42(12):36–42, 2009. doi: 10.1109/MC.2009.385. URL http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=5353460.

[8] Jason Duell. *Pthreads or Processes: Which is Better for Implementing Global Address Space languages?* PhD thesis, University of California, 2007. URL http://upc.lbl.gov/publications/JasonDuell_MS_report_final.pdf.

[9] Ben Elliston, Ross Johnson, Robert Colquhoun, John E. Bossom, Anders Norlander, Tor Lillqvist, Scott Lightner, Kevin Ruland, Mike Russo, Mark E. Armstrong, Lorin Hochstein, Peter Slacik, Mumit Gardian, Aurelio Medina, Graham Dumpleton, Tristan Savatier, Erik Hensema, Rich Peters, Todd Owen, Jason Nye, Fred Forester, Keven D. Clark, David Baggett, Paul Redondo, Scott McCaskill, Jef Gearhart, Arthur Kantor, Steven Reddie, Alexander Terekhov, Thomas Pfaff, Franco Bez, Louis Thomas, David Korn, Jr. Phil Frisbie, Ralf Brese, Prionx@juno.com, Max Woodbury, Rob Fanner, Michael Johnson, Nicholas Barry, Piet van Bruggen, Makoto Kato, Panagiotis E. Hadjidoukas, Will Bryant, Anuj Goyal, Gottlob Frege, Vladimir Kliatchko, Ramiro Polla, Daniel Richard G., and John Kamp. POSIX Threads (Pthreads) for Win32. URL http://sourceware.org/pthreads-win32/.

[10] David Geer. Chip makers turn to multicore processors. *Computer*, 38(5):11–13, 2005. ISSN 0018-9162. doi: 10.1109/MC.2005.160. URL http://ieeexplore.ieee.org/xpl/articleDetails.jsp?arnumber=1430623.

[11] Kalpana Gondi, A. P. Sistla, and V. N. Venkatakrishnan. Minimizing Lifetime of Sensitive Data in Concurrent Programs. In *Proceedings of the 4th ACM conference on Data and application security and privacy*, pages 171–174. ACM, 2014.

[12] 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.

[13] Jason Howard, Saurabh Dighe, SR Vangal, Gregory Ruhl, Nitin Borkar, Shailendra Jain, Vasantha Erraguntla, Michael Konow, Michael Riepen, Matthias Gries, Guido Droege, Tor Lund-Larsen, Sebastian Steibl, Shekhar Borkar, Vivek K. De, and R van der Wijngaart. A 48-core IA-32 processor in 45 nm CMOS using on-die message-passing and DVFS for performance and power scaling. *Solid-State Circuits, IEEE Journal of*, 46(1):173–183, January 2011. ISSN 0018-9200. doi: 10.1109/JSSC.2010.2079450. URL http://dx.doi.org/10.1109/JSSC.2010.2079450.

[14] Intel. Intel Architecture Software Developer's Manual. 2(243190):51–52, 1999.

[15] 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.

[16] M. Kandemir, J. Ramanujam, M. J. Irwin, N. Vijaykrishnan, I. Kadayif, and A. Parikh. Dynamic management of scratch-pad memory space. *Proceedings of the 38th annual Design Automation Conference*, pages 690–695, 2001.

[17] Henry Kasim, Verdi March, Rita Zhang, and Simon See. Survey on parallel programming model. *Network and Parallel Computing*, pages 266–275, 2008. URL http://link.springer.com/chapter/10.1007%2F978-3-540-88140-7_24.

[18] Tongping Liu and Emery D. Berger. SHERIFF: precise detection and automatic mitigation of false sharing. *ACM SIGPLAN Notices*, 46(10):3–18, 2011.

[19] T.G. Mattson, Michael Riepen, Thomas Lehnig, Paul Brett, Werner Haas, Patrick Kennedy, Jason Howard, Sriram Vangal, Nitin Borkar, Greg Ruhl, and Saurabh Dighe. The 48-core SCC processor: the programmer's view. In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–11. IEEE Computer Society, 2010. URL http://portal.acm.org/citation.cfm?id=1884676.

[20] Timothy G. Mattson, Rob Van der Wijngaart, and Michael Frumkin. Programming the Intel 80-core network-on-a-chip terascale processor. In *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, 2008. ISBN 9781424428359. URL http://dl.acm.org/citation.cfm?id=1413409.

[21] 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 Transactions on Design Automation of Electronic Systems (TODAES)*, 5(3):682–704, 2000.

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

[23] Eli Pozniansky and Assaf Schuster. Efficient on-the-fly data race detection in multithreaded C++ programs. *ACM SIGPLAN Notices*, 38(10), 2003.

[24] Ravishankar Rao, Sarma Vrudhula, and Chaitali Chakrabarti. Throughput of multi-core processors under thermal constraints. In *Proceedings of the 2007 international symposium on Low power electronics and design*, pages 201–206. ACM New York, NY, USA, 2007. URL http://portal.acm.org/citation.cfm?id=1283824.

[25] Jai Rawat. Static Analysis of Cache Performance for Real-Time Programming. Master's thesis, Iowa State University, 1993.

[26] Amitabha Roy and Timothy M. Jones. ALLARM: Optimizing Sparse Directories for Thread-Local Data. pages 1–6. IEEE, 2014.

[27] Stefan Savage, Michael Burrows, Greg Nelson, Patrick Sobalvarro, and Thomas Anderson. Eraser: A dynamic data race detector for multithreaded programs. *ACM Transactions on Computer Systems (TOCS)*, 15(4):391–411, 1997.

[28] Herb Sutter. The free lunch is over: A fundamental turn toward concurrency in software. *Dr. Dobb's Journal*, 30(3):202–210, 2005. URL http://mondrian.die.udec.cl/~mmedina/Clases/ProgPar/Sutter-TheFreeLunchisOver.pdf.

[29] Rob F. van der Wijngaart, Timothy G. Mattson, and Werner Haas. Light-weight Communications on Intel's Single-Chip Cloud Computer Processor. *ACM SIGOPS Operating Systems Review*, 45(1):73–83, February 2011. ISSN 01635980. doi: 10.1145/1945023.1945033. URL http://portal.acm.org/citation.cfm?doid=1945023.1945033.

[30] Christoph von Praun and Thomas R. Gross. Static conflict analysis for multi-threaded object-oriented programs. *ACM SIGPLAN Notices*, 38(5):115–128, 2003.

[31] Boris Weissman. Performance counters and state sharing annotations: a unified approach to thread locality. *ACM SIGPLAN Notices*, 33(11):127–138, 1998.

[32] Min Xu, Rastislav Bodík, and Mark D. Hill. A serializability violation detector for shared-memory server programs. *ACM SIGPLAN Notices*, 40(6):1–14, 2005.

[33] Yu Yang, Xiaofang Chen, and Ganesh Gopalakrishnan. Inspect: A runtime model checker for multithreaded C programs. *University of Utah, USA, Tech. Rep*, 2008.

APPENDIX A

CODE REMOVAL ALGORITHMS

Algorithm 5 Remove Pthread Join Calls

```
Input: Program IR (PIR)
 1: for all objects in PIR do
 2:    if object is a function then
 3:       if function name is $pthread\_join$ then
 4:          if function is within a loop then
 5:             Remove the loop from PIR
 6:          else
 7:             Remove the function from PIR
 8:          end if
 9:       end if
10:    end if
11: end for
```

Algorithm 6 Remove Pthread Self Calls

```
Input: Program IR (PIR)
1: for all objects in PIR do
2:    if object is a function then
3:       if function name is $pthread\_self$ then
4:          Replace $pthread\_self$ with $RCCE\_ue$
5:       end if
6:    end if
7: end for
```

Algorithm 7 Remove Pthread Data Types

```
Input: Program IR (PIR)
1: Prepopulate a HashSet with all pthread data types
2: for all objects in PIR do
3:     if object is a variable declaration then
4:         if specifier name $\exists \in$ HashSet then
5:             Remove object from PIR
6:         end if
7:     end if
8: end for
```

Algorithm 8 Remove Pthread API Calls

```
Input: Program IR (PIR)
1: Prepopulate a HashSet with all pthread API calls
2: for all objects in PIR do
3:     if object is a function then
4:         if function name $\exists \in$ HashSet then
5:             Remove object from PIR
6:         end if
7:     end if
8: end for
```

APPENDIX B

CODE ADDITION ALGORITHMS

Algorithm 9 Add RCCE Init Call

```
Input: Program IR (PIR)
 1: for all objects in PIR do
 2:    if object is a procedure then
 3:       if procedure name is $main$ then
 4:          Create a new function called $RCCE\_init$
 5:          As first argument specify &argc
 6:          As second argument specify &argv
 7:          $f \leftarrow$ first statement in $main$
 8:          Insert $RCCE\_init$ into $main$ before $f$
 9:       end if
10:    end if
11: end for
```

Algorithm 10 Add RCCE Finalize Call

```
Input: Program IR (PIR)
 1: for all objects in PIR do
 2:    if object is a procedure then
 3:       if procedure name is $main$ then
 4:          Create a new function called $RCCE\_finalize$
 5:          $l \leftarrow$ second-to-last object in $main$
 6:          Insert $RCCE\_init$ after $l$
 7:       end if
 8:    end if
 9: end for
```

APPENDIX C

BENCHMARK PSEUDOCODE

Algorithm 11 Count Primes (Serial)

```
Input: limit, prime, total
 1: total ← 0
 2: for $i \leftarrow 2$ to $i \leq$ limit do
 3:     $i \leftarrow i + 1$
 4:     prime ← 1
 5:     for $j \leftarrow 2$ to $j < i$ do
 6:         if $i$ mod $j == 0$ then
 7:             prime ← 0
 8:             break
 9:         end if
10:     end for
11:     total ← total + prime
12: end for
13: return total
```

Algorithm 12 Pi Approximation

```
1: step = $\frac{1}{\text{total number of steps}}$
2: while iteration ≤ total number of steps do
3:     $x$ = (iteration + 0.5) × step
4:     sum = sum + $4.0/(1 + x \times x)$
5:     iteration ← iteration + 1
6: end while
7: pi = step × sum
```

Algorithm 13 Stream Add (Serial)

```
Input: ARRAY a, b, c
Input: limit
1: for $j = 0$ to $j < limit$ do
2:     c[j] ← a[j] + b[j]
3: end for
```

Algorithm 14 Stream Copy (Serial)

```
Input: ARRAY a, c
Input: limit
1: for $j = 0$ to $j < limit$ do
2:     c[j] ← a[j]
3: end for
```

Algorithm 15 Stream Scale (Serial)

```
Input: ARRAY b, c
Input: limit
1: for $j = 0$ to $j < limit$ do
2:     b[j] ← 3.0× c[j]
3: end for
```

Algorithm 16 Stream Triad (Serial)

```
Input: ARRAY a, b, c
Input: limit, SCALAR
1: for $j = 0$ to $j < limit$ do
2:     a[j] ← b[j] +3.0× b[j]
3: end for
```